COMPUTING BOOLEAN FUNCTIONS ON ANONYMOUS NETWORKS

Evangelos Kranakis [1] (eva@cwi.nl)
Danny Krizanc [1,2] (krizanc@cs.rochester.edu)
Jacob van den Berg [1] (jvdberg@cwi.nl)

(1) Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

(2) University of Rochester, Department of Computer Science
Rochester, New York, 14627, USA

Abstract

We study the bit-complexity of computing boolean functions on anonymous networks. Let N be the number of nodes, δ the diameter and d the maximal node degree of the network. For arbitrary, unlabeled networks we give a general algorithm of polynomial bit complexity $O(N^4 \cdot \delta \cdot d^2 \cdot \log N)$ for computing any boolean function which is computable on the network. This improves upon the previous best known algorithm which was of exponential bit complexity $O(d^{N^2})$. For symmetric functions on arbitrary networks we give an algorithm with bit complexity $O(N^2 \cdot \delta \cdot d^2 \cdot \log^2 N)$. This same algorithm is shown to have even lower bit complexity for a number of specific networks. We also consider the class of distance regular unlabeled networks and show that on such networks symmetric functions can be computed efficiently in $O(N \cdot \delta \cdot d \cdot \log N)$ bits.

1 Introduction

A very important problem in distributed computing is the designing of efficient algorithms for computing boolean functions in distributed networks of processors. For both practical and theoretical reasons it is useful to minimize the total number of exchanged bits which are necessary in order to compute a certain boolean function, but at the same time keeping the processors as similar to each other as possible.

A distributed network is a simple, connected graph consisting of nodes (vertices) on which the processors are located, and links (edges) along which the interprocess communication takes place. The processors are assumed to have unlimited computational power but may exchange messages only with their neighbors in the network. Initially, each processor is given an input bit, 0 or 1.

The processors follow a deterministic protocol (or algorithm). During each step of the protocol they perform certain computations depending on their input value, their previous history and the messages they receive from their neighbors and then transmit

the result of this computation to some or all of their neighbors. After a finite number of steps, predetermined by the initial conditions and the protocol, the processors terminate their computation and output a certain bit. Let B_N be the set of boolean functions on N variables. Let $\mathcal{N} = (V, E)$ be a network of size N, with node set $V = \{0, 1, \ldots, N-1\}$ and edge set $E \subseteq V \times V$. An input to $\mathcal{N}$ is an N-tuple $I =< b_v : v \in V >$ of bits $b_v \in \{0, 1\}$, where processor v receives as input value the bit b_v. Given a function $f \in B_N$ known to all the processors in the network we are interested in computing the value $f(I)$ on all inputs I. To compute f on input $I =< b_v : v \in V >$ each processor $v \in V$ starting with the input bit b_v should terminate its computation according to the given protocol and output the value b such that $f(I) = b$. A network computes the function f if for each input I, at the end of the computation each processor computes correctly the value $f(I)$. The bit complexity for computing f is the total number of bits exchanged during the computation of f. We are interested in providing algorithms that minimize the bit complexity of boolean functions.

We make the following assumptions regarding the networks and their processors:

1. the processors know the network topology and the size of the network (i.e. total number of processors),
2. the processors are anonymous (this means that they do not know either the identities of themselves or of the other processors),
3. the processors are identical (this means they all run the same algorithm),
4. the processors are deterministic,
5. the network is asynchronous,
6. the network is unlabeled (i.e., there is no global, consistent labeling (or orientation) of the network links).

Note that changing any of the above assumptions changes the computational capabilities and limitations of the model. If the size of the network is not known to the processors then it may not even be possible to compute any nonconstant function, e.g. in the ring [ASW85]. Angluin [Ang80] has shown that if the processors are anonymous and identical there is no algorithm for electing a leader. If we add randomization to the model it becomes possible to improve greatly the average and worst case bit complexity. In synchronous networks information can be gathered not only through message passing but also through the absence of communication during a particular time interval. Labeling the edges of a network can be shown to change the set of functions computable on the network (see [KK90]).

In the sequel we assume that N is the number of processors in a given anonymous network. The simplest topology considered in the study of the bit complexity of computing boolean functions is the ring e.g., [AAHK88], [ASW85], [AS88], [MW86], [PKR84]. It has been shown by [ASW85] that there is an algorithm for computing all boolean functions which are computable on the ring, with bit complexity $O(N^2)$. Moreover, this bit complexity is the same on both oriented and unlabeled rings. In addition, [MW86] show that any nonconstant function has bit complexity $\Omega(N \cdot \log N)$ on the ring, and also construct boolean functions with bit complexity $\Theta(N \cdot \log N)$ on the ring. For the

oriented torus [BB89] give an algorithm with bit complexity $O(N^{1.5})$, and construct non-constant functions with bit complexity $\Theta(N)$. The case of the unlabeled and the oriented hypercube network is studied in [KK90]. For general graphs [YK88] and [YK87] show that the message complexity of computing a boolean function on an arbitrary unlabeled network is $O(N^2 \cdot m)$, where m is the number of links of the network. However, these messages consist of trees of depth N^2 and fanout the corresponding degrees of the nodes of the network. For regular graphs of degree d this translates into an exponential $O(d^{N^2})$ bit complexity ($d = 4$ for the torus, and $d = \log N$ for the hypercube).

In the present paper we study the bit complexity for boolean functions on arbitrary unlabeled networks and on distance regular networks. We show in section 2 that for any unlabeled N-node network of maximal node valency d and diameter δ, every boolean function which is computable on the network can be computed in $O(N^4 \cdot \delta \cdot d^2 \cdot \log N)$ bits, thus significantly improving the previous $O(d^{N^2})$ upper bound of [YK87]. This bound can be significantly improved for computing symmetric functions. In section 3 we give an algorithm for computing symmetric functions with bit complexity $O(N^2 \cdot \delta \cdot d^2 \cdot \log^2 N)$. This same algorithm provides even more efficient algorithms when applied to specific networks. For the case of distance regular networks we show in section 4 how to compute any symmetric function in $O(N \cdot \delta \cdot d \cdot \log N)$ bits. We conclude in section 5 with some discussion and open problems.

2 Unlabeled Networks

In this section we give a general algorithm which computes any boolean function computable on a given network using polynomial bit complexity. One of the results that will be used very frequently in the sequel concerns the computation of certain simple operations, like maximum and set-union on general unlabeled networks. To facilitate and simplify our discussion and avoid unnecessary repetition we state our main algorithm for computing such functions as a separate theorem. First we need a few definitions.

Let $\Diamond$ be a commutative, associative and idempotent binary operation on a set A, i.e. $\Diamond : A \times A \longrightarrow A$ satisfies the following axioms for all $a, b, c \in A$,

- $\Diamond(a, b) = \Diamond(b, a)$ (commutativity),
- $\Diamond(a, \Diamond(b, c)) = \Diamond(\Diamond(a, b), c)$ (associativity),
- $\Diamond(a, a) = a$ (idempotency).

Such operations include maximum, minimum, set-union and set-intersection. For simplicity from now on we will abbreviate $\Diamond(a, b)$ by $a \Diamond b$.

Let $\mathcal{N} = (V, E)$ be an unlabeled network and let $\Diamond$ be an operation satisfying the above three conditions. Let A^N be the set of all N-tuples from elements of A. For any input $I =< i_p : p \in V >\in A^N$ to the network we can define a function $\Diamond : A^N \longrightarrow A$ by the following equation

$$\Diamond(I) = i_0 \Diamond i_1 \Diamond \cdots \Diamond i_{N-1}.$$

(By an abuse of notation we use the same symbol for the binary operation $\Diamond : A \times A \longrightarrow A$ and the function $\Diamond : A^N \longrightarrow A$.) In view of the associativity of $\Diamond$ this function is well defined. As a first step in our goal for providing an algorithm for computing all

(computable) boolean functions we will show that functions, like $\Diamond$, which arise from such binary operations give rise to computable functions.

Theorem 2.1 *Let $\mathcal{N}$ be an unlabeled network with maximal node valency d and diameter δ and let $\Diamond$ be a commutative, associative and idempotent binary operation. There is an algorithm for computing $\Diamond(I)$ for any input $I =< i_p : p \in V > \in A^N$ with bit complexity $O(N \cdot \alpha \cdot \delta \cdot d)$, where α denotes the number of bits necessary to represent an element of A.*

Proof. The idea of the algorithm is rather simple. Each processor sends its initial input value to all its neighbors. After receiving a value from its neighbors it applies the operation $\Diamond$ to the value it already has and the values it receives. Every processor executes these steps δ many times. Eventually every input value to a node of the network will be distributed and accounted for by every other processor. More formally the algorithm is as follows. Let $I =< i_p : p \in V >$ be the input to the network.

Algorithm for procesor p:
Initialize: $value_p[0] := i_p$;
for $i := 0, 1, \ldots, \delta - 1$ **do**
  send $value_p[i]$ to all neighbors of p;
  receive $value_q[i]$ from all neighbors q of p;
  compute $value_p[i+1] := \Diamond(\{value_p[i]\} \cup \{value_q[i] : q \text{ is a neighbor of } p\})$;
od;
output $value_p := value_p[\delta]$.

The proof of correctness of the algorithm is not difficult. By commutativity and associativity it is immaterial the order in which the operation $\Diamond$ is applied to the given values. It can happen that in the course of the execution of the above algorithm by processor p the operation $\Diamond$ is applied more than once to some element a, which is the initial input value to a certain processor q. The number of times $\Diamond$ is applied depends on the number of walks of length less than δ from p to q through the network. However because of the idempotency of the operation $\Diamond$ we have that $a \Diamond a \Diamond \cdots \Diamond a = a$. It follows that all processors will compute exactly the same value $\Diamond(I)$, namely $value_p = \Diamond(I)$, for all p.

It remains to determine the bit complexity of the algorithm. The processors receive through their neighbors elements of A, apply the operation $\Diamond$, create new elements of A and transmit them to their neighbors. The cost of transmitting each of these elements is α, the number of bits necessary to represent an element of A. Each of the N processors transmits a value to its d neighbors once in each of the δ phases of the above algorithm. This gives the desired bit complexity. □

An obvious corollary of the theorem concerns the bit complexity of the OR_N function. This is worth stating separately.

Corollary 2.1 *On an unlabeled N-node network with maximal node valency d and diameter δ the OR_N function can be computed with bit complexity $O(N \cdot \delta \cdot d)$.*

Proof. Apply theorem 2.1 to the operation of binary or, i.e. $a \Diamond b = a \vee b$. □

A simple extension of the lower bound for OR_N on the ring in [ASW85] shows that if the network is regular then OR_N requires $\Omega(N \cdot \delta)$ bits to compute. Thus for this case the above algorithm is optimal to within a factor of d.

Another corollary will be useful in the proof of our general theorem 2.2 about the bit complexity of computable boolean functions on general networks.

Corollary 2.2 *Let $\mathcal{N}$ be an unlabeled N-node network with maximal node valency d and diameter δ. There is an algorithm for computing the set $\{i_p : p \in V\}$ for any input $I =< i_p : p \in V >\in A^N$ with bit complexity $O(N^2 \cdot \alpha \cdot \delta \cdot d)$, where α denotes the number of bits necessary to represent an element of A.*

Proof. Here we apply the main theorem 2.1 to the binary operation union, $\Diamond(a, b) = a \cup b$ where the input to node p is the singleton set $\{i_p\}$. The elements transmitted in the course of the algorithm are subsets of the set $\{i_p : p \in V\}$. Each element can be coded with α bits, and therefore such sets can be coded with $N \cdot \alpha$ bits. □

We are now ready to give our algorithm for computing arbitrary boolean functions on a given unlabeled network. We will prove the following theorem.

Theorem 2.2 *Let $\mathcal{N}$ be an unlabeled N-node network with maximal node valency d and diameter δ. There is an algorithm that computes any boolean function which is computable on the network with bit complexity $O(N^4 \cdot \delta \cdot d^2 \cdot \log N)$.*

Proof. Our algorithm relies on several cost efficient adjustments and improvements of the algorithm of [YK88] using Theorem 2.1. Let $f \in B_N$ be any computable boolean function on the anonymous network $\mathcal{N}$. Let $I =< b_p : p \in V >$ be the input to the network, where b_p is the input to node p. We present the algorithm in three phases.

Phase 1. Each processor chooses an arbitrary labeling for all its incident edges, i.e., the links of p are labeled with the numbers $1, 2, \ldots, deg(p)$, where $deg(p)$ is the degree of p. Now each processor transmits to each neighbor the label it has chosen for the link connecting them. Let $\mathcal{L}$ be the resulting labeling of the network $\mathcal{N}$. Next, each pair (p, q) of processors labels their corresponding link with

$$l(p, q) = (\mathcal{L}(p, q), \mathcal{L}(q, p)).$$

The processors keep this labeling fixed throughout the algorithm. It should be pointed out that this is only a local labeling and not a global orientation of the network; the processors know only the labeling of their corresponding links, and are completely unaware of the choice of labeling by the other processors in the network.

Phase 2. In this phase each processor gathers as much information as possible from the rest of the processors about the input to the network in order to be able to compute correctly the value $f(I)$. Each processor p computes its *view*, $T_{\mathcal{L},I}(p)$ [YK87]. Since $\mathcal{L}$ and I are fixed below we will denote the view of p by T_p. This is a vertex and edge labeled tree of depth N^2. In a sense, each node p "unwraps" the network and forms a tree with itself as root. Since the network is anonymous it cannot use names for the processors, instead it can only label the vertices of the tree with the input bits it receives in the course of the interprocess communication. Thus, the root of T_p is labeled with the input bit b_p and the node corresponding to the node q is labeled with the bit b_q. However it needs to be stressed here that when the processors label a node with the bit b_q they do not necessarily know that the name of the processor they are labeling is q.

The processors need to exchange enough information in order to compute correctly each T_p. They do this by exchanging the views they have constructed. However, trees of depth i have exponential bit complexity $\Omega(d^i)$ and transmitting them is rather expensive. Therefore we must be careful if we want to achieve an algorithm with polynomial bit complexity. In the sequel we concentrate on the issue of coding and transmission of the trees concerned. Processor p computes a sequence of trees T_p^i of depth i, $i = 0, 1, \ldots, N^2$, by executing the following algorithm.

Algorithm for procesor p:
Initialize: $T_p^0 := b_p$ **and** $set_p^0 := \{T_p^0\}$;
for $i := 0, \ldots, N^2$ **do**
  compute the set $set_p^i := \{T_q^i : q \in V\}$;
  code the elements of the set set_p^i with integers $1, \ldots, k$, where $k \leq N$ is the number of elements of set_p^i, by ordering the set set_p^i lexicographically and letting $code(T_q^i) = j$, if T_q^i is the jth tree in this ordering;
  form the tree T_p^{i+1}; it is a tree of depth 1 with root labeled b_p;
  for each neighbor q of p there is an edge labeled $l(p, q)$; its leaves are labeled $code(T_q^i)$, where q is a neighbor of p;
  send the tree T_p^{i+1} to all the neighbors of p;
od;
output $set_p^{N^2}$.

After the trees of level i have been constructed the processors use the set algorithm given in corollary 2.2 to compute the set $\{T_p^i : p \in V\}$. Once all processors know all the trees of depth i there is no need to transmit to each other the decoded full trees themselves. It is sufficient to transmit the codes of the trees, and these can be just integers from 1 up to N. The processors themselves can decode the trees in order to generate the views. To code the trees the processors order them lexicographically and let the code of the tree T be j, if T is the jth tree in this ordering. The processors then form new trees of depth $i + 1$, namely T_p^{i+1}. The tree has a root which is labeled with p's input bit. The leaves of the tree consist of the above codes of the corresponding trees of depth i and the edges have the corresponding labeling. Now the processors transmit these new trees to all their neighbors, etc. As indicated above we iterate this algorithm N^2 times.

Phase 3. At this point all processors have computed the set of all views of depth N^2, namely the set $\{T_p^{N^2} : p \in V\}$. As in [YK87] we define an equivalence relation among trees. Two trees T and T' are equivalent if they are isomorphic including vertex and edge labels, but ignoring names of the vertices. By lemma 3.3 in [YK87] for any two trees if their restrictions to depth N^2 are isomorphic then the full trees themselves must also be isomorphic. Let $[T]_{I,\mathcal{L}}$ denote the equivalence class of T, where the subscript is to stress the dependence of the equivalence class on the input and the chosen labeling. It follows from the above discussion that each processor will be able to find representatives of all the equivalence classes of the full trees. Further, it follows from theorem 4.1 in [YK87] that since f is computable on the network its value depends only on the equivalence classes of the trees above, i.e. for any inputs I, I' and any labelings $\mathcal{L}, \mathcal{L}'$, if $[T]_{I,\mathcal{L}} = [T']_{I',\mathcal{L}'}$, for any trees T, T', then $f(I) = f(I')$. The processors want to compute $f(I)$, but they do not know the input I. To resolve this problem the processor uses its knowledge of the

network topology to construct a labeling $\mathcal{L}'$ and an input I' such that $[T]_{I,\mathcal{L}} = [T]_{I',\mathcal{L}'}$, for all trees T. Certainly, each processor may choose a different input I' and labeling $\mathcal{L}'$. However by exchanging information using corollary 2.2 the processors can agree on a unique input I' and labeling $\mathcal{L}'$. Since the value of f depends only on the equivalence classes of the trees we conclude that $f(I) = f(I')$. Thus it is sufficient to output $f(I')$ and this will be the desired, correct value assumed by f on input I.

This concludes the description of the algorithm. It remains to determine its bit complexity. Phases 1 and 3 either involve local computations which do not require any bit exchanges or simple low cost bit exchanges. The main bit exchanges take place in phase 2. There we have N^2 iterations of the algorithm in corollary 2.2. We need $d \cdot \log N$ bits to represent each of the corresponding trees. This means that the bit complexity of the algorithm is $O(N^4 \cdot \delta \cdot d^2 \cdot \log N)$. □

3 Symmetric Functions

In this section we give an algorithm which computes any symmetric function on an unlabeled network, improving upon the algorithm given above in this case.

Let $\mathcal{N} = (V, E)$ be an unlabeled network of size N, with node set $V = \{0, 1, \ldots, N-1\}$ and edge set $E \subseteq V \times V$. To simplify the analysis below we will consider the network $\mathcal{N}' = (V, E \cup \{(i,i) | i = 0, \ldots, N-1\})$ (i.e., $\mathcal{N}$ with self loops added to each vertex). Let $deg(i)$ and $deg'(i) = deg(i) + 1$ be the valency of node i in $\mathcal{N}$ and $\mathcal{N}'$, respectively. Let $A = (a_{i,j})$ be the adjacency matrix of $\mathcal{N}'$. We associate the stochastic matrix $P = (p_{i,j})$, where $p_{i,j} = a_{i,j}/deg'(i)$, with $\mathcal{N}'$. For each node i, define $\pi_i = deg'(i)/(N + 2M)$, where M is the number of edges of $\mathcal{N}$. Note that π_i can be computed by each processor i from knowledge it has of the topology of the network. We will prove the following theorem.

Theorem 3.1 *Let $\mathcal{N}$ be an unlabeled N-node network with maximal node valency d and let $\mathcal{N}'$ be the network $\mathcal{N}$ with self loops added to each node. Let ρ be the second largest eigenvalue (in absolute value) of the stochastic matrix P associated with $\mathcal{N}'$. There is an algorithm that computes any symmetric function on the network $\mathcal{N}$ with bit complexity $O(\frac{-\log N}{\log \rho} \cdot N \cdot \log N \cdot d)$.*

Proof. The idea of the algorithm is the following. Each processor sends its initial input value to all its neighbors. After receiving the values from all its neighbors the processor updates the value it already has based on the values it receives. Every processor executes these steps S times, where $S = O(\frac{-\log N}{\log \rho})$ is a function of the topology of $\mathcal{N}$ known to all processors. Eventually in this way every input value to a node of the network will be distributed and equally accounted for by every other processor.

Let f be a symmetric boolean function on N variables known to all the processors and let f_k be the value of f on inputs of weight (i.e. number of 1's) k. Let $I = < b_v : v \in V >$ be the input to the network, where b_v is the input bit to node v. More formally the algorithm is as follows.

Algorithm for procesor p:
Input: b_p, f;
Initialize: $value_p[0] := b_p$ and S;
for $i := 0, 1, \ldots, S$ **do**

send $\frac{value_p[i]}{deg'(p)}$ to all neighbors of p;
receive $\frac{value_q[i]}{deg'(q)}$ from all neighbors q of p;
compute $value_p[i+1] := \sum\{\frac{value_q[i]}{deg'(q)} : q \text{ is a neighbor of } p \text{ or } q = p\}$
od;
put $w := \left[\frac{value_p[S]}{\pi_p}\right]$;
Output f_w.

Each processor knows its input bit but does not know the network input configuration I. At the ith stage, $i \leq S$, processor p updates its variable $value_p$ which is an approximation to the number of 1's in the input configuration I times the quantity π_p. At the final stage the processor computes $w = [value_p/\pi_p]$, the nearest integer less than or equal to $value_p/\pi_p$. If the approximation is sufficiently close to the actual value $k\pi_p$, where k is the weight of the input I, then all processors will output the same correct value f_w.

We have to show that all the processors eventually converge to the correct ratio (and hence the resulting value f_w is the same for all the processors) and to bound the value of S. We will use the theory of Markov chains ([Sen81], [Gan59]) in order to complete the proof of correctness of the above algorithm.

Note that $P = (p_{i,j})$ is the $N \times N$ stochastic matrix of a primitive, reversible Markov chain corresponding to a random walk on $\mathcal{N}'$. (In general, the stochastic matrix corresponding to an arbitrary connected, undirected network is only irreducible and need not be primitive. By adding a self loop to each vertex of $\mathcal{N}$ to form the network $\mathcal{N}'$ we guarantee the corresponding Markov chain is primitive.) Its stationary distribution is $(\pi_1, \pi_2, \ldots, \pi_N)$. Let $1 = \lambda_1 > \lambda_2 \geq \ldots \geq \lambda_k$ be the eigenvalues of P and put

$$\rho = \max\{|\lambda_i| : 2 \leq i \leq k\}.$$

Standard arguments (see e.g. [BK89, lemma 2]) show that

$$p_{i,j}^{(r)} = \pi_j + O\left(\sqrt{\frac{\pi_j}{\pi_i}} \cdot \rho^r\right), \tag{1}$$

where $p_{i,j}^{(r)}$ is the (i,j) entry of the matrix P^r. If $M_P = \max_{i,j}\{\sqrt{\pi_j/\pi_i}\}$ then the matrix form of equation (1) is

$$P^r = P^\infty + O\left(M_P \cdot \rho^r \cdot E\right), \tag{2}$$

where E is the matrix of all 1's, and the limit $P^\infty = \lim_{r\to\infty} P^r$ of the chain is an $N \times N$ matrix such that all the entries of its i-th column are equal to π_i. In our case, $M_P = \sqrt{d_{max}/d_{min}}$, where d_{max} (respectively, d_{min}) is the maximal (respectively, minimal) valency of the network $\mathcal{N}'$. Hence, equation (2) becomes

$$P^r = P^\infty + O\left(\sqrt{\frac{d_{max}}{d_{min}}} \cdot \rho^r \cdot E\right). \tag{3}$$

It is easy to see that for any input vector I, $L = IP^\infty$ is the eigenvector of P whose ith entry equals $k\pi_i$, where k is the number of 1's in the input I.

We are interested in the rate of convergence of the limit of IP^r as r tends to infinity. It follows from equation (3) that

$$IP^r = L + O\left(\sqrt{\frac{d_{max}}{d_{min}}} \cdot \rho^r \cdot k \cdot e\right) \tag{4}$$

where e is the row vector consisting of all 1's. During the rth iteration of the above algorithm processor p computes the pth component of IP^r. To guarantee that all the proccessors compute the correct value it is enough to ensure that the error term in (4) is less than $(1/2)\pi_p$, i.e.

$$\sqrt{\frac{d_{max}}{d_{min}}} \cdot \rho^r \cdot k < \frac{1}{2(N+2M)}.$$

This inequality implies that the number of iterations required is $S = O(-\log N/\log \rho)$, if $\rho > 0$. (Of course the case $\rho = 0$ is possible but then the number of required iterations is $S = 2$.) It is not hard to see that during each iteration of the algorithm $O(\log S)$ bits must be transmited by each processor to all of its less than or equal to d neighbors in order to guarantee a sufficient precision of the approximation at the S-th iteration. By [LO81] for any network $\mathcal{N}$ with maximal node valency d and diameter δ the second largest eigenvalue of the stochastic matrix corresponding to the network $\mathcal{N}'$ satisfies the inequality

$$\rho \leq 1 - \frac{1}{N \cdot \delta \cdot (2+d)},$$

i.e. $\log \rho \leq -1/N \cdot \delta \cdot (2+d)$. Hence $\log S = O(\log N)$ and so the bit complexity of the algorithm (number of steps $\times$ number of processors $\times$ maximal number of bits per step per processor) is indeed

$$O\left(-\frac{\log N}{\log \rho} \cdot N \cdot \log N \cdot d\right)$$

as we had to prove. □

As an immediate consequence of the above theorem we get the following bound on the bit complexity of symmetric functions on unlabeled networks.

Theorem 3.2 *Let $\mathcal{N}$ be an unlabeled N-node network with maximal node valency d and diameter δ. There is an algorithm that computes any symmetric boolean function on $\mathcal{N}$ with bit complexity $O(N^2 \cdot \delta \cdot d^2 \log^2 N)$.*

Proof. As above $\log \rho \leq -\frac{1}{N \cdot \delta \cdot (2+d)}$. Combining this with theorem 3.1 we obtain that the bit complexity for computing symmetric functions is $O(N^2 \cdot \delta \cdot d^2 \cdot \log^2 N)$. □

Corollary 3.1 *The bit complexity of computing any symmetric function on an unlabeled d-dimensional torus with $N = n^d$ nodes is $O(N^{1+\frac{2}{d}} \log^2 N)$.*

Proof. The characteristic values of the corresponding adjacency matrix of $\mathcal{N}'$ are given by the formula

$$1 + \sum_{k=1}^{d} 2\cos(\frac{2\pi}{n} i_k), 1 \leq i_1, \ldots, i_d \leq n$$

The second largest eigenvalue of the corresponding stochastic matrix of $\mathcal{N}'$ is $\rho = \frac{1}{2d+1} \cdot (1 + 2d \cdot \cos(\frac{2\pi}{n}))$. Using approximations to the log and cos functions it is easy to show $\log \rho = O(-\frac{1}{n^2})$. Thus, by the theorem, the bit complexity of computing symmetric functions in this case is $O(N^{1+\frac{2}{d}} \log^2 N)$. □

Corollary 3.2 *The bit complexity of computing a symmetric function on an unlabeled n-dimensional hypercube with $N = 2^n$ nodes is $O(N \log^4 N)$.*

Proof. The eigenvalues of the adjacency matrix of the hypercube are $\lambda_i = n - 2i$, $0 \leq i \leq n$. The second largest eigenvalue of the corresponding stochastic matrix of $\mathcal{N}'$ is $\frac{n-1}{n+1}$. Using the inequality $\log(1 - \frac{2}{n+1}) < -\frac{2}{n+1}$, the theorem implies that the bit complexity of computing symmetric functions in this case is $O(N \log^4 N)$. □

Corollary 3.3 *The bit complexity of computing any symmetric function on a random regular graph of valency $2d$ is $O(N \frac{d \log^2 N}{\log d})$ with probability greater than $1 - N^{-\Omega(\sqrt{d})}$.*

Proof. This follows immediately from the theorem and recent results of Friedman et. al. [FKS89] bounding the size of the second largest eigenvalue of random regular graphs. □

4 Distance Regular Graphs

In this section we show that by taking advantage of the topology of distance regular graphs we can derive efficient algorithms for computing symmetric functions on such graphs.

The distance between any two nodes $p, q \in V$ of a network $\mathcal{N}$, denoted $d(p, q)$, is the length of the shortest path between p and q. The circle with center $p \in V$ and radius k, denoted by $C(p; k)$, is the set of nodes $q \in V$ such that $d(p, q) = k$. The set of neighbors of p, denoted $\mathcal{N}(p)$, is the circle $C(p; 1)$. The threshold function $Th_k \in B_N$ is defined to be 1 on inputs of weight at least k and 0 otherwise. (By the weight of an input I we understand the number of occurrences of 1 in the input.)

Distance regular graphs are graphs $\mathcal{N}$ such that for any nodes $p, q \in V$ with $d(p, q) = k$ the quantities

$$\begin{array}{c} \mid C(p;1) \cap C(q;k-1) \mid, \\ \mid C(p;1) \cap C(q;k+1) \mid \end{array}$$

depend only on the distance $d(p, q)$. More formally, for $k = d(p, q)$ we define

$$\begin{aligned} a_k &= \mid \{r \in C(p;1) : d(q,r) = k-1\} \mid, k = 1, 2, \ldots, \delta \\ b_k &= \mid \{r \in C(p;1) : d(q,r) = k+1\} \mid, k = 0, 1, \ldots, \delta - 1, \\ c_k &= \mid \{r \in C(p;1) : d(q,r) = k\} \mid, k = 0, 1, \ldots, \delta. \end{aligned}$$

Such graphs include hypercubes, odd graphs, triangle graphs, complete bipartite graphs, etc. [Big74], [Cam83]. They satisfy several useful properties. We mention only a few obvious ones and refer the reader to [Big74] and [Cam83] for further properties. Distance regular graphs are regular with valency $d = b_0$. By definition, $a_0 = 0$. Moreover, $c_0 = 0$

and $a_1 = 1$. Since, if $d(p,q) = k$ every neighbor of p has distance k, $k-1$ or $k+1$ from q it is clear that $c_k = d - a_k - b_k$. A network $\mathcal{N}$ is distance transitive if for any nodes p, q, p', q' with $d(p,q) = d(p',q')$ there is an automorphism ϕ of the network $\mathcal{N}$ such that $\phi(p) = p'$ and $\phi(q) = q'$. It is easy to see that all distance transitive graphs are distance regular, but the converse is false [Big74].

Now we are ready to prove the main theorem of this section.

Theorem 4.1 *On an unlabeled N-node distance regular network of valency d and diameter δ every symmetric function can be computed in $O(N \cdot \delta \cdot d \cdot \log N)$ bits. Moreover the threshold function Th_k can be computed in $O(N \cdot \delta \cdot d \cdot \log k)$ bits, where $k \leq N$.*

Proof. For any input configuration $I =< b_v : v \in V >$, any processor p and any distance $k \leq \delta$ let $I(p;k)$ be the number of processors x at distance k from the processor p such that $b_x = 1$. To compute a symmetric function it is sufficient for each processor p to know $I(p;k)$, for each $k \leq \delta$. The idea of the proof is to find a (inductive) formula for computing $I(p;k)$ in terms of the previously computed values $I(p;l)$, where $l < k$, and values $I(q,l)$, where $q \in C(p;1)$ is a neighbor of p, $l < k$. We note that

$$\begin{aligned}
\sum_{q \in \mathcal{N}(p)} I(q;k-1) &= \mid \{< q,x >: q \in \mathcal{N}(p), d(q,x) = k-1, b_x = 1\} \mid \\
&= \sum_{b_x=1} \mid \{q \in \mathcal{N}(p) : d(q,x) = k-1\} \mid \\
&= \sum_{b_x=1, d(p,x)=k} \mid \{q \in \mathcal{N}(p) : d(q,x) = k-1\} \mid + \\
&\quad \sum_{b_x=1, d(p,x)=k-1} \mid \{q \in \mathcal{N}(p) : d(q,x) = k-1\} \mid + \\
&\quad \sum_{b_x=1, d(p,x)=k-2} \mid \{q \in \mathcal{N}(p) : d(q,x) = k-1\} \mid \\
&= \sum_{b_x=1, d(p,x)=k} a_k + \sum_{b_x=1, d(p,x)=k-1} c_{k-1} + \sum_{b_x=1, d(p,x)=k-2} b_{k-2} \\
&= a_k \cdot I(p;k) + c_{k-1} \cdot I(p;k-1) + b_{k-2} \cdot I(p;k-2),
\end{aligned}$$

which in turn leads to the following inductive formula

$$I(p;k) = \frac{1}{a_k} \cdot \left(\sum_{q \in \mathcal{N}(p)} I(q;k-1) - (d - a_{k-1} - b_{k-1}) \cdot I(p;k-1) - b_{k-2} \cdot I(p;k-2) \right). \tag{5}$$

Formula (5) and the knowledge of the network topology (i.e. the numbers a_k and b_k) make it possible to construct an efficient algorithm for computing symmetric functions. Let $f \in B_N$ be a symmetric function and let f_k be the value of f on inputs of weight k.

Algorithm for processor p:
Input: b_p, f;
Initialize: $I(p;0) := 1$ if p's input bit is 1 and is $:= 0$ otherwise;
send input bit to all neighbors;
compute $I(p;1) :=$ the number of 1s among the neighbors of p;
for $k := 1, \ldots, \delta - 1$ **do**

send $I(p;k)$ to all the neigbors of p;
compute $I(p;k+1)$ from $I(p;k-1)$, $I(p;k)$ and the $I(q;k)$s,
where q ranges over all neighbors of p, via formula (5);
od;
compute the sum $s := \sum_{k=0}^{\delta} I(p;k)$;
output f_s

The correctness of the algorithm was shown above. It remains to determine its complexity. For $k = 0, \ldots, \delta$ each processor p transmits the number $I(p;k)$ to all its neighbors. This requires transmission of δ messages

$$I(p;0), \ldots, I(p;\delta)$$

(each of length less than or equal to $\log N$ bits) to each of the d neighbors of p, i.e. $O(\delta \cdot d \cdot \log N)$ bits per processor for a total of $O(N \cdot \delta \cdot d \cdot \log N)$.

The proof of the bit complexity of computing the threshold function Th_k employs the previous algorithm. Observe that when the number of 1s at a certain distance from a processor exceeds the threshold value k then we only need to transmit k which requires $\log k$ bits. □

An important corollary to the above is the case of the hypercube.

Corollary 4.1 *On the unlabeled hypercube, every symmetric function can be computed in* $O(N \cdot \log^3 N)$ *bits. Moreover the threshold function* Th_k *can be computed in* $O(N \cdot \log^2 N \cdot \log k)$ *bits, where* $k \leq N$.

Proof. Let $n = \log N$. This is an immediate consequence of the fact that the hypercube is distance regular. It is easy to show that in the notation of section 4, $a_k = k$, $b_k = n - k$ and $c_k = 0$. The resulting inductive formula (which is a special case of formula (5)) is the following:

$$b(p;k) = \frac{1}{k} \cdot \left(\sum_{q \in D(p;1)} b(q;k-1) - (n-k+2) \cdot b(p;k-2) \right) .\square \qquad (6)$$

5 Conclusions and Open Problems

The present paper has been concerned with the problem of determining algorithms with polynomial bit complexity for computing boolean functions on anonymous distributed networks. The main result of section 2 provides such an algorithm for any unlabeled network $\mathcal{N}$ with bit complexity $O(N^4 \cdot \delta \cdot d^2 \cdot \log N)$. It would be interesting however if we could improve on this bit complexity.

We have been able to find more efficient algorithms for computing symmetric functions on arbitrary networks (theorem 3.2) and very efficient algorithms for symmetric functions on the class of distance regular networks (theorem 4.1). Nevertheless these algorithms are still not known to be optimal and improvements are possible.

An interesting special case is that of the hypercube network. Based upon the results of [ASW85] for unlabeled and oriented rings and [BB89] for oriented tori we conjecture that there are more efficient algorithms for computing boolean functions on the unlabeled

and oriented hypercube than those provided here. Preliminary results on these questions are presented in [KK90].

There have been few studies in the literature regarding lower bounds. The only network for which this question has been studied extensively is the ring [MW86], [AAHK88], [DG87]. [PKR84] studies the question for the extrema finding function but relies on specific properties of this function. [YK88] give lower bounds for the message complexity of computing boolean functions for broad classes of networks. However, very little is known about lower bounds on the bit complexity of boolean functions on the anonymous torus or hypercube, not to mention the general case of unlabeled networks.

If we allow the processors to flip coins in the course of the computation then this changes entirely the rules of the game. It is now possible to introduce algorithms with improved average and worst case bit complexity. Also, the class of functions computable in this model may be different. For the case of rings this has been studied by [AS88]. For general networks [SS89] and [MA89] have given algorithms with low message complexity for the problem of constructing a rooted spanning tree (which can then be used to compute boolean functions efficiently). It would be very interesting to examine more thoroughly the bit complexity for the case of general anonymous networks.

6 Acknowledgements

We are grateful to L. Meertens, J. Hastad and J. Tromp for many fruitful conversations. C. Attiya and T. Tsantilas were very helpful with the bibliography.

References

[AAHK88] Karl Abrahamson, Andrew Adler, Lisa Higham, and David Kirkpatrick. Randomized evaluation on a ring. In Jan van Leeuwen, editor, *Distributed Algorithms, 2nd International Workshop, Amsterdam, The Netherlands, July 1987*, volume 312, pages 324 – 331, Heidelberg, 1988. Springer Verlag Lecture Notes in Computer Science.

[Ang80] Dana Angluin. Local and global properties in networks of processors. In *12th Annual ACM Symposium on Theory of Computing*, pages 82 – 93, 1980.

[AS88] Hagit Attiya and Mark Snir. Better computing on the anonymous ring. Technical Report RC 13657 (number 61107), IBM T. J. Watson Research Center, November 1988. 33 pages.

[ASW85] Chagit Attiya, Mark Snir, and Manfred Warmuth. Computing on an anonymous ring. In *4th Annual ACM Symposium on Principles of Distributed Computation*, pages 196 – 203, 1985.

[BB89] Paul W. Beame and Hans L. Bodlaender. Distributed computing on transitive networks: The torus. In B. Monien and R. Cori, editors, *6th Annual Symposium on Theoretical Aspects of Computer Science, STACS*, pages 294–303, Heidelberg, 1989. Springer Verlag Lecture Notes in Computer Science.

[Big74] Norman Biggs. *Algebraic Graph Theory.* Cambridge University Press, 1974.

[BK89] A. Broder and A. Karlin. Bounds on the cover time. *Journal of Theoretical Probability*, 2(1):101 – 120, 1989.

[Cam83] Peter J. Cameron. Automorphism groups of graphs. In Lowell W. Beineke and Robin J Wilson, editors, *Selected Topics in Graph Theory, Volume 2*, chapter 4, pages 89 – 127. Academic Press Inc., 1983.

[DG87] P. Duris and Z. Galil. Two lower bounds in asynchronous distributed computation. In *Proceedings 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 326 – 330, 1987.

[FKS89] Joel Friedman, Jeff Kahn, and Endre Szemerédi. On the second eigenvalue of random regular graphs. In *21st Annual ACM Symposium on Theory of Computing*, pages 587 – 598, 1989.

[Gan59] F. R. Gantmacher. *Matrix Theory.* Chelsea Publishing Company, 1959. Translated from the Russian.

[KK90] E. Kranakis and D. Krizanc. Computing boolean functions on a distributed hypercube network, 1990. unpublished manuscript.

[LO81] H. J. Landau and A. M. Odlyzko. Bounds for eigenvalues of certain stochastic matrices. *Linear Algebra and its Applications*, 38:5 – 15, 1981.

[MA89] Y. Matias and Y. Afek. Simple and efficient election algorithms for anonymous networks. In J.-C. Bermond and M. Raynal, editors, *Distributed Algorithms, 3nd International Workshop, Nice, France, September 1989*, volume 392, pages 183 – 194, Heidelberg, 1989. Springer Verlag Lecture Notes in Computer Science.

[MW86] S. Moran and M. Warmuth. Gap theorems for distributed computation. In *5th Annual ACM Symposium on Principles of Distributed Computation*, pages 131 – 140, 1986.

[PKR84] J. Pachl, E. Korach, and D. Rotem. A new technique for proving lower bounds for distributed maximum finding algorithms. *J. of the ACM*, 31(4):905 – 918, October 1984.

[Sen81] E. Seneta. *Non-negative Matrices and Markov Chains.* Springer Series in Statistics. Springer Verlag, 1981. 2nd edition.

[SS89] B. Schieber and M. Snir. Calling names on nameless networks. In *8th Annual ACM Symposium on Principles of Distributed Computation*, pages 319–328, 1989.

[YK87] M. Yamashita and T. Kameda. Computing on an anonymous network. Technical Report 87-16, Laboratory for Computer and Communication Research, Simon Fraser University, 1987. 27 pages.

[YK88] M. Yamashita and T. Kameda. Computing on an anonymous network. In *7th Annual ACM Symposium on Principles of Distributed Computation*, pages 117 – 130, 1988.