Network Flows for Functions

Virag Shah Bikash Kumar Dey D. Manjunath
Department of Electrical Engineering
Indian Institute of Technology Bombay
Mumbai, India, 400 076
virag4u@gmail.com,{bikash,dmanju}@ee.iitb.ac.in

Abstract

We consider in-network computation of an arbitrary function over an arbitrary communication network. A network with capacity constraints on the links is given. Some nodes in the network generate data, e.g., like sensor nodes in a sensor network. An arbitrary function of this distributed data is to be obtained at a terminal node. The structure of the function is described by a given computation schema, which in turn is represented by a directed tree. We design computing and communicating schemes to obtain the function at the terminal at the maximum rate. For this, we formulate linear programs to determine network flows that maximize the computation rate. We then develop fast combinatorial primal-dual algorithm to obtain ϵ-approximate solutions to these linear programs. We then briefly describe extensions of our techniques to the cases of multiple terminals wanting different functions, multiple computation schemas for a function, computation with a given desired precision, and to networks with energy constraints at nodes.

I. Introduction

Motivated by sensor network applications, there has been significant interest in computing functions of distributed data inside the network. A typical scenario that is considered is as follows. Sensor nodes, distributed in a sensor field, can make measurements of their environment, perform reasonable amounts of computation and also communicate with other nodes. The interest of the sensor network is not so much in the measurement values made by the sensors but of some function of these variables, say Θ. Since the nodes in the network can perform computation, they could participate in the computation of Θ. Thus the interest is in distributed computation of a function of distributed data. This has also been called 'in-network function computation.' In this setting, it is typically assumed that the variables form a time sequence and that they can be generated at any rate; equivalently, an infinite sequence is readily available. Thus, in this setting it is natural to want to compute Θ at the best rate possible. In this paper, we introduce novel network flow techniques to design a computation and communication scheme that maximizes the rate at which Θ is computed. Though network flow techniques have been used widely to study multiple unicast [1]–[4] problems, our work develops such techniques for the first time for function computation.

Early work on in-network computation was on the asymptotic analysis of the number of transmissions needed to compute specific functions in noisy broadcast networks. e.g., [5]–[7]. In recent works, it is assumed that the node locations are from a realization of a suitable random point process, hence the resulting communication graph of the network is a random graph, e.g., [8]–[11]. In this setting a probabilistic characterization of the asymptotic (in the number of nodes) computation rate for different classes of functions, such as 'type-threshold functions' and 'type sensitive functions' [8], have been obtained.

Another class of work considers simplistic networks with small number of correlated sources [12]–[15]. Much of this work takes the information theoretic perspective in which the objective is to find encoding rate regions for reliably communicating the desired function. This class of work allows block coding to achieve better rates. There has been some recent work in the network coding literature on distributed function computation [16]–[18]. They consider larger and more complex networks with independent sources. However, designing optimal coding schemes and finding capacity is a difficult problem except for very special functions or networks [16], [17].

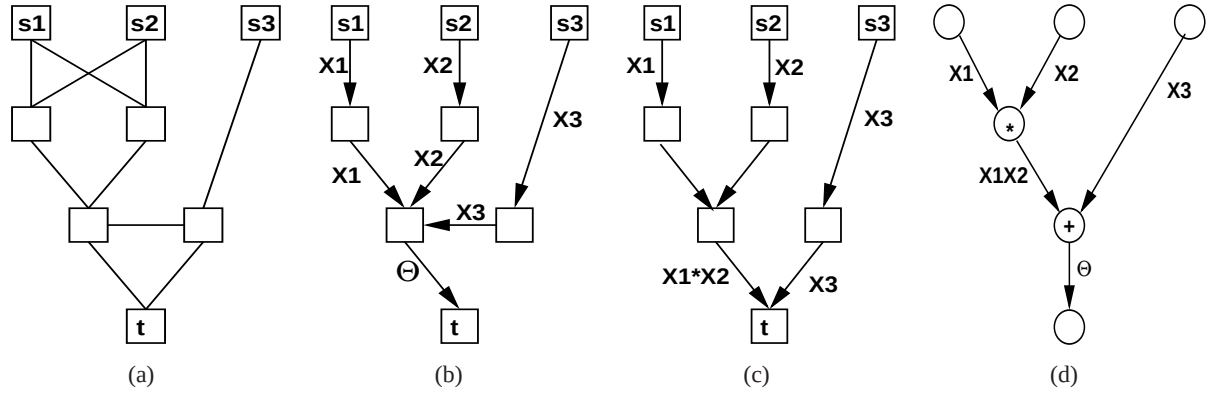


Fig. 1. Computing $\Theta = X_1X_2 + X_3$ over a network. (a) A network to compute $\Theta = X_1X_2 + X_3$. (b) A possible embedding that computes at Θ at unit rate. (c) An alternative embedding. (d) A schema to compute Θ.

In this paper we make a significant departure from the above. We consider arbitrary functions of the distributed data for which a computation schema is described by a directed tree. A computation schema defines a sequence of operations to compute the function. An arbitrary communication network over which Θ is to be computed is assumed given. Our techniques work for networks with both directed as well as undirected links with capacity constraints, though we present our results only for networks with undirected links. There are some similarities of our work with that of graph embedding. e.g., [19]–[21] but there are significant differences in the modeling assumptions and in the embedding objectives. Such work typically assume the target network to be a 'regular network' like a hypercube or a mesh and all link capacities are assumed equal. The embedding objective is to minimize the parameters like 'dialation.'

A. An Example and Motivation

Let us consider the function $\Theta(X_1, X_2, X_3) = X_1X_2 + X_3$ of three variables generated at three sources s_1, s_2, and s_3 respectively. A terminal node t is required to obtain the function $\Theta(X_1, X_2, X_3)$. We assume that all the three data symbols are from the same alphabet $\mathcal{A}$. The computation of the function can be broken into two parts, namely, first computing X_1X_2, and then adding X_3. These two operations can be done at different nodes in the network in the above order. This decomposition of the computation can be represented by the graph shown in Fig. 1(d). Such a graphical representation of the computation will henceforth be called a computation tree. Each edge represents a *unique* function of the source symbols.

Now consider computing $\Theta(X_1, X_2, X_3)$ in the network shown in Fig. 1(a) where each edge has unit capacity. There are multiple ways of receiving this function at the terminal t depending on what computations are done at what nodes and along what paths the data flows. Two such ways of computing this function are shown in Figs. 1(b) and 1(c). These are called 'embeddings', defined formally in Sec. II. It is clear that intelligent time-sharing between these various embeddings may give higher number of computation per use of the network on average than using only one such embedding. This raises the natural question: what is the maximum rate of computing that can be achieved on a given network and how to achieve it?

B. Organization and Summary of Contributions

We begin by describing the model in detail in the next section. Section III presents the main contributions of this paper. Here we formulate a linear program, *Embedding-Edge-LP*, that optimally allocates flows on the embeddings. We then present another LP, *Node-Arc-LP,*, based on a flow conservation law. This LP can be solved in polynomial time. We then describe an algorithm, *Algorithm 1*, that converts the flow rates obtained from *Node-Arc-LP* into a flow allocation on the embeddings. We then present a fast primal-dual algorithm which finds a solution to achieve at least $(1 - \epsilon)$ fraction of the optimal rate. We

call such a solution an ϵ-approximate solution. This algorithm uses an oracle subroutine which finds a minimum cost embedding of the computation tree in the network. We provide an efficient algorithm, *OptimalEmbedding(L)*, to obtain the same. This algorithm is also of independent interest. Four interesting extensions of our results are presented in Sec. IV. First, we allow multiple computing schema for computing the same function. Then we consider multiple terminals computing distinct functions of disjoint sets of sources. For this problem, we modify our techniques to maximize the weighted sum-rate of computations, and also to maximize the rate-tuple in a given direction. In the third extension, we consider the problem of computing a function with a desired precision which is achieved by allowing possibly different precision for each type of data. In the fourth extension, we consider a network with energy-constrained nodes, and assume that each type of data, i.e., each edge of the computation tree, requires some fixed but different amount of energy to compute/generate, transmit, and receive.

II. The model and the notation

The communication network is an undirected, simple, connected graph $\mathcal{N} = (V, E)$ where V is a set of n nodes and E is a set of m undirected edges. Each edge $uv \in E$ represents a half duplex link with a total non-negative capacity $c(uv)$. In the network, $S = \{s_1, s_2, \ldots, s_\kappa\} \subset V$ is the set of κ source nodes. Source s_i has an infinite sequence of data values $\{X_i(k)\}_{k\geq 0}$ where $X_i(k)$ belongs to a finite alphabet $\mathcal{A}$. The link capacities are expressed in $|\mathcal{A}|$-ary unit. X_i is used to denote a representative element of the sequence. Let $X \stackrel{\triangle}{=} [X_1, \ldots X_\kappa]$. Without loss of generality, we assume that each source node in the network generates exactly one data sequence. If a source node generates two or more data sequences then this can be represented by multiple source nodes connected by infinite capacity links. We also assume that there is only one terminal node.

A given function $\Theta : \mathcal{A}^\kappa \to \mathcal{A}$ of X needs to be obtained at the terminal node t for each k at the best possible rate. A computation schema for Θ is given and represented by a directed tree $\mathcal{G} = (\Omega, \Gamma)$ where Ω is the set of nodes and Γ is the set of edges. The elements of Ω are labelled $\mu_1, \mu_2, \ldots, \mu_{|\Omega|}$ where $\mu_1, \mu_2, \ldots, \mu_\kappa$ are the source nodes, $\mu_{|\Omega|}$ is the terminal node that obtains Θ and the rest are computing nodes that compute different functions of X. Further, the nodes in Ω are labelled according to a topological order such that for $i > j$ there is no directed path in $\mathcal{G}$ from μ_i to μ_j. The source nodes have in-degree zero and out-degree one and the terminal node has in-degree one and out-degree zero. All other nodes have in-degree greater than one and out-degree exactly one. Similarly, the elements of Γ are labelled $\theta_1, \theta_2, \ldots, \theta_{|\Gamma|}$ with $\theta_1, \theta_2, \ldots, \theta_\kappa$ being the outgoing edges from $\mu_1, \mu_2, \ldots, \mu_\kappa$ respectively, and $\theta_{|\Gamma|}$ being the incoming edge into $\mu_{|\Omega|}$. The remaining edges are labeled according to a topological order, i.e., for any $i < j$, there is no path from the head node of j to the tail node of i. The nodes and edges of $\mathcal{G}$ can be labeled as above in $O(|\Gamma|) = O(\kappa)$ time.

For any edge $\theta \in \Gamma$, let $tail(\theta)$ and $head(\theta)$ represent, respectively, the tail and the head nodes of the edge θ. Let $\Phi_\uparrow(\theta)$ and $\Phi_\downarrow(\theta)$ denote, respectively, the predecessors and the successors of θ, i.e.,

$$\begin{aligned} \Phi_\uparrow(\theta) &\stackrel{\triangle}{=} \{\eta \in \Gamma | head(\eta) = tail(\theta)\} \text{ and} \\ \Phi_\downarrow(\theta) &\stackrel{\triangle}{=} \{\eta \in \Gamma | tail(\eta) = head(\theta)\}. \end{aligned}$$

Each edge θ of $\mathcal{G}$ represents a distinct function of X that can be computed from the functions corresponding to the edges in $\Phi_\uparrow(\theta)$. Further, each function takes values from the same alphabet $\mathcal{A}$. (We remark here that this is not unreasonable even when all the computations are over real numbers because computations are performed using a fixed precision.)

Let $N(v) \stackrel{\triangle}{=} \{u \in V | uv \in E\}$ denote the set of neighbors of a node $v \in V$. We also denote the set of neighbors and itself by $N'(v) = N(v) \cup \{v\}$. A sequence of nodes $v_1, v_2, \cdots, v_l$, $l \geq 1$, is called a path if $v_i v_{i+1} \in E$ for $i = 1, 2, \ldots, l-1$. The set of all paths in $\mathcal{N}$ is denoted by $\mathcal{P}$. With abuse of notation, for such a path P, we will say $v_i \in P$ and also $v_i v_{i+1} \in P$. The nodes v_1 and v_l are called respectively the start node and the end node of P, and are denoted as start(P) and end(P).

As discussed in Sec. I, a function with a given computation tree can be computed along any "embedding" of the tree in the network as shown in Fig. 1. We are now ready to formally define an embedding of a computation tree.

Definition: An embedding is a mapping $B : \Gamma \to \mathcal{P}$ such that

1) start$(B(\theta_l)) = s_l$ for $l = 1, 2, \ldots, \kappa$
2) end$(B(\eta)) =$ start$(B(\theta))$ if $\eta \in \Phi_{\uparrow}(\theta)$
3) end$(B(\theta_{|\Gamma|})) = t$.

We denote the set of embeddings of $\mathcal{G}$ in $\mathcal{N}$ by $\mathcal{B}$. Our aim is to determine the flows on these embeddings so as to maximize the total flow. An edge in the network may carry different functions of the source data in an embedding. We thus define the number of times an edge $e \in E$ is used in an embedding B as $r_B(e) = |\{\theta \in \Gamma | e \text{ is a part of } B(\theta)\}|$. Note that $|r_B(e)| \leq |\Gamma|$ for any edge, and $r_B(e) = 0$ for an edge e which is not used by the embedding B. Further, an edge may also be used to carry flows on different embeddings. Therefore in an assignment of flows on different embeddings, i.e., in a particular timesharing scheme, the edge may carry multiple types of data (i.e., different functions of X) of different amounts.

III. Linear programs and algorithms

In this section, we present our main contributions.

- In Section III-A, we give a basic linear program, the *Embedding-Edge LP*, which characterizes our problem.
- In Section III-B, we give an alternate LP, the *Node-Arc LP*, that can be solved in polynomial time. We then present an algorithm which obtains a solution of the *Embedding-Edge LP* with the same rate from a solution of the *Node-Arc LP*.
- Drawing parallels from multi-commodity flow techniques, we give, in Section III-C, the dual of our *Embedding-Edge LP* and present a fast primal-dual algorithm to compute an ϵ-approximate solution. This algorithm needs a subroutine which finds a 'minimum weight embedding' of the computation tree in the network for given edge-weights. We present an efficient exact algorithm for this purpose. This algorithm is of independent interest, for instance, for computing functions over a network with power limited, but with infinite bandwidth, links.

Note that, if start$(B(\theta_i)) =$ end$(B(\theta_i))$, i.e., if $B(\theta_i)$ consists of a single node, then in that embedding the data θ_i is generated as well as used (i.e., not forwarded to another node) in that node.

A. The Embedding-Edge LP

As discussed in Sec. I and Sec. II, the function for a particular sample of the data can be computed over the network using any embedding of the computation tree in the network. Let $\mathcal{B}$ be the set of all embeddings of $\mathcal{G}$ in $\mathcal{N}$. For any embedding $B \in \mathcal{B}$, let $x(B)$ denote the average number of function symbols computed using the embedding B per use of the network. We present below a linear program to maximize the computation rate $\lambda = \sum_{B \in \mathcal{B}} x(B)$. Recall that $r_B(e)$ represents the number of times the edge e is used in the embedding B.

Embedding-Edge LP: Maximize $\lambda = \sum_{B \in \mathcal{B}} x(B)$ subject to

1. Capacity constraints

$$\sum_{B \in \mathcal{B}} r_B(e) x(B) \leq c(e), \ \forall e \in E \tag{1}$$

2. Non-negativity constraints

$$x(B) \geq 0, \ \forall B \tag{2}$$

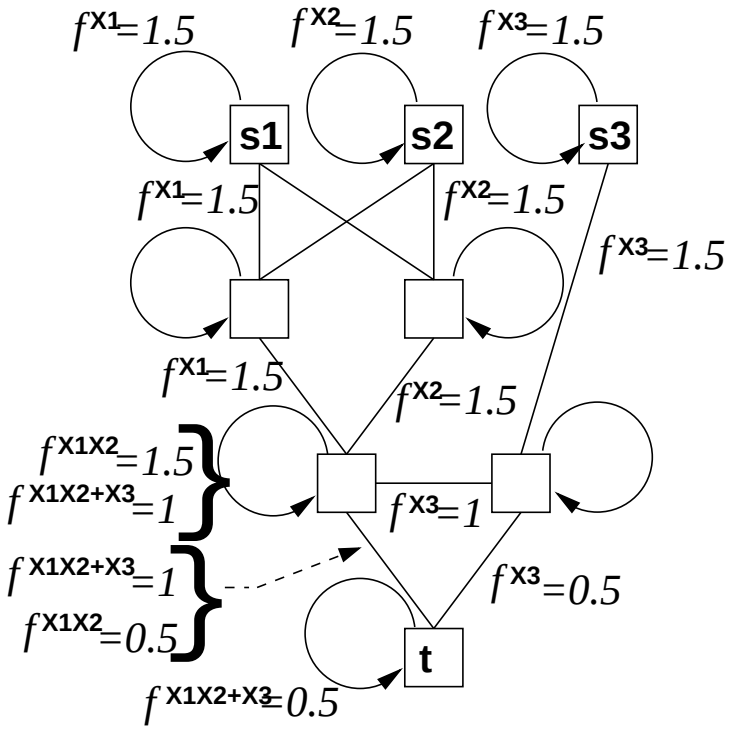


Fig. 2. The aggregate edge-flow values for a flow of 0.5 on the embedding in Fig. 1(c) and a flow of 1 on the embedding in Fig. 1(b).

This LP finds an optimal fractional packing of the embeddings of $\mathcal{G}$ into $\mathcal{N}$. Similar formulations have been considered widely in literature in the context of multi-commodity flow [2], [22] and other packing problems [2].

In multi-commodity flow problems, a solution of the so called *Path-edge LP* readily gives a way of achieving the corresponding rates. However, since in our problem, the data is to be mixed according to different embeddings for different realizations of data, one needs to carefully device a protocol to schedule the computation and communication at the nodes and edges in such a way that data from different realizations are not mixed. Such a protocol is presented in the appendix.

B. The Node-Arc LP

Note that the cardinality of $\mathcal{B}$ can be exponential in $|V|$. Hence the complexity of the *Embedding-Edge LP* is exponential in the network parameters if any other structure of the problem is not used. In the multi-commodity flow literature, another LP formulation, called the *Node-Arc LP*, based on the flow conservation principle is well-known which can be solved in polynomial time. In the following, we formulate a node conservation based LP for our problem. For this LP, we assume that each node in the network has a virtual self-loop of infinite capacity. The data flowing in the self-loop represents the data generated at that node. This may be the source data generated at the sources or the intermediate or final values computed at the node. For example, if a node computes X_1X_2 from X_1 and X_2 it receives, and then computes $X_1X_2 + X_3$ by using the computed X_1X_2 and received X_3, then both X_1X_2 and $X_1X_2 + X_3$ will be assumed to be flowing in its self-loop. Example of the flows on the edges and the self-loops corresponding to a particular flow assignment on two embeddings is shown in Fig. 2.

The variables in our *Node-Arc LP* are

$$\left\{f^{\theta}_{uv}, f^{\theta}_{vu} | uv \in E, \theta \in \Gamma\right\} \cup \left\{f^{\theta}_{uu} | u \in V, \theta \in \Gamma\right\} \cup \{\lambda\}.$$

where, f^{θ}_{uv} represents the flow of type $\theta \in \Gamma$ flowing through the edge $uv \in E$ from u to v, f^{θ}_{uu} denotes the flow of type θ flowing in the self-loop at u and λ represents the total rate of the function computation.

The linear program consists of capacity constraint on the edges of $\mathcal{N}$, a flow-conservation rule on the nodes of $\mathcal{N}$, and non-negativity constraint on the flows f_{uv}^{θ}. The flow conservation rule is based on the fact that an intermediate node in $\mathcal{N}$ can, apart from forwarding the flows it receives, generate a flow of type θ on its self-loop by terminating the same amount of incoming flows of type $\eta \in \Phi_{\uparrow}(\theta)$. Each source node s_l, in addition, generates λ amount of flow of type θ_l. Similarly, the terminal node t terminates λ amount of flow of type $\theta_{|\Gamma|}$. The *Node-Arc LP* is as follows. Recall that $N'(v)$ denotes the set of the neighbors of v and itself.

Node-Arc LP: Maximize λ subject to following constraints any node $v \in V$

1. Functional conservation of flows:

$$f_{vv}^{\eta} + \sum_{u \in N(v)} f_{vu}^{\theta} - \sum_{u \in N'(v)} f_{uv}^{\theta} = 0, \ \ \forall \theta \in \Gamma \setminus \{\theta_{|\Gamma|}\} \text{ and } \forall \eta \in \Phi_{\downarrow}(\theta). \tag{3}$$

2. Conservation and termination of $\theta_{|\Gamma|}$:

$$\sum_{u \in N(v)} f_{vu}^{\theta_{|\Gamma|}} - \sum_{u \in N'(v)} f_{uv}^{\theta_{|\Gamma|}} = \begin{cases} -\lambda & v = t \\ 0. & \text{otherwise} \end{cases} \tag{4}$$

3. Generation of θ_l $\forall l \in \{1, 2, \ldots, \kappa\}$:

$$f_{vv}^{\theta_l} = \begin{cases} \lambda & v = s_l \\ 0. & \text{otherwise} \end{cases} \tag{5}$$

4. Capacity constraints

$$\sum_{\theta \in \Gamma} \left(f_{uv}^{\theta} + f_{vu}^{\theta} \right) \leq c(uv), \ \forall uv \in E. \tag{6}$$

5. Non-negativity constraints

$$f_{uv}^{\theta} \geq 0, \forall uv \in E \text{ and } \forall \theta \in \Gamma \tag{7}$$

$$f_{uu}^{\theta} \geq 0, \forall u \in V \text{ and } \forall \theta \in \Gamma \tag{8}$$

$$\lambda \geq 0. \tag{9}$$

This LP has $O(\kappa m)$ number of variables, $O(\kappa m)$ number of non-negativity constraints (one for each variable), and $O(\kappa n + m)$ number of other constraints. Hence it can be solved in polynomial time.

The above LP gives a set of flow values on each link. Now we briefly describe and present an algorithm, Algorithm 1, which, from any feasible solution of this LP, obtains a corresponding feasible solution for the *Embedding-Edge LP* that achieves the same λ.

Each iteration of the *while* loop finds an embedding with a non-zero flow and removes the corresponding edge-flows to obtain another feasible solution with a reduced rate. This continues until λ amount of flow has been extracted. The i-th iteration of the *for* loop finds the mapping of θ_i in the embedding. While exploring the nodes to find the mapping of θ_i, it checks for the presence of a cycle of flow of type θ_i. It removes such a cycle if detected.

Proof of correctness of Algorithm 1: The proof of the following statements ensures the correctness of the algorithm.

1) In the third line inside the *for* loop, there exists a $u \in N'(v)$ such that $f_{uv}^{\theta} > 0$.
2) If a cycle of redundant flow is found and removed in the first *if* block inside the *for* loop, then the remaining flows still satisfy the constraints in the LP with λ replaced by $\lambda - \lambda'$.

Algorithm 1: Finding equivalent solution of the *Embedding-Edge LP* from a feasible solution of the *Node-Arc LP*.

input : Network graph $\mathcal{N} = (V, E)$, capacities $c(e)$, set of source nodes S, terminal node t, computation tree $\mathcal{G} = (\Omega, \Gamma)$, and a feasible solution to its *Node-Arc LP* that consists of the values of λ, $f^{\theta}_{uv}\ \forall\theta \in \Gamma$, $\forall uv \in E$, and $f^{\theta}_{uu}\ \forall\theta \in \Gamma,\ \forall u \in V$.

output: Solution $\{x(B)|B \in \mathcal{B}\}$ to the *Embedding-Edge LP* with $\sum_{B\in\mathcal{B}} x(B) = \lambda$.

Initialize $x(B) := 0, B(\theta_i) = \emptyset$ (the null sequence), $\forall B \in \mathcal{B}$ and $\forall\theta_i \in \Gamma, \lambda' = 0$

while $\lambda' \neq \lambda$ **do**
- $z(t) := \lambda$;
- $B(\theta_{|\Gamma|}) := t$;
- **for** $i := |\Gamma|$ **to** 1 **do**
  - $v := B(\theta_i)$; `// valid, as` $B(\theta_i)$ `has of only one node at this step`
  - $u :=$ an element in $N'(v)$ such that $f^{\theta_i}_{uv} > 0$;
  - **if** $u \neq v$ *and* $u \in B(\theta_i)$ **then**
    - `// A cycle of redundant flow found: remove the flow from all the edges in the cycle`
    - Let P be the path in $B(\theta_i)$ upto the first appearance of u in it.;
    - Delete P from $B(\theta_i)$. ;
    - $y := \min_{u'v'\in\{uv\}\cup P}\left(f^{\theta_i}_{u'v'}\right)$;
    - $f^{\theta}_{u'v'} := f^{\theta}_{u'v'} - y\ \forall u'v' \in \{uv\} \cup P$
  - **end**
  - **else**
    - $z(u) := \min\left(z(v), f^{\theta_i}_{uv}\right)$;
  - **end**
  - **if** $u \neq v$ **then**
    - Prefix u in $B(\theta_i)$;
    - $v := u$;
    - Jump to the second statement inside the **for** loop ;
  - **end**
  - **else**
    - $B(\eta) := u, \forall\eta \in \Phi_{\uparrow}(\theta_i)$;
  - **end**
- **end**
- $x(B) := z(s_1)$; `// Flow extracted on` B
- $\lambda' := \lambda' + x(B)$; `// Total flow extracted`
- `// Remove` $x(B)$ `amount of flow from all the edges in` B.
- $f^{\theta}_{u'v'} := f^{\theta}_{u'v'} - x(B)\ \forall\theta \in \Gamma$ and $\forall u'v' \in B(\theta)$;
- `// Remove` $x(B)$ `amount of flow from all the relevant self-loops.`
- $f^{\theta}_{v'v'} := f^{\theta}_{v'v'} - x(B)\ \forall\theta \in \Gamma$ and $v' = \text{start}(B(\theta))$;

end

3) At the end of each iteration of the *while* loop, the remaining flows still satisfy the constraints in the LP with λ replaced by $\lambda - \lambda'$.
4) The algorithm terminates in finite time.

We now outline a proof of each of these statements. We prove the statements 1)–3) for a certain iteration of the loops while assuming that all the above claims are true in all the previous iterations of the *while* and *for* loops.

Proof of 1: The current values of the flows satisfy all the constraints in the *Node-Arc LP* with λ replaced by $\lambda - \lambda'$. The algorithm ensures that in this step, the total outgoing flow $\sum_{u \in N(v)} f^{\theta}_{vu} \geq z(v) > 0$. So, by constraints (3) and (4), the total of incoming and generated flows $\sum_{u \in N'(v)} f^{\theta}_{uv} > 0$. Hence the statement follows.

Proof of 2: We will prove that a cyclic flow on a cycle $v_1, v_2, \cdots, v_l, v_1$ satisfies all the constraints in the *Node-Arc LP* with $\lambda = 0$. Then clearly after subtracting this flow from the edges of the cycle, the remaining flows in the network will still satisfy the constraints with the same λ as before. For a cyclic flow of type θ of volume y, the flow values are $f^{\theta}_{v_i v_{i+1}} = y$ for $i = 1, 2, \cdots, l-1$, $f^{\theta}_{v_l v_1} = y$, and all other flow values are equal to 0. So, for any node, any nonzero incoming flow is 'compensated' by the same amount of outgoing flow of the same type. All flow values in the self-loops are also 0. So clearly these flows satisfy the constraints in the LP with $\lambda = 0$. This completes the proof.

Proof of 3: Again, we will prove that the removed $x(B)$ amount of flows on the edges of an embedding and on the self-loops themselves satisfy the constraints in the LP with $\lambda = x(B)$. Then the remaining flows will also satisfy the constraints with λ replaced by $\lambda - x(B)$. The subtracted flow values are $f^{\theta}_{uv} = x(B)$ for $uv \in B(\theta)$, $f^{\theta}_{uu} = x(B)$ for $u = \text{start}(B(\theta))$, and all other flow values 0. We can verify that these flows satisfy the constraints in the *Node-Arc LP*.

Proof of 4: The *Node-Arc LP* has $O(m|\Gamma|)$ number of variables f^{θ}_{uv} and f^{θ}_{uu}. Each deletion of flows through a cycle, or through an embedding, makes at least one of these variables zero. Since the number of steps in each iteration is finite, the algorithm ends in finite time. ■

It can be checked that the overall complexity of Algorithm 1 is $O(\kappa^2 m^2)$.

C. Primal-dual algorithm and min-cost embedding

The *Node-Arc LP* and the subsequent algorithm to find an optimal solution of the *Embedding-Edge LP* has polynomial-time complexity. For the multi-commodity flow problem, and for more general packing problems, Garg and Konemann [2] gave a faster primal-dual algorithm to find an ϵ-approximate solution. The algorithm uses a hypothetical subroutine/oracle. For the multi-commodity flow problem, the subroutine finds the shortest paths between the source-terminal pairs. We now give a similar fast algorithm to find an ϵ-approximate solution to the *Embedding-Edge LP*.

We first provide the dual of the *Embedding-Edge LP*. The dual has the variables $L = (l(e))_{e \in E}$ corresponding to the capacity constraints in the primal. The dual LP is given as follows.

Dual of Embedding-Edge LP: Minimize $D(L) = \sum_{e \in E} c(e) l(e)$ subject to
1. Constraints corresponding to each $x(B)$ in primal:

$$\sum_{e \in B} r_B(e) l(e) \geq 1, \ \forall B \tag{10}$$

2. Non-negativity constraints:

$$l(e) \geq 0, \ \forall e \in E \tag{11}$$

We define the weight of an embedding B as

$$w_L(B) = \sum_{e \in B} r_B(e) l(e).$$

It can be checked (similar to [2]) that the dual LP is equivalent to finding $\min_L \frac{D(L)}{\alpha_L}$, where

$$\alpha_l = \min_B w_l(B)$$

is the cost of the minimum cost embedding for L.

The *Embedding-Edge LP* is a fractional packing LP of the type considered by Garg and Konemann [2] and Plotkin *et al.* [23]. A polynomial time primal-dual algorithm was presented in [2] for such LPs assuming the existence of an efficient oracle subroutine which finds a 'shortest path.' For a packing LP $\max\left\{a^T x | Ax \leq b, x \geq 0\right\}$ and its dual LP $\min\left\{b^T y | A^T y \geq a, y \geq 0\right\}$, the shortest path is defined as $\sum_i A(i,j)y(i)/a(j)$ [2]. It is easy to see that for our LP, the 'shortest path' corresponds to the embedding with minimum weight, $\arg\min_B w_L(B)$. Algorithm 2 gives the instance of the primal-dual algorithm for our problem.

Algorithm 2: Algorithm for finding approximately optimal x and λ

input : Network graph $\mathcal{N} = (V, E)$, capacities $c(e)$, set of source nodes S, terminal node t, computation tree $\mathcal{G} = (\Omega, \Gamma)$, the desired accuracy ϵ
output: Primal solution $\{x(B), B \in \mathcal{B}\}$

Initialize $l(e) := \delta/c(e)$, $\forall e \in E, x(B) := 0, \forall B \in \mathcal{B}$;
while $D(l) < 1$ **do**
  B^* := OptimalEmbedding(L) ; `// OptimalEmbedding(`L`) outputs` $\arg\min_B w_L(B)$
  e^* := edge in B^* with smallest $c(e)/r_{B^*}(e)$;
  $x(B^*) := x(B^*) + c(e^*)/r_{B^*}(e^*)$;
  $l(e) := l(e)(1 + \epsilon \frac{c(e^*)/r_{B^*}(e^*)}{c(e)/r_{B^*}(e)}),\ \forall e \in B^*$;
end
$x(B) := x(B)/\log_{1+\epsilon} \frac{1+\epsilon}{\delta}, \forall B$;

We now describe, and then provide below, the subroutine OptimalEmbedding(L) which finds a minimum weight embedding of $\mathcal{G}$ on $\mathcal{N}$ with a given length function L. For each edge θ_i, starting from θ_1, it finds a way to compute θ_i at each network node at the minimum cost possible. It keeps track of that minimum cost and also the 'predecessor' node from where it receives θ_i. If θ_i is computed at that node itself then the predecessor node is itself. This is done for each θ_i by a technique similar to the Dijkstra's algorithm. Computing θ_i for $i \in \{1, 2, \ldots, \kappa\}$ at the minimum cost at a node u is equivalent to finding the shortest path to u from s_i. We do this by using Dijkstra's algorithm. For any other i, the node u can either compute θ_i from $\Phi_\uparrow(\theta_i)$ or receive it from one of its neighbors. To take this into account, unlike Dijkstra's algorithm, we initialize the cost of computing θ_i with the cost of computing $\Phi_\uparrow(\theta_i)$ at the same node. With this initialization, the same principle of greedy node selection and cost update as in Dijkstra's algorithm is used to find the optimal way of obtaining θ_i at all the nodes. Finally, the optimal embedding is obtained by backtracking the predecessors. Starting from t, we backtrack using predecessors from which $\theta_{|\Gamma|}$ is obtained, till we hit a node whose predecessor is itself. This node is the start node of $B(\theta_{|\Gamma|})$ and the end node of $B(\eta)$ for all $\eta \in \Phi_\uparrow(\theta_{|\Gamma|})$. The complete embedding is obtained by continuing this process for each θ_i in the reverse topological order.

Correctness of OptimalEmbedding(L): It is sufficient to show that, during each phase i, the algorithm computes optimal values for $\omega_u(\theta_i)$ and $\sigma_u(\theta_i)$, for each node u in $\mathcal{N}$. We prove this by induction on the pair $(i, |\Psi|)$ according to the lexicographic ordering. For $i \in \{1, \ldots, \kappa\}$ and for all $|\Psi|$, this follows from the correctness of Dijkstra's algorithm. Now, assuming the optimality of $\omega_u(\theta_i)$ and $\sigma_u(\theta_i)$ till all iterations before $(i, |\Psi|)$, we prove the statement for $(i, |\Psi|)$. Suppose v is the element added to Ψ in the current iteration. We consider two cases:

Case 1: $\Psi = \{v\}$: The cost of computing (and not receiving from another node) θ_i at any node u is $\sum_{\eta \in \Phi_\uparrow(\theta_i)} \omega_u(\eta)$. The algorithm chooses v which has the minimum $\sum_{\eta \in \Phi_\uparrow(\theta_i)} \omega_u(\eta)$ among all nodes

Procedure OptimalEmbedding(L)

input : Network graph $\mathcal{N} = (V, E)$, Length function L, set of source nodes S, terminal node t, computation tree $\mathcal{G} = (\Omega, \Gamma)$.
output: Embedding B^* with minimum weight under L

for $i = 1$ **to** $|\Gamma|$ **do**
  if $i \in \{1, 2, \ldots, \kappa\}$ **then**
    $\omega_u(\theta_i) := \infty, \forall u \in V - \{s_i\}$;
    $\omega_{s_i}(\theta_i) := 0$ and $\sigma_{s_i}(\theta_i) := s_i$;
  end
  else
    $\omega_u(\theta_i) := \sum_{\eta \in \Phi_\uparrow(\theta_i)} \omega_u(\eta), \forall u \in V$;
    $\sigma_u(\theta_i) := u, \ \forall u \in V$;
  end
  $\Psi := \emptyset$; $\bar{\Psi} := V$;
  while $|\Psi| < n$ **do**
    $v := \arg\min_{u \in \bar{\Psi}} \omega_u(\theta_i)$;
    $\Psi := \Psi \cup \{v\}$;
    $\bar{\Psi} := \Psi - \{v\}$;
    foreach $u \in N(v)$ **do**
      if $\omega_v(\theta_i) + l(uv) < \omega_u(\theta_i)$ **then** $\omega_u(\theta_i) := \omega_v(\theta_i) + l(uv)$ and $\sigma_u(\theta_i) := v$;
    end
  end
end
$B^*(\theta_{|\Gamma|}) := t$;
for $i = |\Gamma|$ **to** 1 **do**
  $u := B^*(\theta_i)$; `// valid, as` $B^*(\theta_i)$ `consists of only a node at this step`
  while $\sigma_u(\theta_i) \neq u$ **do**
    Prefix $\sigma_u(\theta_i)$ to $B^*(i)$;
    $u := \sigma_u(\theta_i)$;
  end
  $B(\eta) := u \ \forall \eta \in \Phi_\uparrow(\theta_i)$;
end

$u \in V$ and assigns $\omega_v(\theta_i) = \sum_{\eta \in \Phi_\uparrow(\theta_i)} \omega_v(\eta)$ and $\sigma_v(\theta_i) = v$. If these are not optimal, then it must be more efficient for v to receive θ_i which is computed at some other node u. But that implies $\sum_{\eta \in \Phi_\uparrow(\theta_i)} \omega_u(\eta) < \sum_{\eta \in \Phi_\uparrow(\theta_i)} \omega_v(\eta)$, which is a contradiction to the choice of v.

Case 2: $\{v\} \subsetneq \Psi$: Suppose there is a more efficient way of receiving θ_i at v than from the node selected as $\sigma_v(\theta_i)$ and that is to compute θ_i at a node u and receive it along a path $P_{u,v}$. Let the corresponding cost be $\omega'_v(\theta_i)$. First, if $u \in \Psi'$, then the present cost $\left(\leq \sum_{\eta \in \Phi_\uparrow(\theta_i)} \omega_u(\eta)\right)$ at u is less than the present value of $\omega_v(\theta_i)$, which is a contradiction to the choice of v. Thus $u \in \Psi$. Let u' be the last node in $P_{u,v}$ from Ψ, and v' be the first node in $P_{u,v}$ from Ψ'. Then $\omega'_v(\theta_i) \geq \omega_{u'}(\theta_i) + l(u'v') \geq \omega_{v'}(\theta_i) \geq \omega_v(\theta_i)$ — a contradiction. Here the first inequality follows since $u' \in \Psi$. The second inequality follows from the update rule followed during the inclusion of u' in Ψ. The last inequality follows from the choice of v.

Complexity of OptimalEmbedding(L) and the primal-dual algorithm: Let us consider the first *for* loop in OptimalEmbedding(L). Each iteration of this loop is the same as Dijkstra's algorithm except for the initialization. Thus, the for loop, excluding the initialization step, can be run in $O(m + n \log n)$ time using

Fibonacci heap implementation. The initialization step requires $O(n|\Phi_\uparrow(\theta_i)|)$ time for each iteration. The second *for* loop has $O(n\kappa)$ complexity. So the overall algorithm takes $O(\kappa(m + n\log n))$ time.

The number of iterations in the primal-dual algorithm is of the order $O(\epsilon^{-1}m\log_{1+\epsilon}(m))$. Thus the overall complexity of the algorithm is $O\left(\epsilon^{-1}\kappa m(m + n\log n)\log_{1+\epsilon}(m)\right)$.

IV. Extensions

1. Multiple trees for the same function: It may be possible to compute a function in different sequences of operations which are expressed by different computation trees. For example, the 'sum' function $f(X_1, X_2, X_3) = X_1+X_2+X_3$ may be computed by any of the computation sequences $\big((X_1+X_2)+X_3\big)$, $\big(X_1+(X_2+X_3)\big)$, or $\big(X_2+(X_1+X_3)\big)$. In general, suppose multiple computation trees $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_\nu$ are given for computing the same function. Let $\mathcal{B}_i$ denote the set of all embeddings of $\mathcal{G}_i$ for $i = 1, 2, \ldots, \nu$. Let $\mathcal{B} = \cup_i\mathcal{B}_i$ denote the set of all embeddings. Under this definition of $\mathcal{B}$, the *Embedding-Edge LP* for this problem is the same as that for a single tree. The new OptimalEmbedding(L) algorithm finds an optimal embedding for each $\mathcal{G}_i$ and chooses the one with minimum weight as the optimal embedding in $\mathcal{B}$. This can be used in the same primal-dual algorithm to find an ϵ-approximate solution.

Some edges of different trees may represent an identical function of the sources. For example, for the function $X_1+X_2+X_3+X_4$, an edge corresponding to the function X_1+X_2 is present in each of the trees corresponding to $\Big(\big((X_1+X_2)+X_3\big)+X_4\Big)$, $\big((X_1+X_2)+(X_3+X_4)\big)$, and $\Big(\big((X_1+X_2)+X_4\big)+X_3\Big)$. For this reason, OptimalEmbedding(L) algorithm can be made more efficient by running iterations for each function rather than each edge. The initialization of $\omega_u(\theta)$ changes correspondingly, to take into account all possible ways of computing that function. Rest of the algorithm remains the same.

The particular function $\Theta(X_1, X_2, \ldots, X_\kappa) = X_1 + X_2 + \ldots + X_\kappa$ is of special theoretical as well as practical interest. There are many, of the order of $\kappa!$, sequences of additions of data and corresponding trees to get this function. With the above modification, our OptimalEmbedding(L) algorithm has complexity exponential in κ and linear in m. As a result, our primal-dual algorithm gives an ϵ-approximate solution in exponential complexity in κ and quadratic in m. The problem is equivalent to the much investigated multicast problem. For this problem, and consequently for the function 'sum', the oracle finds a minimum weight Steiner tree. This is well-known to be NP-hard on κ. Approximate (but not ϵ-approximate for any given ϵ) polynomial complexity algorithms are known (see [24] and citations therein) for finding a minimum weight Steiner tree. This can also be used to find approximate solution to the multicast, and hence the 'sum', in polynomial complexity [24].

2. Multiple functions and multiple terminals: Suppose the network has multiple terminals $t_1, t_2, \ldots, t_\gamma$ wanting functions $\Theta_1(X^{(1)}), \Theta_2(X^{(2)}), \ldots, \Theta_\gamma(X^{(\gamma)})$ respectively. Here $X^{(i)}$ is the data generated by a set of sources $S^{(i)}$. The sets $S^{(i)}; i = 1, 2, \ldots, \gamma$ are assumed to be pairwise disjoint. For each function Θ_i, a computation tree $\mathcal{G}_i$ is given. Let us consider the problem of communicating the functions to the respective terminals at rates $\lambda_1, \lambda_2, \ldots, \lambda_\gamma$. The problem is to determine the achievable rate region which is defined as the set of $\mathbf{r} = (\lambda_1, \lambda_2, \ldots, \lambda_\gamma)$ for which a protocol exists for transmission of the functions at these rates. This region can be approximately found by solving either of the following problems.

(i) For any given non-negative weights $\alpha_1, \alpha_2, \ldots, \alpha_\gamma$, what is the maximum achievable weighted sum-rate $\sum_{i=1}^{\gamma}\alpha_i\lambda_i$?

For this problem, we consider embeddings of the computation trees $\mathcal{G}_i$ into the network for each terminal t_i. Let $\mathcal{B}_i$ denote the set of all embeddings of $\mathcal{G}_i$. Then the *Embedding-Edge LP* for this problem is to maximize $\sum_{i=1}^{\gamma}\alpha_i\sum_{B\in\mathcal{B}_i}x(B)$. The constraints are the same as before with $\mathcal{B}$ defined by $\mathcal{B} = \cup_i\mathcal{B}_i$. The weight of an embedding $B \in \mathcal{B}$ under a weight function L is defined as $\alpha_i w_L(B)$ if $B \in \mathcal{B}_i$. The new OptimalEmbedding(L) algorithm finds an optimal embedding for each $\mathcal{G}_i$ and chooses the one with minimum weight. This can be used in the same primal-dual algorithm to find an ϵ-approximate solution. It is also easy to obtain a *Node-Arc LP* for this problem by minor modifications to that for a single function computation at a single terminal.

(ii) For any non-negative demands $\alpha_1, \alpha_2, \ldots, \alpha_\gamma$, what is the maximum λ for which the rates $\lambda\alpha_1, \lambda\alpha_2, \ldots, \lambda\alpha_\gamma$ are concurrently achievable?

Here, we define an embedding to be a tuple $B = (B_1, B_2, \ldots, B_\gamma)$, where $B_i \in \mathcal{B}_i$ is an embedding of the computation tree $\mathcal{G}_i$. The *Embedding-Edge LP* for this problem is the same as that for the single terminal problem with $r_B(e)$ defined as $r_B(e) = \sum_{i=1}^{\gamma} \alpha_i |\{\theta \in \Gamma_i | e \text{ is a part of } B_i(\theta)\}|$ and $\mathcal{B} = \mathcal{B}_1 \times \mathcal{B}_2 \times \ldots \times \mathcal{B}_\gamma$. The weight of an embedding B under a weight function L is defined as $\sum_{i=1}^{\gamma} \alpha_i w_L(B_i)$. The new OptimalEmbedding(L) algorithm finds an optimal embedding B by separately finding optimal embeddings B_i for each $\mathcal{G}_i$. This can be used in the same primal-dual algorithm to find an ϵ-approximate solution. Again, we can easily obtain a *Node-Arc LP* by minor modification to that for a single function computation at a single terminal.

3. **Computing with a precision:** In practice, the source data may be real-valued, and communicating such a data requires infinite capacity. In such applications, it is common to require a quantized value of the function at the terminal with a desired precision. This may, in turn, be achieved by quantizing various data types with pre-decided precisions and thus different data type may require different number of bits to represent them. Suppose the data type denoted by θ is represented using $b(\theta)$ bits. Then the *Embedding-Edge LP* and its dual for this problem are the same as before except that the definition of $r_B(e)$ is changed to $r_B(e) = \sum_{\theta \in \Gamma: e \text{ is a part of } B(\theta)} b(\theta)$. In the *Node-Arc LP*, the capacity constraints are changed to

$$\sum_{\theta \in \Gamma} \left(f_{uv}^{\theta} + f_{vu}^{\theta}\right) b(\theta) \leq c(uv), \ \forall uv \in E.$$

In the *OptimalEmbedding(L)* algorithm, $l(uv)$ is replaced by $l(uv)b(\theta_i)$ inside the **foreach** loop.

4. **Energy limitted sensors:** Suppose, instead of capacity constraints on the links, each node $u \in V$ has a total energy $E(u)$. Each transmission and reception of θ require the energy $E_{T,\theta}$ and $E_{R,\theta}$ respectively. Generation of one symbol of θ or computation of one symbol of θ from $\Phi_{\uparrow}(\theta)$ requires the energy $E_{C,\theta}$. The objective is to compute the function at the terminal maximum number of times with the given total node energy at each node.

For an embedding B, if $B(\theta) = v_1, v_2, \cdots, v_l$, then $tr(B(\theta)) = \{v_1, v_2, \cdots, v_{l-1}\}$ denotes the transmitting nodes, and $rx(B(\theta)) = \{v_2, v_3, \cdots, v_l\}$ denotes the receiving nodes of θ. If $l = 1$, then $tr(B(\theta)) = rx(B(\theta)) = \emptyset$. For B, the energy load on the node u is given by

$$E_B(u) = \sum_{\theta: \text{start}(B(\theta))=u} E_{C,\theta} + \sum_{\theta: u \in tx(B(\theta))} E_{T,\theta} + \sum_{\theta: u \in rx(B(\theta))} E_{R,\theta}.$$

The capacity constraint in the *Embedding-Edge LP* is replaced by the energy constraint on the nodes

$$\sum_{B \in \mathcal{B}} x(B) E_B(u) \leq E(u) \ \forall u \in V,$$

where an empty sum is defined to be 0. The dual of the *Embedding-Edge LP* is: Minimize $D(L) = \sum_{u \in V} E(u) l(u)$ subject to

1. Constraints corresponding to each $x(B)$ in primal:

$$\sum_{u \in B} E_B(u) l(u) \geq 1, \ \forall B \tag{12}$$

2. Non-negativity constraints:

$$l(u) \geq 0, \ \forall u \in V. \tag{13}$$

The weight or cost of an embedding can be defined as

$$w_L(B) = \sum_{u \in B} E_B(u) l(u).$$

The *OptimalEmbedding(L)* is modified in the weight initialization and weight update. The weight initialization is done as $\omega_{s_i}(\theta_i) := E_{C,\theta_i}$ for source data and $\omega_u(\theta_i) := E_{C,\theta_i} + \sum_{\eta \in \Phi_\uparrow(\theta_i)} \omega_u(\eta)$ for other data. The weight update at u is now done as $\omega_u(\theta_i) := \omega_v(\theta_i) + E_{T,\theta_i} + E_{R,\theta_i}$ if $\omega_v(\theta_i) + E_{T,\theta_i} + E_{R,\theta_i} < \omega_u(\theta_i)$. After suitable modification, the primal-dual algorithm with the modified *OptimalEmbedding(L)* algorithm finds an ϵ-approximate solution.

In the *Node-Arc LP*, the capacity constraints are replaced by energy constraints at the nodes:

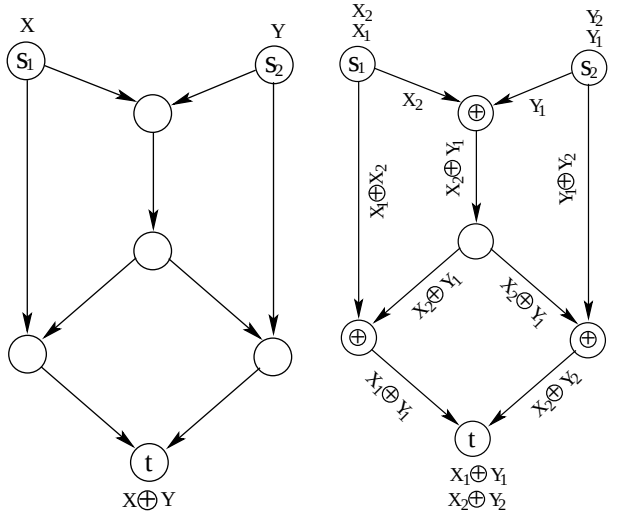
$$\sum_{\theta \in \Gamma} f^\theta_{uu} E_{C,\theta} + \sum_{\theta \in \Gamma} \sum_{v \in N(u)} (f^\theta_{uv} E_{T,\theta} + f^\theta_{vu} E_{R,\theta}) \leq E(u) \quad \forall u \in V.$$

V. Discussion and conclusion

In this paper, we have laid the foundations for network flow techniques for distributed function computation. Though we have obtained results for computation trees, we believe that much of our techniques can be extended to larger classes of functions, for instance, fast Fourier transform (FFT), that can be represented by more general graphical structures like directed acyclic graphs and hypergraphs where each edge or hyper-edge represents a distinct function of the sources. The sum function discussed in Sec. IV is one such function representable by a hypergraph.

Our computation framework does not allow block coding, i.e., coding across different realizations of the data. Such coding has been used in the information theory and network coding literature. Block coding can, in general, offer better computation rate. For example, consider the directed butterfly network as shown in Fig. 3 with two binary source nodes (with source processes denoted by X and Y) and a terminal node with a XOR target function $\Theta(X, Y) = X \oplus Y$. It can be checked that the maximum rate achievable by routing-like schemes, i.e., without using inter-realization coding, is 1.5. On the other hand, the scheme shown in Fig. 3(b) using inter-realization coding achieves a rate of 2. However, for more general functions, finding the optimal rate and designing optimal coding schemes is a difficult problem under this framework. Further, for undirected multicast networks, it is known that the inter-realization coding can achieve a rate strictly less than twice the rate achieved by routing [25]. We expect that similar results will hold for function computation over undirected networks.

Altogether, we believe that results in this paper opens many new avenues for further research.



(a) The butterfly network. Each edge has capacity 1 bit/use

(b) A rate-2 solution using cross-realization coding

Fig. 3. The butterfly network with XOR target function $\Theta(X, Y) = X \oplus Y$

VI. Acknowledgement

The authors would like to thank A. Diwan for fruitful discussions. This work was supported in part by Bharti Centre for Communication at IIT Bombay and a project from the Department of Science and Technology (DST), India.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows*. Prentic Hall Inc, 1993.
[2] N. Garg and J. Konemann, "Faster and simpler algorithms for multicommodity flow and other fractional packing problems," In Proc. FOCS, 1998.
[3] T. Leighton, F. Makedon, S. Plotkin, C. Stein, S. Tragoudas, and E. Tardos, "Fast approximation algo- rithms for multicommodity flow problems," *J. Comput. System Sci.*, vol. 50, pp. 228–243, 1995.
[4] F. Shahrokhi and D. Matula, "The maximum concur- rent flow problem," *J. ACM,*, vol. 37, pp. 318334, 1990.
[5] R. G. Gallager, "Finding parity in simple broadcast networks," *IEEE Transactions on Information Theory*, vol. 34, pp. 176–180, 1988.
[6] E. Kushilevitz and Y. Mansour, "Computation in noisy radio networks," in *Proceedings of the 9th annual ACM-SIAM Symposium on Discrete Algorithms*, 1998, pp. 236–243.
[7] U. Feige and J. Kilian, "Finding or in noisy broadcast network," *Information Processing Letters*, vol. 73, no. 1–2, pp. 69–75, January 2000.
[8] A. Giridhar and P. R. Kumar, "Computing and communicating functions over sensor networks," *IEEE Journal on Selected Areas in Communications*, vol. 23, no. 4, pp. 755–764, April 2005.
[9] L. Ying, R. Srikant, and G. Dullerud, "Distributed symmetric function computation in noisy wireless sensor networks with binary data," in *Proc. of the 4th International Symposium on Modeling and Optimization in Mobile, Ad-Hoc and Wireless networks (WiOpt)*, April 2006, pp. 1–9.
[10] Y. Kanoria and D. Manjunath, "On distributed computation in noisy random planar networks," in *Proceedings of IEEE International Symposium on Information Theory*, Nice, France, June 2007.
[11] S. Kamath and D. Manjunath, "On distributed function computation in structure-free random networks," in *Proceedings of IEEE International Symposium on Information Theory*, Toronto, Canada, July 2008.
[12] J. Korner and K. Marton. How to encode the modulo-two sum of binary sources. *IEEE Trans. Inform. Theory*, 25(2):219–221, 1979.
[13] T. S. Han and K. Kobayashi. A dichotomy of functions $f(x, y)$ of correlated sources (x, y). *IEEE Trans. Inform. Theory*, 33(1):69–86, 1987.
[14] Alon Orlitsky and J. R. Roche. Coding for computing. *IEEE Trans. Inform. Theory*, 47(3):903–917, 2001.
[15] H. Feng, M. Effros, and S. A. Savari. Functional source coding for networks with receiver side information. In *Proceedings of the Allerton Conference on Communication, Control, and Computing*, September 2004.
[16] B. K. Rai and B. K. Dey, "Sum-networks: system of polynomial equations, reversibility, insufficiency of linear network coding, unachievability of coding capacity," *Submitted to IEEE Trans. Inform. Th., available at http://arxiv.org/abs/0906.0695*.
[17] R. Appuswamy, M. Franceschetti, N. Karamchandani, and K. Zeger, "Network coding for computing part i : Cut-set bounds," *Submitted to IEEE Trans. Inform. Th., available at http://arxiv.org/abs/0912.2820*.
[18] M. Langberg and A. Ramamoorthy, "Communicating the sum of sources in a 3-sources/3-terminals network," in *Proceedings of IEEE International Symposium on Information Theory*, (Seoul, Korea), 2009.
[19] F. T. Leighton, M. J. Newman, A. G. Ranade, and E. J. Schwabe, "Dynamic tree embeddings in butterflies and hypercubes," *SIAM Journal on Computing*, vol. 21, no. 4, pp. 639–654, 1992.
[20] O. Wohlmuth and F. Mayer-Lindenberg, "A method for them embedding of arbitrary communication topologies into configurable parallel computers," in *Proceedings of the 1998 ACM Symposium on Applied Computing*, 1998, pp. 569–574.
[21] V. Heun and E. W. Mayr, "Efficient dynamic embeddings of arbitrary binary trees into hypercubes," *Journal of Algorithms*, vol. 43, pp. 51–84, 2002.
[22] G. Karakostas, "Faster approximation schemes for fractional multicommodity flow problems," *ACM Trans. Algorithms*, vol. 4, 2008, pp. 1–17.
[23] S. Plotkin and D. Shmoys and E. Tardos, "Fast approximation algorithms for fractional packing and covering problems," *Math. Oper. Res.*, vol. 20, pp. 257–301, 1995.
[24] M. Saad and T. Terlaky and A. Vannelli and H. Zhang, "Packing trees in communication networks," *J. Comb. Optim.*, vol. 16, pp. 402–423, 2008.
[25] Z. Li and B. Li, "Network coding in undirected networks," Proc. 38th CISS, Princeton, NJ, Mar. 2004, pp. 257–262.

Appendix A
The protocol

We now outline a communication and computation protocol designed to receive the function at the terminal at a rate that is greater than $\sum_{B\in\mathcal{B}} x(B) - \epsilon$ for any given solution of the *Embedding-Edge LP*. First, the flow values $x(B)$ are rounded to lower rational numbers so that the total flow r is still greater than $\sum_{B\in\mathcal{B}} x(B) - \epsilon$. With abuse of notation, we use the same notation $x(B)$ to denote these rounded values of $x(B)$ in the rest of this subsection. All these flows are then multiplied by the least common multiple N of the denominators of the flows $x(B); B \in \mathcal{B}$. Let the resulting values be $n(B); B \in \mathcal{B}$.

Clearly $\sum_{B\in\mathcal{B}} n(B) = rN$. Let us fix an order in the embeddings $B_1, B_2, \ldots, B_{|\mathcal{B}|}$. The protocol consists of computation at the nodes and communication across the links in a block/frame of N consecutive uses of the network. In each frame, a link e can carry upto a total of $Nc(e)$ symbols in both directions. Our protocol will require sending integer number of symbols in N uses of e in each direction. We assume that this is possible as long as the total number of symbols transmitted in both directions is at most $Nc(e)$. We assume that computation at nodes is done instantaneously, and a frame sent across a link is available at the receiving node at the end of the frame. The receiving node can forward the data on another edge in the next frame or use it to compute something else for transmission in the next or later frames.

In our protocol, the data stream generated at each source is divided into blocks of rN symbols, and the terminal computes rN number of corresponding function values in each frame. Out of the rN computations, the first $n(B_1)$ are carried out using the embedding B_1, the next $n(B_2)$ are carried out using the embedding B_2, and so on. In each direction on each link, the transmissions corresponding to different embeddings are ordered in the same order as the embeddings. Further, if uv is in $B(\theta_i)$ as well as $B(\theta_j)$ (assume $i < j$ without loss of generality), then uv carries the data for (B, θ_i) first and then the data for (B, θ_j). Formally, in each frame and in each direction, a link uv in $\mathcal{N}$ carries a subframe, possibly empty, of data for each (B, θ) pair, where $B \in \mathcal{B}, \theta \in \Gamma$. These subframes are transmitted in the lexicographic order on (B, θ). Since the subframes for different (B, θ) may be available at u with different delay, these subframes will not correspond to the same frame of source data. In the following, we explicitly describe the subframes carried by uv in the k-th frame.

Let $\mathbf{y}^k_{B,\theta}$ denote the $n(B)$ symbols of data of type θ corresponding to the $n(B)$ symbols of source data in the k-th frame corresponding to the embedding B. That is, $\mathbf{y}^k_{B_1,\theta}$ denotes the $n(B_1)$ symbols of data of type θ corresponding to the first $n(B_1)$ symbols of source data in the k-th frame, $\mathbf{y}^k_{B_2,\theta}$ denotes the $n(B_2)$ symbols of data of type θ corresponding to the next $n(B_2)$ symbols of source data in the k-th frame, and so on. In each frame, uv carries a subframe of data for each (B, θ) pair. The subframe corresponding to (B, θ) is empty if $uv \notin B(\theta)$. Formally,

$$\mathbf{y}^k_{uv,B,\theta} = \begin{cases} \mathbf{y}^k_{B,\theta} \text{ if } uv \in B(\theta), \\ \emptyset \text{ otherwise.} \end{cases}$$

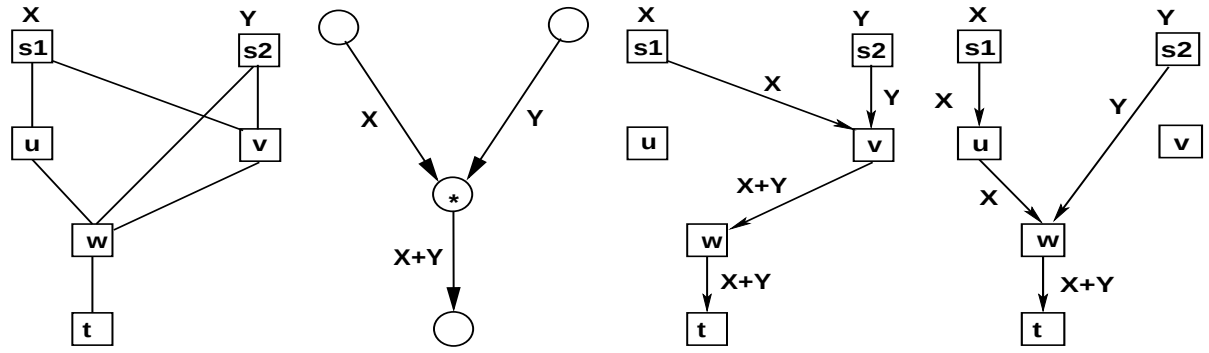
This subframe corresponds to the k-th block of source data. These subframes may be available at u with variable delay due to variable path lengths from the sources along different embeddings. Let us define the depth or delay $d(u, B, \theta)$ as

$$d(uv, B, \theta) = \begin{cases} \infty \text{ if } uv \notin B(\theta) \\ 0 \text{ if } uv \in B(\theta), u = s_i, \theta = \theta_i \\ 1 + \max\{d(wu, B, \eta) | \eta \in \Phi_\uparrow(\theta), wu \in B(\eta)\} \\ \quad \text{if } uv \in B(\theta), u = \text{start}(B(\theta)), (u, \theta) \neq (s_i, \theta_i) \\ d(wu, B, \theta) + 1 \text{ if } (u, \theta) \neq (s_i, \theta_i), wu, uv \in B(\theta). \end{cases} \tag{14}$$

So, the subframe $\mathbf{y}^k_{uv,B,\theta}$, which has $n(B)$ symbols if $uv \in B(\theta)$ and which corresponds to the k-the frame of source data, will be transmitted in the $(k + d(uv, B, \theta))$-th frame on uv. The infinite value for $uv \notin B(\theta)$ indicates that the corresponding data does not flow through uv from u to v.

Example: Consider the network and the computation tree shown in Fig. 4. The edges of the computation tree are labeled by the functions they carry, that is, X, Y, and $X+Y$. For embedding B_1, $d(s_1v, B_1, X) = 0$, $d(s_2v, B_1, Y) = 0$, $d(vw, B_1, X + Y) = 1$, $d(wt, B_1, X + Y) = 2$, and all other delay values are ∞. For embedding B_2, $d(s_1u, B_2, X) = 0$, $d(s_2w, B_2, Y) = 0$, $d(uw, B_2, X) = 1$, $d(wt, B_2, X + Y) = 2$, and all other delay values are ∞.

The data transmitted in the k-th frame from u to v on the link uv, in order of transmission, is thus $\mathbf{y}^{k-d(uv,B_1,\theta_1)}_{uv,B_1,\theta_1}, \mathbf{y}^{k-d(uv,B_1,\theta_2)}_{uv,B_1,\theta_2}, \ldots, \mathbf{y}^{k-d(uv,B_1,\theta_{|\Gamma|})}_{uv,B_1,\theta_{|\Gamma|}}, \mathbf{y}^{k-d(uv,B_2,\theta_1)}_{uv,B_2,\theta_1}, \mathbf{y}^{k-d(uv,B_2,\theta_2)}_{uv,B_2,\theta_2}, \ldots, \mathbf{y}^{k-d(uv,B_2,\theta_{|\Gamma|})}_{uv,B_2,\theta_{|\Gamma|}}, \ldots, \mathbf{y}^{k-d(uv,B_{|\mathcal{B}|},\theta_1)}_{uv,B_{|\mathcal{B}|},\theta_1}$,



(a) A network to compute $\Theta = X + Y$. (b) A computation tree for Θ. (c) A embedding B_1. (d) An embedding B_2.

Fig. 4. A network, a computation tree and two embeddings

$\mathbf{y}_{uv,B_{|\mathcal{B}|},\theta_2}^{k-d(uv,B_{|\mathcal{B}|},\theta_2)}, \ldots, \mathbf{y}_{uv,B_{|\mathcal{B}|},\theta_{|\Gamma|}}^{k-d(uv,B_{|\mathcal{B}|},\theta_{|\Gamma|})}$. It is easy to see that the required flow of function values will be computed on each embedding by this protocol. If the communication starts with the frame number 0 and ends with the K-th frame of source data, then the subframes are empty for $k < d(uv, B_i, \theta_j)$ and for $k > K + d(uv, B_i, \theta_j)$. In particular, a subframe $\mathbf{y}_{uv,B_i,\theta_j}^{k-d(uv,B_i,\theta_j)}$ is empty if $uv \notin B_i(\theta_j)$.

Example: In the above example, suppose a solution of the *Embedding-Edge LP* is $x(B_1) = 1$ and $x(B_2) = 0.5$. Then $N = 2$, and $n(B_1) = 2, n(B_2) = 1$. Each data stream is divided into frames of 3 symbols, out of which the first 2 symbols flow over B_1 and the last symbol flows over B_2. In the k-th frame, the link uw carries only one non-empty subframe for B_2 containing one 'X' symbol. That subframe $\mathbf{y}_{uw,B_2,X}^{k-1}$ corresponds to the last symbol of the $(k-1)$-th frame of data. The link wt carries one subframe of two '$X + Y$' symbols for B_1 and another subframe of one '$X + Y$' symbol for B_2. These subframes $\mathbf{y}_{wt,B_1,X+Y}^{k-2}$,$\mathbf{y}_{wt,B_2,X+Y}^{k-2}$ correspond to the first two symbols of the $(k-2)$-th data frame and the last symbol of the $(k-2)$-th data frame respectively.

To implement the protocol, any node u needs to know N, $n(B)$ for all embeddings with non-zero $n(B)$, and $d(uv, B, \theta)$ and $d(vu, B, \theta)$ for all such embeddings B, $\theta \in \Gamma, v \in N(u)$. The values of $d(uv, B, \theta)$ can be found in $O(nb|\Gamma|)$ time, where b is the number of embeddings for which $n(B) > 0$. In the following, we give the sequence of actions taken by any node u.

1. The node maintains an input queue for each (B, θ) pair for which $d(vu, B, \theta) < \infty$ for some $v \in N(u)$.

2. For the k-th frame received from v on the link vu, the node u knows the 'composition', i.e., how many symbols for which (B, θ) pair are received on that frame and in what order. This is because the frame contains a non-empty subframe corresponding to (B, θ) if and only if $d(vu, B, \theta) \leq k$. Such a non-empty frame contains exactly $n(B)$ symbols. The transmission of all the non-empty frames is ordered in the lexicographic ordering of (B, θ). For any received frame on any link, u puts each received subframe in its respective input queue. If u is a source, it also takes the rN generated symbols and creates the subframes of lengths $n(B)$ for all the relevant embeddings. Those are also placed in respective queues.

3. After queueing all the received and generated data in the k-th frame, u prepares the data to be transmitted on each link uv in the next, that is $(k+1)$-th, frame of N transmissions. The non-empty subframes for this transmitted frame are those for which $d(uv, B, \theta) \leq k + 1$. If there is an input queue for (B, θ), i.e., if such a data subframe is received at u, then this subframe of $n(B)$ symbols is taken from the respective input queue. Otherwise, this subframe is generated from the subframes from the queues for $(B, \eta); \eta \in \Phi_{\uparrow}(\theta)$. If such a queue for (B, η) contains multiple subframes of $n(B)$ symbols, then the oldest of them is taken. For instance, in our example (Fig. 4), for constructing the subframe $\mathbf{y}_{wt,B_2,X+Y}^{k}$ at w for the k-th frame, w takes a subframe from its input queue (B_2, X) and a subframe from the input queue (B_2, Y) and adds them. At this time, in the first queue, there is only one subframe $\mathbf{y}_{uw,B_2,X}^{k-2}$ which is used now. But in the second queue, there are two subframes $\mathbf{y}_{vw,B_2,Y}^{k-1}$ and $\mathbf{y}_{vw,B_2,Y}^{k-2}$ available, out of

which the older subframe $\mathbf{y}_{vw,B_2,Y}^{k-2}$ is used.