Meeting Your Neighbours

Avery Miller
a4miller@cs.toronto.edu
University of Toronto, Toronto, Canada

June 17, 2011

Abstract

In the *complete neighbourhood learning* problem, each processor in a static ad hoc radio network must discover all other processors that are within its transmission range. This paper contains a survey of the important upper and lower bounds that are already known for this problem. We provide novel ways of presenting the algorithms, as well as proofs of correctness and running-time analyses that were omitted in some of the original publications.

1 Introduction

There are many examples of algorithms for static ad hoc networks in which it is assumed that processors can discover all other nearby processors. In topology control algorithms [1, 2, 3], each processor adjusts its transmission power (or, in the case of directional antennas, its transmission direction) to form a network topology with certain properties (e.g., a minimum-energy spanning tree). In the case of routing algorithms [4, 5], processors must find a way of forwarding a message from source to destination using a path of transmitting processors. Some knowledge about the topology is needed to avoid simply flooding the network. In the gossiping task, each processor wishes to send a message to all other processors in the network. In [6], Chlebus et al. showed how to efficiently complete the gossiping task once each node knows its neighbourhood. Regardless of the task, many algorithms assume that an underlying network layer deals with medium-access control (MAC). This layer is responsible for scheduling processor transmissions in such a way as to prevent two processors that are close to one another from interfering with each other. Many medium-access control protocols [7, 8] need information about which processors are close to one another. It is important to note that *partial* neighbourhood learning, as accomplished by the randomized neighbourhood learning algorithms in the literature, is not sufficient in many of these situations: algorithm correctness can no longer be ensured if the assumption that each processor initially knows its complete neighbourhood is no longer true.

The purpose of this paper is to collect important results in the theory of complete neighbourhood learning and to present them in an instructive manner. The algorithms in this paper provide examples of ways to use certain combinatorial objects (cover-free codes, selectors, and selective families) to construct transmission schedules with desirable properties. These properties are useful in many contexts outside of neighbourhood learning. To emphasize this, the presentation of the neighbourhood learning algorithm in section 3.3 is separated into subroutines that solve problems of independent interest: broadcasting, leader election, gathering, distributing, and learning the topology of a connected network of processors with similar degrees. For some algorithms, this paper also provides some additional details about the proof of correctness and running-time analysis that are not available in the literature.

In section 2, we give our general assumptions about the problem and model. Then, in section 3.1, we present a strong lower bound for complete neighbourhood learning, due to Krishnamurthy et al. [9], when it is assumed that algorithms avoid all transmission collisions. In the remainder of the paper, we will describe several algorithms that perform asymptotically better than this bound. These algorithms may produce schedules in which transmission collisions occur. To ensure that these collisions do not prevent the processors in the network from ultimately learning about their neighbourhood, the algorithms make use of cover-free codes (section 3.2), selectors and selective families (section 3.3).

2 Preliminaries

A static ad hoc network consists of n processors at arbitrary fixed locations. Each processor p possesses a unique identifier number $ID(p)$ from the range $\{1,\ldots,N\}$. For a set of processors P, we denote by $ID(P)$ the set $\{ID(p) \mid p \in P\}$. It is assumed that the universe of IDs is much larger than the size of the network, i.e., N is super-polynomial in n. The topology of a network is represented as an undirected graph, with a vertex for each processor and an edge joining each pair of neighbours. The maximum degree of the network is denoted by Δ. For any processor p, we denote by $nbrhd(p)$ the set of processors that are neighbours of p. This set is known as p's *neighbourhood*. We consider the task of *complete neighbourhood learning*, where each processor p must determine the set $ID(nbrhd(p))$. Unless specified otherwise, each processor initially knows its own ID and the value of N. We assume that no processors leave or join the network during an algorithm's execution.

At any given time, a processor can either *transmit* or *listen*, but not both. We consider networks where the processors share a single radio channel, which means that two signals that reach the same point at the same time interfere with one another. A processor in *listen* mode that receives two such signals simultaneously only hears noise, and we say that a *collision* has occurred. We say that a listening processor *receives* a message in the case when exactly one of its neighbours transmits. A listening processor cannot distinguish between the normal background noise that is present when no neighbour is transmitting and the noise heard during a collision.

Each processor possesses a clock which divides time into equal-length *slots*, $(t_0,t_1,\ldots)$. Each slot is long enough to allow the complete transmission of any message. We consider *synchronous* models, in which it is assumed that all clocks run at the same rate, that slot boundaries coincide across all processors, and that each processor begins its local algorithm at the same time. Each processor locally computes its *transmission schedule*, which specifies the slots during which it will transmit. The model allows any set of processors to transmit during a single slot. A *collision-free* algorithm ensures that the processors are scheduled in such a way that no collisions occur.

3 Complete Neighbourhood Learning

The simplest approach to neighbourhood learning uses *round-robin* scheduling, which assigns a different slot for each ID in $\{1,\ldots,N\}$. A processor can only transmit in the slot assigned to its ID. A simple example of such a schedule is to assign slot i to processor with ID i. The length of this schedule is N, which gives an upper bound of N slots for complete neighbourhood learning. We will now see that there is no efficient deterministic collision-free algorithm for complete neighbourhood learning. In fact, in the class of collision-free algorithms, the simple round-robin procedure is nearly optimal.

3.1 A Collision-Free Lower Bound

Assume that all processors know the value of N, but do not have any information about the value of n. Krishnamurthy et al. [9] show that, for any deterministic collision-free neighbourhood learning algorithm, every processor must listen during at least $N-n$ time slots. In general, since it may be the case that $N >> n$, this lower bound implies that no deterministic collision-free algorithm can do significantly better than an algorithm that uses round-robin scheduling.

The proof of their result relies on an indistinguishability argument. Before proceeding, a few definitions are needed. At any point in time, the *state* of a processor is a representation of the values of its local variables. The *system state* at any point in time is the current state of all processors. The *mode* of a processor during time slot t is one of $\{listen, transmit, terminated\}$. The *system mode* during time slot t is the mode of all processors during time slot t. A *network execution* is an alternating sequence of system states and system modes, starting with an initial system state. The *history* h_i of a processor p_i is what happens to it in each time slot. Specifically,

- $h_i(t) = \perp$ if, during slot t, p_i is in transmit mode, if no neighbour of p_i transmits, a collision occurs at processor p_i, or p_i has terminated;
- $h_i(t) =$ the message that p_i receives in slot t, otherwise.

Two executions are *indistinguishable* to processor p_i up to time slot t if p_i has the same initial state and the same history in both executions up to, but not including slot t in both executions. The indistinguishability argument used to prove the lower bound uses the fact that, if two executions are indistinguishable to processor p_i up to time slot t, then p_i has the same mode during time slot $t+1$ in both executions.

For any algorithm $\mathscr{A}$, the authors prove that, for an arbitrary processor p_i, there must be a different slot in which p_i is in listen mode for each ID that is not held by a processor in the network. For any network $G = (V, E)$ and any k such that no processor in V has ID k, they define $G_k = (V \cup \{p_k\}, E_k)$ where $E_k = E \cup \{p_i, p_k\}$. Then, they define t_k to be the first time slot in the execution of $\mathscr{A}$ in network G_k during which p_k transmits and p_i is listening (i.e., p_i has not terminated). The value of t_k is well-defined since, otherwise, the entire executions of algorithm $\mathscr{A}$ in both G and G_k would be indistinguishable to p_i, which means that p_i would learn the same set of neighbours in both G and G_k. To processor p_i, the executions of algorithm $\mathscr{A}$ up to slot t_k in both G and G_k are indistinguishable: p_i has the same initial state in both executions, since it does not know n; also, in G_k, no processor receives a transmission from p_k before slot t_k. It follows that p_i is listening during slot t_k during the execution of $\mathscr{A}$ in G.

Finally, for any two distinct IDs a, b such that no processor in V has ID a or b, the authors prove that $t_a \neq t_b$. This can be proven by contradiction. Assume that $a \neq b$ but $t_a = t_b$. Consider the network $G_{a,b} = (V \cup \{p_a, p_b\}, E_{a,b})$, where $E_{a,b} = E \cup \{p_i, p_a\} \cup \{p_i, p_b\}$. Without loss of generality, suppose that, in the execution of $\mathscr{A}$ in $G_{a,b}$, p_i first receives a message from p_a before it first receives a message from p_b. Suppose that t' is the time slot during which p_i first receives a message from p_a in the execution of $\mathscr{A}$ in $G_{a,b}$. Up to time slot $t_{min} = \min\{t', t_a\}$, the executions of $\mathscr{A}$ in G_a and $G_{a,b}$ are indistinguishable to all processors. It follows that p_a transmits during slot t_{min} in both executions, that is, $t_{min} = t_a = t'$. Similarly, since $t_a = t_b$, the executions of $\mathscr{A}$ in G_b and $G_{a,b}$ are indistinguishable to all processors up to time slot $t_{min} = \min\{t', t_a\} = \min\{t', t_b\}$. Since p_b transmits during slot t_b in the execution of $\mathscr{A}$ in G_b, it will also transmit during $t_b = t_a = t_{min}$ in the execution of $\mathscr{A}$ in $G_{a,b}$. Therefore, we have shown that both p_a and p_b transmit during slot t_{min} in the execution of $\mathscr{A}$ in $G_{a,b}$, which causes a collision at p_i, a contradiction.

Note that the above argument also works when processors have an approximate upper bound n_u on the size of the network G. Namely, as long as $n_u \geq n+2$, then we can argue that the initial state of p_i in networks G, G_a, G_b, and $G_{a,b}$ are indistinguishable.

3.2 An Algorithm Based on Cover-Free Codes

3.2.1 Mathematical Background

Given a set of t-bit binary strings, we *combine* them by performing a bitwise OR, e.g., if $S = \{011001, 010010\}$, then $comb(S) = 011011$. A set C of t-bit binary strings is called a *superimposed code of order d* if all combinations of at most d strings from C are distinct. Superimposed codes were first introduced by Kautz and Singleton [10] as *uniquely decipherable codes*.

A superimposed code can be represented using matrix notation. A $t \times N$ binary matrix represents a code with N binary strings, each having length t. As an example, consider the superimposed code represented by the following matrix. As all combinations of at most 2 columns are distinct, this matrix represents a superimposed code of order 2.

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Given two t-bit strings a, b, we write $a \leq b$ to denote that each bit of a is a 1 only if the corresponding bit in b is a 1. A set C of t-bit binary strings is called a *cover-free code of order d* if, for any $a \in C$ and any $B \subseteq C$ of cardinality at most d, $a \leq comb(B)$ implies that $a \in B$. The $n \times n$ identity matrix represents a cover-free code of order n. We note that the matrix in the example above does not represent a cover-free code of order 2, since the first column is less than or equal to the combination of the second and third columns.

The matrix representation of cover-free codes can be characterized by focusing on the rows rather than the columns. We will see that this characterization is helpful in creating transmission schedules for neighbourhood learning. The following characterization of cover-free code matrices is well-known [11].

Lemma 1. *A matrix M represents a cover-free code of order d if and only if, for any nonempty set S of at most $d+1$ columns of M and for any $c \in S$, there exists a row r such that $M_{r,c} = 1$ and, for each $c' \in S \backslash c$, $M_{r,c'} = 0$.*

Proof. First, suppose that the matrix M represents a cover-free code. Consider any nonempty set S of at most $d+1$ columns of M and any $c \in S$. By the definition of cover-free code, $c \not\leq comb(S \backslash c)$, so there exists a row r such that $M_{r,c} = 1$ and, for each $c' \in S \backslash c$, $M_{r,c'} = 0$.

Next, suppose that for any nonempty set S of at most $d+1$ columns of M and for any $c \in S$, there exists a row r such that $M_{r,c} = 1$ and, for each $c' \in S \backslash c$, $M_{r,c'} = 0$. Consider any column a and any set of columns B of cardinality at most d. Let $S = \{a\} \cup B$. Then $|S| \leq d+1$, so there exists a row r such that $M_{r,a} = 1$ and, for each $c' \in S \backslash a$, $M_{r,c'} = 0$. If $a \notin B$, then $(S \backslash a) = B$. Thus, $a \not\leq comb(B)$, as required. □

3.2.2 The Algorithm

Suppose that we are given a cover-free code C of order d in a $t \times N$ matrix M. We can construct a transmission schedule of length t by assigning to each processor p_j in the network a unique column c_j of the matrix, and having processor p_j transmit its ID in slot i if and only if $M_{i,j} = 1$. If $\Delta \leq d$, then Lemma 1 implies that, for each set S of at most $\Delta + 1$ processors, for each processor s in S, there exists a time slot in which s transmits and all processors in $S \backslash s$ stay silent. So, by assigning to S an arbitrary processor p and its (up to) Δ neighbours, it follows that p will receive a message from each of its neighbours. The running time of this algorithm depends on t, the length of the cover-free code of order d.

Porat and Rothschild [20] construct explicit cover-free codes of order d that have length $O(\min\{N, d^2 \log N\})$. In the worst case, either the algorithm has no non-trivial upper bound on Δ, or $\Delta \in \Omega(n)$, so the running time is $O(\min\{N, n^2 \log N\})$.

Keshavarzian et al. [12] give a condition on the parameters of constant-weight codes such that the resulting codes are cover-free. A set C of L-bit binary strings is called an (L, W, h) *constant-weight code* if every string in C has exactly W ones and every pair of strings in C differ in at least h bits. The authors show that, if $W \leq \frac{(h\Delta/2)-1}{\Delta-1}$, then the resulting code is a cover-free code of order Δ. To see why this is true, we will use the following fact.

Lemma 2. *Consider two strings a, b of length L such that each string has exactly W ones. Suppose that the Hamming distance between a and b is ℓ. Let x be the number of positions where a and b are both 1. Then, $W = x + \ell/2$.*

Proof. Consider the number of 1's that appear in a plus the number of ones that appear in b, which we know is equal to $2W$. This can be expressed as: (the number of positions i such that $a_i = 1$ and $b_i = 0$) + (the number of positions j such that $a_j = 0$ and $b_j = 1$) + 2*(the number of positions k such that $a_k = b_k = 1$). The first two terms sum to ℓ and the last term is equal to $2x$. $\square$

Consider any set S of $\Delta + 1$ strings from C. Lemma 2 implies that, for any two strings in S, the number of positions where both strings are 1 is $W - h/2$. Fix an arbitrary string $s \in S$. For each string $s' \in S - \{s\}$, there are at most $W - h/2$ positions in which both s and s' are 1. Therefore, if there are $1 + \Delta(W - h/2)$ positions in s that are 1, there must be a position i such that $s_i = 1$ and $s'_i = 0$ for all $s' \in S - \{s\}$. By considering the matrix with the strings of C as columns, this satisfies the characterization of cover-free codes of order Δ in Lemma 1.

For an (L, W, h) constant-weight code C, the relationship between $|C|$ and (L, W, h) is not well understood asymptotically. In other words, we don't know general bounds on the length of a constant-weight code given the constraints $|C| \geq N$ and $W \leq \frac{(h\Delta/2)-1}{\Delta-1}$. However, given specific values of W, h, and N, one can check if an appropriate code has been constructed. Results about specific codes that have been constructed can be found in [13], [14], [15].

3.3 An Algorithm Based on Selectors and Selective Families

The most efficient deterministic algorithm for complete neighbourhood learning that we know of is due to Gasieniec et al. [16]. The algorithm uses $O(n \log^2 N \log^2 n)$ slots. After presenting the necessary mathematical background, we will describe the neighbourhood algorithm from the ground up by presenting each subroutine before presenting the algorithm in which it is used. This presentation emphasizes the fact that each subroutine is a non-trivial solution to a problem of independent interest.

3.3.1 Selectors

From [17], a family $\mathscr{F}$ of subsets of $\{1, \ldots, N\}$ is called a *w-selector*, if, for any two disjoint sets $X, Y \subseteq \{1, \ldots, N\}$ with $w/2 \leq |X| \leq w$ and $|Y| \leq w$, there exists a set S in $\mathscr{F}$ such that $|S \cap X| = 1$ and $S \cap Y = \emptyset$.

Chrobak et al. [17] prove that there exists a w-selector of size $c(w \log N) + 1$ where $c = (4/\log(32/31))$. An explicit construction of selectors of size $O(w\text{polylog(N)})$ using dispersers is provided by Indyk [18]. Chlebus and Kowalski [19] extend this construction to a more general definition of selectors.

3.3.2 Selective Families

Recall, from section 3.2.1, a set C of t-bit binary strings is called a *cover-free code of order d* if, for any $a \in C$ and any $B \subseteq C$ of cardinality at most d, $a \leq comb(B)$ implies that $a \in B$. Given a binary matrix whose columns form a cover-free code of order d, we can view each row as a characteristic vector that specifies a set of columns. Suppose that each processor in a network is assigned a unique column, and that row i specifies the set of processors that may transmit during slot i. By Lemma 1, for any non-empty subset S of at most $d + 1$ columns and any column $c \in S$, there is a row i with a 1 in column c and a 0 in each column in $S \backslash c$. In other words, the processor that is assigned column c will transmit during slot i and no other processor in the subset S will transmit during slot i. The family of sets specified by the rows of M is known as a strongly $(d + 1)$-selective family. We proceed by defining various types of selective families and discussing their importance in creating transmission schedules.

Definition 1. *For $k > 0$, a family $\mathscr{F}$ of subsets of $\{1, \ldots, N\}$ is called* strongly k-selective *if for any nonempty set $Z \subseteq \{1, \ldots, N\}$ such that $|Z| \leq k$, for each $z \in Z$, there is a set $S \in \mathscr{F}$ such that $S \cap Z = \{z\}$.*

If each set from a strongly k-selective family is assigned to a time slot, then we are guaranteed that, for any subset Z of at most k processors, each processor in Z transmits during a slot in which no other processor from Z transmits. For example, if we know that processor p has a neighbourhood of size at most k, we know that *each* of p's neighbours will transmit during a slot in which neither p nor any other neighbour of p transmits.

The explicit cover-free codes of size $O(\min\{N, k^2 \log N\})$ constructed by Porat and Rothschild [20] lead to strong k-selective families of the same size. This upper bound is a logarithmic factor away from the best known lower bound of $\Omega(\frac{k^2}{\log k} \log N)$, which was proven by Füredi [21].

If we weaken the condition on how $\mathscr{F}$ intersects with Z, we can define other useful selective families. Using a selective family that satisfies the following condition, we can ensure that *at least half* of p's neighbours will each transmit during a slot in which neither p nor any other neighbour of p transmits, provided that p has no more than k neighbours.

Definition 2. *For $k > 0$, a family $\mathscr{F}$ of subsets of $\{1, \ldots, N\}$ is called* linearly k-selective *if for any nonempty set $Z \subseteq \{1, \ldots, N\}$ such that $|Z| \leq k$, for at least half of the elements $z \in Z$, there exist $S \in \mathscr{F}$ such that $|S \cap Z| = \{z\}$.*

Gasieniec et al. [16] introduce linearly k-selective families. By taking the union of 2^i-selectors for $i = 0, \ldots, \lceil \log k \rceil - 1$, they prove the existence of a linearly k-selective family of size $O(\min\{N, k \log N\})$. Using the explicit construction of selectors by Indyk [18], this procedure leads to explicit linearly k-selective families of size $O(\min\{N, k \log^{O(1)} N)\}$. A general lower bound on the size of selective families, proven by Chlebus and Kowalski [19], implies that any linearly k-selective family must have size $\Omega(k \log(N/k))$.

Finally, if we further weaken the condition on how $\mathscr{F}$ intersects with Z, the resulting selective families can be used to ensure that *at least one* of p's neighbours will transmit during a slot in which no other neighbours of p transmit, provided that p has no more than k neighbours.

Definition 3. *For $k > 0$, a family $\mathscr{F}$ of subsets of $\{1, \ldots, N\}$ is called* k-selective *if for any nonempty set $Z \subseteq \{1, \ldots, N\}$ such that $|Z| \leq k$, there exists $z \in Z$ and a set $S \in \mathscr{F}$ such that $S \cap Z = \{z\}$.*

Clementi et al. [22] prove that k-selective families have size $\Theta(k \log(N/k))$, though the upper bound is non-constructive. The explicit linearly k-selective families described above are also k-selective families, which yields explicit k-selective families of size $O(\min\{N, k \log^{O(1)} N\})$.

3.3.3 Broadcasting

In the broadcasting task, a specified source processor has a message that needs to be shared with all processors in the network. A processor is *informed* if it knows the source message. In [17], Chrobak, Gasieniec, and Rytter describe a $O(n \log n \log N)$ broadcasting algorithm for networks where processors have no knowledge of the network's topology other than its size n.

The processors follow a transmission schedule that is defined in advance using selectors, although a processor only transmits if it is informed. A processor that is not informed is called *dormant*. The schedule is defined in stages and uses $\lceil \log_2 n \rceil + 1$ different selectors $\mathscr{S}_0, \ldots, \mathscr{S}_{\lceil \log_2 n \rceil}$. For each $j \in \{0, \ldots, \lceil \log_2 n \rceil\}$, $\mathscr{S}_j$ is a 2^j-selector. We denote by $\mathscr{S}_{j,\ell}$ the ℓ^{th} set in selector $\mathscr{S}_j$, and let m_j be the number of sets in $\mathscr{S}_j$. The schedule is easiest understood by considering the following matrix. Each row represents a stage of the algorithm and each column corresponds to one of the $\lceil \log_2 n \rceil + 1$ selectors (each selector is repeated to fill the entire column). Each entry of the matrix is a set of processors that are allowed to transmit, as defined by the associated selector. The schedule proceeds in row-major order.

$$\begin{bmatrix}
\mathscr{S}_{0,1} & \mathscr{S}_{1,1} & \mathscr{S}_{2,1} & \cdots & \mathscr{S}_{\lceil\log n\rceil,1} \\
\mathscr{S}_{0,2} & \mathscr{S}_{1,2} & \mathscr{S}_{2,2} & \cdots & \mathscr{S}_{\lceil\log n\rceil,2} \\
\mathscr{S}_{0,3} & \mathscr{S}_{1,3} & \mathscr{S}_{2,3} & \cdots & \mathscr{S}_{\lceil\log n\rceil,3} \\
\vdots & \vdots & & \vdots & \vdots \\
\mathscr{S}_{0,m_0} & \mathscr{S}_{1,m_0} & \mathscr{S}_{2,m_0} & \cdots & \mathscr{S}_{\lceil\log n\rceil,m_0} \\
\mathscr{S}_{0,1} & \mathscr{S}_{1,m_0+1} & \mathscr{S}_{2,m_0+1} & \cdots & \mathscr{S}_{\lceil\log n\rceil,m_0+1} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \mathscr{S}_{1,m_1} & \vdots & \vdots & \vdots \\
\vdots & \mathscr{S}_{1,1} & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots
\end{bmatrix}$$

The algorithm runs for a fixed number of stages. The authors calculate that $O(n\log N)$ stages are sufficient to ensure that all processors are informed upon termination. We present a more thorough version of their proof. In what follows, let $c = (4/\log(32/31))$.

An informed processor is called a *frontier* processor if it has a dormant neighbour, otherwise it is called an *inner* processor. We say that a processor p *upgrades its status* if one of the following occurs:

- p goes from dormant to frontier
- p goes from frontier to inner
- p goes from dormant to inner

Fact 3. *A processor can upgrade its status at most twice.*

Fact 4. *All processors are informed if and only if no frontier processors exist.*

Lemma 5. *Consider an arbitrary stage s. If there are dormant processors at the beginning of stage s, then there exists an integer $j > 0$ such that at least j processors upgrade their status after the beginning of stage s but before the beginning of stage $s + 2cj\log N$.*

Proof. Let F be the set of frontier processors at the beginning of stage s, and let $f = \lfloor \log_2 |F| \rfloor + 1$. For each $i = 1, \ldots, f$, consider the set Y_i of processors that go from dormant to informed in the next $c2^i \log n$ stages. There are two cases to consider:

1. There exists k such that $|Y_k| \geq 2^k$.

   In this case, the result follows by choosing $j = 2^k > 0$.

2. For all k, $|Y_k| \leq 2^k$.

   In this case, we will show that all frontier processors become inner within the next $2c|F|\log n$ stages. The result follows by choosing $j = |F|$. By Fact 4, $|F| > 0$ if there are dormant processors at the beginning of stage s.

   It is sufficient to show that an arbitrary dormant processor v that has at least 1 neighbour in F at the beginning of stage s becomes informed in the next $2c|F|\log n$ stages. Letting X be v's neighbours in F, and setting $g = \lfloor \log_2 |X| \rfloor + 1 \leq f$, it follows that $2^{g-1} \leq |X| < 2^g$. Figure 1 illustrates the relationship between v and the processors in F and Y_g. By definition, a 2^g-selector contains a set S such that $|S \cap X| = 1$ and $S \cap Y_g = \emptyset$. Namely, this means that running a 2^g-selector starting at stage s ensures that v will become informed, since:

   - exactly one element in X (i.e., exactly one neighbour of v in F) will transmit,
   - no processors in Y_g will transmit,

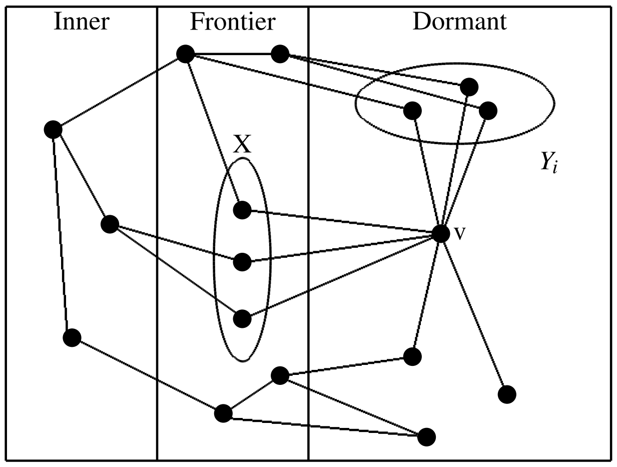

Figure 1: Dormant processors and their neighbouring frontier processors

- by the definition of Y_g, any processors that are dormant at the beginning of stage s and are not in Y_g are still dormant, so they will not transmit.

From section 3.3.1, the length of a 2^g-selector is at most $c2^g \log n \leq c2^f \log n \leq 2c|F| \log n$. This means that, starting at stage s, each set of a 2^g-selector (and in particular, S) will be scheduled in the next $2c|F| \log n$ stages, as required.

□

Informally, we will apply Lemma 5 repeatedly: starting at stage 0, some number j_0 of processors upgrade their status in the next $2cj_0 \log N$ stages, then, starting at the beginning of stage $2cj_0 \log N$, some number j_1 of processors upgrade their status in the next $2cj_1 \log N$ stages, and so on, until all processors are informed. More formally, we define a sequence of stages $s_0, s_1, \ldots$ inductively. First, let $s_0 = 0$. For $i \geq 0$, if there are dormant processors at the beginning of stage s_i:

- define j_i to be the maximum j such that j processors upgrade their status after the beginning of stage s_i but before the beginning of stage $s_i + 2cj \log N$,
- define $s_{i+1} = s_i + 2cj_i \log N$ (equivalently, $s_{i+1} = 2c\left(\sum_{m=0}^{i} j_m\right) \log N$).

By Lemma 5, if there are dormant processors at the beginning of stage s_i, then $j_i > 0$, which implies that $s_{i+1} > s_i$. Finally, for $i \geq 0$, define U_i to be the set of processors that upgrade their status after the beginning of stage s_i but before the beginning of stage s_{i+1}.

Consider the smallest integer α such that no processors are dormant at the beginning of s_α. Namely, all processors are informed at the beginning of stage $s_\alpha = 2c\left(\sum_{m=0}^{\alpha-1} |U_m|\right) \log N$. From Fact 3, it follows that $\sum_{m=0}^{\alpha-1} |U_m| \leq 2n$. Thus, $s_\alpha \leq 4cn \log N$, as required.

As each stage consists of $\log_2 n+1$ slots, the total number of time slots is $(\log_2 n+1)4cn\log N \in O(n\log n\log N)$. Note that the algorithm does not treat the source processor any differently than other informed processors. Also, Lemma 5 does not use the fact that there is a single source. Therefore, the algorithm still works if it is executed in a network where several processors are already informed. This property makes the algorithm useful as a subroutine for leader election.

3.3.4 Leader Election

In the leader election problem, all processors in a connected network must choose the ID of a single processor in the network. In [17], the authors solve this task by describing a $O(n\log n\log^2 N)$ algorithm in which all processors determine the largest ID of all processors in the network.

Their algorithm works like binary search: in each iteration, the processors start with a range of candidate IDs (initially $[1,\dots,N]$), calculate the midpoint of that range, and then run a version of the broadcasting algorithm from section 3.3.3 in which any processors with ID greater than the range's midpoint begin the algorithm as "informed" with message HIGHER. Each processor updates its range of candidate IDs for the next iteration depending on whether or not it received the HIGHER message in the current iteration: if HIGHER is received, the processor discards the candidate IDs that are less than or equal to the range's midpoint, otherwise, it discards the IDs that are greater than the midpoint. In $\log_2 N$ iterations, only one candidate ID will remain and it will be equal to the maximum ID found in the network.

3.3.5 Gathering and Distributing

Consider a network G with a designated leader λ known by all processors. Further, suppose that all processors know the entire topology of G. Let *layer s* be the set of processors that are exactly s hops away from λ. Fix an arbitrary $s>0$. Assume that each of the n_s processors in layer s has a unique message. In the *gathering task*, all of these messages must be received by λ. It is not necessary that these messages be received by all nodes in intermediate layers. In the *distributing task*, a message from λ must be received by all nodes in layer s. Similarly, it is not necessary that the message from λ be received by all nodes in intermediate layers (as in the broadcasting task). In [16], Gasieniec et al. describe algorithms that complete the gathering task using at most $2n_s+s-2$ slots and the distributing task using at most n_s+s-1 slots. We now present these two algorithms, in turn.

The gathering algorithm sends messages in a layer-by-layer fashion, from layer s down to layer 1. Consider the bipartite subgraph induced by the edges between the layers s and $s-1$. The idea is that all of the messages are passed from layer s to a subset L_{s-1} of processors in layer $s-1$ that dominate the processors in layer s. The processors in layer s are scheduled to transmit so that no collisions occur at processors in L_{s-1}. This is equivalent to colouring the processors in layer s so that every pair of processors that share a neighbour in L_{s-1} are coloured differently. This procedure is repeated to send messages from L_{s-1} to a set L_{s-2}, and so on, until all messages have reached L_1. Finally, each of the processors in L_1 transmits one at a time, so that all messages reach $L_0=\{\lambda\}$.

We now describe how the algorithm colours the processors and how it constructs the sets $L_{s-1},\dots,L_1$. These are performed completely locally, that is, no transmission slots are used when computing the colouring of processors or the sets $L_{s-1},\dots,L_1$. This is why such a strong assumption about topology knowledge is needed. Define L_s to be all processors in layer s. Fix a $j\in\{2,\dots,s\}$ and assume that L_j has already been constructed.

To colour the processors in L_j, we maintain two sets that are initially empty: S_j contains the processors that will transmit sequentially (i.e. they are all given different colours), while P_j contains the processors that will transmit in parallel during a single slot (i.e. they are all given the same colour). Consider the processor $p_a\in L_j\backslash(S_j\cup P_j)$ with smallest ID. Find the processor $p_b\in L_j\backslash(S_j\cup P_j)$ with smallest ID that shares a common neighbour with p_a in layer $j-1$. If no such p_b exists, then add p_a to P_j. Otherwise, add p_a and p_b to S_j. This process is repeated until $S_j\cup P_j=L_j$. Note that no two processors in P_j share a common neighbour in layer $j-1$.

The set L_{j-1} is constructed in tandem with the colouring procedure. To help with the description of the distributing algorithm later in this section, we will actually construct two sets, S'_{j-1} and P'_{j-1} (initially empty). When the construction of these sets is complete, their union will give us the set L_{j-1}. During the colouring procedure, whenever two processors p_a and p_b are added to S_j, if either p_a or p_b does not have a neighbour in S'_{j-1}, then the common neighbour of p_a and p_b in layer $j-1$ with smallest ID is added to S'_{j-1}. Whenever a processor p_a is added to P_j, if S'_{j-1} does not contain a neighbour of p_a, then the neighbour of p_a in layer $j-1$ with smallest ID is added to P'_{j-1}. Note that each processor in P_j has at most one neighbour in P'_{j-1}: for each processor added to P_j, at most one processor is added to P'_{j-1}, so, if some $p_a \in P_j$ has two neighbours in P'_{j-1}, then one of those neighbours was added to P'_{j-1} when some $p_b \neq p_a$ was added to P_j. This impossible since no two processors in P_j share a common neighbour in layer $j-1$.

Once the construction and colouring of L_j is complete, for all $j \in \{1,\ldots,s\}$, the transmission of messages is performed. More specifically, for $j = s,\ldots,2$, the processors in S_j transmit sequentially (from lowest ID to highest ID), then all processors in P_j transmit in parallel. Finally, all of the processors in L_1 transmit sequentially (from lowest ID to highest ID).

For arbitrary j, note that L_j was partitioned into S_j and P_j, so it follows that every processor in L_j has transmitted. When the processors in P_j transmit in parallel, no collisions occur in layer $j-1$ since no two of these processors have a common neighbour in layer $j-1$. Finally, note that each processor in L_j has at least one neighbour in $S'_{j-1} \cup P'_{j-1} = L_{j-1}$. A simple induction on j shows that the messages sent by processors in L_j are received by $L_0 = \{\lambda\}$.

The algorithm depends on processors knowing about edges between processors in different layers. It does not depend on knowledge about edges between processors in the same layer. This fact will be used in section 3.3.6.

To calculate the number of slots used by the algorithm, separately consider the slots for sequential transmissions and the slots for parallel transmissions. First, note that there is at most 1 slot of parallel transmissions for each layer j, which contributes s total slots to the running time. Next, within layer j, $|L_j| - |P_j| = |S_j|$ is the number of sequential slots. Since at most one processor was added to S'_{j-1} each time that two processors were added to S_j, it follows that $|S_j| \geq 2|S'_{j-1}|$. Furthermore, at most one processor was added to P'_{j-1} for each of the processors in P_j. So, $|L_{j-1}| = |P'_{j-1}| + |S'_{j-1}| \leq |P_j| + |S_j|/2 = (|L_j| + |P_j|)/2$. Hence, $|S_j| = |L_j| - |P_j| = 2|L_j| - (|L_j| + |P_j|) \leq 2|L_j| - 2|L_{j-1}|$. Summing over all j, the total number of sequential transmissions is at most $2|L_s| - 2 = 2n_s - 2$. Therefore, the algorithm uses at most $2n_s + s - 2$ slots.

Next, we consider the distributing algorithm. The idea is to "reverse" the flow of information that occurs in the gathering algorithm. Namely, we can create a directed graph from the gathering schedule in the following way: for all j, for each transmission by a processor p in L_j that is received by a processor q in L_{j-1}, add a directed edge (p,q) to the graph. The distributing algorithm reverses the direction of all of these edges, and schedules processors so that the message from λ is passed along all of them.

More specifically, for all $j = s,\ldots,0$, all processors compute the sets S'_j and P'_j as in the gathering algorithm above. The first transmission is by λ. Then, for each $j = 1,\ldots,s-1$: the processors in S'_j transmit sequentially (from lowest ID to highest ID), and then all processors in P'_j transmit in parallel.

Note that, at the end of each iteration j, all processors in $S'_j \cup P'_j = L_j$ have transmitted. When the processors in P'_j transmit in parallel, no collisions occur at processors in P_{j+1}, since, as we observed earlier, each processor in P_{j+1} has at most one neighbour in P'_j. Finally, recall that each processor in L_{j+1} has a neighbour in $S'_j \cup P'_j$. A simple induction on j shows that, at the end of an iteration j, all processors in L_{j+1} have received the message transmitted by λ.

To calculate the number of slots used by the distributing algorithm, separately consider the slots for sequential transmissions and the slots for parallel transmissions. First, note that there is at most 1 slot of parallel transmissions for each layer $j \in \{1,\ldots,s-1\}$, which contributes $s-1$ total slots to the running time. Next, recall that, for $j \in \{1,\ldots,s\}$, $|S_j| \geq 2|S'_{j-1}|$ and $|S_j| = 2|L_j| - 2|L_{j-1}|$, so $|S'_{j-1}| \leq |S_j|/2 = |L_j| - |L_{j-1}|$. Summing over all $j = 2,\ldots,s$, we get that, in the distributing algorithm, at most $|L_s| - |L_0| = n_s - 1$ sequential slots are used by processors in layers $0,\ldots,s-1$. Adding the $s-1$ slots for parallel transmissions, and 1 slot for λ's transmission, we get that the number of slots used by the distributing algorithm is at most $n_s + s - 1$.

3.3.6 Learning the Topology of a Connected Network of Processors with Similar Degrees

In [16], Gasieniec et al. describe an algorithm where each processor in a connected network learns the complete topology of the network in $O(n \log n \log^2 N)$ slots. It is assumed that there is an integer $d \geq \sqrt{n}$ (known by all processors in the network) such that:

- each processor has degree at least d or has a neighbour with degree at least d,
- each processor that has degree greater than $4d$ initially knows the identity of its neighbours.

The first part consists of choosing a leader λ for the network, using the $O(n \log n \log^2 N)$-slot algorithm described in section 3.3.4. The processors exactly k hops away from λ are referred to as *layer k*, and let n_k denote the number of processors in layer k. The set of processors at most k hops away from λ is referred to as H_k. The second part of the algorithm, which takes $O(n \log^2 n + n \log N)$ slots, proceeds in phases: during each phase $k > 0$, each processor in H_k learns the complete topology of the subgraph induced by H_k. The algorithm terminates when no new information is learned by λ during a phase, as the complete topology has been learned. Therefore, if diameter of the network is D, the number of phases is bounded above by $D+1$. Each phase consists of several stages, which we now describe in more detail.

1. *each processor in layer k learn that they are in layer k, learns the identity of all of its neighbours in layer $k-1$, and learns the topology of the subgraph induced by H_{k-1}*

   At the beginning of phase k, all processors in layer $k-1$ know the complete topology of the subgraph induced by H_{k-1}. More specifically, all processors in layer $k-1$ know the identity of all other processors in layer $k-1$. Therefore, they can transmit, in turn, from lowest ID to highest ID, so that each processor in layer k has heard from their neighbours in layer $k-1$. Each transmission contains the transmitter's ID and the entire topology of the subgraph induced by H_{k-1}.

2. *the leader λ and processors in layers $k-1$ and k learn about all of the edges between layers $k-1$ and k*

   This stage consists of $O(\log d)$ iterations. This stage can be viewed as incrementally discovering new edges between layers $k-1$ and k. Processors in layer k that have been discovered in an iteration j do not participate in future iterations.

   At the beginning of iteration 0, the processors in layer $k-1$ with degree greater than $4d$ know about their neighbours in layer k. Let E_0 be the set of these known edges between layers $k-1$ and k. The set E_0 is gathered at λ using the procedure described in section 3.3.5 with $s = k-1$, which takes $2n_{k-1} + k - 1$ slots. Then, E_0 is distributed to all of the processors in layer $k-1$ using the procedure described at the end of section 3.3.5 with $s = k-1$. Finally, the processors in layer $k-1$ transmit, one at a time, in increasing order by ID, so that all processors in layer k receive E_0.

   In subsequent iterations $i > 0$: each processor in layer $k-1$ with degree at most $4d$ receives a transmission from at least half of their undiscovered neighbours in layer k (the details are provided below). Each such transmission from a processor in layer k contains a list of all of its edges to processors in layer $k-1$ (which it learned in stage 1). Let E_i be the set of all of these learned edges between layer $k-1$ and k. Then, as above, the set E_i is gathered at λ, distributed to all of the processors in layer $k-1$, and then sent to all processors in layer k. The processors in layer k that appear in E_i have been discovered in iteration i, so they do not participate in future iterations. Note that, at the end of iteration i, all processor in layers $k-1$ and k have learned about the edges in $\bigcup_{m=0}^{i} E_m$.

We now go back and consider the first step of each iteration $i > 0$. In this step, a linearly $(4d/2^{i-1})$-selective family is scheduled, but only the undiscovered processors in layer k transmit. From section 3.3.2, this takes $O((d/2^i)\log N)$ slots. From section 3.3.2, processors with at most $(4d/2^{i-1})$ transmitting neighbours in layer k will learn about at least half of these neighbours. Since processors that are discovered before iteration i do not participate in iteration i, a straightforward induction argument shows that, at the beginning of iteration i, each processor in layer $k-1$ has at most $(4d/2^{i-1})$ undiscovered neighbours in layer k.

After iteration $i = \log_2 d + 3$ of this stage, all of the edges between layers $k-1$ and k are known by λ and all processors in layers $k-1$ and k. Each iteration i takes $O((d/2^i)\log N + n_{k-1} + k)$ slots, so the entire stage takes $O(d\log N + n_{k-1}\log n + k\log n)$ slots.

3. *all processors in H_{k-1} learn about all of the edges between layers $k-1$ and k*

   In the previous stage, λ learns about all of the edges between layers $k-1$ and k. Also, we assumed that, at the beginning of phase k, all processors in H_{k-1} know the entire topology of the subgraph induced by H_{k-1}. So, using a $O(k\log^2 n)$ broadcasting algorithm from [23] for networks where the entire topology is known, λ sends the list of all edges between layers $k-1$ and k to all processors in the subgraph induced by H_{k-1}.

4. *each processor in layer k learns which nodes in layer k are its neighbours and sends this information to λ*

   Each processor in layer k knows the identity of all other processors in layer k, so they can transmit, in turn, from lowest ID to highest ID. The neighbours of a node in layer k are those nodes from which receives a transmission during this process. Using the procedure described in section 3.3.5, the information about edges between processors in layer k can be gathered at λ. The total number of slots used in this stage is $3n_k + k - 2$.

5. *the entire topology of the subgraph induced by H_k is sent by λ to all processors in H_k*

   By the end of the previous stage, λ has learned the entire topology of the subgraph induced by H_k. This information is sent to all processors in H_{k-1} using the $O(k\log^2 n)$ broadcasting algorithm from [23] for networks where the entire topology is known. Then, the processors in layer $k-1$ transmit, in turn, from lowest ID to highest ID, to send the entire topology of the subgraph induced by H_k to all processors in layer k.

The number of slots used in phase k is $O(d\log N + n_{k-1}\log n + n_k + k\log^2 n)$. Summing over all phases $k = 1,\dots,D+1$, we get $O(dD\log N + D^2\log^2 n + n\log n)$ slots. The authors prove that $D = O(n/d)$, which, along with the fact that $d \geq \sqrt{n}$, gives the desired $O(n\log N + n\log^2 n)$ upper bound. Therefore, the leader selection step of the algorithm dominates the running time, giving an upper bound of $O(n\log n\log^2 N)$.

3.3.7 The Neighbourhood Learning Algorithm

Gasieniec et al. [16] provide an algorithm in which all processors learn their complete neighbourhood in $O(n\log^2 N\log^2 n)$ slots. The value of n is known, however the algorithm can be easily adapted to the case where only a linear upper bound on n is known. The algorithm proceeds in three stages:

1. Each processor p gets an estimate $\tilde{\Delta}_p$ of its degree, Δ_p, such that $\frac{1}{2}\Delta_p \leq \tilde{\Delta}_p \leq \Delta_p$.
2. Each small-degree processor (i.e. $\Delta_p < \lceil\sqrt{n}\rceil$) learns about all of its neighbours.
3. Each large-degree processor (i.e. $\tilde{\Delta}_p \geq \lceil\sqrt{n}\rceil$) learns about all of its neighbours.

In stage 1, running a linearly n-selective family on all processors ensures that each processor learns about at least half of its neighbours. This takes $O(n\log N)$ slots (see Section 3.3.2). In stage 2, running a strongly $\lceil\sqrt{n}\rceil$-selective family on all processors ensures that each small-degree processor receives a message from each of its neighbours. This takes $O(n\log N)$ slots (see Section 3.3.2).

In stage 3, the algorithm actually achieves something quite strong: in the subgraph induced by $nbrhd(\{p \mid \tilde{\Delta}_p \geq \lceil\sqrt{n}\rceil\})$, each processor learns the entire topology of the connected component in which it is contained. This is achieved using $O(n \log^2 N \log^2 n)$ slots. We now outline the main steps of this stage. We denote by $nbrhd(P)$ the union of all neighbourhoods of processors in P, that is, $nbrhd(P) = \bigcup_{p \in P} nbrhd(p)$.

Let $P_i = \{p \mid \tilde{\Delta}_p \geq 2^i \lceil\sqrt{n}\rceil\}$. First, for each $i \in \{0, \dots, \frac{1}{2}\log n\}$, each processor learns whether or not it belongs to $nbrhd(P_i)$. More specifically, for each $i \in \{0, \dots, \frac{1}{2}\log n\}$ in turn, the processors in P_i run an n-selective family. This is sufficient since, when the processors P_i run an n-selective family, a processor q with $\tilde{\Delta}_q < 2^i \lceil\sqrt{n}\rceil$ receives a message if and only if $q \in nbrhd(P_i)$. Running these $O(\log n)$ selective families takes $O(n \log N \log n)$ slots.

The remainder of the algorithm consists of phases $i = \frac{1}{2}\log n - 1, \dots, 0$. In phase i, each processor in $nbrhd(P_i)$ learns the entire topology of its connected component of the subgraph induced by $nbrhd(P_i)$. The construction of this topology occurs inductively, taking advantage of the fact that $nbrhd(P_i) \supseteq nbrhd(P_{i+1})$. Phase i consists of the processors in $nbrhd(P_i)$ running the algorithm described in Section 3.3.6 with $d = 2^i \lceil\sqrt{n}\rceil \geq \sqrt{n}$. Recall that the algorithm assumes that:

1. each processor has degree at least d or has a neighbour with degree at least d,
2. each processor that has degree greater than $4d$ initially knows the identity of its neighbours.

Assumption 1 is satisfied since, for each $p \in P_i$, $\Delta_p \geq \tilde{\Delta}_p \geq 2^i \lceil\sqrt{n}\rceil$. Assumption 2 is satisfied since any processor with degree greater than $4d = 2^{i+2}\lceil\sqrt{n}\rceil$ has estimated degree greater than $\frac{1}{2}(2^{i+2}\lceil\sqrt{n}\rceil) = 2^{i+1}\lceil\sqrt{n}\rceil$, so it belongs to P_{i+1}, and hence learned about its neighbours in the previous phase. After phase $i = 0$, each processor in P_0, i.e., each processor with $\tilde{\Delta}_p \geq 2^0 \lceil\sqrt{n}\rceil$, has learned about all of its neighbours, as required. The algorithm from Section 3.3.6 uses $O(n \log n \log^2 N)$ slots, and there are $\frac{1}{2}\log n$ phases, which means that stage 3 uses $O(n \log^2 n \log^2 N)$ slots.

4 Conclusions

For the task of complete neighbourhood learning, we have seen that there does not exist a collision-free deterministic algorithm that does significantly better than the round-robin upper bound of N slots. Since, in general, N can be much larger than the number of processors in the network, this is quite a strong negative result. However, we have also seen that combinatorial objects such as selectors and selective families can be used to design algorithms that perform drastically better than round-robin. Our hope is that these objects (or similar ones) can be further exploited to solve open questions for the neighbourhood learning task.

To our knowledge, the lower bound for collision-free algorithms presented in this paper is the only non-trivial bound in our model. An interesting open problem is finding non-trivial lower bounds for more general classes of algorithms that solve complete neighbourhood learning.

The complete neighbourhood learning task is completely open for various models of mobile ad hoc networks. In a model where processors travel along continuous trajectories with bounded speed on the line or plane, Ellen et al. [24] have created an algorithm that maintains neighbourhood information, but they leave open the problem of initially acquiring this information. We do not know of any algorithms for complete neighbourhood learning in this model (though one has to carefully specify the problem when neighbourhoods can change during the execution of the algorithm). We know of one result by Cornejo et al. [25] that solves a small part of the task. Their algorithm assumes that there exists an underlying MAC layer protocol that handles collisions and guarantees that messages be delivered within a known bounded amount of time.

Another interesting model of mobile networks is the time-varying graph model [26]. In this model, the vertices of the graph represent network processors and the labelled edges of the graph represent time intervals during which the endpoints of the edge are within communication range (during such an interval, the edge is said to be *activated*). However, this graph is only available from the global view of the network. Processors must run local algorithms that do not know which edges will be activated, when they will be activated, nor for how long. In a particular class of these graphs, the edges are activated at periodic intervals, which can be useful for modelling systems where processors repeat the same trajectory in a loop (such as low-orbiting satellites or bus routes). Complete neighbourhood learning, defined as each processor learning the period of activation and length of activation of all incident edges, is an interesting open problem in this class of time-varying networks. After neighbourhood learning is complete, the processors would be able to anticipate when and for how long each incident edge will be activated. This information would be very useful when solving other problems, such as routing.

Acknowledgements

The author wishes to thank his advisor, Faith Ellen, whose helpful insights and guidance led to a vastly improved presentation of this research. Also, the author appreciates the valuable feedback from the anonymous reviewers of this paper. The author was supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Li, L., Halpern, J.Y., Bahl, P., Wang, Y., Wattenhofer, R.: Analysis of a cone-based distributed topology control algorithm for wireless multi-hop networks. In: Proceedings of ACM Symposium on Principle of Distributed Computing (PODC). (2001) 264–273

[2] Ramanathan, R., Hain, R.: Topology control of multihop wireless networks using transmit power adjustment. In: Proceedings of IEEE INFOCOM. (2000) 404–413

[3] Wattenhofer, R., Li, E.L., Bahl, P., Wang, Y.: Distributed topology control for wireless multihop ad-hoc networks. In: Proceedings of IEEE INFOCOM. (2001) 1388–1397

[4] Choudhury, R.R., Vaidya, N.H.: Impact of directional antennas on ad hoc routing. In: Proceedings of Personal and Wireless Communications. (2003) 590–600

[5] Johnson, D.B., Maltz, D.A.: Dynamic Source Routing in Ad Hoc Wireless Networks. In: Mobile Computing. Kluwer Academic Publishers (1996) 153–181

[6] Chlebus, B.S., Gąsieniec, L., Gibbons, A., Pelc, A., Rytter, W.: Deterministic broadcasting in unknown radio networks. In: Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). (2000) 861–870

[7] Bao, L., Garcia-Luna-Aceves, J.J.: Transmission scheduling in ad hoc networks with directional antennas. In: Proceedings of ACM MOBICOM. (2002) 48–58

[8] Choudhury, R.R., Yang, X., Vaidya, N.H., Ramanathan, R.: Using directional antennas for medium access control in ad hoc networks. In: Proceedings of ACM MOBICOM. (2002) 59–70

[9] Krishnamurthy, S., Thoppian, M.R., Kuppa, S., Chandrasekaran, R., Mittal, N., Venkatesan, S., Prakash, R.: Time-efficient distributed layer-2 auto-configuration for cognitive radio networks. Computer Networks **52**(4) (2008) 831–849

[10] Kautz, W., Singleton, R.: Nonrandom binary superimposed codes. IEEE Transactions on Information Theory **10**(4) (1964) 363–377

[11] Du, D., Hwang, F.: Combinatorial Group Testing and Its Applications. 2nd edn. World Scientific (2000)

[12] Keshavarzian, A., Uysal-Biyikoglu, E., Herrmann, F., Manjeshwar, A.: Energy-efficient link assessment in wireless sensor networks. In: Proceedings of IEEE INFOCOM. Volume 3. (2004) 1751–1761

[13] Brouwer, A.E., Shearer, J.B., Sloane, N.J.A., Smith, W.D.: A new table of constant weight codes. IEEE Transactions on Information Theory **36** (1990) 1334–1380

[14] Chee, Y.M., Xing, C., Yeo, S.L.: New constant-weight codes from propagation rules. IEEE Transactions on Information Theory **56** (2010) 1596–1599

[15] Smith, D.H., Hughes, L.A., Perkins, S.: A new table of constant weight codes of length greater than 28. Electronic Journal of Combinatorics **13** (2006)

[16] Gąsieniec, L., Pagourtzis, A., Potapov, I., Radzik, T.: Deterministic communication in radio networks with large labels. Algorithmica **47**(1) (2007) 97–117

[17] Chrobak, M., Gąsieniec, L., Rytter, W.: Fast broadcasting and gossiping in radio networks. Journal of Algorithms **43** (2002) 177–189

[18] Indyk, P.: Explicit constructions of selectors and related combinatorial structures, with applications. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). (2002) 697–704

[19] Chlebus, B., Kowalski, D.: Almost optimal explicit selectors. In: Proceedings of Fundamentals of Computation Theory. (2005) 270–280

[20] Porat, E., Rothschild, A.: Explicit non-adaptive combinatorial group testing schemes. In: Proceedings of International Colloquium on Automata, Languages and Programming (ICALP). (2008) 748–759

[21] Füredi, Z.: On r-cover-free families. Journal of Combinatorial Theory, Series A **73**(1) (1996) 172–173

[22] Clementi, A., Monti, A., Silvestri, R.: Distributed broadcast in radio networks of unknown topology. Theoretical Computer Science **302** (2003) 337–364

[23] Chlamtac, I., Weinstein, O.: The wave expansion approach to broadcasting in multihop radio networks. IEEE Transactions on Communications **39**(3) (1991) 426–433

[24] Ellen, F., Subramanian, S., Welch, J.: Maintaining information about nearby processors in a mobile environment. In: Proceedings of International Conference on Distributed Computing and Networking (ICDCN). (2006) 193–202

[25] Cornejo, A., Viqar, S., Welch, J.: Reliable neighbor discovery for mobile ad hoc networks. In: Proceedings of the 6th International Workshop on Foundations of Mobile Computing (DIALM-POMC). (2010) 63–72

[26] Casteigts, A., Flocchini, P., Mans, B., Santoro, N.: Deterministic computations in time-varying graphs: Broadcasting under unstructured mobility. In: IFIP International Conference on Theoretical Computer Science. (2010) 111–124