UNIVERSITY OF TARTU

Institute of Computer Science
Computer Science Curriculum

Janar Jõgi

Function Computation in Networks

Bachelor's Thesis (9 ECTS)

Supervisor: Vitaly Skachek, PhD

Tartu 2017

Funktsiooni arvutus võrkudes

Lühikokkuvõte: Ruutimine, mis kasutab ainuüht parimat teekonda sõnumite edastamiseks, on praegusel hetkel peamine meetod informatsiooni edastamiseks võrgus. Väljapakutud alternatiiviks on võrgukodeerimine, mis lubab kogu võrgul osaleda informatsiooni edastamises, saates kodeeritud infot läbi mitme teekonna ja taastades algse sõnumi vastuvõtjas. Mõningate rakenduste korral on algsete sõnumite taastamise asemel vaja funktsiooni üle nende sõnumite. Nimetame seda funktsiooni arvutuseks võrgus. Selline lähenemine lubab arvutusi teha teekonna jooksul, mil sõnum liigub allikatest saajateni. See töötab hästi näiteks võrkudes, kus ühendatud on palju piiratud arvutusvõimsusega väikseid seadmeid. Situatsioon, mis IoT esiletõusuga ilmneb aina tihedamini. Kuna funktsiooni arvutus võrkudes on suhteliselt uus mõiste, ei ole veel täiesti suudetud mõista võrgu funktsionaalarvutuse rakendatavust ja teoreetilise jõudlikkuse piire. Käesolev töö keskendub kindlale sihtfunktsioonide perekonnale ja tuvastab võrgu omadusi, et funktsionaalarvutus oleks edukas. See töö esitab kodeerimislahendusi, mis lubavad edukalt võrgus funktsionaalarvutusi läbi viia, kus sõnumiteks on üksikud sümbolid. Tulemused on seejärel laiendatud suvalise sümbolite arvuga sõnumitele, kasutades sarnast kodeerimislahendust.

Võtmesõnad: Funktsionaalarvutus võrgus, kodeerimine võrgus, andmevõrgud, sensorvõrgud, ruutimine

CERCS: P175

Function Computation in Networks

Abstract: Routing, that uses a single best path in the network, is currently the primary method for information transfer in networks. A proposed alternative to routing is called network coding that allows for the whole network to participate in the transmission of information by sending the coded data using multiple paths and then reconstructing the original message at the receiver. In some applications instead of reconstructing the original messages a function of those messages needs to be obtained. The corresponding problem is called a problem of function computation in the network. This approach allows for efficient en-route computing that works especially well with many small connected devices with limited computational capacities, a situation that appears often with the rise of the IoT. Since network function computation is a relatively new concept, the applicability and theoretical performance limits of this approach are not yet fully understood. The current work focuses on a certain family of target functions and identifies properties a network must have for function computation to be feasible. We propose encoding solutions that allow for successful network function computation. The results are then extended to packets with arbitrary number of symbols using a similar encoding scheme.

Keywords: Network function computation, network coding, data networks, sensor networks, routing

CERCS: P175

Contents

1 Introduction

Current dominating strategy in networking is breaking the source messages apart into packets, and then forwarding packets using routing, which finds the single best path per packet. Network coding provides an alternative to this by finding a way to encode source messages into packets and send them along multiple paths in the network. The data is processed in the nodes, but the entire network is used in information transmission and fewer transmissions are required to transmit data. This way one can better utilize the network and achieve optimal transmission rate especially near maximum traffic volume, called network capacity.

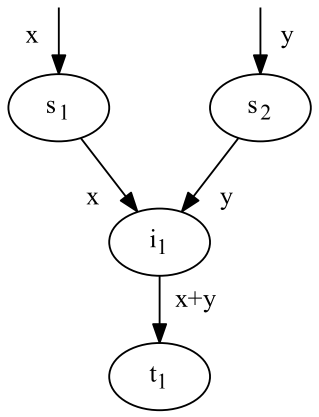


Figure 1: An example of network computation at vertex i_1 with two sources s_1 and s_2, one sink t_1 and a target sum function.

The problem studied in this work is related to network coding. It is called network computation. Network coding can be viewed as a special case of network computation where the function to be computed is the identity function. In function computation it is not needed to replicate the source messages at the receivers, instead a function of those messages needs to be computed. For example, given a wireless sensor network in a greenhouse, we may want to find a general function of the source messages such as finding the average temperature. We may want to find a function of the messages from the sources (wireless sensors) to find the average temperature in the greenhouse. Clearly this problem can be solved with routing by simply sending sensor data directly to the computing node and processing the data there. However, making use of the wireless sensor network and their computing power we can instead let the network compute the function and achieve the same result, saving the communication bandwidth.

As it is shown on Figure 1, in case of routing, two separate transmissions are needed for the sink node t_1 to first receive the inputs and then compute the target function of x_1 and x_2, assuming the edges can only carry one packet at a time. Network function computing can improve this by computing the function at i_1 and forwarding the result to the sink node. In the end we have only required a single transmission to compute the target function instead of two.

In this work, we consider several problems related to function computation in the network. The first problem is finding a set of conditions when a target function, in any network with unit-capacities and individual symbols as messages, will be computable regardless of the number of sources and receivers. This builds upon the work done in [1] that limited itself to the sum function. The authors managed to show that for two sinks and n receivers or n receivers and two sinks, as long as the max-flow between the receivers and sources is more than or equal to one, then the function is computable. It is also shown in [1] that for three sources and three sinks scenarios, the previous does not apply. Our research will determine what missing property of the target function causes it to fail and find conditions for a function to be computable in any network regardless of the number of sources of receivers.

We build upon the existing works, such as [2], that derives a simple upper bound for computing capacity for function computation in networks and then finds a variety of lower bounds for multiple classes of functions like divisible, symmetric, λ-exponential and λ-bounded functions. Previous work limits itself to networks with many sources but just one sink. While previous works have adopted a information theoretical approach then this work adopts a more algebraic, encoding oriented approach.

We start by explaining the basics of network flow followed by the explanation of the network model under consideration. Afterwards we define more specific concepts related to encoding, the act of computation in networks with unit messages or messages with a specified integer length. Our first goal is to specify a set of properties that a function and a network must have in order for a function to be computable assuming unit capacities and individual symbols as messages. Then we find similar conditions for a case when vectors of symbols instead of individual symbols are used.

2 Related works

R. Koetter and M. Médard [3] provide an algebraic framework for network coding. They find both necessary and sufficient conditions for determining whether a set of connections is feasible for a general network coding problem.

A book by S. Even [4, p. 85-108] provides the basics of network flow and provides some definitions and groundwork used in this work.

This thesis continues the line of work started by R. Appuswamy et al [2]. The authors define the notion of computational capacity, the maximal information rate which allows for computation of the given function of the receiver node inputs. Given multiple sources of data, a single sink node and a target function, the authors derive upper and lower bounds for computational capacity under these conditions. First they find a simple upper bound for the computational capacity for any target function. Then they consider 4 different classes of target functions: divisible, symmetric, λ-exponential and λ-bounded functions and find lower bounds for all of them. They conclude that with the identity function the bounds are tight yet with arbitrary functions the bounds may not be tight. While our work does not involve finding computational capacities, the classification of functions and the analysis of the effects their properties have on network coding, is a major part of this work.

This work is closely related to the work done by A. Ramamoorthy and M. Langberg in [1] which deals with simpler cases where the sum of source symbols is computed over a network. The edges have unit capacity and carry only one symbol. Authors limit themselves to two sources and n sinks, n sources and two sinks and three sources and three sinks scenarios. They derive necessary and sufficient conditions for feasibility of connections when the sum is computed in a network with two sources or two sinks. They also show the insufficiency of these previous conditions when the network has three sources and three sinks which exemplifies the complexity of network function computation. We look deeper into the reasons why computations in networks can fail and seek to further analyse various networks that allow for successful computation of certain classes of target functions.

The work [6] analyses the prospects of function computation in reverse butterfly networks. The authors find various computing and routing capacities for network computation in those specific types of networks.

Another paper related to sum networks is [7] where the authors construct a large family of sum networks for which computational capacity can be determined. Their work follows from [8] and generalizes their results and answers affirmatively that smaller sum networks with certain specified capacity can exists.

The notion of network computation has been associated with the rise of IoT as seen from [9]. As the number of small connected devices increases so does the need for more efficient flow of information in networks that does not overwhelm existing network infrastructure. The authors proposed an architecture that uses network function computation to perform en-route data proccessing using existing communication infrastructure instead of simply transmitting it to a central hub such as a cloud, for processing.

3 Model and definitions

3.1 Basics of network flow

We first define a directed graph $G(V, E)$ as a structure that consists of a set of vertices V and a set of edges E. We assume that both V and E are finite. Edges or directed edges are an ordered pair of vertices denoted as (v_1, v_2) or $v_1 \to v_2$, where the first and the second component of the pair denote the starting and ending point of the edge respectively.

We now define the notion of network flow. The following definition is taken from S. Even [4, p. 86-87]: Consider a directed graph $G(V, E)$ with no self-loops or parallel edges with two vertices s and t called source and sink, respectively. Let $\varepsilon_i(v)$ and $\varepsilon_o(v)$, $\forall v \in V$, denote the set of in-edges and out-edges originating from vertex v, respectively. We are given a capacity function, $c : E \to R^+$. A positive real number $c(e)$ is called the capacity of edge e. A flow function $f : E \to R$ is the assignment of a real number $f(e)$ to every edge e such that the following holds:

$$\forall e \in E \;\; 0 \leq f(e) \leq c(e) \tag{1}$$

$$\forall v \in V/\{s, t\} \;\; \sum_{e \in \varepsilon_i(v)} f(e) = \sum_{e \in \varepsilon_o(v)} f(e). \tag{2}$$

A total flow between s and t is defined as $F = \sum_{e\in\varepsilon_i(t)} f(e) - \sum_{e\in\varepsilon_o(t)} f(e)$. Take two sets $S \subset V$ and $\bar{S} = V \setminus S$ where $s \in S$ and $t \in \bar{S}$. May $(S, \bar{S})$ denote a set of edges that are directed from a vertex in S to a vertex in $\bar{S}$, called a forward cut and similarly we denote $(\bar{S}, S)$ as a backwards cut. A cut is defined as $(S, \bar{S}) \cup (\bar{S}, S)$. Capacity of the cut is $c(S)$ which is defined as follows:

$$c(S) = \sum_{e \in (S, \bar{S})} c(e) \tag{3}$$

The relationship between a cut and the total flow between a sink and a source node gives rise to the following theorem presented in [4, p. 87, Corollary 5.1].

Max-flow min-cut theorem. *For every flow function f, with total flow F, and $\forall s \in S \subset (V \setminus \{t\})$, if $F = c(S)$ then F is maximum and the capacity of cut S is minimum.*

The maximum flow between a source and a sink node has an integer value. It can be shown that in the network with integer edge capacities, there exists a maximum flow with integer flow in each edge. The previous theorem allows us to then assume the existence of at least one path in the network with the specified capacity between a source and a sink node.

3.2 Flow network model

We define a k-symbol flow network $N(G, S, P)$ as a connected directed acyclic graph $G = G(V, E)$ which contains a set of source nodes $S = \{s_1, ..., s_n\} \subset V$ and a set of sink nodes $P = \{p_1, ..., p_m\} \subset V$. We assume that $S \bigcap P = \emptyset$ and that the source nodes have no in-edges. In addition, all intermediate vertices with no in-edges, are omitted from the network. All edges in the network have a capacity of one. For all $e \in E$, let $tail(e)$ denote the starting vertex and $head(e)$ the ending vertex for the directed edge e, $v_i \to v_j$.

We call $N'(G'(V', E'), S, P)$ a subnetwork of $N(G(V, E), S, P)$ if $V' \subseteq V$ and $E' \subseteq E$ and $G'(V', E')$ is a graph.

Definition 3.1 (Tree network)**.** *We define a tree network as a flow network where between any source and sink node pair (s_i, t_j), a single directed path must exist starting from s_i and ending in t_j.*

3.3 Computation in networks

We define the length of a vector as the number of its components. Each source node has a vector of length k, composed of symbols from the alphabet A, associated with it. Let $X^k = \{\mathbf{x}^{s_1}, ..., \mathbf{x}^{s_n}\}$ denote the set of vector inputs in source nodes $s_1, ..., s_n \in S$. These inputs are also called message vectors. Let x_i denote the i-th component of the vector $\mathbf{x}$. If the source node inputs are vectors with one component then we refer to them by their component symbols with subscript denoting the source node they originate from $X = \{x_{s_1}, ..., x_{s_l}\}$.

We assume that one edge can carry one vector of length k. We define an encoding function $h(e)$ whose image is the k-length vector over alphabet A, being carried by an edge e, $\forall e \in E$, $v = tail(e)$:

$$h(e) : \begin{cases} \prod_{e' \in \varepsilon_i(v)} A^k \to A^k & v \notin S \\ A^k \to A^k & v \in S \end{cases}$$

An important concept used in this thesis is maximum flow from source nodes to the sink nodes. May max-flow(s_i — t_j) denote the maximum-flow between all source and sink node pairs (s_i, t_j). Since we made the assumption that all edges have capacity of one and can carry one vector then max-flow(s_i — t_j) ≥ 1 would imply the existence of a minimum cut with capacity greater than or equal to one, between any source and sink node. This implies we have at least one edge-disjoint path between all of the source and sink node pairs.

We define a family of target functions as $\bar{f}_r$: $A^r \to A$ for each $r \geq 1$. For notational clarity we no longer specify r and infer it from the number of arguments.

For further notational convenience, we define a function $g_{\bar{f}} : \prod_k A^k \rightarrow A^k$ that is applied to a sequence of n k-length vectors $\mathbf{x}^1, \mathbf{x}^2, ..., \mathbf{x}^n$, such that:

$$g_{\bar{f}}(\mathbf{x}^1, ..., \mathbf{x}^n) = (\bar{f}(x_1^1, ..., x_1^n), \bar{f}(x_2^1, ..., x_2^n), ..., \bar{f}(x_k^1, ..., x_k^n)). \tag{4}$$

This mapping allows us to apply the target function to appropriate components of vectors and makes dealing with message vectors in k-vector networks much easier.

We say that a target function $\bar{f}$ in a network $N(G, S, P)$ is computable at the vertex u if u can compute a vector $g_{\bar{f}}(\mathbf{x}^{s_1}, ..., \mathbf{x}^{s_n})$, $\mathbf{x}^{s_1}, ..., \mathbf{x}^{s_n} \in X$ from the messages it obtains. In the case of unit capacity networks the previous simplifies to u receiving $\bar{f}(x_{s_1}, ..., x_{s_n})$.

We say that a target function $\bar{f}$ is computable in the network $N(G, S, P)$ if it is computable at all the sink nodes.

Definition 3.1 (Binary divisibility). *Let $\bar{f}$: $A^n \rightarrow A$ be a family of target functions with arguments $x_1, ..., x_n \in A$ where $n \geq 2$. Let $S = \{i_1, ..., i_k\} \subset \{1, 2, ..., n\}$ be a subset where $i_1 \leq i_2 \leq ... \leq i_k$. Let $\mathbf{x}_S$ denote a vector $(x_{i_1}, ..., x_{i_k})$. We say that the target function $\bar{f}$ is binary divisible to a function f if for $|S| \geq 2$ there exists a two argument function called binary function $f : A \times A \rightarrow A$, that for some partition $S = S_1 \cup S_2$, $S_1 \cap S_2 = \emptyset$*

$$\bar{f}(\mathbf{x}_S) = f(\bar{f}_1(\mathbf{x}_{S_1}), \bar{f}_2(\mathbf{x}_{S_2})) \tag{5}$$

where $\bar{f}_1$, $\bar{f}_2$ belong to a family $\bar{f}$ and $\bar{f}_j(\mathbf{x}_{S_j})$ is binary divisible to f or $|\mathbf{x}_{S_j}| = 1$.

Definition 3.2 (Associativity). *A binary function $f : A \times A \rightarrow A$ is associative when $f(x, f(y, z)) = f(f(x, y), z), \forall x, y, z \in A$.*

Definition 3.3 (Commutativity). *A binary function $f : A \times A \rightarrow A$ is commutative when $f(x, y) = f(y, x), \forall x, y, z \in A$.*

Definition 3.4 (Idempotence). *A binary function $f : A \times A \rightarrow A$ is idempotent when, $\forall x \in A,\ \ f(x, x) = x$.*

Lemma 3.5. *If we assume that $\bar{f}$ is binary divisible and f is associative then from* (5)

$$\begin{aligned}\bar{f}(x_1, ..., x_n) &= f(f(...f(f(x_1, x_2), x_3)...), x_n) = \\ &= f(f(...f(x_1, f(x_2, x_3))...), x_n) = ... = f(x_1, f(...f(x_{n-2}, f(x_{n-1}, x_n))))\end{aligned} \tag{6}$$

In fact since for associative functions order of computation is irrelevant then it is easy to see that this can be generalized as follows:

Lemma 3.6. *If $\bar{f}$ is binary divisible to f that is associative then*

$$\bar{f}(x_1, ..., x_i, \bar{f}(y_1, ..., y_m), x_{i+1}, ..., x_n) = \bar{f}(x_1, ..., x_i, y_1, ..., y_m, x_{i+1}, ..., x_n) \tag{7}$$

The proof of this lemma is straightforward. We can freely choose the order of computation of f and still compute the same function. For example, we can define an arithmetic sum function as a binary function $f(x_1, x_2) = x_1 + x_2$, and since sum is associative, we can define a n-argument function as $\bar{f}(x_1, ..., x_n) = f(f((...f(f(x_1, x_2), x_3)...), x_{n-1}), x_n)$. If in addition we assume that f is commutative then we can change the order of the arguments as well. A familiar property in network computation known as symmetry:

Lemma 3.7. *If $\bar{f}$ is binary divisible to f that is associative and commutative then*

$$\bar{f}(x_1, ..., x_i, x_j, ..., x_n) = \bar{f}(x_1, ..., x_j, x_i, ..., x_n) \tag{8}$$

Therefore, if our target function is divisible into an associative and commutative binary function then the target function is symmetric.

It is straightforward to verify that the following derives from (7) and (8).

Lemma 3.8. *If $\bar{f}$ is binary divisible to f that is associative, commutative and idempotent then if $x_i = x_j$*

$$\bar{f}(x_1, ..., x_i, ..., x_j, ..., x_n) = \bar{f}(x_1, ..., x_i, ..., x_n) = \bar{f}(x_1, ..., x_j, ..., x_n) \tag{9}$$

The following lemmas are useful when dealing with k-length vector messages. For fixed k-length vectors $\mathbf{x}^1, ..., \mathbf{x}^n \in A^k$ and $\mathbf{y}^1, ..., \mathbf{y}^m \in A^k$.

Lemma 3.9. *If $\bar{f}$ is binary divisible to f that is associative then*

$$\begin{aligned} &g_{\bar{f}}(\mathbf{x}^1, ..., \mathbf{x}^i, g_{\bar{f}}(\mathbf{y}^1, ..., \mathbf{y}^m), \mathbf{x}^{i+1} ..., \mathbf{x}^n) \\ &\qquad = g_{\bar{f}}(\mathbf{x}^1, ..., x^i, \mathbf{y}^1, ..., \mathbf{y}^m, x^{i+1} ..., \mathbf{x}^n) \end{aligned} \tag{10}$$

Proof:

$$\begin{aligned} &g_{\bar{f}}(\mathbf{x}^1, ..., \mathbf{x}^i, g_{\bar{f}}(\mathbf{y}^1, ..., \mathbf{y}^m), \mathbf{x}^{i+1} ..., \mathbf{x}^n) \\ &\overset{(4)}{=} g_{\bar{f}}(\mathbf{x}^1, ..., \mathbf{x}^i, (\bar{f}(y_1^1, ..., y_1^m), ..., \bar{f}(y_k^1, ..., y_k^m)), \mathbf{x}^{i+1} ..., \mathbf{x}^n) \\ &\overset{(4)}{=} (\bar{f}(x_1^1, ..., x_1^i, \bar{f}(y_1^1, ..., y_1^m), x_1^{i+1}, ..., x_1^n), \\ &\quad \bar{f}(x_2^1, ..., x_2^i, \bar{f}(y_2^1, ..., y_2^m), x_2^{i+1}, ..., x_2^n), ... \\ &\quad , \bar{f}(x_k^1, ..., x_k^i, \bar{f}(y_k^1, ..., y_k^m), x_k^{i+1}, ..., x_k^n)) \\ &\overset{(7)}{=} (\bar{f}(x_1^1, ..., x_1^i, y_1^1, ..., y_1^m, x_1^{i+1}, ..., x_1^n), \\ &\quad \bar{f}(x_2^1, ..., x_2^i, y_2^1, ..., y_2^m, x_2^{i+1}, ..., x_2^n), ... \\ &\quad , \bar{f}(x_k^1, ..., x_k^i, y_k^1, ..., y_k^m, x_k^{i+1}, ..., x_k^n)) \\ &\overset{(4)}{=} g_{\bar{f}}(\mathbf{x}^1, ..., \mathbf{x}^i, \mathbf{y}^1, ..., \mathbf{y}^m, \mathbf{x}^{i+1} ..., \mathbf{x}^n) \quad \square \end{aligned}$$

Lemma 3.10. *If $\bar{f}$ is binary divisible to f that is associative and commutative then*

$$g_{\bar{f}}(\mathbf{x}^1, ..., \mathbf{x}^i, \mathbf{x}^j, ..., \mathbf{x}^n) = g_{\bar{f}}(\mathbf{x}^1, ..., \mathbf{x}^j, \mathbf{x}^i, ..., \mathbf{x}^n) \tag{11}$$

Proof:

$$\begin{aligned} g_{\bar{f}}(\mathbf{x}^1, ..., \mathbf{x}^i, \mathbf{x}^j ..., \mathbf{x}^n) &\overset{(4)}{=} (\bar{f}(x_1^1, ..., x_1^i, x_1^j, ..., x_1^n), ..., \bar{f}(x_k^1, ..., x_k^i, x_k^j, ..., x_k^n)) \\ &\overset{(8)}{=} (\bar{f}(x_1^1, ..., x_1^j, x_1^i, ..., x_1^n), ..., \bar{f}(x_k^1, ..., x_k^j, x_k^i, ..., x_k^n)) \\ &\overset{(8)}{=} g_{\bar{f}}(\mathbf{x}^1, ..., \mathbf{x}^j, \mathbf{x}^i ..., \mathbf{x}^n) \ \square \end{aligned}$$

Once again the following corollary follows from the two previous lemmas.

Corollary 3.11. *If $\bar{f}$ is binary divisible to f that is associative, commutative and idempotent then if $\mathbf{x}^i = \mathbf{x}^j$*

$$g_{\bar{f}}(\mathbf{x}^1, ..., \mathbf{x}^i, ..., \mathbf{x}^j ..., \mathbf{x}^n) = g_{\bar{f}}(\mathbf{x}^1, ..., \mathbf{x}^i ..., \mathbf{x}^n) = g_{\bar{f}}(\mathbf{x}^1, ..., \mathbf{x}^j ..., \mathbf{x}^n) \tag{12}$$

4 Computability in one-symbol networks

In this section we assume that all source vector inputs are individual symbols from the alphabet A. The first theorem seeks to find a collection of properties neccessary for a function in order to be computable in any network regardless of the number of sources and sinks.

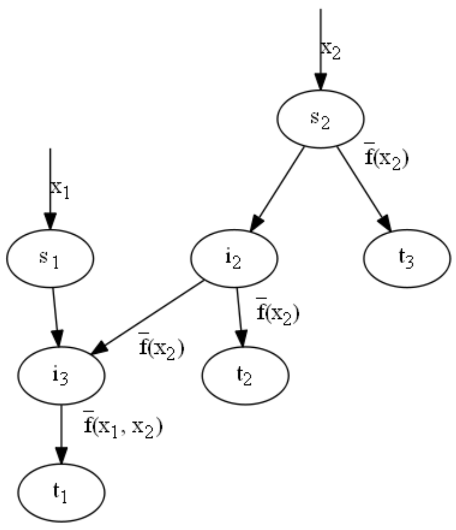


Figure 2: An example of the greedy encoding being applied in a network with two sources s_1, s_2 and three sinks t_1, t_2, t_3.

Definition 4.1 (Greedy encoding)**.** *Given an alphabet A and a series of inputs from source nodes $x_{s_1}, ..., x_{s_n} \in X \subseteq A$. $\forall e \in E$, let the encoding function $h(e)$ denote the symbol result of the computation at node $v = tail(e)$ being carried by an edge e from node v with respect to a target function $\bar{f}$ for a network $N(G, S, P)$:*

$$h(e) = \begin{cases} h(e_1) \;\; e_1 \in \varepsilon_i(v)) & v \notin S \text{ and } |\varepsilon_i(v)| = 1 \\ \bar{f}(h(e_1), h(e_2), ..., h(e_k)) \\ \qquad\qquad e_1, ..., e_k \in \varepsilon_i(v) & v \notin S \text{ and } |\varepsilon_i(v)| > 1 \\ \bar{f}(x_v) & v \in S \end{cases} \tag{13}$$

Every node in the network will encode over its inputs and forward the resulting code to neighboring nodes through all of its outgoing edges. The sink node will do processing of its inputs regardless of whether it has any outgoing edges or not. More formally, a sink node t returns a function of its inputs $\bar{f}(h(e_1), h(e_2), ..., h(e_k))$

$e_1, ..., e_k \in \varepsilon_i(t)$. The computation in the node and the resulting code on the edges is determined by the greedy encoding procedure. To summarize the above definition, if the parent node of the edge is not a source node but has only one in-edge then we forward the information from the previous node. If the parent node of the edge is not a source node but has more than one in-edge then we forward the target function of the parent node inputs. If the parent node is a source node then we simply forward the target function of the symbol itself. Figure 2 shows the application of greedy encoding to a network with all the three previously mention cases included. It should be noted that the max-flow condition is not satisfied for this network between all of the source and sink node pairs and therefore, only one sink node can compute the target function successfully.

Theorem 4.2. *In an acyclic one-symbol network $N(G, S, P)$ with alphabet A where $|S| = n, |P| = k$, $\forall i \in \{1, ..., n\}$, max-flow($s_i$ — t_j) ≥ 1 where the target function $\bar{f}$ binary divisible into an associative, commutative and idempotent binary function f, the target function is computable in the network.*

Proof: If there is only one source x_s and sink node t_j then according to the encoding: if they are adjacent then source node simply forwards the function of its symbol, if not then the intermediary symbols simply relay this information from the source node. In either case the sink node receives $\bar{f}(x_s)$ and function is computable. Now assume we have more than one source node. Let $x_{s_1}, ..., x_{s_n} \in X$ denote the source symbols. Since there must be more than one edge-disjoint path to this sink node the result of the computation at the receiving node t_j according to the encoding (13) must be:

$$\bar{f}(h(e_1), h(e_2), ..., h(e_k)), \;\; e_1, ..., e_k \in \varepsilon_i(t_j).$$

At any node other than a source node, we have an arbitrary number of incoming edges and for any of those edges, one of the following must apply:

1. it is an out-edge of a source node;

2. it is an out-edge of a node with one in-edge;

3. is it an out-edge of a node with many in-edges.

The output of the application of the encoding function on an edge will depend on which of the three aforementioned cases applies for that edge. We begin by proving transitions for all three different outputs of the encoding function:

1. Assume that edge e_i is the out-edge of a source node s_j:

$$\bar{f}(h(e_1), ..., h(e_i), ..., h(e_k)) \overset{(13)}{=} \bar{f}(h(e_1), ..., \bar{f}(x_{s_j}), ..., h(e_k)) \overset{(7)}{=} \bar{f}(h(e_1), ..., x_{s_j}, ..., h(e_k)) \quad (14)$$

2. Assume that edge e_i is the out-edge of a node that has one in-edge e':

$$\bar{f}(h(e_1),...,h(e_i),...,h(e_k)) \overset{(13)}{=} \bar{f}(h(e_1),...,h(e'),...,h(e_k)) \tag{15}$$

3. Finally assume that edge e_i is the out-edge of a node that has q in-edges $e'_1,...,e'_q$:

$$\begin{aligned}&\bar{f}(h(e_1),...,h(e_i),...,h(e_k))\\ &\quad\overset{(13)}{=} \bar{f}(h(e_1),...,\bar{f}(h(e'_1),...,h(e'_q)),...,h(e_k)) \overset{(7)}{=}\\ &\qquad\bar{f}(h(e_1),...,h(e'_1),...,h(e'_q),...,h(e_k))\end{aligned} \tag{16}$$

For every in-edge $e_1,...,e_k \in \varepsilon_i(t_j)$ we evaluate its encoding function and use one of the three previously proven transitions. Let us assume that t_j has d in-edges originating from source nodes, u in-edges originating from a node with one in-edge and an arbitrary amount of in-edges originating from nodes with many in-edges with each node having $h_1,...,h_b$ in-edges respectively. The expression will take the following form:

$$\begin{aligned}&\bar{f}(h(e_1),...,h(e_k)) =\\ &\quad\bar{f}(r_1,...,r_d,h(e'_1),...,h(e'_u),h(e''_1),...,h(e''_{h_1}),h(e'''_1),...,h(e'''_{h_2}),\\ &\qquad...,h(e''''_1),...,h(e''''_{h_b}))\end{aligned}$$

Since the only nodes without in-edges are source nodes then we continue to evaluate the encoding functions for edges and apply the appropriate transition until our expression contains only source input symbols. Because our network graph is finite and acyclic then the procedure must terminate by obtaining a function with a collection of source symbols as arguments:

$$\begin{aligned}&\bar{f}(h(e_1),...,h(e_k))\\ &\quad= \bar{f}(r_1,...,r_d,h(e'_1),...,h(e'_u),h(e''_1),...,h(e''_{h_1}),h(e'''_1),...,h(e'''_{h_2}),\\ &\qquad...,h(e''''_1),...,h(e''''_{h_b})) = ... = \bar{f}(r_1,r_2,...,r_l),\ \ r_1,r_2,...,r_l \in X\end{aligned}$$

By using associativity and commutativity of f we can rearrange the arguments in such a way that duplicate elements are removed by using that f is idempotent. By using properties (7) and (8), if $r_i = r_j$ then $\bar{f}(r_1,r_2,...,r_i,...,r_j,...,r_l) = \bar{f}(r_i,r_j,r_1,r_2,...,r_l) = \bar{f}(r_i,r_1,r_2,...,r_l)$. Therefore,

$$\bar{f}(r_1,r_2,...,r_l) = \bar{f}(x_{s_1},...x_{s_n})$$

and because sink t_j was arbitrary the proof is complete. □

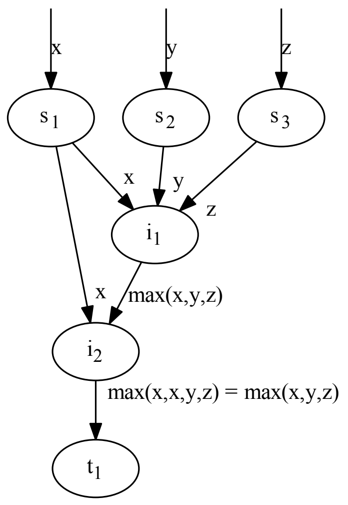


Figure 3: An example of computing a maximum function in a network with three sources and one sink.

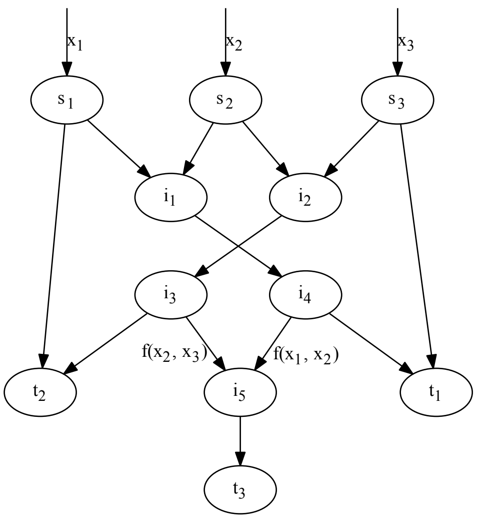


Figure 4: Graph similar to the one presented by A. Ramamoorthy and M. Langberg [1] to show the incomputability of the sum function even with max-flow condition satisfied.

An example of a target function that is binary divisible to an associative, commutative and idempotent binary function is the maximum function. Computing the maximum function in a network using greedy encoding is shown on Figure 3. As seen on the graph, source s_1 sends its message two times causing duplication. However, since maximum function is idempotent, the duplicate inputs will not interfere with the computation.

The previous proof well exemplifies the primary hurdles with function computation in networks. The relevance of the order of function computations can greatly limit the number of computable functions in a network, requiring very specific networks in order to work so that computations are done in a certain order. The associativity property eliminates the need for these highly specialized networks. Commutativity enables us to include all possible sequences of source messages. In addition, we may have to deal with multiples of the same symbol in the network if it gets sent out of multiple edges from a node which can interfere with our function computation. This interference is also represented in a counter-example for 3 source and 3 sink scenario by A. Ramamoorthy and M. Langberg [1]. A scenario loosely represented on Figure 4. The target function in this case is the sum and while the authors do not explicitly mention duplication as the root cause of incomputability, it is clear from the graph that it is indeed the case, since i_3 receives a duplicated message from s_2 through i_1 and i_2 which results in an incorrect computation. To eliminate this we need both associativity and commutativity in order to make use of the idempotence property or the possibility of messages multiplying in the network must be removed by some other method.

In the next theorem we state a sufficient condition for a target function to be computable without demanding idempotence. It will require a tree network which is a familiar concept for normal routing in networks, where spanning trees are desired to avoid duplication of packets. As we are about to show, such network graph structures can be useful in network function computation as well.

Theorem 4.3. *In a one-symbol tree network $N(G, S, P)$ with alphabet A, $|S| = n, |P| = k$, $\forall i \in \{1, ..., n\}$, max-flow($s_i$ — t_j) $= 1$ where the target function $\bar{f}$ is binary divisible to an associative and commutative function f, the target function is computable in the network.*

Proof: Let $x_{s_1}, ..., x_{s_n} \in X \subseteq A$ denote the source input symbols. The symbol computed at the sink node t_j according to the encoding (13) can be expressed as:

$$\bar{f}(h(e_1), h(e_2), ..., h(e_k)), \;\; e_1, ..., e_k \in \varepsilon_i(t_j)$$

Similarly to the proof of Theorem 4.2 we will use the encoding scheme and the binary divisibility of our target function to an associative binary function to go recursively upwards from the sink, making use of transitions (14), (15) and (16). After fully evaluating the expression, the computation at the sink node t_j will be a target function with source symbols as arguments:

$$\bar{f}(h(e_1), h(e_2), ..., h(e_k)) = \bar{f}(r_1, r_2, ..., r_l), \;\; l \geq k, \;\; r_1, ..., r_l \in X$$

We observe that the arguments must contain no multiples of source symbols. We do this by using our assumption that the network is a tree network meaning between any source and sink node pair, a single directed path must exist. The assumption was necessary since no indempotence is assumed in this theorem. Lets assume that we have a duplicate input in the function $r_i = r_j = x_{s_b}$. Then there must exist a vertex a, upstream from or equivalent to t_j, that encoded these duplicates originally (which cannot be a source node). Node a will encode $h(e) = \bar{f}(l_1, ..., r_i, ..., r_j, ..., l_q)), \;\; q = |\varepsilon_i(a)| \;\; e \in \varepsilon_o(a)$. Since this is the node that originally encoded these duplicates there must be two distinct in-edges e' *and* e'' to node a that carry $h(e') = \bar{f}(..., r_i, ...)$ and $h(e'') = \bar{f}(..., r_j, ...)$ respectively. Since $r_i = r_j = x_{s_b}$ then they must have originated from the same source node s_b. Therefore there must exist a path from s_b to $tail(e')$ and a path to $tail(e'')$. Now it's clear that there exists two different paths between nodes a and s_b once using the edge e' and second using the edge e''. Since a was upstream from t_j, meaning theres a path from a to t_j, then it causes a contradiction with the assumption that our network is a tree network. It is clear now that $\bar{f}(r_1, r_2, ..., r_l) = \bar{f}(x_{s_1}, ...x_{s_n})$ cannot contain multiples of the same source inputs and because sink t_j was arbitrary the proof is complete. □

Corollary 4.4. *The results of Theorem 4.3 hold even if the network contains a subnetwork that is a tree network, in this case we can ignore the edges not contained in the tree network.*

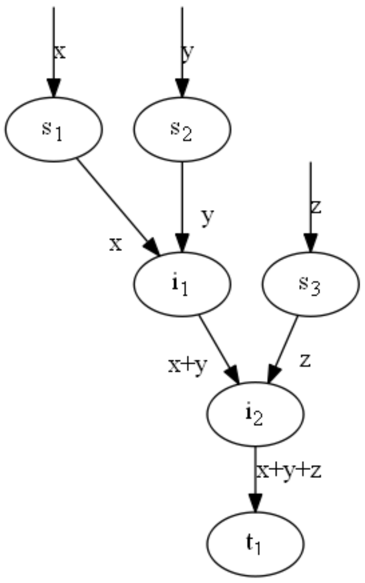


Figure 5: An example of computing a target sum function in a tree network with three sources s_1, s_2, s_3 with input symbols x, y, z and one sink t_1

An example of a successful network computation scenario is visualized on Figure 5 with the target function being the arithmetic sum which is binary divisible to an associative and commutative but not idempotent binary function. The structure of the graph prevents duplication of inputs x, y and z by eliminating multiple paths to the sink node t_1 instead of restricting the choice of a target function.

In the proof of Theorem 4.3 we do not explicitly use the commutativity property. Without the property, Theorem 4.3 would only apply to a specific sequence of source nodes, analysis of which fell outside the scope of this thesis.

As it is clear from the previous theorems, we must choose between two rather significant restrictions either for the target function or for the network, to ensure a successful network computation.

5 Computability in k-symbol networks

We now consider the case where sources carry vectors of symbols called message vectors. We show how the approach in Section 4 is extendable from single symbols to vectors. This case facilitates the requirement of a slightly different encoding scheme.

Definition 5.1 (Greedy vector encoding)**.** *Given an alphabet A and a series of inputs from source nodes $\mathbf{x}^{s_1}, ..., \mathbf{x}^{s_n} \in X^k$, $X \subseteq A$. $\forall e \in E$, let the encoding function $h(e)$ denote the vector result of the computation at node $v = tail(e)$ being carried by an edge e with respect to a target function $\bar{f}$ for a network $N(G, S, P)$:*

$$h(e) = \begin{cases} h(e_1), \;\; e_1 \in \varepsilon_i(v) & v \notin S \;\; and \;\; |\varepsilon_i(v)| = 1 \\ g_{\bar{f}}(h(e_1), ..., h(e_l)) \\ \quad e_1, ..., e_l \in \varepsilon_i(v) & v \notin S \;\; and \;\; |\varepsilon_i(v)| > 1 \\ g_{\bar{f}}(\mathbf{x}^v) & v \in S \end{cases} \tag{17}$$

Once again the sink node will do processing of its inputs regardless of whether it has any outgoing edges or not. A sink node t returns a function of its inputs $g_{\bar{f}}(h(e_1), ..., h(e_l))$, $e_1, ..., e_l \in \varepsilon_i(t)$. The new encoding solution functions similarly to the previous case (Section 4) but instead of individual symbols the edges now carry fixed k-length vectors. During the encoding process the target function will be greedily applied to to the i-th component of each incoming vector until a new k-length vector has been composed of the results and will be passed to the neighboring vertices. We can now extend Theorems 4.2 and 4.3 to deal with vector messages in the same manner as unit messages.

Theorem 5.2. *In an acyclic k-symbol network N(G,S,P) with alphabet A where $|S| = n, |P| = m$, $\forall i \in \{1, ..., n\}$, max-flow($s_i$ — t_j) ≥ 1 and the target function $\bar{f}$ binary divisible into an associative, commutative and idempotent binary function f then the target function is computable in the network.*

Proof: Let us assume the existence of a single source s_1 and an arbitrary sink node t_j then according to the encoding (17) and the max-flow condition: if they are adjacent then s_1 will encode and forward $g_{\bar{f}}(\mathbf{x}^{s_1}) = (\bar{f}(x_1^s), \bar{f}(x_2^{s_1}), ..., \bar{f}(x_k^s))$ otherwise the intermediary symbols simply relay this information to the sink from the source node. In either case the sink node receives $g_{\bar{f}}(\mathbf{x}^{s_1})$ hence the target function is computable.

Now lets assume we have more than one source node. Let $\mathbf{x}^{s_1}, ...\mathbf{x}^{s_n} \in X$ denote the source message vectors. Since there must be at least one edge-disjoint path to this sink node and because our network is finite and acyclic, then the greedy encoding procedure must terminate. Vector computed at the receiving node t_j according to the encoding must be $g_{\bar{f}}(h(e_1), ..., h(e_l))$, $e_1, ..., e_l \in \varepsilon_i(t_j)$. We now show that this is indeed equivalent to $g_{\bar{f}}(\mathbf{x}^{s_1}, ..., \mathbf{x}^{s_n})$.

Again we assume that at any node other than a source node, we have an arbitrary number of incoming edges and for any of those edges, one of the following must apply:

1. it is an out-edge of a source node;
2. it is an out-edge of a node with one in-edge;
3. is it an out-edge of a node with many in-edges.

The output of the application of the encoding function on an edge will depend on which of the three aforementioned cases applies for that edge. We begin by proving transitions for all three different outputs of the encoding function:

1. Assume that edge e_i is the out-edge of a source node s_j:
$$g_{\bar{f}}(h(e_1), ..., h(e_i), ..., h(e_l)) \overset{(17)}{=} g_{\bar{f}}(h(e_1), ..., g_{\bar{f}}(\mathbf{x}^{s_j}), ..., h(e_l)) \overset{(10)}{=} g_{\bar{f}}(h(e_1), ..., \mathbf{x}^{s_j}, ..., h(e_l)) \quad (18)$$
2. Assume that edge e_i is the out-edge of a node that has one in-edge e':
$$g_{\bar{f}}(h(e_1), ..., h(e_i), ..., h(e_l)) \overset{(17)}{=} g_{\bar{f}}(h(e_1), ..., h(e'), ..., h(e_l)) \quad (19)$$
3. Finally assume that edge e_i is the out-edge of a node that has r in-edges $e'_1, ..., e'_r$:
$$g_{\bar{f}}(h(e_1), ..., h(e_i), ..., h(e_l)) \overset{(17)}{=} g_{\bar{f}}(h(e_1), ..., g_{\bar{f}}(h(e'_1), ..., h(e'_r)), ..., h(e_l)) \overset{(10)}{=} g_{\bar{f}}(h(e_1), ..., h(e'_1), ..., h(e'_r), ..., h(e_l)) \quad (20)$$

For every in-edge $e_1, ..., e_l$ we evaluate its encoding function and use one of the three previously proven transitions. Lets assume t_j has d in-edges originating from source nodes, u in-edges originating from a node with one in-edge and an arbitrary amount of in-edges originating from nodes with many in-edges with each node having $h_1, ..., h_b$ in-edges respectively. The new expression will take the following form:

$$g_{\bar{f}}(h(e_1), ..., h(e_l)) = g_{\bar{f}}(\mathbf{r}^1, ..., \mathbf{r}^d, h(e'_1), ..., h(e'_u), h(e''_1), ..., h(e''_{h_1}), h(e'''_1), ..., h(e'''_{h_2}), ..., h(e''''_1), ..., h(e''''_{h_b}))$$

Since the only nodes without in-edges are source nodes then we continue to evaluate the encoding functions for edges and apply the appropriate transition until our expression contains only source vector inputs:

$$g_{\bar{f}}(h(e_1), ..., h(e_l)) = ... = g_{\bar{f}}(\mathbf{r}^1, ..., \mathbf{r}^q), \ \ \mathbf{r}^1, ..., \mathbf{r}^q \in X^k$$

Since multiple paths between any two nodes are possible then the expression may contain multiples of the same source input vector. However, since our target function is binary divisible to an associative, commutative and idempotent binary function then from Corollary 3.11:

$$g_{\bar{f}}(\mathbf{r}^1, ..., \mathbf{r}^q) = g_{\bar{f}}(\mathbf{x}^{s_1}, ..., \mathbf{x}^{s_n})$$

and because sink t_j is arbitrary, the proof is complete. □

Theorem 5.3. *In a k-symbol tree network $N(G, S, P)$ with alphabet A where $|S| = n, |P| = k$, $\forall i \in \{1, ..., n\}$, max-flow($s_i$ — p_j) $= 1$ and our target function $\bar{f}$ is binary divisible to an associative and commutative function then $\bar{f}$ is computable in the network.*

Proof: Let t_j be an arbitrary sink node in our network. Let $\mathbf{x}^{s_1}, ..., \mathbf{x}^{s_n} \in X$ denote the source message vectors. Once again we begin at the sink node p_j and identically to Theorem 5.2 we fully evaluate all the encoding functions and transform them using our previously proven transitions (18), (19) and (20). We obtain an expression that only includes the source vector inputs:

$$g_{\bar{f}}(h(e_1), ..., h(e_l)) = ... = g_{\bar{f}}(\mathbf{r}^1, ..., \mathbf{r}^q), \ \ \mathbf{r}^1, ..., \mathbf{r}^q \in X^k$$

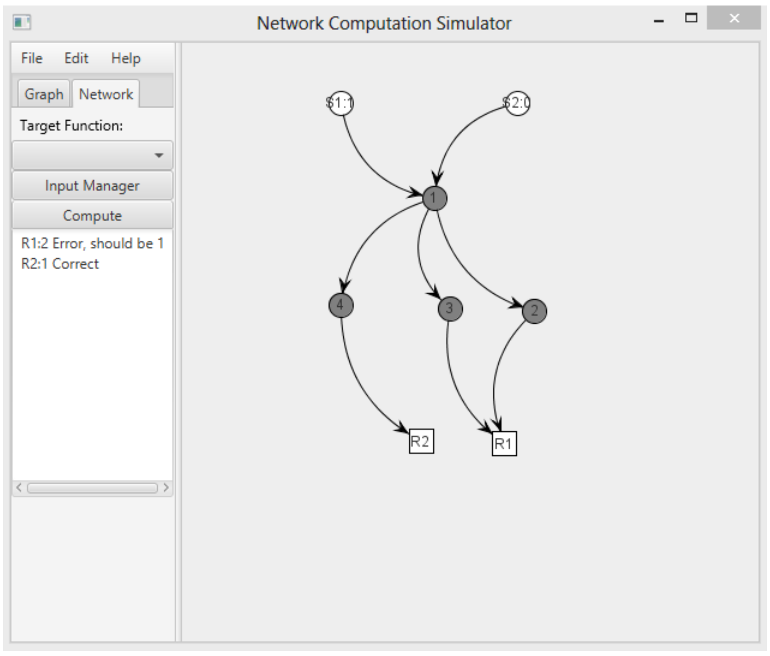
We must now verify that this expression does not contain duplicates of the same input vectors. The proof of this is very similar to the one presented in Theorem 4.3. Since the network is a tree network then between any source and sink node pair, a single directed path must exist. Let the duplicate vectors in our case be $\mathbf{r}^i = \mathbf{r}^j = \mathbf{x}^{s_b}$. There must exist vertex a, upstream from or equivalent to t_j, that was the first to encode these duplicates. Node a will encode $h(e) = g_{\bar{f}}(\mathbf{l}^1, ..., \mathbf{r}^i, ..., \mathbf{r}^j, ..., \mathbf{l}^q))$, $\ q = |\varepsilon_i(a)| \ e \in \varepsilon_o(a)$. Since this is the node that originally encoded these duplicates there must be two distinct in-edges e' *and* e'' to node a that carry $h(e') = g_{\bar{f}}(..., \mathbf{r}^i, ...)$ and $h(e'') = g_{\bar{f}}(..., \mathbf{r}^j, ...)$ respectively. Since $\mathbf{r}^i = \mathbf{r}^j = \mathbf{x}^{s_b}$ then they must have originated from the same source node s_b. Therefore there must exist a path from s_b to $tail(e')$ and a path to $tail(e'')$. Now it is clear that there exist two different paths between nodes a and s_b, one of them using the edge e' and the other using the edge e'', and since a was upstream from or equivalent to t_j, this is a contradiction with the assumption that the network is a tree network. Hence, no duplicates of the same source input vector can exist. □

Corollary 5.1. *The results of Theorem 5.3 hold even if the network contains a subnetwork that is a tree network, in this case we can ignore the edges not contained in the tree network.*

We have managed to take the original Theorems 4.2, 4.3 and successfully extended them to deal with messages of various length which make our results more applicable to packet based networks used today.

6 Software for visualization

Software for visualization and testing of network function computation was developed in Java, using JUNG framework for creating and manipulating graphs and JavaFX for developing the user interface. The software allows the creation of a network with an arbitrary number of sources and sinks. Source inputs can then be defined and an attempt to compute the arithmetic sum using network function computation is then made. Software provides feedback such as whether the computation was successful or not and what are the results at each of the sink nodes.



R1:2 Error, should be 1
R2:1 Correct

Figure 6: Network function computation simulator reporting a failed computation.

After the user creates a network using the graph creation tools, the software simulates the Greedy encoding procedure defined in Section 4. Software is limited to the arithmetic sum target function, uses integers as its alphabet and only uses individual symbols as messages instead of packets. Figure 6 shows a picture of the program after the user has set the source inputs using "Input Manager" and pressed "Compute" that runs the simulation of the Greedy encoding procedure. Feedback regarding the computation is provided in the text panel. The software was used to study the problem of computing functions in the network, to verify the results of Theorem 4.3 and to gain intuition about network computation in general.

7 Conclusions

We have proposed encoding solutions and examined the computability of binary divisible target functions in one-, and k-symbol networks where between all of the source and sink node pairs at least one path exists. We showed that for one-symbol networks our encoding solution will successfully compute the required function if the target function is binary divisible to an associative, commutative and idempotent function. We also found that by assuming that the network is a tree network the idempotence property will not be needed. A new encoding solution allowed for the previous theorems to be extended to k-symbol networks with fixed-length vector messages. While it is clear from this work that the total number of transmissions for network computation with the presented encoding can decrease, we did not rigorously examine the advantage the proposed encoding solution has over ordinary routing, nor did we examine the achievable transmission rates and as such they remain open questions.

References

[1] Ramamoorthy, A., & Langberg, M. (2013). Communicating the sum of sources over a network. IEEE Journal on Selected Areas in Communications, 31(4), 655-665.

[2] Appuswamy, R., Franceschetti, M., Karamchandani, N., & Zeger, K. (2011). Network coding for computing: Cut-set bounds. IEEE Transactions on Information Theory, 57(2), 1015-1030.

[3] Koetter, R., & Médard, M. (2003). An algebraic approach to network coding. IEEE/ACM Transactions on Networking, 11(5), 782-795.

[4] Even, S. (2011). Graph algorithms. Cambridge University Press.

[5] Giridhar, A., & Kumar, P. R. (2005). Computing and communicating functions over sensor networks. IEEE Journal on Selected Areas in Communications, 23(4), 755-764.

[6] Appuswamy, R., Franceschetti, M., Karamchandani, N., & Zeger, K. (2009, June). Network computing capacity for the reverse butterfly network. In ISIT (pp. 259-262).

[7] Tripathy, A., & Ramamoorthy, A. (2014, September). Sum-networks from undirected graphs: construction and capacity analysis. In Communication, Control, and Computing (Allerton), 52nd Annual Allerton Conference on (pp. 651-658).

[8] Rai, B. K., & Das, N. (2013, October). On the capacity of sum-networks. In Communication, Control, and Computing (Allerton), 51st Annual Allerton Conference on (pp. 1545-1552).

[9] Vukobratovic, D., Jakovetic, D., Skachek, V., Bajovic, D., Sejdinovic, D., Karabulut Kurt, G., & Fischer, I. (2016). CONDENSE: A reconfigurable knowledge acquisition architecture for future 5G IoT. IEEE Access, 4, 3360-3378.