A Robust Architecture for Distributed Inference in Sensor Networks

Mark A. Paskin
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720
mark@paskin.org

Carlos E. Guestrin
Berkeley Research Center
Intel Corporation
Berkeley, CA 94704
guestrin@cs.cmu.edu

ABSTRACT

Many inference problems that arise in sensor networks can be formulated as a search for a global explanation which is consistent with local information known to each node. Examples include probabilistic inference, pattern classification, regression, and constraint satisfaction. Centralized inference algorithms for these problems often take the form of message passing on a special type of data structure called a junction tree, which has the important property that local consistency between adjacent nodes is sufficient to ensure global consistency between all pairs of nodes.

In this paper we present an architecture for distributed inference in sensor networks which is robust to unreliable communication and node failures. In our architecture, the nodes of the sensor network assemble themselves into a stable spanning tree, and use an asynchronous message passing algorithm to transform the spanning tree into a junction tree for the inference problem. Using asynchronous message passing on this junction tree, the nodes can solve the inference problem efficiently and exactly. We also present an efficient distributed algorithm for optimizing the choice of junction tree so that the communication and computational cost of inference can be minimized. We present experimental results for three applications—distributed sensor calibration, optimal control, and sensor field modeling—on data from a real sensor network deployment.

1. INTRODUCTION

Sensor networks consist of nodes which can measure characteristics of their local environment, perform local computations, and communicate with each other over a wireless network. In recent years, advancements in hardware and low-level software have led to viable, multi-hundred node sensor networks that can instrument unstructured environments at an unprecendented scale. The most popular application of sensor networks to date has been environmental monitoring. In these deployments the sensor data is downloaded from the network for later analysis [15, 14] or the network aggregates the measurements using simple local operations that compute, for example, averages, maxima, or histograms [10, 11].

More advanced applications, such as tracking and actuation, require sensor networks that can solve significantly more complex problems like sensor fusion, data modelling, prediction, and optimal control. Solving these inference problems requires combining all of the nodes' local measurements to generate a globally consistent view of the environment, or in the case of actuation, coherent controls to change it. For example, a node with a temperature sensor can measure only the temperature at its location; if the node's sensor is biased, it is impossible to infer the true temperature from the measurement. However, as we show in the next section, by combining this local information with the measurements of the other sensors, we can solve a global inference problem that automatically calibrates the temperature sensors at all nodes.

Most existing inference algorithms for sensor networks focus on solving specific tasks such as computing contour levels of sensor values [16], distributed sensor calibration [2], or target tracking [20]. In this paper, we present the first general architecture for inference in sensor networks which can solve a wide range of inference problems including probabilistic inference problems (e.g., sensor calibration and target tracking), regression (e.g., data modelling and contour finding), and optimization (e.g., actuator control, decision-making and pattern classification). At the core of the architecture is a powerful data structure called a *junction tree*, which allows all of these inference problems to be solved by simple asynchronous message passing algorithms [1].

Recently, there have been some proposals to use existing centralized inference algorithms in sensor networks [5, 18, 3]. However, these inference approaches are not as general as ours, and more importantly, they do not fully address the practical issues that arise in real deployments: communication over wireless networks is unreliable due to noise and packet collisions; the wireless network topology changes over time; and, nodes can fail for a number of reasons, often because the battery dies. To address these challenges, we have found that it is insufficient to implement existing algorithms on the sensor network architecture; fundamentally new algorithms are required.

To address these robustness issues we propose a novel architecture consisting of three distributed algorithms: *spanning tree formation*, *junction tree formation*, and *message passing*. The nodes of the sensor network first organize them-

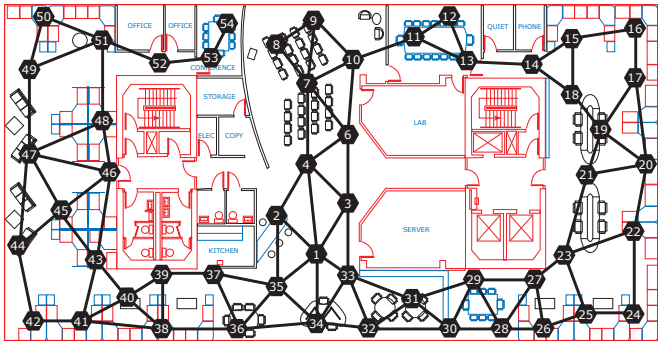


Figure 1: ***Lab deployment. The Markov graph for the nodes' temperature variables is overlaid; neighboring temperature variables are directly correlated, and those joined by paths are indirectly correlated.***

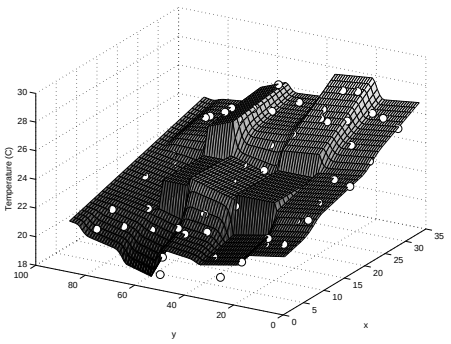


Figure 2: ***The temperature measurements from the deployment in Figure 1 and a regressed function.***

selves into a spanning tree so that adjacent nodes have high-quality wireless connections. By communicating along the edges of this tree, the nodes compute the information necessary to transform the spanning tree into a junction tree for the inference problem. In addition, these two algorithms interact to optimize the spanning tree so the computation and communication required by inference is minimized. Finally, the inference problem is solved exactly via asynchronous message passing on the junction tree. By quickly responding to changes in each others' states, these three algorithms can efficiently recover from communication and node failures. We demonstrate the power of our architecture on three separate inference tasks using data from a real sensor network deployment.

1.1 Inference problems in sensor networks

Our inference architecture is useful for solving a wide variety of inference problems that arise in sensor networks. In this section we present three such problems, probabilistic inference, regression, and optimal control, to give a sense of the range of problems addressed.

1.1.1 Probabilistic inference

Probabilistic inference is a powerful tool for solving problems where we must reason with partial or noisy information [4]. These problems often arise in sensor networks, where the sensor measurements give an incomplete view of the instrumented environment. We will focus on a particular application to which probabilistic inference may be applied: distributed sensor calibration. After a sensor network is deployed, the sensors can be adversely affected by the environment, leading to biased measurements. The distributed sensor calibration task involves automatic detection and removal of these biases [2]. This is possible because the quantities measured by nearby nodes are correlated but the biases of different nodes are independent.

Figure 1 shows a sensor network of 54 nodes that we deployed in our lab; the data from this deployment indicates that the temperatures at the nodes' locations fluctuate throughout the day, but that there are significant correlations between the temperatures at nearby nodes. We fit a probabilistic model to this data set where each node i has three associated variables: its observed temperature measurement M_i, the true (unobserved) temperature at its location T_i, and the (unobserved) bias of its temperature sensor B_i.

The probability model for this data has two main parts: a prior model representing the correlations between temperatures at various locations, and sensor models for each node which give the likelihood of a possible sensor measurement given the true temperature and sensor bias. The temperature prior is a structured model called a Markov network [4], shown in Figure 1, which compactly represents direct and indirect correlations between the temperature variables. The complete joint probability density is given by:

$$\begin{aligned}&\Pr\{T_1,\ldots,T_N,B_1,\ldots,B_N,M_1,\ldots,M_N\}\\ &\propto \underbrace{\prod_{(i,j)\in\mathcal{E}} \psi_{ij}(T_i,T_j)}_{\text{temperature prior}} \prod_{n=1}^{N} \underbrace{\Pr\{B_n\}}_{\text{bias prior}} \underbrace{\Pr\{M_n \mid B_n, T_n\}}_{\text{measurement model}} \end{aligned} \quad (1)$$

where $\psi_{ij}(T_i,T_j)$ is a factor (or part) of the Markov network that defines the prior over temperature. There is one such factor for each of the edges in the Markov network in Figure 1, and they are learned from previous observations from the network.

Assume we have obtained some sensor measurements $\overline{\mathbf{m}}_{1:N} = (\overline{m}_1,\ldots,\overline{m}_N)$. By plugging these into the joint density (1) and marginalizing out all variables except T_i, we get the posterior distribution of T_i given the measurements:

$$\Pr\{t_i \mid \overline{\mathbf{m}}_{1:N}\} \propto \sum_{\mathbf{b}_{1:N}} \sum_{\mathbf{t}_{\setminus i}} \Pr\{t_i, \mathbf{t}_{\setminus i}, \mathbf{b}_{1:N}, \overline{\mathbf{m}}_{1:N}\} \quad (2)$$

This probability distribution for T_i represents an estimate of the temperature at node i using all of the sensor measurements to filter out sensor bias and noise. Under our model learned from the deployment data, we calculate that if the sensor biases have a standard deviation of 1°C, then in expectation these posterior estimates eliminate 44% of the bias. As the sensor network becomes denser, leading to more strongly correlated variables, a larger fraction of the bias can be automatically eliminated. For details on this model, please see [17].

1.1.2 Regression

Many current sensor network deployments are used in a "data gathering mode": all of the network's measurements

are uploaded to a central location. Transmitting all of the measurements can be wasteful because the measurements at nearby locations are often correlated (such as the temperature measurements of the previous example). In other words, the effective dimensionality of the data in these networks is often significantly lower than the total number of sensor measurements. Regression, or function fitting, is a powerful and general framework for maintaining the structure of the sensor field while significantly decreasing the communication required to access it [8]. In *linear regression*, the sensor field is modeled by a weighted combination of basis functions:

$$\widehat{f}(x,y,t) \triangleq \sum_{j=1}^{k} w_j b_j(x,y,t).$$

Here, $\widehat{f}(x,y,t)$ represents an approximation to the value of the sensor field at location (x,y) at time t, and the $b_j(x,y,t)$ are basis functions which are chosen in advance. The weights w_j are optimized to minimize the sum squared error between the observed measurements and the model $\widehat{f}$:

$$\mathbf{w}^* \triangleq \operatorname*{argmax}_{\mathbf{w}} \sum_{i=1}^{N} \left(\overline{m}_i - \sum_{j=1}^{k} w_j b_j(x_i,y_i,t_i) \right)^2, \quad (3)$$

where (x_i,y_i,t_i) is the location of node i and the time measurement $\overline{m}_i$ was obtained. The optimal weights $\mathbf{w}^*$ can be found by solving a linear system; these weights and their basis functions then provide a structured summarization of the original data. Figure 2 shows the result of fitting such a function to temperature data from our deployment.

In the general case, computing the optimal weights requires solving a dense linear system, which is expensive to solve in distributed settings. *Kernel linear regression* is a specialization of this technique that can effectively model local correlation in the data with significantly less computational cost. In this case, each basis function has bounded support (a region of influence), and the optimal weights are the solution to a *sparse* linear system. As shown in [8], these regression problems can be solved efficiently using the junction tree data structure and message passing algorithm.

1.1.3 Optimal control

Other interesting inference problems arise when nodes can control their environment to achieve some end. Consider a greenhouse deployment where nodes actuate the blinds to achieve specific desired light levels at different locations. The light level measured by each node will depend on the state of nearby blinds, and nearby nodes may have conflicting desires. To achieve the setting of the blinds that are best for all of the nodes, we can specify for each node a *reward function* which specifies its local utility for a setting of the blinds given its current light measurement. For example, the reward function for node 1 may be $Q_1(A_1,A_2,A_3;\overline{m}_1)$ indicating the utility of all settings A_1,A_2,A_3 of the nearby blinds given its current measurement $\overline{m}_1$. Given this reward function, node 1's preferred blind setting is given by:

$$\operatorname*{argmax}_{a_1,a_2,a_3} Q_1(a_1,a_2,a_3;\overline{m}_1).$$

In the general problem where we have multiple nodes, each with its own desired light level. Here, balance the nodes' competing desires by maximizing the sum of all reward functions:

$$\operatorname*{argmax}_{\mathbf{a}} \sum_i Q_i(\mathbf{a}_i;\overline{m}_i), \quad (4)$$

where $\mathbf{a}$ is a setting for all blinds, and $\mathbf{a}_i$ is the restriction of this setting to the blinds that affect the light at the location of node i. If solved naïvely, this optimization problem is very expensive; e.g., if each blind setting $A_i \in \{\mathsf{up},\mathsf{down}\}$, then enumerating the possible settings requires exponential time. Fortunately, junction trees again provide an abstraction for solving this problem efficiently. We refer the reader to [9] for more details, and for an algorithm for obtaining the functions Q_i.

1.2 Message passing on junction trees

The three inference problems described above are instances of a large class of inference problems that can be solved efficiently by message passing on a data structure called a **junction tree**. In this section we provide an introduction to inference using junction trees; for more detail see [1].

1.2.1 Inference problems

In this section we show how each of these three inference problems is an example of a general class of inference problems. This requires us to introduce a couple of simple abstractions: variables, which are the unknown quantities of interest; and factors, which represent partial knowledge about sets of variables. To make these notions concrete, we will use probabilistic inference as a running example and mention only briefly how regression and optimal control can also be represented in these terms; for more detail see [8, 9]. For readers familiar with database systems, we also make connections to the relational algebra used in relational query processing, which is yet another example of this problem class.

Each of the problems described above can be viewed as inference about some set of unknown quantities which we will call **variables**. We will use capital letters to represent variables, e.g., $X,Y,\ldots$, and boldface capital letters to represent sets of variables, e.g., $\mathbf{C},\mathbf{D},\ldots$. Each inference problem has an associated set of variables, which we will denote $\mathbf{V}$.

Example 1 (variables). In probabilistic inference, $\mathbf{V}$ is the set of unobserved random variables; in the calibration problem described above, $\mathbf{V}$ contains for each node i of the sensor network the unknown temperature T_i at node i's location and the bias B_i of its temperature sensor. (The temperature measurement M_i is an observed quantity, so it is not in $\mathbf{V}$.)

In regression, the variables in the problem are the optimal weights w_i of the basis functions, while in the control problem, the variables are the optimal controls A_i. In relational query processing, variables are attributes of the database. □

Solving these inference problems requires us to operate on information about the variables: some of this information may be obtained from sensor measurements, and some may be prior knowledge. We will encode both types of knowledge in terms of factors. A **factor** represents partial knowledge

about a subset of the variables, which is called the **domain** of the factor. We use lower case Greek letters (e.g., ϕ, ψ, π, etc.) to represent factors, and we write $\phi : \mathbf{D}$ to represent that ϕ is a factor whose domain is $\mathbf{D} \subseteq \mathbf{V}$.

Example 2 (factors). In probabilistic inference, a factor is a term of the posterior joint probability density, such as a local prior or conditional probability density function, or a likelihood function. Consider the joint probability density

$$\Pr\{t_1, b_1, m_1\} = \Pr\{t_1\} \Pr\{b_1\} \Pr\{m_1 \,|\, t_1, b_1\}$$

over the variables T_1, B_1, M_1. If we observe $M_1 = \overline{m}_1$, then the posterior joint density is

$$\Pr\{t_1, b_1 \,|\, \overline{m}_1\} \propto \Pr\{t_1\} \Pr\{b_1\} \Pr\{\overline{m}_1 \,|\, t_1, b_1\} \qquad (5)$$

i.e., it is proportional to a product of functions over the unobserved variables. The factors of this problem are

$$\begin{aligned} \phi_1 : \{T_1\} \quad &\text{defined by} \quad \phi_1(t_1) = \Pr\{t_1\} \\ \phi_2 : \{B_1\} \quad &\text{defined by} \quad \phi_2(b_1) = \Pr\{b_1\} \\ \phi_3 : \{T_1, B_1\} \quad &\text{defined by} \quad \phi_3(t_1, b_1) = \Pr\{\overline{m}_1 \,|\, t_1, b_1\} \end{aligned}$$

Because functions are hard to represent and manipulate, these factors are represented by a set of parameters. For example: if the variables T_1 and B_1 each have k values, then $\phi_3(t_1, b_1)$ is often represented as a $k \times k$ table of function values; if they are continuous and Gaussian, then $\phi_3(t_1, b_1)$ can be represented by a 2×1 vector and a 2×2 matrix that parameterize a log-quadratic function [4].

In the control problem, the factors are the Q_i functions. In regression, each factor consists of a matrix and a vector summarizing the impact of some measurements on components of the optimal weight vector. In relational queries, a factor is a relation (or table of tuples), and its domain is its attribute set. □

Inference consists of two operations on factors; the first is combination. $\otimes$ is an operator that combines the information represented by two factors into a single factor. The **combination of ϕ and ψ** is written $\phi \otimes \psi$, and its domain is the union of the domains of ϕ and ψ:

$$\phi : \mathbf{C} \text{ and } \psi : \mathbf{D} \implies \phi \otimes \psi : \mathbf{C} \cup \mathbf{D} \qquad (6)$$

We require that $\otimes$ is symmetric and associative, so that combination is an order-independent operation.

Example 3 (combining factors). In probabilistic inference, combination is simply multiplication. For example, the combination of two factors $\phi : \{T_1, T_2\}$ and $\psi : \{T_1, T_3\}$ is $\pi : \{T_1, T_2, T_3\}$ where

$$\pi = \phi \otimes \psi \iff \pi(t_1, t_2, t_3) = \phi(t_1, t_2) \times \psi(t_1, t_3)$$

Thus, the posterior density (5) can be written as

$$\bigotimes_{i=1}^{3} \phi_i \stackrel{\triangle}{=} \phi_1 \otimes \phi_2 \otimes \phi_3$$

(Order independence makes this notation unambiguous.)

In regression and control problems, the combination operation corresponds to addition; in the former we add matrices and vectors, and in the latter Q_i functions. In the relational algebra, $\otimes$ is the join operator $\bowtie$, which combines two relations to produce a relation with the union of their attributes. □

The second operation is summarization. $\bigoplus$ is an operator that takes a factor ψ and a set of variables $\mathbf{S}$ and produces a new factor summarizing the information ψ represents about $\mathbf{S}$. The **summary of ϕ to $\mathbf{S}$** is written $\bigoplus_{\mathbf{S}} \phi$, and its domain contains those variables in the domain of ϕ that are also in $\mathbf{S}$:

$$\phi : \mathbf{D} \implies \bigoplus_{\mathbf{S}} \phi : \mathbf{D} \cap \mathbf{S} \qquad (7)$$

Like combination, we require that summary is an order-independent operation: $\bigoplus_{\mathbf{S}} \bigoplus_{\mathbf{T}} \phi = \bigoplus_{\mathbf{S} \cap \mathbf{T}} \phi$.

Example 4 (summarizing factors). In probabilistic inference, the summary operator is marginalization. For example, the summary of $\phi : \{T_1, T_2, T_3, T_4\}$ down to $\{T_1, T_2\}$ is computed by marginalizing out all variables but T_1 and T_2:

$$\psi = \bigoplus_{\{T_1, T_2\}} \phi \iff \psi(t_1, t_2) = \sum_{t_3} \sum_{t_4} \phi(t_1, t_2, t_3, t_4)$$

In our control problem, the summarization operation computes the maximum possible value of a Q function for settings of a subset of the control variables. In regression, the summarization operation corresponds to steps of the Gaussian elimination algorithm for solving linear systems of equations. In the relational algebra, the summary operator is the project operator π, which eliminates unneeded attributes. (It can also include a selection operators σ which selects a subset of the tuples.) □

We now have the concepts necessary to give a formal definition of the inference problem. We have a collection of factors $\Phi = \{\phi_1, \phi_2, \ldots, \phi_k\}$ that represent our knowledge about the variables. In the inference problem, we must combine these factors to integrate our knowledge, and then summarize the result to a set of **query variables $\mathbf{Q}$**:

$$\beta = \bigoplus_{\mathbf{Q}} \bigotimes_{i=1}^{k} \phi_i \qquad (8)$$

Example 5 (inference problems). The posterior marginal of T_1 given $M_1 = \overline{m}_1$ is given by marginalizing out all variables except T_1 out of (5):

$$\Pr\{t_1 \,|\, \overline{m}_1\} \propto \sum_{b_1} \Pr\{t_1\} \Pr\{b_1\} \Pr\{\overline{m}_1 \,|\, t_1, b_1\}$$

This is an inference problem of the form (8) where $\mathbf{Q} = \{T_1\}$:

$$\beta(t_1) = \Pr\{t_1 \,|\, \overline{m}_1\} \iff \beta = \bigoplus_{\{T_1\}} \bigotimes_{i=1}^{3} \phi_i$$

The inference problems in regression and control are specified in a similar fashion, leading to the solution of Equations (3) and (4). In the case of regression, the result of inference is the optimal weights for a subset of the basis functions; in control the result specifies the maximum possible utility obtainable for different settings of some control

variables. In the relational algebra, the result of inference is a relation formed by joining relations and applying projection and selection operations. □

1.2.2 Algebraic structure in inference problems

So that our inference problems can be solved efficiently, we require that the combination and summary operators satisfy some simple algebraic properties. In addition to the order-independence properties mentioned above, we define the **null factor** $\mathbf{1} : \emptyset$, which represents a lack of information about any variables, so that $\psi \otimes \mathbf{1} = \psi$ and $\bigoplus_{\mathbf{S}} \mathbf{1} = \mathbf{1}$.

The crucial property that yields efficient inference algorithms is that combination must distribute over summary. Formally, this is written

$$\psi : \mathbf{D} \text{ and } \mathbf{D} \subseteq \mathbf{S} \implies \bigoplus_{\mathbf{S}} (\psi \otimes \phi) = \psi \otimes \bigoplus_{\mathbf{S}} \phi \quad (9)$$

In other words, when computing a summary of a combination of factors, we can "push" the summary past any factors whose domains are contained in the summary set.

Example 6 (algebraic structure). In probabilistic inference, it is easy to verify the order-independence of combination and summary, as these properties are inherited directly from multiplication and addition. Distributivity also follows since

$$a(b + c) = ab + ac.$$

For example, let $\phi : \{T_2, T_3\}$ and $\psi : \{T_1, T_2\}$ be two factors. Say we wish to summarize their combination to $\{T_2, T_3\}$:

$$\bigoplus_{\{T_2,T_3\}} \phi \otimes \psi = \sum_{t_1} \phi(t_2, t_3) \times \psi(t_1, t_2) \quad (10)$$

$$= \phi(t_2, t_3) \times \sum_{t_1} \psi(t_1, t_2) = \phi \otimes \bigoplus_{\{T_2,T_3\}} \psi \quad (11)$$

If T_1, T_2, and T_3 each take k values, then $\pi(t_2, t_3)$ is a table of k^2 entries. If we used (10) to compute it, we would require $O(k^3)$ time and space to form the combination $\phi \otimes \psi$, which is a table with k^3 entries. Using (11) instead avoids forming this table, and requires only $O(k^2)$ time and space.

Our control problem similarly exploits the distributivity of addition over maximization:

$$a + \max(b, c) = \max(a + b, a + c).$$

In regression, this property translates into the distributivity of matrix addition over Gaussian elimination steps. In the relational algebra, distributivity allows us to "push" projections and selections past joins to optimize query processing. □

Understood procedurally, the inference problem (8) can be interpreted as a directive to combine together all sources of knowledge into a single factor, and then summarize the result to a set of query variables. But this method of solving the inference problem forces us to compute a factor over all of the variables, which can be expensive or intractable. For example: in the regression problem each factor is represented by a matrix and vector that requires space quadratic in the size of the domain; in the probabilistic inference and optimization problems, the factors may be tables of numbers whose sizes scale exponentially with the domain size.

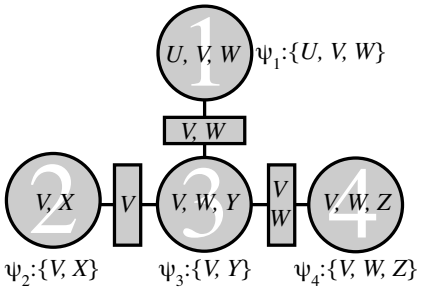


Figure 3: ***An example junction tree. The nodes are numbered 1–4, and are labelled with their cliques; e.g.,*** $\mathbf{C}_4 = \{V, W, Z\}$***. Next to each node*** i ***is its factor*** $\psi_i : \mathbf{D}_i$***. Each edge*** (i, j) ***is labelled with a box containing the separator*** $\mathbf{S}_{ij} = \mathbf{C}_i \cap \mathbf{C}_j$***.***

When the factors of an inference problem have significant locality structure, i.e., the factors' domains are small, the algebraic properties of combination and summary can lead to efficient algorithms for solving the inference problem (8). In particular, the distributivity property (9) allows us to perform summarizations early, instead of forming the unweildy combination of all of the factors. We now present the junction tree algorithm, which uses this property to its full potential to minimize the cost of inference.

1.2.3 Junction trees

To take full advantage of this locality structure, we require a data structure called a junction tree.

Definition 1 (junction tree). A **junction tree** is an undirected tree where each node i has

- a factor $\psi_i : \mathbf{D}_i$ (called the **local factor at** i) and
- a set of variables $\mathbf{C}_i \supseteq \mathbf{D}_i$ that contains the domain of ψ_i (called the **clique at** i)

and the **running intersection property** holds:

> If a variable X is in two cliques $\mathbf{C}_i$ and $\mathbf{C}_j$ in the junction tree, then it must also be in all cliques on the (unique) path between $\mathbf{C}_j$ and $\mathbf{C}_j$.

For each edge (i, j), the **separator** $\mathbf{S}_{ij} \triangleq \mathbf{C}_i \cap \mathbf{C}_j$ contains the variables shared by the cliques at its endpoints.

Example 7 (junction trees). Figure 3 shows an example junction tree with its cliques, factors, and separators. Notice that for each node i, the clique $\mathbf{C}_i$ contains the domain $\mathbf{D}_i$ of its factor ψ_i; e.g., $\mathbf{C}_3 = \{V, W, Y\} \supseteq \{V, Y\} = \mathbf{D}_3$. Notice also that the running intersection property holds: for example, because $V \in \mathbf{C}_4$ and $V \in \mathbf{C}_1$, it must be (and is) in $\mathbf{C}_3$, which is on the path between them. □

Below we present an inference algorithm which achieves global consistency by ensuring local consistency between neighbors in the junction tree. The running intersection property is required because if two nodes i and j must agree on their knowledge regarding a variable X, then the nodes on the path between them must be willing to "carry information" about X.

1.2.4 Message passing

The junction tree representation forms the basis of an efficient algorithm for solving the inference problem. In this algorithm each node of the junction tree passes a single message (which is a factor) to each of its neighbors. By combining its local factor ψ_i with the messages it receives, each node i can solve an inference problem whose the query variables are its associated clique $\mathbf{C}_i$. The messages are given by:

Definition 2 (message). Let j and k be neighbors in the junction tree. The **message from j to k** is the factor

$$\mu_{jk} \triangleq \bigoplus_{\mathbf{S}_{jk}} \left[\psi_j \otimes \bigotimes_{i \in n(j) \setminus k} \mu_{ij} \right] \tag{12}$$

where $n(j)$ is the set of neighbors of j in the junction tree.

Node j computes the message by combining its local factor with the messages it receives from its other neighbors, and then summarizing the result to the separator $\mathbf{S}_{jk}$; this summarizes away all variables that are not in $\mathbf{C}_k$, the clique of the destination node. Definition 2 is recursive, as messages depend upon other messages; the recursion "bottoms out" at the leaves of the junction tree: the message a leaf node sends to its neighbor does not depend on any other messages. Because the junction tree is a tree, there are schedules for computing the messages in which no message is computed more than once [1, 4].

By combining its local factor with its incoming messages, each node obtains a factor which we call its "result":

Definition 3 (result). The **result at j** is the factor

$$\rho_j \triangleq \psi_j \otimes \bigotimes_{i \in n(j)} \mu_{ij} \tag{13}$$

As we have alluded to above, the result at j is exactly the solution to the inference problem whose query variables are given by the clique at j:[1]

Theorem 1. *The result at j satisfies*

$$\rho_j = \bigoplus_{\mathbf{C}_j} \bigotimes_{i=1}^{k} \phi_i \tag{14}$$

This means that after all of the messages have been passed in the junction tree, every node becomes an "expert" about its clique of variables, in that it has the solution to the inference problem where the query variables are given by its clique. Moreover, since each message is used to compute the results at several nodes, the simultaneous solution of these inference problems is extremely efficient.

Example 8 (results). When a junction tree is used to solve a probabilistic inference problem, each node i has immediate access to the posterior distribution of its variables $\mathbf{C}_i$ given all of the observed measurements. In kernel regression, it has the optimal weights of all basis functions necessary to predict the sensor field in its local region. In control, each node obtains information that enables it to locally compute the optimal setting for its control variables. □

It is also possible to use asynchronous message passing to solve the inference problem. In this scheme, each node initializes its incoming messages to the null factor $\mathbf{1}$ and then sending a message to each of its neighbors. Whenever a node j receives a new (version of a) message from a neighbor i, it recomputes and retransmits its message to all neighbors but i. Eventually each node receives correct versions of all of its incoming messages, and the nodes stop sending messages. This scheme is less efficient because messages are recomputed many times, but as we will see, it is more useful in distributed settings.

Another important property of the junction tree data structure is that from the clique and separator sizes we can easily determine the computational complexity of the inference algorithm. Each message μ_{ij} is a factor over the corresponding separator $\mathbf{S}_{ij}$ and each result ρ_i is a factor over the corresponding clique $\mathbf{C}_i$. Thus, the sizes of the messages and the local storage required at each node depend upon the sizes of the cliques and the separators. The computational complexity required to compute messages and results also scales with the clique sizes (in a problem-specific way). Thus, finding a junction tree with small cliques and separators important for efficient inference.

1.3 Overview of the architecture

In the inference problems described above, our knowledge is represented by a collection of factors $\Phi = \{\phi_1, \ldots, \phi_k\}$. These factors typically represent two types of knowledge about the variables: knowledge that derives from observations, and prior knowledge (which may come from experts or from observations made in the past). Both sorts of knowledge are represented by factors. Clearly, those factors that derive from measurements are produced by—and are most naturally stored at—the nodes that make the measurements. The remaining factors, which represent prior information, are partitioned across the nodes; they may be stored on the nodes before deployment, or "downloaded" to the nodes after deployment using dissemination techniques such as directed diffusion [12].[2]

For notational simplicity, we use $\psi_n : \mathbf{D}_n$ to represent the

[1] See [1] for a proof; the key idea is that the messages use distributivity to "push" summaries past combinations, and the running intersection property prevents the summaries from being "pushed in" too far.

[2] In more sophisticated implementations, we may choose to store replicates of each "prior knowledge" factor on several nodes, so that if a small number of nodes fail, the sensor network still has access to all of the factors; see [17] for a technique for redundant distribution of factors in probabilistic inference.

combination of all factors that are stored at node n. Since the factors are partitioned across the nodes we have

$$\bigotimes_{n=1}^{N} \psi_n = \bigotimes_{i=1}^{k} \phi_i \tag{15}$$

which guarantees that this is an equivalent representation of our knowledge. If we were to now organize the nodes of the sensor network into an undirected tree, then we would have a distributed data structure that is almost a junction tree: a tree where each node n has a factor ψ_n—all that would be missing are the cliques associated with each node. This hints at a three-layer architecture for distributed inference:

1. The **spanning tree layer** (§2) allows each node to select a set of neighbors with good communication links such that the nodes are organized in a spanning tree.
2. The **junction tree layer** (§3) allows the nodes compute cliques and separators so that Definition 1 is satisfied. This completes the description of a junction tree that is "embedded" in the network.
3. The **inference layer** (§4) allows the nodes to asynchronously pass the inference messages (12) over the edges of the junction tree, each node eventually converging to the correct result of inference for its clique.

Each of these layers is implemented as a distributed algorithm that runs on every node of the sensor network. Rather than running in sequence, the three layers run concurrently, responding to changes in each others' states. For example, if communication along an edge of the spanning tree suddenly becomes unreliable, the spanning tree must be changed, which causes the junction tree layer to recompute the cliques, which in turn causes the inference layer to recompute new messages.

It can be helpful to view this three-layer architecture in terms of a programming metaphor. We start with a problem we would like to solve (the inference problem) and a physical architecture for computation (the sensor network). The spanning tree layer is responsible for defining a *logical architecture* on which to solve this problem, in that it determines the communication pattern: only neighbors in the tree will communicate with each other. The junction tree layer then uses this logical architecture to determine where summary operators should be placed in the network; it is, in effect, compiling the inference program to the logical architecture. The message passing layer then executes the inference program on the architecture. The next three sections examine these layers in detail.

2. SPANNING TREE FORMATION

As described above, the first step towards distributed inference is to organize the nodes of the sensor network into a spanning tree so that adjacent nodes have high-quality communication links. While this sounds simple, it is actually one of the more difficult problems to be solved. The nodes of a sensor network observe only local information, but spanning trees have three non-local properties: they are connected; they are acyclic; and they are undirected, in that neighbors both agree that they are adjacent. When the connectivity is static and each node knows its neighbors and link qualities, efficient distributed minimum cost spanning tree algorithms are possible [6]. Unfortunately, none of these properties hold in sensor networks: link qualities are asymmetric and change over time (see Figure 4); nodes must discover new neighbors and estimate their associated link qualities; and, nodes must detect when neighbors disappear.

Spanning trees are a basic component of many distributed algorithms because they are a simple communication structure that permit global coordination among a set of distributed processors. For example, spanning trees are often used for multi-hop routing in ad hoc networks and sensor networks [19, 10]. Our application has different requirements than routing, and as a result, we found it necessary to develop a distributed spanning tree algorithm specifically for our architecture. In addition to being correct and robust to failure, we require a spanning tree algorithm with two properties:

- it must be **stable**, in that its topology remains fixed whenever possible; and
- it must be **flexible**, in that it can choose between a wide variety of different topologies.

While these properties are important for spanning tree algorithms used in routing, they are not crucial: the main goal is to move packets through the network. In our setting, the spanning tree defines a "logical architecture" for our application, and in this capacity it determines the computation and communication required to solve the inference problem. Thus, our spanning tree algorithm must be as stable as possible so that the inference algorithm can make progress; and, the spanning tree algorithm must be able to flexibly choose between different spanning trees, so that we can minimize the cost of inference.

There is another important difference between our setting and that of multi-hop routing. In routing, the spanning tree algorithm is privy to useful information that is not easily available in our application. For example, the spanning tree algorithm described in [19] detects cycles by inspecting the originator of each packet; if a node receives a packet that it originated, the node knows it is part of a cycle and takes steps to break it. Our application does not naturally generate information of this sort, and so our spanning tree algorithm must employ other methods to avoid cycles.

2.1 Overview of the algorithm

Our spanning tree algorithm based upon ideas from a few different protocols. The basic structure of our algorithm is inspired by the IEEE 802.1D protocol. The nodes collaborate to identify the node with the lowest identifier (e.g., MAC address); this node is elected the **root** of the network. In addition, each node i selects a **parent** node, which has a high quality connection with i and which has a path to the root via its parent. When all nodes agree on the root and each has found a valid parent, the network has formed a correct *directed* spanning tree; children notify their parents to make the tree undirected.

Each node starts by selecting itself as the root (and also as its parent). To coordinate with the other nodes, each node i periodically broadcasts a **configuration** message, which conveys its current choice of root r_i and parent p_i (as well as other information). The interval between these periodic broadcasts is called an **epoch**, and it is the same for all nodes. Nodes use the configuration messages they receive to update their own choice of root and parent. For example, if node i with root r_i receives a configuration message from node j with root $r_j < r_i$, then node i selects node j as its parent and updates its root to r_j; this is the basic mechanism by which the nodes elect the node with the lowest identifier as the root. Nodes also use configuration information to learn about their children: if node i receives a configuration message from node j that designates i as its parent, i learns that j is its child.

The local state at node i consists of a cache of **node information records**. Each such record corresponds to another node j in the network that node i has learned about, and includes the configuration most recently received from node j, link quality information about node j, and other information. New records are inserted into the cache when nodes are first discovered, and they are flushed from the cache when the corresponding node is believed to have failed.[3]

To prevent stale configuration information, each configuration message has another field called the **root pulse**. This is an integer that each node copies to its configuration message from that of its parent. The only node in the network that changes the root pulse is the root itself; in each successive epoch the root increments its pulse. Thus, if the root is functioning and a node has a valid path to the root, that node will see the root pulse (as conveyed by its parent's configuration) increasing. If the root has failed or node i's path is invalid, the root pulse will stop, and node i knows to choose another parent.

More specifically, the root pulse is used to detect stale information as follows. In node i's cache, the node information record for node j includes the configuration most recently received from node j, and also its associated **configuration age**, which is the number of epochs in which the configuration's root pulse has not changed. A configuration is viewed as stale if its age exceeds a prespecified constant, `MAX-CFG-AGE`. When a configuration is first received from node j (or when the root r_j of j's configuration changes), the age is set to $1 +$ `MAX-CFG-AGE`; this ensures that a configuration is initially viewed as stale until a "live pulse" is observed (to prevent stale information from being propagated around the network indefinitely). Node i increments this configuration age once each epoch, and resets it to zero if it receives from j a new configuration message with the same root but a higher root pulse.

2.2 Link quality estimation

In addition to coordinating the nodes' decisions regarding the tree topology, the configuration messages form the basis of our link quality estimation technique. The configuration messages are issued with a known period, and so once node i learns of node j's existence, it can interpret an epoch without an error-free configuration message from j as a failed transmission. We use an exponentially-weighted moving average of the fraction of successfully received configuration messages as an estimate of the **in-bound** link quality on a link [19]; this is a dynamic estimate $\tilde{p}_{j\to i}$ of the probability $p_{j\to i}$ that a packet transmitted by node j will be received by node i.[4]

By listening to other nodes, a node i can learn about its in-bound link qualities, but not about its **out-bound** link qualities; i.e., node i cannot learn the success rate of its transmissions. Because link qualities can be (and often are) asymmetric (see Figure 4), we need a technique for nodes to learn about their out-bound link qualities. We accomplish this by augmenting each configuration message with **in-bound** link quality information; i.e., when node i broadcasts its configuration message, it includes information on the nodes that it can hear, and their associated in-bound link quality estimates. To keep the size of the configuration message small, each configuration message can report quantized link qualities for a small number of nodes; a node can cycle through the link qualities it reports in successive configuration messages. The net effect is that if node i can hear node j, it will eventually obtain an estimate $\tilde{p}_{i\to j}$ of

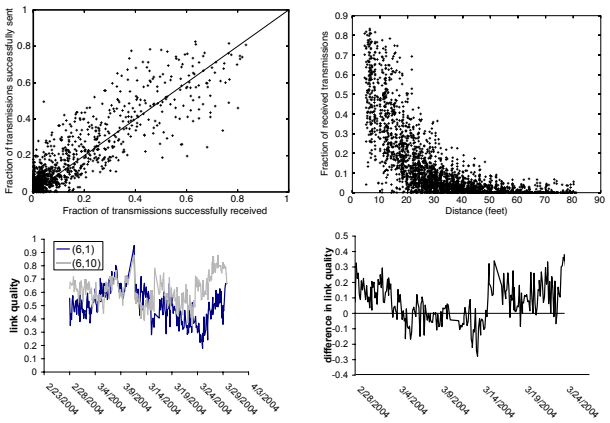


Figure 4: ***Link qualities in the sensor network deployment. The upper left plot shows the asymmetry of link qualities: each point corresponds to a pair of nodes, and gives the fraction of successfully sent and received transmissions. The upper right plot shows how in-bound link quality varies with distance. Note that link quality has large variance even at short distances. The lower left plot shows how the in-bound link quality from two transmitters change over a month's time, and the lower right plot shows the difference in received link quality between two parents; because it is both positive and negative, the best choice of parent can change over time.***

[3]In practice a node may hear infrequently from many distant nodes; this can cause its node information cache to grow unmanageably large. To maintain a fixed size on the cache, it is important to choose a cache management strategy that adds new records conservatively[19].

[4]These estimates are reliable only after the transmitter has been observed for a reasonable period of time. For this reason, each node information record stores the (exponentially-weighted) counts of received and lost configuration messages and also the **record age**, which is the amount of time the record has been in the cache. The link quality is assumed to be zero until the record age exceeds a predefined threshold.

$p_{i\to j}$, the probability that its transmissions are received by node j; if no estimate is received it is assumed to be zero.

Once a node i has trustworthy estimates of its in-bound link quality $\tilde{p}_{j\to i}$ from node j and its out-bound link quality $\tilde{p}_{i\to j}$ to node j, it can compute the **reliable transmission cost** between itself and node j as

$$\tilde{c}_{ij} \triangleq \frac{1}{\tilde{p}_{i\to j} \times \tilde{p}_{j\to i}} \tag{16}$$

Under the assumption that the success of the transmissions are independent events, $\tilde{c}_{ij}$ is an estimate of the expected total number of packet transmissions node i and node j would have to perform in order to successfully transmit a packet and then successfully acknowledge its receipt. Each node uses these costs to optimize its choice of parent, as described in §2.3.

We also rely upon configuration messages to detect node loss. If node i experiences an extended period in which no configuration messages arrive from node j, this can indicate that node j has died. Given the wide variety of link qualities (even within a single sensor network deployment—see Figure 4), it is undesirable to use a predefined silence period to diagnose node loss: five consecutively lost messages is a strong indication of a dead node if the link is of excellent quality, but it says little about a node with a low-quality link. Instead, in our algorithm, node i assumes node j has died if it has been silent for n consecutive epochs and $(1-\tilde{p}_{j\to i})^n$, the probability n successive configuration messages would be lost given the link quality is $\tilde{p}_{j\to i}$, is less than a small threshold (e.g., 1×10^{-7}).

2.3 Parent selection

To complete the description of the algorithm, we must describe how nodes select their parents. Each node i periodically scans its node information cache to find the node j that has the lowest reliable transmission cost $\tilde{c}_{ij}$ such that the following constraints are satisfied.

1. Node j must have selected the lowest root of all potential parents; this ensures the nodes all agree on the root.

2. The age of node j's configuration must not exceed the maximum permitted configuration age; this ensures stale information is not used to make parent decisions.

3. If i currently has a parent k that offers a path to the same root as node j, then node j's configuration must have a larger pulse value than node k's configuration does; this prevents node i from selecting one of its descendants as a parent, which would create a cycle.

Given these rules for parent selection, the algorithm is guaranteed to converge to a correct spanning tree if the network is not partitioned, and if link qualities remain stable for a long enough period of time. (It is not guaranteed to converge to the minimum cost spanning tree, however, because the nodes make independent parent decisions.)

2.4 Descendant testing

We have found experimentally that the algorithm described above yields stable spanning trees, but that it is limited in the spanning trees it will select—i.e., it is not flexible. The reason is that condition (3) above is a very conservative method for avoiding cycles; in practice it means that a node will almost never select a parent that is further than itself from the root, even if it would greatly reduce the cost.

We address this problem by adding a second root pulse called the **descendant test pulse**. Like the root pulse, this is an integer that is periodically incremented by the root and is propagated from parent to child throughout the spanning tree. Unlike the root pulse, any node may choose to temporarily halt the progression of the descendant test pulse to perform a descendant test. When a node i fixes the descendant test pulse that it broadcasts, then the descendant test pulse observed (and broadcast) by its descendants will also halt. If node i then observes another node j with a descendant test pulse that is higher than the one it is broadcasting, then i knows that j is not a descendant, and could possibly serve as a parent. In this way, node i learns of a larger set of possible parents, and can make better decisions about how to minimize its cost.

One shortcoming of this scheme is that a descendant test is less useful if nearby nodes are performing descendant tests—especially when an ancestor is performing a descendant test (in which case the descendant test pulse has already been halted). This makes the timing the tests important. Nodes can make intelligent decisions about when to perform descendant tests by snooping on the descendant test pulses broadcast by nearby nodes and by performing them only when they could possibly lead to the discovery of a better parent.

3. JUNCTION TREE FORMATION

Recall that each node i in the sensor network starts with a factor $\psi_i : \mathbf{D}_i$. Once a spanning tree has been constructed, the nodes have formed a distributed data structure similar to a junction tree: a tree where each node has a factor (see Figure 5). To make this into a junction tree, we must also specify the clique $\mathbf{C}_i$ for each node i of the network. According to Definition 1, these cliques must satisfy two properties: each node's clique must include the domain of its factor (i.e., $\mathbf{C}_i \supseteq \mathbf{D}_i$ for all nodes i); and, we must have the running intersection property: if two cliques $\mathbf{C}_i$ and $\mathbf{C}_j$ have the same variable X, then all nodes on the unique path between them must also carry X.

In this section, we present a robust, distributed algorithm which passes messages between neighbors in the spanning tree in order to compute the unique set of minimal cliques that satisfy these two properties. Because the spanning tree topology determines the cliques of the junction tree, we also present a robust, distributed algorithm for optimizing the spanning tree to minimize the clique sizes; this minimizes the communication and computation required by inference.

3.1 Ensuring the running intersection

We will begin by presenting the algorithm under the assumption that there is a stable, valid spanning tree and that

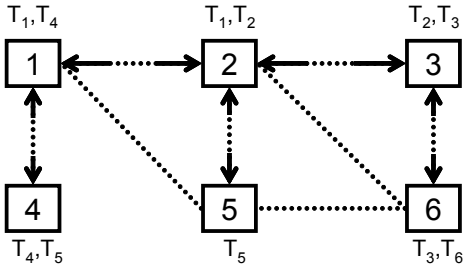


Figure 5: ***Example of the initial spanning tree in a six node network; the dotted lines indicate high-reliability links, the links used in the spanning tree are shown with arrows. Next to each node i is the domain $\mathbf{D}_i$ of its local factor. Note that the running intersection is not satisfied; $\mathbf{D}_4$ and $\mathbf{D}_5$ include T_5, but $\mathbf{D}_1$ and $\mathbf{D}_2$ do not.***

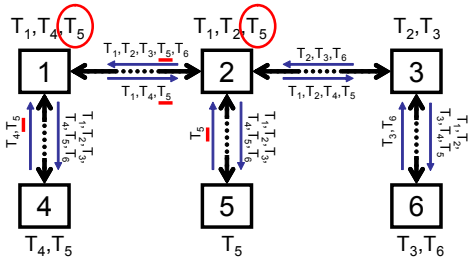


Figure 6: ***The reachable variables messages for Figure 5. Each node i is now labelled with its clique $\mathbf{C}_i$. The reachable variables message $\mathbf{R}_{32} = \{T_2, T_3, T_6\}$ is obtained by the union of $\mathbf{R}_{63} = \{T_3, T_6\}$ with the local variables for node 3, $\mathbf{D}_3 = \{T_2, T_3\}$. The circled variables were added to satisfy the running intersection property, e.g., T_5 is included in $\mathbf{C}_2$ because it appears in $\mathbf{R}_{12}$ and $\mathbf{R}_{52}$, as shown by the underlined variables in the messages.***

communication between neighbors is reliable. We then describe its generalization to the case where the spanning tree is changing and communication is not reliable. Finally, we describe techniques for minimizing communication.

3.1.1 Message passing algorithm

Each node learns its cliques and separators using a message passing algorithm in which it sends a message to and receives a message from each neighbor. Let i be a node and j be a neighbor of i; the **variables reachable to j from i** are:

$$\mathbf{R}_{ij} \triangleq \mathbf{D}_i \cup \bigcup_{\substack{k \in n(i) \\ k \neq j}} \mathbf{R}_{ki}, \tag{17}$$

where $n(i)$ are i's neighbors in the spanning tree. These messages are defined recursively, just like the messages of junction tree inference (§1.2). Node i computes $\mathbf{R}_{ij}$ by collecting the variables that can be reached through each neighbor but j and adding the domain $\mathbf{D}_i$ of its local factor; then it sends $\mathbf{R}_{ij}$ as a message to j. Figure 6 shows the reachable variables messages for the example of Figure 5.

If a node receives two reachable variable messages that both include some variable X, then it knows that it must also carry X to satisfy the running intersection property. Formally, node i computes its clique using

$$\mathbf{C}_i \triangleq \mathbf{D}_i \cup \bigcup_{\substack{j,k \in n(i) \\ j \neq k}} \mathbf{R}_{ji} \cap \mathbf{R}_{ki}. \tag{18}$$

For example, in Figure 6, node 2 receives two reachable variables messages that contain T_5, and so its clique must include T_5, as shown. These messages also allow each node i to locally compute its separator with a neighbor j via $\mathbf{S}_{ij} = \mathbf{C}_i \cap \mathbf{R}_{ji}$.

Just like the message passing algorithm used for inference (§1.2), this message passing algorithm can easily be made asynchronous. Each node initializes its incoming reachable variables messages to be empty. Each time node i receives a new reachable variables message from a neighbor j, it recomputes its reachable variables messages to all neighbors but j, and transmits them if they have changed from their previous values; in addition, it recomputes its clique and separators. This algorithm is guaranteed to converge to the unique, minimal set of cliques that preserve the running intersection property for the underlying spanning tree.

3.1.2 Robust, distributed implementation

In our presentation above, we made two simplifying assumptions. First, we assumed reliable communication between neighbors in the spanning tree. While this is not true at the physical network layer, it can be implemented at the transport layer using message acknowledgements; by hypothesis, the spanning tree consists of high-quality wireless links. Second, we assumed that the reachable variables messages were transmitted after the spanning tree algorithm had run to completion. The algorithm cannot be implemented in this way, however, because in a sensor network, there is no way to determine when a distributed algorithm has completed: a node can never rule out the possibility that a previously unknown node will later join the network, for example.

Our algorithms therefore run concurrently on each node, responding to changes in each others' states. When the spanning tree algorithm on a node adds or removes a neighbor, the junction tree algorithm is informed and reacts by updating its reachable variables messages. If node i obtains a new neighbor j, then $\mathbf{R}_{ij}$ is computed and sent to j; if j is removed from i's neighbor set then for all other neighbors k, $\mathbf{R}_{ik}$ is recomputed and retransmitted (if it has changed from its previous value). This tight interaction between the algorithms permits the junction tree to reorganize quickly when changing link qualities, interference, or node failures cause the spanning tree to change.

3.1.3 Minimizing communication

This junction tree algorithm is the only part of our architecture where nodes must reason about "global" aspects of the model. In general, the reachable variables messages require space linear in the total number of variables in the model; for example, if j is a leaf in the spanning tree, then $\mathbf{R}_{ij}$ must include $\mathbf{V} - \mathbf{D}_j$, which is typically the majority the model's variables. For large models, then, it is important to choose a compact encoding of these messages to minimize communication cost. For example, if the variables are represented by integer identifiers, the we can compactly encode large reachable variables messages using a set of intervals.

As we have described the algorithm above, $\mathbf{R}_{jk}$ is retransmitted whenever it changes, which can happen when j receives a new reachable variables message from another neighbor. A great deal of communication can be saved if instead of sending the new value of $\mathbf{R}_{jk}$, node j sends a "patch" which allows node k to compute the new value from the old one. Say that $\mathbf{R}'_{jk}$ is the new value, and that $\mathbf{R}_{jk}$ is the old value that node k currently has. Instead of sending $\mathbf{R}'_{jk}$ to node k, node j can send an **add set** and a **drop set**:

$$\begin{aligned} \mathbf{A}_{jk} &\triangleq \mathbf{R}'_{jk} - \mathbf{R}_{jk} \\ \mathbf{D}_{jk} &\triangleq \mathbf{R}_{jk} - \mathbf{R}'_{jk} \end{aligned}$$

Then node k can compute

$$\mathbf{R}'_{jk} = (\mathbf{R}_{jk} - \mathbf{D}_{jk}) \cup \mathbf{A}_{jk}$$

Sending $\mathbf{A}_{jk}$ and $\mathbf{D}_{jk}$ is much more efficient than sending $\mathbf{R}'_{jk}$, especially because retransmissions are often required to ensure k successfully received the message. However, the correctness of this optimization hinges on nodes j and k agreeing upon the previous reachable variables message, $\mathbf{R}_{jk}$. If previous patches from node j have not been received by node k, then node k's view of $\mathbf{R}_{jk}$ will be out of sync with node j's. To prevent this from happening, node j may not send a new patch until node k has acknowledged the previous patch.

This rule ensures that once nodes j and k agree on the current value of $\mathbf{R}_{jk}$ that changes are propagated correctly; however, extra work is needed to ensure the initial condition: when nodes j and k become aware that they are neighbors, they must come to agreement on the value of $\mathbf{R}_{jk}$. In the simplest case, this is easy: both nodes start with $\mathbf{R}_{jk}$ being empty. But the asymmetry of neighbor relations can cause tricky situations. For example, assume j and k know that they are neighbors in the spanning tree. Due to changing link qualities, it can happen that k chooses to drop its link to j in favor of another node, and then reverses this decision, choosing j again as its neighbor. If during this period k's configuration messages are not received by j, it will appear to j that k never stopped being its neighbor. In this case, j's view of $\mathbf{R}_{jk}$ will not have changed, but k has reinitialized it to be empty.

To ensure correctness even under these circumstances, it is necessary to have neighbors signal to each other when their shared state must be reinitialized. This is accomplished with a simple handshake protocol. When node k adds j as a neighbor, it sends $\mathbf{R}_{kj}$ to node j with a `RESET` flag. This signals to node j that $\mathbf{R}_{kj}$ is the current reachable variables from k and that k's view of $\mathbf{R}_{jk}$ is invalid. Node j responds with $\mathbf{R}_{jk}$, and the flag `REACHABLE`, which provides k with a current view of its reachable variables. Once this exchange is successfully completed (using acknowledgements for reliability), all following messages are patches, consisting of an add set and a drop set.

When the model is very large and communication is very expensive, even the techniques described above may not be sufficient to satisfy application cost constraints. In this case, problem-specific structure may also be used to reduce the size of the reachable variables messages. For example, if it can be guaranteed that two nodes which have a variable in common are no more than k hops away in the spanning tree, then each variable in a reachable variables message can be given a "time to live", which is decremented each time it is propagated across an edge. When the time to live reaches zero, the variable does not need to be propagated further.

3.2 Optimizing the junction tree

The algorithm above transforms the spanning tree into a junction tree by computing the unique set of minimal cliques that satisfy Definition 1. Note that different spanning trees can give rise to junction trees with different clique and separator sizes; for example, if in Figure 5 node 5 had chosen to connect to node 1 instead of node 2, the node 2's clique would not need to include the variable T_5. The size of a node's clique determines the amount of computation it must perform, and the separator sizes determine the amount of communication required by neighbors in the tree. These facts motivate a *tree optimization algorithm* which chooses a spanning tree that gives rise to a junction tree with small cliques and separators.

The input to this algorithm is a cost function that can be computed by the nodes of the network. If our objective is to minimize the communication required to solve the inference problem, we may choose the cost of a junction tree to be

$$\sum_{i=1}^{N} \sum_{j \in n(i)} \tilde{c}_{i \to j} \times \mathsf{num\_packets}(\mathbf{S}_{ij}) \tag{19}$$

where $\tilde{c}_{i \to j}$ is the reliable transmission cost of transmitting a packet from node i to node j (as estimated by the spanning tree algorithm—see (16)) and $\mathsf{num\_packets}(\mathbf{S}_{ij})$ is the number of packets required to serialize the inference message that would be sent from node i to node j. This cost function takes into account the link quality between neighbors and the size of the message that would be sent. We could also use each node's processor power and clique size to take into account the computational cost of computing the messages.

Finding the spanning tree that minimizes this cost function is NP-hard (by a simple reduction from centralized junction tree optimization [4]), but we can define an efficient distributed algorithm for greedy local search through the space of spanning trees. The local move we use to move through tree space is (legal) edge swaps; for example, in Figure 5 node 5 can swap its edge to 1 for an edge to 2 or an edge to 6. Thus, we rely upon the spanning tree algorithm in §2 to build up a good spanning tree using link quality information only; then the tree optimization algorithm repeatedly improves the current tree by performing edge swaps when it would reduce the communication and/or computation required to solve the inference problem.

Nodes learn about a legal edge swap, and the change to the global cost function (19) that would occur if it was implemented, using a distributed dynamic programming algorithm. By starting an **evaluation broadcast** along one of its spanning tree edges, a node can learn about alternatives for the edge and their relative costs. For example, in Figure 5, suppose node 5 initiates an evaluation broadcast by sending to its neighbor in the spanning tree, node 2, a message $\texttt{EVAL}(5, 2)$, meaning "find legal alternatives for the edge $5 \leftrightarrow 2$." Node 2 then propagates $\texttt{EVAL}(5, 2)$ to its

neighbors, nodes 1 and 3. When node 1 receives the message, it sees that the originator, 5, is a potential neighbor, and propagates the message to 5 *outside the spanning tree*. When node 5 receives the $\texttt{EVAL}(5, 2)$ message from node 1, it learns of a legal swap: it can trade its edge to 2 for an edge to 1. Similarly, when node 3 receives the evaluation request from node 2, node 3 propagates it to node 6, which then propagates it to node 5 outside of the spanning tree; in this way node 5 learns that $5 \leftrightarrow 6$ is another alternative for the edge $5 \leftrightarrow 2$.

In general, swapping spanning tree edges has non-local effects on the cliques and separators of the induced junction tree, so a node cannot assess the relative cost of an edge swap locally. However, the relative cost can be assessed efficiently by an extension of the evaluation broadcast scheme described above. The key idea is that if the edge $5 \leftrightarrow 2$ were swapped for the edge $5 \leftrightarrow 1$, *only the reachable variables messages on the cycle* $5 \leftrightarrow 2 \leftrightarrow 1 \leftrightarrow 5$ *would change.* This is a direct consequence of the definition of the reachable variables messages (17). Similarly, if the edge $5 \leftrightarrow 2$ were swapped for the edge $5 \leftrightarrow 6$, only the reachable variables messages on the cycle $5 \leftrightarrow 2 \leftrightarrow 3 \leftrightarrow 6 \leftrightarrow 5$ would change. Therefore, to evaluate the relative cost of an edge swap, only the nodes on the **swap cycle**, i.e., the cycle closed by the new edge, must be involved in the computation. If the swap cycle is small, then the relative cost of a swap can be computed by a small number of nodes.

Moreover, by augmenting the evaluation messages with some compact reachable variables information, the nodes along the swap cycle can assess the change in cost *even without knowing for what edge the* $5 \leftrightarrow 2$ *edge will be swapped.* This allows the nodes on the swap cycle to add in their local contributions to the relative cost as the evaluation broadcast messages are propagated. Once node 5 has received its evaluation message from 1 it learns of the legal swap and its effect on the global cost. If the swap reduces the cost, this information is provided to the spanning tree algorithm, which effects the change.

To accomplish this, each `EVAL` message has five fields:

`originator`	the node that is considering swapping an edge to one of its neighbors
`neighbor`	the neighbor of `originator` for which an alternative is sought
`running-cost`	the change in cost accumulated around the swap cycle up to the `EVAL` recipient
`add-set`	the set of additional variables that would be reachable to the `EVAL` recipient from its successor on the swap cycle if the swap were performed
`drop-set`	the set of variables that would no longer be reachable to the `EVAL` recipient from its predecessor on the swap cycle if the swap were performed

Say a node i wishes to search for an alternative to a current neighbor j. Then node i sends to node j the message

$$\texttt{EVAL}\left(i, j, -(\tilde{c}_{i\to j} \times \mathsf{num\_packets}(\mathbf{S}_{ij})), \mathbf{R}_{ij}, \mathbf{R}_{ij}\right)$$

The running cost is started at $-(\tilde{c}_{i\to j} \times \mathsf{num\_packets}(\mathbf{S}_{ij}))$, because if node j were no longer a neighbor of node i, then node i would no longer incur the communication cost of sending an inference message to node j. The add and drop sets are initialized to $\mathbf{R}_{ij}$, because if the swap were to take place, the nodes on the swap cycle would no longer receive node i's reachable variables via node j, but instead by node i's new neighbor.[5]

Now suppose a node k has received from node h an evaluation message $\texttt{EVAL}(i, j, c, \mathbf{A}, \mathbf{D})$ that indicates i is searching for a replacement for its neighbor j. Node k will propagate the broadcast to all neighbors but the sender h, so that the evaluation broadcast will traverse all possible swap cycles. When computing the evaluation message to a neighbor ℓ, k is assuming that ℓ is its next hop on a swap cycle. Because the cost of a swap depends upon the swap cycle traversed, the evaluation message that node k sends to ℓ is different than the one it sends to the other neighbors. In particular, node k computes its message to ℓ as follows:

1. Node k computes the clique and separators it would have if the reachable variables messages from the previous and next nodes on the swap cycle were

$$\mathbf{R}'_{hk} \triangleq \mathbf{R}_{hk} - \mathbf{D} \tag{20}$$

$$\mathbf{R}'_{\ell k} \triangleq \mathbf{R}_{\ell k} \cup \mathbf{A} \tag{21}$$

2. Node k computes the change in its cost as

$$\Delta c_k \triangleq \sum_{m\in n(k)} \tilde{c}_{k\to m} \times \mathsf{num\_packets}(\mathbf{S}'_{km}) - \sum_{m\in n(k)} \tilde{c}_{k\to m} \times \mathsf{num\_packets}(\mathbf{S}_{km})$$

   where $\mathbf{S}'_{km}$ is the separator computed with (20) and (21).

3. Using (20) and (21), node k uses (17) to compute $\mathbf{R}'_{k\ell}$, the reachable variables message k would send to h if the swap occurred.

4. Node k sends the following message to node ℓ:

$$\texttt{EVAL}\left(i, j, c + \Delta c_k, \mathbf{A}, \mathbf{R}_{k\ell} - \mathbf{R}'_{k\ell}\right)$$

If node k has a wireless link to the request originator i, then k is the last node on a swap cycle. In this case, node k follows the same procedure as above to send an evaluation message to node i. When node i receives this message, the cost computation is almost complete; all that remains is for node i to perform step (1) above and then add

$$\tilde{c}_{i\to k} \times \mathsf{num\_packets}(\mathbf{S}'_{ik})$$

[5]In fact, node i could equivalently send to j the message

$$\texttt{EVAL}\left(i, j, -(\tilde{c}_{i\to j} \times \mathsf{num\_packets}(\mathbf{S}_{ij})), \mathbf{S}_{ij}, \mathbf{S}_{ij}\right)$$

This message is more compact because $\mathbf{S}_{ij}$ is typically far smaller than $\mathbf{R}_{ij}$. And, it is equally correct because $\mathbf{S}_{ij}$ is the set of variables that are on both sides of the edge $i \leftrightarrow j$; if a variable in $\mathbf{R}_{ij}$ is not in $\mathbf{S}_{ij}$ then it has no effect on the result of the cost computation.

to the running cost it received from k; this is the extra cost that node i would incur from the swap because it would have to send an inference message to node k.

After a node starts an evaluation broadcast, it should wait to receive all (or most) of the responses. If the node were to implement the first swap it heard about, then the cliques and separators along the swap cycle would change, rendering any subsequent responses stale. Therefore, after issuing an evaluation broadcast, the node waits for a fixed period to collect responses. Each time a response is received, the collection period is incremented. When the collection period ends, the swaps and their costs are reported to the spanning tree algorithm, which implements the best swap possible.

As explained above, the effects an edge swap has on the junction tree exhibit only partial locality. If two nodes were to undertake edge swaps at the same time and their swap cycles overlap, then the resulting change in cost may be different than the individual cost estimates would indicate. This makes timing the swap evaluations important. To avoid simultaneous evaluations and swaps, we use a simple heuristic based on snooping on the broadcast channel. A node will only start an evaluation broadcast if it has not heard reachable variables messages or another evaluation broadcast for a period of time. If one is heard, the node will back off and wait longer before it starts its evaluation. When there are no conflicting edge swaps, this distributed algorithm will converge to a junction tree that is a local minimum of the cost function.

As we have described it, the communication pattern of the tree optimization algorithm makes evaluating the cost of swaps expensive: when node i starts an evaluation broadcast via a neighbor j, the evaluation messages are propagated to all nodes on the j side of the $i \leftrightarrow j$ edge. In large networks, this can be very expensive. Fortunately, it is possible to prove that once the running cost becomes positive, it can never decrease as the evaluation messages propagate around a swap cycle. Because we are not interested in swaps that increase the tree cost, we can halt propagation of the evaluation messages whenever the running cost becomes positive. Another sensible method to reduce the communication cost is to use a hop count limit to limit the local search.

4. ASYNCHRONOUS MESSAGE PASSING

To solve the inference problem using asynchronous message passing, we must make some simple changes to the algorithm in §1.2. First, messages must be transmitted reliably, which is accomplished using acknowledgments. The remaining changes are required because we cannot assume the junction tree is fixed and correct: just as with the spanning tree algorithm, nodes do not (and could not) ever receive an indication that the junction tree assembly is complete. Therefore, the inference algorithm must react to changes in the junction tree as follows:

- When node i gets a new neighbor j, the inference algorithm initializes the incoming message $\mu_{ji} = \mathbf{1}$, and computes and transmits the outgoing message μ_{ij}.
- When node i loses a neighbor j, the inference algorithm discards the incoming message μ_{ji}, recomputes the messages to the remaining neighbors, and then retransmits them.
- When node i learns that its separator with a neighbor j has changed to $\mathbf{S}'_{ij}$, the inference algorithm recomputes the outgoing message to j using (12) with the new separator and transmits it to j.[6]

In addition, the inference algorithm reacts to inference messages as described in §1.2: whenever a node j receives a new (version of a) message from a neighbor i, it recomputes and retransmits its message to all neighbors but i. If the spanning tree eventually stabilizes, then the junction tree will also stabilize; in this case these rules guarantee that the inference messages will eventually converge to the correct values, and that after this point nodes will stop passing inference messages.

In some problems, it is possible to make intelligent decisions about when retransmitting a message is not worth the communication cost. For example, if node j has transmitted μ_{jk} to node k and it then receives a new message μ_{ij} from another neighbor, it often happens that the new message it would send to k, μ'_{jk}, is not that different from the previous value. In probabilistic inference, it is possible to obtain error bounds by computing the Kullback–Liebler divergence between the two messages [13]; in linear regression, one can analyze the differences between the linear constraints to estimate how they will propagate [7]. This can be an effective way to trade communication cost for approximation error.

We conclude the presentation of our inference architecture by returning to a point that has become a motif of this paper: in a sensor network, there is no way for a node to know when a distributed algorithm has terminated, because the node cannot exclude the possibility that a new node will later be introduced into the network. This fact motivated us to design our algorithms so that they can react to changes in each others' states. But now we have reached the top of our algorithm stack, and we must consider how an application will use the results of inference when it cannot be sure that the inference algorithm has run to completion.

Certainly the solution to this problem will be application specific, but it seems clear that in general it is useful for the inference algorithm to guarantee that at any point during its execution, each node's **partial result**—i.e., the quantity (13) which is computed when not all of the final versions of the messages have arrived—is useful. Some inference algorithms naturally have this property. For example, in the regression problem, each message represents the impact a subset of the measurements have on the optimal parameters; thus, if a node solves for its optimal weight vector without final versions of all of the messages, it is simply failing to account for measurements made at nodes that it cannot communicate with [8]. Other inference algorithms do not naturally have this property; for example, the partial results of the traditional algorithm for probabilistic inference

[6] If $\mathbf{S}'_{ij} \subset \mathbf{S}_{ij}$, then the following optimization avoids passing two messages: then node i updates the incoming message from j by $\mu'_{ji} = \bigoplus_{\mathbf{S}'_{ij}} \mu_{ji}$ and does not retransmit a new message to j; when node j learns of the change to $\mathbf{S}_{ij}$, it does the same thing.

can be arbitrarily far from the correct results. To make these algorithms useful for inference in sensor networks, extra work is necessary; for example, see [17] for a new message passing algorithm for probabilistic inference algorithm that resolves the problem.

5. EXPERIMENTAL RESULTS

To validate our architecture and algorithms, we deployed 54 Intel–Berkeley motes in our lab (Figure 1) and collected temperature measurements every 30 seconds for a period of 4 weeks. We also collected link quality statistics and computed the fraction of transmissions each mote heard from every other mote. Using this link quality information, we designed a sensor network simulator that modeled the actual deployment. (Designing, testing, and experimenting with our algorithms would have been far more difficult in the actual deployment.) This simulator uses an event-based model to simulate lossy communication between the nodes at the message level: messages are either received or not, depending upon samples from the link quality model. The simulator's programming model is also event-based—algorithms are coded in terms of responses to message events—and we expect that our algorithmic implementations can be transferred to the real sensor network without significant changes.

5.1 Junction tree experiments

To demonstrate the communication pattern of the junction tree layer, we performed a simple experiment to compute the total communication used by all reachable variables messages for a particular inference problem (described in §5.3). Figure 7(a) shows the total amount of communication required by the transmitted reachable variables messages in each epoch (0.1 time units). At the beginning of the simulation there is a prolonged period before the spanning tree is built; the nodes wait until they have accurate link quality estimates before they begin building this tree. After a stable spanning tree has been found, the nodes eventually enforce the running intersection property, resulting in a valid junction tree.

After a spanning tree is built, it can be lost; this typically occurs in the delay between a node changing its parent and the old and new parent learning of the change. Note that the communication required to rebuild the junction tree after the spanning tree has changed is significantly less than the cost required to build the first junction tree; this is because the majority of the junction tree does not change. Finally, note that after a correct junction tree is formed, communication eventually stops; this pattern is also present in the message passing layer, where after all messages have been passed, the network quiesces.

We ran another experiment to test the distributed spanning tree optimization algorithm. This experiment used the inference problem described in §5.2. We chose our communication cost function so that the cost of a (directed) edge is proportional to the expected number of transmitted bytes necessary to successfully communicate an inference message, taking into account retransmissions. The piecewise constant curve in Figure 7(b) represents the current cost of the spanning tree when one exists. Offline, we used a combination of simulated annealing, greedy local search, and random restarts to find a local minimum of this cost function[7]; its cost is plotted as the horizontal line in Figure 7(b). Note that the initial spanning tree, which is selected using only link quality information, is significantly more expensive than the hypothesized optimum, but that the distributed optimization algorithm eventually finds trees whose cost is less than twice this hypothesized optimum.

5.2 Calibration experiments

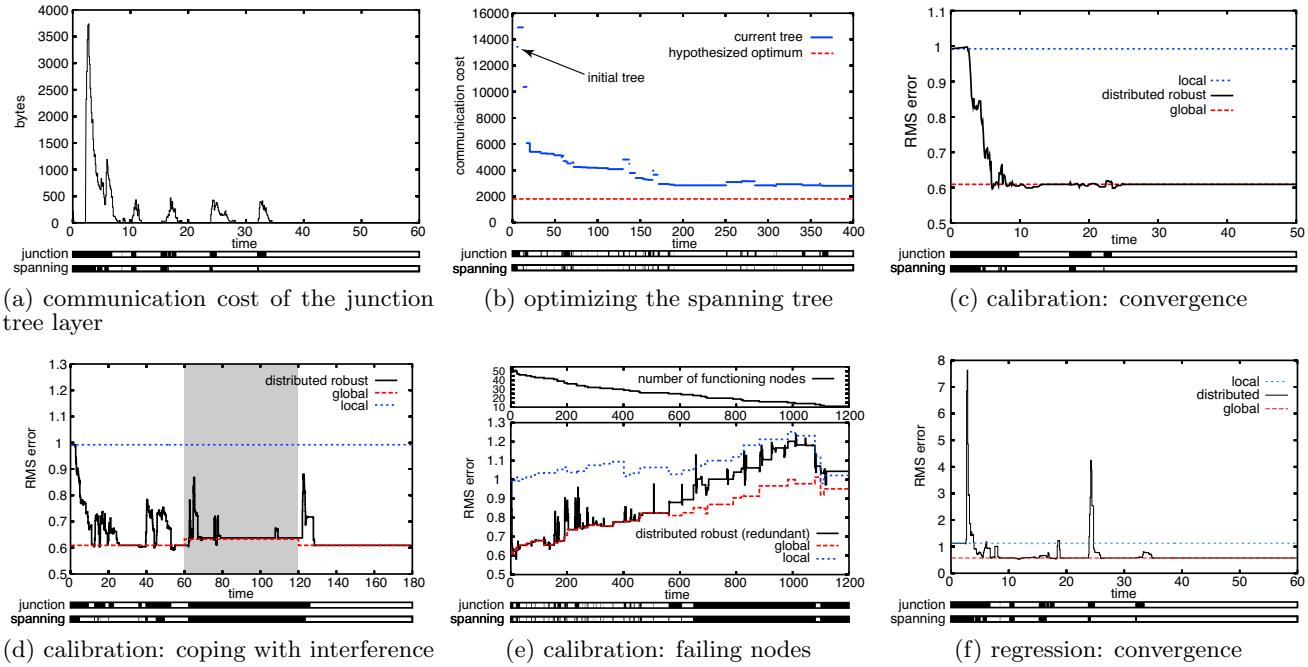
The next set of experiments were performed on the distributed sensor calibration problem described in §1.1.1. Using the temperature data from the lab, we learned a multivariate Gaussian distribution over the temperature variables that has the Markov graph shown in Figure 1. (Mote 5 failed shortly after deployment, which explains its absence in the figure, and also justifies our efforts to develop algorithms robust to such failures.) The model was augmented with bias variables for each temperature measurement, which were distributed i.i.d. from $\mathcal{N}(0, 1°\mathrm{C})$.

To set up our distributed sensor calibration task we sampled a true, unobserved bias for each node, and created a set of biased measurements by adding these biases to a held-out test set of measurements. The inference task is for the nodes to compute their posterior mean bias estimates, and the error metric we use is the root mean squared error (RMS) from their estimates to the (unobserved) biases we sampled.

Our first experiment demonstrates the inference architecture in the simplest setting, where link qualities are stable. Figure 7(c) visualizes a trace of the inference architecture when the robust message passing algorithm of [17] is used to solve the probabilistic inference problem. (In these experiments, the optimization algorithm was turned off for simplicity.) The main panel of Figure 7(c) plots the RMS error of three inference algorithms. The line marked *global* refers to centralized inference using all of the measurements. In this case, the posterior mean bias estimates of global inference have 0.61 RMS error; because the bias is additive, this number also represents the average error in the posterior mean temperature measurements. Thus, by solving the global inference problem the nodes can automatically eliminate approximately 39% of the bias. The line marked *local* refers to local inference, where each node's posterior is computed using only its measurement. Local inference performs about as well as predicting zero bias, achieving a 0.99 RMS error; this is expected, correlated measurements from different nodes are required for automatic calibration.

The third curve, *distributed robust*, refers to our architecture combined with the robust probabilistic message passing algorithm. This plot graphically demonstrates the key properties of the algorithm: before any messages have been passed, the partial results coincide with the estimates given by local inference; at convergence, the estimates coincide with those of centralized global inference; and, before all messages have been passed, the estimates are informative approximations. Looking closely, we can see that before the junction tree is valid, and even before a complete spanning tree is constructed, the estimates of the robust message pass-

[7]The local moves used in simulated annealing and local search were edge swaps that are less restrictive than those used by the spanning tree layer.



(a) communication cost of the junction tree layer

(b) optimizing the spanning tree

(c) calibration: convergence

(d) calibration: coping with interference

(e) calibration: failing nodes

(f) regression: convergence

Figure 7: ***Experimental results. The x-axis of all of these plots is time. The two bars under the graphs are: the bottom bar is white when a valid spanning tree has been constructed, and black when it has not; the top bar shows when the running intersection property has been enforced.***

ing algorithm quickly approach those of centralized global inference.

To test the algorithms' robustness to long-term communication failure, we ran the same experiment, but this time we introduced a period where interference causes the network to be segmented into two parts. At time 60, all messages between the left half of the network (nodes 1-35) and the right half (nodes 36-54) are lost; at time 120 the communication is restored. During the period of interference the nodes on the left half of the network do not have access to the measurements made on the right, and vice-versa; therefore the global inference error curve in Figure 7(d) changes: it is computed for each node using the posterior conditioned on the measurements only on its side of the network.

In Figure 7(d) we can see that, when combined with our architecture, the robust message passing algorithm achieves convergence before and after the inference period, but that the interference prevents a (complete) spanning tree from being formed. In spite of this, the robust message passing algorithm converges with an error that is very close to the optimum. In this period, each half of the network forms its own junction tree and uses message passing. The reason robust message passing does not converge to exactly the same result as global inference is because some prior factors needed by the left half of the network have been distributed to right half, and vice versa.

We also tested the architecture's performance under simulated node failures. The time to failure for each node was sampled i.i.d. from an exponential distribution. To avoid losing prior factors when nodes are lost, we distributed each prior factor redundantly to three nodes. Figure 7(e) shows the results of this experiment. As each node dies, its measurement is lost, and so the inference problem to be solved is changing over time; this accounts for the changing error values for global and local inference. Notice that the network can form a junction tree and solve the inference problem exactly past 500 seconds, when only 26 of the original 53 nodes are still functioning.

5.3 Regression experiments

Our next experiment evaluates our architecture on the regression task of §1.1.2. Using the distributed regression formulation and messages described in [8], we defined a regression problem on the temperature data from our lab deployment. We defined seven overlapping regions of influence, each of which has three local basis functions: a constant term and two linear terms. In addition, we included a basis function with constant value for the entire lab. The result of fitting this model to our temperature data is shown in Figure 2.

In our regression task, each node uses its local estimate of the optimal model parameters to predict the measurement of its five nearest neighbors, along with its own measurement. Figure 7(f) shows the resulting root mean squared error for this task. As with the calibration case, this graph shows three curves: the *local* curve corresponds to each node using its own measurement to predict its neighbors' measurements; the *global* curve corresponds to fitting the regression parameters offline, and using the resulting model for prediction; the *distributed* line uses our architecture and the distributed regression messages so that each node locally predicts its neighbors' values using its current estimates of the basis function coefficients. As with the calibration case, we see that the results obtained by our distributed algorithm quickly converge to those obtained by the optimal offline so-

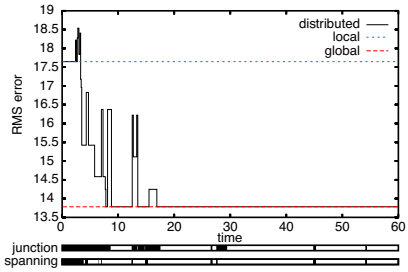


Figure 8: ***Results for optimal control problem.***

lution to the regression problem.

5.4 Optimal control experiments

Our third inference problem is an instance of the control problem described in §1.1.3. Based on our lab deployment, we defined an actuation problem where 16 blinds around the lab can be moved to change the light conditions; each blind is controlled by a specific node of the network. We define the light at each node in the lab as the sum of the lights at the nearby windows, decayed linearly with distance. The light values at the nodes are observed, but the light intensities at the windows are unknown; they were estimated by solving a sparse regression problem similar to kernel regression. Like the temperature experiments, this experiment used real light measurements obtained from our deployment.

Each actuating node has five possible controls, which raise and lower the blinds by varying amounts. Each node of the network has a desired light value that is 40 lux greater than its current value. The goal is to find positions for all the blinds that minimize the mean squared deviation from the desired light values. Notice that in this problem each blind affects many nodes, which means that to choose the optimal controls, conflicting goals must be resolved.

Our results, shown in Figure 8, again compare three methods: in the *local* curve each actuating node chooses the blind setting that best fits its desires; the *global* curve corresponds to the optimal solution obtained offline; the *distributed* curve uses our architecture to optimize the setting in a distributed fashion, where each actuating node chooses the control setting that it currently views as the best global solution. As with calibration and regression, we see that the control strategy obtained by our distributed algorithm quickly converges to that obtained by the optimal offline solution.

6. CONCLUSIONS

We presented the first robust and general architecture for inference in sensor networks which can solve a wide range of inference problems including probabilistic inference, regression, and optimization. In particular, we have presented distributed algorithms which can construct stable, valid junction trees, even in the presence of communication and node failures; we have also presented distributed algorithms to optimize this junction tree to minimize the cost of inference, and to solve the inference problem. We demonstrated the architecture on three applications, using data from a real sensor network deployment. Our experiments results also demonstrate that the inference algorithm quickly converges to the correct answer for all three applications, even in the presence of communication and node failures.

An important feature of our architecture is that it does not rely on a network layer that provides multi-hop routing (which is often difficult or impossible in sensor networks). This is due in part to the communication pattern of our algorithms: only neighbors in the tree must communicate with each other. Another reason is that our architecture tightly couples the application and networking layers so that both network-related and application-specific information can be used to minimize communication and computation; for example, the tree optimization algorithm uses link quality information and information about the current junction tree's structure to find edge swaps that will reduce the cost of inference. We expect that this tight coupling between the application and networking layers will be useful for other types of in-network data processing.

General architectures that address a range of sensor network applications (as well as the robustness issues of real systems) will significantly increase the usefulness of sensor network technology. We believe that the work presented herein provides a solid step towards this goal.

Acknowledgements. We gratefully acknowledge Wei Hong, Samuel Madden, and Romain Thibaux for their help deploying the sensor network, and Alec Woo for helpful discussions about the spanning tree algorithm which led to the descendant testing strategy presented in §2.4. Mark Paskin was supported by ONR N00014-00-1-0637 and an Intel Graduate Research Internship.

7. REFERENCES

[1] Aji, S. M., and McEliece, R. J. The generalized distributive law. *IEEE Transactions on Information Theory 46* (March 2000), 325–343.

[2] Byckovskiy, V., Megerian, S., Estrin, D., and Potkonjak, M. A collaborative approach to in-place sensor calibration. In *IPSN*, 2003.

[3] Coates, M. J. Distributed particle filtering for sensor networks. In *IPSN*, 2004.

[4] Cowell, R., Dawid, P., Lauritzen, S., and Spiegelhalter, D. *Probabilistic Networks and Expert Systems*. Springer, NY, 1999.

[5] Crick, C., and Pfeffer, A. Loopy belief propagation as a basis for communication in sensor networks. In *UAI*, 2003.

[6] Gallagher, R., Humblet, P., and Spira, P. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems 5* (January 1983), 66–77.

[7] Golub, G., and Van Loan, C. *Matrix Computations*. 1989.

[8] Guestrin, C., Thibaux, R., Bodik, P., Paskin, M. A., and Madden, S. Distributed regression: an efficient framework for modeling sensor network data. In *IPSN*, 2004.

[9] Guestrin, C. E., Koller, D., and Parr, R. Multiagent planning with factored MDPs. In *NIPS-14*, 2001.

[10] Hellerstein, J. M., Hong, W., Madden, S., and Stanek, K. Beyond average: Towards sophisticated sensing with queries. In *IPSN*, 2003.

[11] Intanagonwiwat, C., Estrin, D., Govindan, R., and Heidemann, J. Impact of network density on

data aggregation in wireless sensor networks. In *ICDCS*, 2002.

[12] Intanagonwiwat, C., Govindan, R., and Estrin, D. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *MobiCom*, 2000.

[13] Kjærulff, U. Approximation of bayesian networks through edge removals. Research Report IR-93-2007, Aalborg University, 1993.

[14] Mainwaring, A., Polastre, J., Szewczyk, R., Culler, D., and Anderson, J. Wireless sensor networks for habitat monitoring. Tech. Rep. IRB-TR-02-006, Intel Research, 2002.

[15] Michael Hamilton et al. Habitat sensing array, first year rep., 2003.

[16] Nowak, R., and Mitra, U. Boundary estimation in sensor networks: Theory and methods. In *IPSN*, 2003.

[17] Paskin, M. A., and Guestrin, C. E. Distributed inference in sensor networks. Submitted to the *UAI*, 2004. Online at `http://paskin.org/uai04-draft.pdf`.

[18] Plarre, K., and Kumar, P. R. Extended message passing algorithm for inference in loopy gaussian graphical models. *Ad Hoc Nets.*, 2004.

[19] Woo, A., Tong, T., and Culler, D. Taming the underlying challenges of reliable multihop routing in sensor networks. In *SenSys*, 2003.

[20] Zhao, F., Liu, J., Liu, J., Guibas, L., and Reich, J. Collaborative signal and information processing: An information directed approach. *Proceedings of the IEEE 91*, 8 (2003), 1199–1209.