**January, 1991**

LIDS-P 2050

**Research Supported By:**

Vinton Hayes Fellowship

# Finding Small Dominating Sets in Stationless Mobile Packet Radio Networks

Parekh, A.K.J.

LIDS-P-2050

# Finding Small Dominating Sets in Stationless Mobile Packet Radio Networks

Abhay K. Parekh [1]
Laboratory for Information and Decision Systems
Massachusetts Institute of Technology
parekh@lids.mit.edu

**Abstract**

We model a snapshot of a mobile packet radio network as an undirected graph. The nodes of the graph are processors, that communicate along their incident edges by broadcast. The radios do not know the size of the network, and start out with no topological information. Our goal is to select a small subset of the nodes of the graph so that every other node is adjacent to at least one node in the subset. Such a subset is called a dominating set. Finding the smallest dominating set of a graph is known to be NP-hard. An event driven distributed algorithm is presented that picks a dominating set of size at most $N - \sqrt{2M+1}$, for any network with $N$ nodes and $M$ links. Since communication is by broadcast, the messages cannot be exclusively routed on subgraphs such as spanning trees. For the class of $\delta$-regular graphs of diameter $d$, it takes $O(N\delta^d)$ messages to learn the entire topology of the graph. We show that for graphs of regular degree, $\delta$, our algorithm has communication complexity of $O(|D_g|\delta^3 + N\delta)$, where $D_g$ is the dominating set picked by the algorithm. The time complexity is $O(|D_g|)$. Thus the algorithm is efficient for graphs with diameter greater than 3.

**Key Words:** Distributed Algorithms, Mobile Communications, Dominating Sets, Approximation Algorithms.

---

[1]Room 35-303, MIT, Cambridge MA 02139. This research was partially funded by a Vinton Hayes fellowship.

# 1 Introduction

Fixed stations are often used to co-ordinate the broadcasts of the users of a mobile radio network. For example in cellular telephone networks, all calls are relayed through base stations local to the calling parties. The aim of this co-ordination is to minimize collisions in the multiaccess channel, which occur when a radio receives several packets at overlapping times. Since the range of the radios is limited, a message may have to be relayed over several hops before it reaches its destination, and the stations play a central role in achieving this function as well.

In some networks, however, such as those formed by battleships or tanks, these stations may not be present, and the co-ordination must be performed by the users themselves. One way to accomplish this is to select certain "strategically located" users as stations or leaders. Every other radio is assigned to one of these leaders so that the network is partitioned into clusters. The leaders are selected so that if a radio, $r$ is in a cluster with leader $l$, then $r$ and $l$ are within range of each other. This architecture was first proposed in a paper by Baker and Ephridemes [1]. They also proposed a synchronous distributed algorithm for selecting cluster leaders, but their scheme can result in a large number of clusters being created—sometimes almost all of the nodes are selected as leaders. However, it is desirable to minimize the the number of clusters, since having many stations lessens the extent to which co-ordination can occur (for example, if all the users become stations there are no users left to co-ordinate). Typically, users that can communicate with *many* other users i.e. those with high connectivity, should be selected.

Since the users are mobile, this selection of stations has to be performed periodically, to reflect the changing topology of the network. In [6], a multiaccess scheme is proposed that adapts to small changes in the topology, and that does not have to be re-run in its entirety. The scheme is token-based, and *all* the users of the network share a single token. However, when radio ranges are small, there may be many pairs of radios that can broadcast simultaneously without causing collisions. Thus, in such scenarios, the token passing scheme can lead to poor utilization of the channel.

Since the operability of any given user can not be assured (especially in a tactical scenario!), the scheme should be distributed in nature. Further, it is desirable to limit a node's knowledge of the topology to those radios that close to it, i.e. those currently within a few hops. In addition, synchronization requirements should be minimal.

To summarize, the protocol should ensure that every user is assigned a station, and that there are not too many stations. It should be truly distributed in that every node should run the same algorithm, and be driven by events rather than by time-outs. In this paper

we will describe a selection scheme with these properties. The scheme is efficient in terms of number of messages broadcast, when that the network is *sparsely connected.*

We model a snapshot of the network as a finite unconnected undirected graph, $G(V, E)$, with $N$ nodes and $M$ edges. Every node is a user, has a unique identity number, and runs the same protocol as every other user. An edge is defined for a node pair (i,j) if and only if $i$ and $j$ can exchange messages without going through any other processor(s). Nodes broadcast messages to all of their neighbors. Messages arriving at a node are queued, and then processed on a first come first serve basis. The delay across any edge is variable but bounded by $t_{max}$, and messages are assumed to be received along a direction of an edge in the order that they were transmitted. Errors in transmission are not considered i.e. it is assumed that there are underlying acknowledgement schemes. Other lower level protocols such as error detection and message formatting are also assumed. Finally, while the goal of creating the clusters is to minimize collisions, an additional level of multiaccess control is required to facilitate the protocol that establishes these stations in the first place. A number of suitable multiaccess schemes have been proposed in the literature, for example see Chapter 4 of [2]. We assume that such a scheme exists and disregard the effects of collisions in our protocol. An important property of the protocol will be to limit the total number of messages to traverse the edges of the graph. Our model is a variation of those presented in [4] and [10].

In terms of the model, we want to find a set of nodes such that every node not in the set is adjacent to at least one node in the set. Such a set is known in graph theoretic terminology, as a *dominating* set (DS). Note that the set $V$ is an example of a dominating set, but one that we would not want to pick. In this paper we concentrate on minimizing the cardinality of the dominating set, i.e, on picking as few stations as possible. Such dominating sets are called minimum dominating sets (MDS). The cardinality of the MDS is called the *domination number* of the graph. Unfortunately, finding the domination number of a graph is a difficult problem, and the decision version of the problem is NP-complete [5]. Thus, our goal changes to finding good (distributed) heuristics to the MDS problem. There are many ad-hoc schemes that one can devise, but understanding how these schemes perform relative to the domination number is much more difficult. We will focus on a simple greedy heuristic that can be parallelized in an asynchronous model, and for which we can demonstrate that the dominating sets picked are not too large. We note that for the protocol described here to be practical, it must incorporate the function of *linking the clusters* together for inter-cluster communication. However, we believe that this function can be accommodated in our protocol with some additional work.

The rest of the paper is organized as follows: A sequential greedy heuristic, *Greedy*, is

presented and analyzed in Section 2. In Section 3, we present the distributed version of this heuristic, $DISTG$. The correctness of $DISTG$ is established in Section 4, and in Section 5 the complexity of $DISTG$ analyzed. Finally, pseudo PASCAL code for $DISTG$ is given in an appendix.

## 2 The *Greedy* heuristic

Number the nodes of a given graph G(V,E) from 1 to $|V|$. Let the neighbors of a node $i$, be denoted $N(i)$, and let $\bar{N}(i) = \{i\} \cup N(i)$. We sometimes refer to $\bar{N}(i)$ as the *closed neighborhood* of $i$. Also, for any set $S \subset V$, let $N(S) = \cup_{j \in S} \bar{N}(j)$.

Define a node to be "covered" in the beginning of an iteration, if either it or one of its neighbors is in the set of nodes picked by the heuristic so far. Initially, no nodes are covered. For the $k^{th}$ iteration, let the uncovered neighbors of node $i$, including $i$ if it is uncovered, be $N_k^u(i)$. So if $U_k$ is the set of uncovered nodes at the beginning of the $k^{th}$ iteration, then $N_k^u(i) = \bar{N}(i) \cap U_k$. Now for every iteration $k$ define $i_c$ to be least numbered neighbor of $i$ (it may be covered or uncovered) such that:

$$|N_k^u(i_c)| = \max_{j \in \bar{N}(i)} |N_k^u(j)|.$$

What this means is that $i_c$ is the most highly connected node (in the current iteration, $k$), that $i$ can exchange messages with directly. Think of $i_c$ as node $i$'s candidate for election in the current iteration. Finally, define a node, $i$ as *Elected* if it is the most highly connected neighbor of all of the nodes in $N^u(i)$, i.e., $i$ is *Elected* in iteration $k$ if

$$i = j_c, \quad \forall j \in N_k^u(i).$$

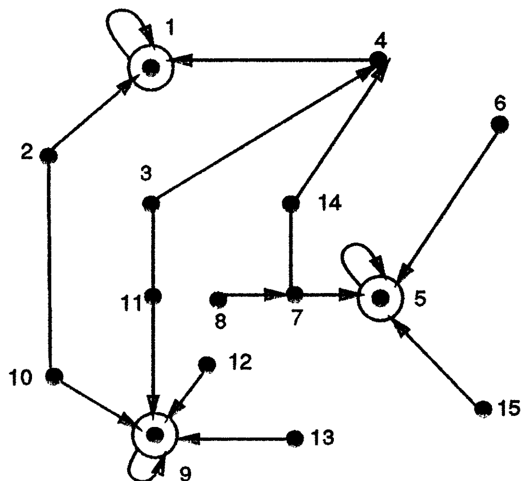
So, an *Elected* node is the candidate for *every* one of its uncovered neighbors.

Now *Greedy* is defined as follows:

> In each iteration, put all *Elected* nodes in the dominating set, and end when all the nodes are covered.

In the 15 node example of Figure 1, notice that nodes 1, 5, and 9 are *Elected* in the first iteration, after which only nodes 8 and 14 remain uncovered. They both choose node 7 in iteration 2, making node 7 *Elected.* Thus the DS picked by greedy is $\{1, 5, 7, 9\}$. *Greedy* terminates after this because all the nodes are covered.

Note that *Greedy* is sequential, but may pick more than one node for the dominating set in a single iteration. Our distributed algorithm will parallelize *Greedy.* However, before




An arrow from node $j$ to $i$ indicates that $j_c = i$. *Elected* nodes are circled.

Figure 1: The first iteration of *Greedy* on a 15 node graph.

presenting the algorithm, we would like to examine if *Greedy* picks small dominating sets relative to the domination number.

Compare *Greedy* to an even simpler sequential scheme $SG$ (for Simpler Greedy), that is the following: At each iteration put into the dominating set the least numbered node $i$, that has the largest number of uncovered neighbors (including itself if it is uncovered). Stop when all nodes are covered. The reason we are interested in $SG$ is that it picks only one node per iteration, and is thus easier to analyze. Notice that it takes four iterations to terminate on our example above. The nodes picked to be included in the dominating set are 9, 1, 5 and 7, in that order. This DS is identical to the one picked by *Greedy*. We now show that this is true for all graphs:

**Theorem 1** *If $D_{sg}$ and $D_g$ are dominating sets returned by $SG$ and Greedy for a graph, $G(V, E)$, then $D_{sg} = D_g$.*

**Proof.** For any graph $G$, with a numbering on its nodes, both $SG$ and *Greedy* have unique solutions. Let $D_{sg} = \{g_1, g_2, ...g_{|DS_g|}\}$, where $g_k$ is the node picked in the $k^{th}$ iteration of $SG$. Define $S_i$ to be the set of uncovered nodes covered by $g_i$ when it is picked. Let $|S_i| = m_i$.

Now let $T_k$ be the set of nodes picked in iterations $1, 2, ..., k$ of *Greedy*. Observe that 2 nodes cannot be picked in the same iteration of *Greedy* if they have any uncovered nodes in

the intersection of their neighborhoods. If $i \in D_g$, then define $C_i$ to be the set of uncovered nodes covered by $i$ when it is picked. Clearly the $C_i$'s are mutually disjoint.

We show the following by induction on $k$, the number of iterations of *Greedy*:

(i) $T_k \subset D_{sg}$

(ii) $\{g_1, ..., g_k\} \subset T_k$

(iii) If $g_i \in T_k$, then $S_i = C_{g_i}$.

Conditions (i) and (ii) are sufficient to show the Theorem.
Basis $k = 1$: Trivially, $g_1 \in T_1$. This satisfies condition (ii). Now suppose that there exists $i \in T_1$, for some $i \neq g_1$. Clearly, $i \notin S_1$. Suppose $i$ is covered in iteration $w$ of $SG$. Then $g_w$ is the first node in $\bar{N}(i)$ to be selected by $SG$. Since $i$ is *Elected* in the first iteration of *Greedy*, it must be the least numbered node of maximum degree in $\bar{N}(i)$. Therefore, $g_w = i$. We also have that $S_w = C_i$. Thus conditions (i) and (iii) are also satisfied.
*Inductive Step:* $k = K + 1$: We first show that condition (ii) is satisfied. By the induction hypothesis, $T_K$ properly contains $g_1, ..., g_k$, and so we need only consider the case $g_{K+1} \notin T_K$. By conditions (i) and (iii) of the hypothesis, we know that all of the nodes in $S_{K+1}$ must be uncovered at the beginning in the $K + 1^{st}$ iteration of *Greedy*. Then since $\{g_1, ..., g_K\} \subset \cup_{i=1}^{K} T_i$, it follows that $g_{K+1}$ is *Elected*, and therefore *Greedy* picks it in iteration $K + 1$.

Now we show the validity of conditions (i) and (iii): Suppose there exists $i \neq g_{K+1}$, which is *Elected* in the $K + 1^{st}$ iteration of *Greedy*. By the induction hypothesis, if $g_w \in T_K$, then $S_w = C_{g_w}$. Also, $T_K \subset D_{sg}$. Thus, none of these nodes in $N^u_{K+1}(i)$ is covered in the first $k$ iterations of $SG$. Suppose $i$ is covered in iteration $\omega$ of $SG$. Then $g_\omega$ is the most highly connected neighbor of all the nodes in $N^u_{K+1}(i)$. But $i$ is Elected in iteration $K + 1$ of *Greedy* and so $g_\omega = i$. This shows condition (i). Also, the set of uncovered nodes covered by $i$ when it is picked by $SG$ is exactly $N^u_{K+1}(i)$, which satisfies condition (iii). □

$SG$ is an analog of a greedy algorithm that has been analyzed by Chvatal [3] and others [7], [8] for finding small set covers. The focus there has been on comparing the cardinality of the set cover returned by the algorithm to that of the smallest set cover, in the worst case. Since any dominating set problem can be formulated as a set covering problem, the results for the set covering algorithm can be specialized to our problem. A result almost directly obtained from the work of Chvatal [3] is that

$$\frac{|D_{sg}|}{D_o} \leq \sum_{i=1}^{(\delta+1)} \frac{1}{i}, \tag{1}$$

where $\delta$ is the maximum degree of a node in the graph, and $D_o$ is domination number of

the graph.

We have also shown elsewhere [9] that the following upper bound on $D_o$ due to Vizing [11], applies to $|D_{sg}|$ as well:

$$|D_{sg}| \leq N + 1 - \sqrt{2M + 1}. \quad (2)$$

Since $|D_g| = |D_{sg}|$, the bounds of (1) and (2) apply to $D_g$ as well. We now present the event driven distributed algorithm that parallelizes *Greedy*.

## 3 The Distributed Greedy Algorithm *DISTG*

A naive implementation of *Greedy* would be for every node to discover the topology of the graph and then to compute the dominating set for itself. However, since the nodes have to *broadcast* all their messages, this might result a very large number of messages to be sent. The problem is that it is not possible to restrict communication to proceed along subgraphs such as spanning trees. The distributed algorithm presented in this section only requires a node $i$, to ascertain $\bar{N}(N(i))$.

*DISTG* consists of two phases. In the first phase, every node determines its neighbors and its neighbors' neighbors. It takes two broadcasts (per node) to do this: In the first broadcast every node, $i$ sends out its identity, and in the second broadcast it sends out the identities of all its neighbors. The phase terminates since the link delays are bounded by $t_{max}$.

The second phase of DISTG is entirely event-driven in its communication–there are no time outs. The algorithm is conducive to a high degree of parallelism—different parts of the network may be in entirely different stages at the same (real) time.

A node starts Phase 2 of the algorithm with the following knowledge:

(i) Its unique identity number.

(ii) The identities of its neighbors and its neighbors' neighbors.

Note that a node does not know the total number of nodes.

The nodes that are assigned to a particular station are said to be in its cluster, and the station itself is called the cluster head or cluster leader. Nodes that elect a cluster head become part of that node's cluster. As in *Greedy*, all the uncovered nodes in $\bar{N}(i)$ must vote for $i$, for $i$ to become a cluster head. A key aspect of DISTG is synchronization so that *only* nodes that would be *Elected* in *Greedy* are selected. The reader is urged to keep this fact in mind while reading explanations of the various subroutines in the next section.

### 3.1 Data Structures

The following data structures are at each node. For notational purposes we focus on a particular node, $i$:

Connectivity: a list of edges that describes $\bar{N}(N(i))$.
Statuslist: a list that contains tuples (j,status) for all $j \in \bar{N}(N(i))$, where a node's status can be one of FREE, HEAD, or COVERED depending on whether it is uncovered, a cluster leader, or covered but not a cluster head. Notationally, we refer to the status of $j$ as j.status.
Myleader: the identity of the cluster leader of $i$.
Myvote: the lowest indexed node of $\bar{N}(i)$ that has a maximum cardinality set of FREE neighbors. (This is $i_c$ described earlier.)
Votelist: a list that contains tuples (j,vote) for all FREE neighbors j of $i$, such that j.vote is $j$'s most recently communicated vote to $i$.
Uplist: a list of tuples such that $(a, b) \in$ Uplist if necessary changes in information on node $b$ which are caused by the declaration of $a$ as a leader, have been made at node $i$.
Wait: a list of nodes such that $a \in$Wait iff there is at least one node in N(i) from which new information is likely to be received because of the declaration of node $a$ as a cluster head.
3Sent: a list of nodes such that $l \in$ 3Sent only if $i$ is 3 hops away from a cluster head, $l$, and $i$ has broadcast the necessary information pertaining to this fact.

### 3.2 Initialization

Phase 2 of DISTG is now presented. Again, we focus on how it works at a node, $i$. The initialization at the end of phase 1 as follows:
Wait = $\phi$ Myleader= $i$;
status.j=FREE $\forall j \in \bar{N}(N(i))$;
Uplist, Votelist=$\phi$;
3Sent=$\phi$;
Send($Myvote$);

In the next few sections we describe the algorithm in words and figures.The actual code (written in pseudo PASCAL) is in Appendix A. We can view the algorithm as a protocol executed at every node, so in our description we focus on a particular node, $i$. Messages received from $i$'s neighbors trigger the execution of various subroutines of the protocol. When a node declares itself to be a leader, it broadcasts a **Type 0** messages, and in general, a message triggered by the receiving of a **Type k-1** message is a **Type k** message.

### 3.3 Trying to become a Leader

As the values of Myvote are received from neighbors, Votelist is updated. Once values of Myvote have been received from all neighbors, the procedure *TryHead* is executed: Node $i$ checks to see if it is the choice of all its neighbors. If so, it declares itself a cluster leader, changes its status to HEAD, and terminates the algorithm after broadcasting the change of status to its neighbors. It also broadcasts N(i); we will see why this is done later, in our discussion of the routines *TwoAway* and *OneThreeAway*. The message broadcast by $i$ as a result of declaring itself a leader, is called a **Type 0** message. *TryHead* is also executed, as shown later, in response to various messages. In general, $i$ must be elected by all of its FREE neighbors.

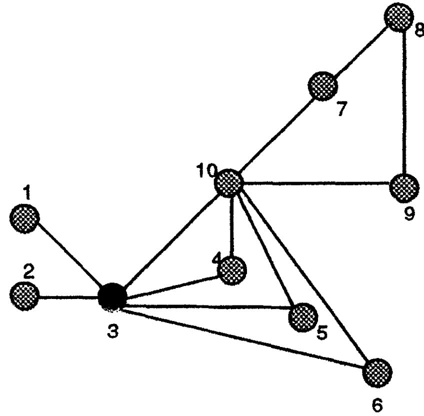
### 3.4 What to do if a neighbor declares itself a Leader

Let us now consider the response of $i$ to the message that one of its neighbors, $j$, is a head, i.e., if $i$ receives a Type 0 message from $j$. If node i.status = FREE it must change this to COVERED, and in addition Statuslist, Votelist, and Uplist must be modified to record all changes in $N(i)$ due to $j$'s declaration. In other words, $i$ must recompute its set of FREE neighbors. Since $i$ has (in general) fewer FREE neighbors now, it may no longer be the most highly connected neighbor of any other node. Thus, $i$ must pass on the fact of $j$'s declaration to its neighbors. More specifically, this is necessary because:

(a) If $i$ has changed its status, its neighbors must know that if they were to declare themselves cluster heads, they would cover at least one less node. Also, they must now disregard the value of *Myvote*.

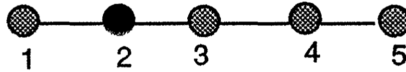
(b) Suppose *Myvote* $=$ $i$ for some neighbor $p$. It is clear that this value can change, once the number of FREE nodes covered by $i$ does. Assume that the value changes to $q$, where $q$ is a node that would declare itself to be cluster leader, if only its FREE neighbor $p$, would change its value of Myvote to $q$. Then we see that in order for $q$ to be able to change its status, $i$ must communicate the new status of the node $i$, along with the information for $p$ to calculate the size of $i$'s new FREE neighborhood.

Hence, $i$ broadcasts the fact that it is covered by $j$, and N(j). Note that if there are no FREE nodes in $N(i)$, $i$ will broadcast the information and then terminate. Another thing to note is that $i$ does not send a value of *Myvote* since it is now covered. This situation is illustrated by Figure 2. Observe (from the code in Appendix A) that if $i$ broadcasts, it includes $j$ in Wait. The reasons for doing this are best illustrated by the simple example of 5 nodes connected in a line—see Figure 3.




After node 3 declares itself a HEAD, nodes 7 and 9 will choose node 8 instead of node 10.

Figure 2:




It can easily be seen that the only node that can declare itself to be a HEAD, after WAIT=$\phi$ for the first time, is 2. This node sends out the information that it is a leader to nodes 1 and 3. Node 3 receives this, and in executing Event *DoCovered*, sends out its broadcast. Now suppose $2 \notin WAIT$. Observe that node 4's most current value of Myvote, that node 3 possesses is 3. Its only other neighbor is not free, and so event *Tryhead* has occurred. Hence 3 becomes a cluster head. Similarly, 4 and 5 will also become cluster leaders. By ensuring that $WAIT \neq \phi$, we force 3 to wait until node 4 has had a chance to reevaluate its value of Myvote, which is clearly 4 itself.

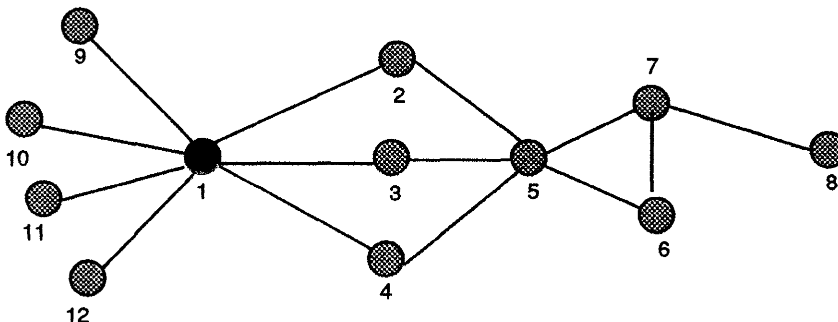
Figure 3: Nodes 3, 4 and 5 will become HEADs if 2 $\notin$WAIT at node 3.

### 3.5 What to do if a neighbor's neighbor declares itself a Leader

We now move to the case when $i$ receives a message sent by a node $j$, whose neighbor $l$, has just declared itself a cluster leader (when node $i$ receives a **Type 1** message from node $j$). This case is dealt with by the routine *TwoAway*. If $i$ is just one hop away from $l$ then *DoCovered* will be executed subsequently, if it has not done so already. The interesting case arises when $i$ is two hops away from $j$. Then the algorithm determines $C_l$, the set of all of $i$'s non-head neighbors, adjacent to $l$. These are the neighbors of $i$ that are affected by $l$'s declaration. (Note that $C_l$ can be determined because the **Type 1** message received by $i$ contains $N(l)$.) Thus when $i$ receives a **Type 1** message pertaining to a leader $l$ for the first time, it has enough information to modify all of its variables correctly. The pair

$(l, i)$ is added to Uplist, and all subsequent **Type 1** messages pertaining to $l$ are ignored.

Again, $i$ should ensure that it will wait if there is a chance of it declaring itself a cluster leader incorrectly. (Figure 4 shows this condition, and also illustrates some of the points made earlier.) If, however, $i$ is none of its FREE neighbors' most highly connected neighbor,




Node 1 has declared itself a leader, and node 5 has received a **Type 1** message from, say node 3, intimating it of this fact. Now 5 has only 2 FREE neighbors left (6 and 7), both of which has initially chosen 5 to be their most highly connected neighbor. If $l \notin$ WAIT (at node 5), then it is possible that node 5 would declare itself to be a leader, which it clearly should not be able to do. When $l \in$WAIT (at node 5), 5 must wait to be informed that 7 is now the most attractive candidate.

Figure 4:

then it is not necessary to change include $l$ in WAIT. If $i$ is FREE, the new value of *Myvote* must be broadcast. This is calculated easily from Connectivity and Statuslist.

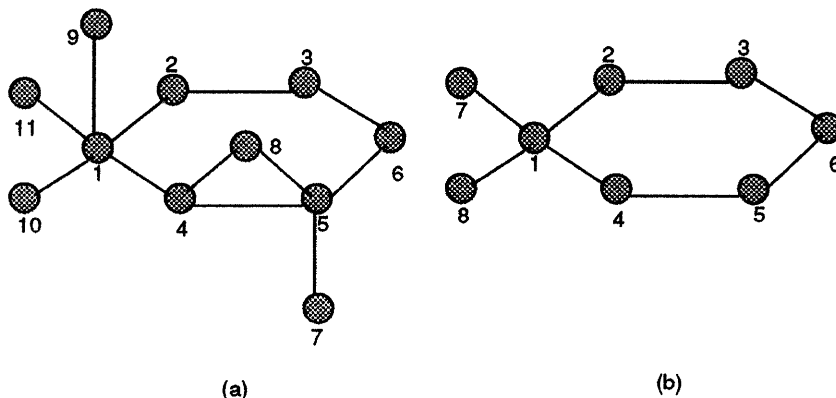
### 3.6 What to do upon receiving a Type 2 message

We now consider $i$'s response to a **Type2** message. There are 3 possibilities: $i$ is either 1, 2, or 3 hops away. In the first and second cases, no communication is necessary, but $i$ must update its variables. This is done as shown in routine *OneThreeAway* of the code. It is possible that WAIT=$\phi$, and $i$ may execute *TryHead* subsequently.

If $i$ is three hops away from $l$ it must send an updated value of *Myvote*. Again, $i$ has all the information necessary to do this the very first time it receives a **Type 2** message pertaining to $l$: Node $i$ computes $E_l$, the set of neighbors of $i$ that are two hops away from $l$, and not HEADs. Node $i$ can then find its most highly connected neighbor from Connectivity and Statuslist. Hence, *Myvote* can be broadcast without waiting to hear from the other nodes, if in fact i.status=FREE. To prevent $i$ from rebroadcasting another **Type 3** message pertaining to $l$, we include $l$ in the set 3Sent.

Observe that while $i$ cannot calculate the new values of *Myvote* of all members of $E_l$, it

need not wait to hear from all of them before executing TryHead. This is because all of $i$'s neighbors that preferred it before $l$'s declaration will continue to do so after the declaration. Figure 5 illustrates this point.




Node 1 declares itself a HEAD. Suppose node 6 receives a **Type 2** message from node 3, but has not received one from node 5 as yet. In (a) we see that node 6 declares itself a HEAD, and in (b) it does not. In both cases, node 6's decisions would not have been different if it *had* heard from node 5 before receiving node 3's message.

Figure 5:

### 3.7 What to do upon receiving a Type 3 Message

The only remaining case is the action taken on receiving a **Type 3** message: When this happens $i$ checks if it is 2 hops away from $l$. If it is not, then it must be 3 or 4 hops away and merely updates its Votelist. If it is 2 hops away, it replaces the new value of Myvote and modifies Uplist appropriately. It then checks to see if the conditions have been met to remove $l$ from WAIT.

## 4 Correctness

Observe that the *Myvote* values in $DISTG$ help determine which nodes are *Elected* in the network a particular stage of the algorithm. In the following, we will argue that exactly those nodes that are *Elected* under *Greedy* are declared cluster leaders by Procedure $TryHead$ in $DISTG$.

After the initial values of Myvote have been received at the least numbered node with the highest outdegree, and WAIT$=\phi$ for the first time, the node will declare itself to be a cluster head, broadcast, and terminate. This ensures that the algorithm will in fact begin.

Note that this node is *Elected* in the first iteration of *Greedy.*

The next Lemma will be useful in the proof.

**Lemma 1** *If 2 nodes declare themselves to be cluster leaders at the same time, they must have no FREE neighbors in common.*

**Proof.** By contradiction. Let nodes $p$ and $q$ declare themselves to be leaders simultaneously, and let their neighborhoods have some FREE node, $r$ in common. Since both nodes consider themselves to be *Elected*, $r$ must have considered each one of them its most highly connected neighbor at different times. Suppose, without loss of generality that $q$ was an earlier value than was $p$. When $r$ changed its vote to $p$, it must have done so because of a decrease in node $r$'s estimate of $|N^u(q)|$ (Here, $N^u(q)$ is the set of FREE neighbors of node $q$.) It is now argued that $r$ could not have learned of this decrease from $q$ itself.

Suppose $q$ knew that its FREE neighborhood had decreased, before its declaration. When $q$ broadcast this **Type 1** and **Type 2** message, it had to have set UpdateStatus($l$)=WAIT for some $l$, implying that WAIT$\neq \phi$. Thus it must have received the modified value of $r.vote$ before removing $l$ from WAIT, i.e. making it possible for WAIT to be $\phi$, and so it could not consider itself to be *Elected.* This contradiction shows that $q$ could not have known of the decrease in its FREE neighborhood before declaring itself a cluster head, implying that $r$ could not have heard of the decrease from $q$.

So suppose $r$ heard from some $u \neq q$, and let $z$ be the node whose declaration resulted in the decrease in node $r$'s estimate of $|N^u(q)|$. It is clear that $q$ and $z$ must have at least one node, $g$ in the intersection of their closed neighborhoods which was FREE before $z$'s (and $q$'s) declaration. Suppose $g = q$. Then $Myvote = z$ at $q$, and $q$ could not have declared itself a HEAD without hearing about $z$'s declaration. If $g = z$ then z.vote=q at $q$, implying that $z$ could not have declared itself HEAD. So $g$ is some node other than $q$ and $z$. Now since $z$'s declaration precedes $q$'s, it follows that when $g$ was FREE, its vote was for $z$. Then Nstatus(g)=z at $q$ when it declares itself HEAD because $q$ does not know of $z$'s declaration. This contradiction shows that $g$ does not exist, implying that $r$'s estimate of $|N^u(q)|$ does not change by $z$'s declaration $\Rightarrow$ $Myvote$ at $r$ does not change $\Rightarrow$ $p$ could not have declared itself a cluster head until it learned that $q$ had. □

Now suppose that a node $i$ declares itself a cluster leader at some time, $T$ of the algorithm. There are 2 possible cases:

(1) Node $i$ was a FREE node just before it became a HEAD.

(2) Node $i$ was a COVERED node just before it became a HEAD.

**Case 1**:

In general some of $i$'s neighbors are free, and others already covered when $i$ becomes a leader. Note that none of the neighbors can be leaders, since $i$ is FREE. Now, suppose that $N(i)$ consists of *no* covered nodes. Then the value of $|N^u(i)|$ has not changed since the beginning of the algorithm. Since, $i$ is going to declare itself a HEAD, all of its neighbors must have regarded it as their most highly connected neighbor at some earlier time in the algorithm. Now observe that values $|N^u(i)|$ never increase, implying that since none of the nodes in $|N(i)|$ has been covered, $i$ still has to be the most highly connected node of all its neighbors.

But suppose that some of $i$'s neighbors are COVERED nodes. Let $j$ be the neighbor to be most recently covered before time $T$. Note that this time must be strictly before $T$, because no 2 nodes can simultaneously become cluster leaders if they have a Free node in common. The Free node in this case is $i$. Now suppose that some FREE neighbor of $i$, say $p$ does not have $Myvote = i$ at time $T$. Obviously, $i$ does not know this since it is about to declare itself a HEAD. This means that $i$ was $p$'s most highly connected neighbor at some earlier time. This situation could only have changed by a subsequent *decrease* in the value of $|N^u(i)|$. The last time this happened was when $j$ was COVERED. At this time, $i$ must have sent a type 2 message to node $p$, and could not have had $WAIT = \phi$. At time $T$, WAIT $= \phi$ at $i$, which means that node $p$ considered $i$ to be its mostly highly connected neighbor even after $|N^u(i)|$ had reduced to its value at time $T$. This contradicts our assumption, and $DISTG$ always picks a *Elected* node in this case.

**Case 2**:

Since $i$ is COVERED at time $T$, at least one of its neighbors has already declared itself a cluster head. Let $j$ be the neighbor to become a head most recently before $T$. Before $i$ received the **Type 0** message from $j$ it could not have considered itself *Elected* because j, its neighbor did, and we know that 2 neighbors can never consider themselves *Elected* simultaneously. So after receiving the **Type 0** message from $j$, $i$ put $j$ in WAIT, and broadcast a **Type 1** message to all its neighbors. All its free neighbors must have declared it to be the most highly connected node, and therefore we know that even after the most recent reduction in $|N^u(i)|$, due to a **Type 0** message, node $i$ is an *Elected* node. However, observe that $|N^u(i)|$ could also have been reduced because of **Type 1** messages received. That this cannot lead to an error has already been shown in Case 1. From this we conclude that if $i$ is COVERED at time $T$, it cannot declare itself to be a leader unless it is *Elected.*

We still have to show that the algorithm never terminates before covering all the nodes, and that it never deadlocks. The second of these issues is resolved easily in light of the preceding discussion. $DISTG$ can never deadlock because there are always *Elected* nodes

in the graph, until all nodes are covered. This follows from the fact that *Greedy* does not deadlock. Now we show that the algorithm will never terminate if there is even a single uncovered node in the network. This can be seen from the stopping conditions at a node: either the node is a cluster leader, or it has no FREE neighbors. An uncovered node always has status FREE, and so none of such a node's neighbors can terminate until it has been covered.

This completes our argument of the correctness of $DISTG$.

# 5 Complexity

Initially, the list *Connectivity* must be established at each node. This takes $2N$ broadcasts. All subsequent communication is triggered by the declaration of a leader. The number of broadcasts resulting from node $i$ becoming a leader is no more than $F_i = |N(N(N(i)))|$. So, if $D_g = \{d_1, ..., d_{|D_g|}\}$, the total number of broadcasts $\leq \sum_{i=1}^{|D_g|} F_{d_i} + 2N$. Now observe that the number of elemental messages in each of **Type 0**, **Type 1** and **Type 2** messages sent is bounded by the size of the neighborhood of some cluster leader, plus 2. **Type 3** messages consist of only 2 elemental messages. Let $\delta$ be the size of the largest neighborhood. Also, let $T_i = |N(N(i))|$. Defining $C$ to be the total number of elemental messages broadcast we have:

$$C \leq (\sum_{i=1}^{|D_g|} T_{d_i}(2 + |N(d_i)|) + (F_{d_i} - T_{d_i})2) + N + N\delta.$$

Simplifying:

$$C \leq (\sum_{i=1}^{|D_g|} T_{d_i}|N(d_i)| + 2F_{d_i}) + N(1 + \delta). \tag{3}$$

To obtain an accurate expression for $C$, we will make the assumption that the network is regular i.e. all nodes have the same degree, $\delta$. Then, $T_i \leq 1 + \delta^2$ and $F_i \leq 1 + \delta - \delta^2 + \delta^3$. Substituting in (3):

$$C \leq |D_g|(2 + 3\delta - 2\delta^2 + 3\delta^3) + N(1 + \delta).$$

Now note that

$$|N(d_i)| = \frac{2M}{N} = \delta, \forall i.$$

Making use of the bound for $|D_g|$ derived in Theorem 2 we have:

$$C \leq (N - \sqrt{N\delta + 1})(2 + 3\delta - 2\delta^2 + 3\delta^3) + N(1 + \delta).$$

For large $\delta$ and $N$ we have

$$C \leq O(\delta^3|D_g| + N\delta).$$

When the naive algorithm is used, in which every node to learns the entire topology of the graph and then computes the dominating set according to *Greedy*, it can be shown that the communication complexity is $O(N\delta^d)$, where $d$ is the diameter of the graph. Thus $DISTG$ is an efficient protocol when the graph is sparse, so that the longest path originating at any node $i$ is much longer than 3. When this is not true, the naive algorithm may be better.

The Time complexity, $T$ is the number of iterations the heuristic would take if it worked synchronously. Every time a leader is declared, 4 types of messages are broadcast. So

$$T \leq 4|D_g| = 4(N - \sqrt{2M+1} + 1) \Rightarrow T = O(N - \sqrt{2M+1}) = O(N - \sqrt{N\delta+1}).$$

# 6 Acknowledgment

I am grateful to Professor Robert Gallager for his help in the performing and writing of this work.

# References

[1] D. J. Baker and A. Ephridemes, *The architectural organization of a packet radio network via a distributed algorithm*, IEEE Transactions on Communications, COM-29 (1981), pp. 1694–1701.

[2] D. Bertsekas and R. Gallager, *Data Networks*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[3] V. Chvatal, *A greedy heuristic for the set covering problem*, Mathematics of Operations Research, 4 (1979), pp. 233–235.

[4] R. G. Gallager, *Distributed minimum hop algorithms*, Tech. Rep. P1175, MIT Laboratory for Information and Decision Systems, 1982.

[5] M. R. Garey and D. S. Johnson, *Computers and Intractability—A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York, 1979.

[6] Y. I. Gold and S. Moran, *A correction algorithm for token-passing sequences in mobile communications networks*, Algorithmica, 4 (1989), pp. 329–341.

[7] D. S. Hochbaum, *Approximation algorithms for the set covering and vertex covering problems*, SIAM Journal on Computing, 11 (1982), pp. 555–556.

[8] D. S. Johnson, *Approximate algorithms for combinatorial problems*, Journal of Computer System Science, 9 (1974), pp. 256–278.

[9] A. K. Parekh, *Analysis of a greedy heuristic for finding small dominating sets in graphs*, Tech. Rep. 2018, MIT Laboratory for Information and Decision Systems, 1991. To appear in *Information Processing Letters*.

[10] A. Segall, *Distributed protocols*, IEEE Transactions on Information Theory, 1 (1983), pp. 23–35.

[11] V. G. Vizing, *A bound on the external stability number of a graph*, Doklady A. N., 164 (1965), pp. 729–731.

## APPENDIX A—Code for DISTG

executed at node $i$:

```
Event TryHead:
begin
    if (Wait = φ) AND (j.vote = i∀j ∈Votelist) then        (*is i Elected? *)
        begin
            Myleader=i;
            i.status = HEAD;
            Send(i is a head, N(i));        [type0 message]
            TERMINATE;
        end
end

Event DoCovered:
executed when Rec(Type0 from j):
begin
    if (j, i) ∉ Uplist then
    begin
        if Myleader=i then Myleader=j; (*assign i to leader j*)
        (* Find i's affected neighbors *)
        B_j = {k : k ∈ N(j) and k.status ≠ HEAD }
        for all p ∈ B_j do
        begin
            p.status = COVERED
            Delete p from Votelist
        end
        A_j = N̄(i) ∩ B_j              j.status = HEAD;
        (* Modify the datastructures *)
        for all k ∈ A_j do
        begin
            Uplist=Uplist∪{(j, k)}; (*Since i ∈ A_j, (l, i) ∈Uplist *)
        end ;
        (* Any nodes in N(i) that do not know that j is a HEAD are told so. *)
        if ∃k ∈ N(i) − N̄(j) such that (j,k) ∉ Uplist then
        begin
            Wait = Wait ∪{k}
            Send(i covered by j, N(j)); [type1 message]
        end
        else
        begin
            Wait = Wait ∪{k}
            TryHead;
        end ;
        if {k : k ∈ N̄(i), k.status=FREE} = φ then TERMINATE;
    end
end
```

```
Event TwoAway:
executed when Rec(Type1 message from j; l is leader)
begin
(* Continue only if i is 2 hops away, and has not heard about j's declaration *)
    if (l, i) ∉Uplist and l ∉ N(i) then
    begin
        Uplist=Uplist∪ (l,j);
        C_l = N(i) ∩ N(l) ∩ {k :k.status≠HEAD};
        l.status = HEAD;
        (*Modify datastructures*)
        for all k ∈ C_l do
        begin
            Delete k from Votelist;
            Uplist = Uplist∪{(l, k)};
        end ;
        for all p ∈ N(l) ∩ N(N̄(i)) s.t. p.status≠HEAD do
            p.status=COVERED;
        for i.status=FREE then Recalculate Myvote;
        Send(l,N(l), and [Myvote if i.status=FREE]); [type2 message]
        Uplist=Uplist∪{(l, i)};
        if ∃p : (l, p) ∉ Uplist, p.vote=i then
            WAIT = WAIT ∪{i}
        else
        TryHead;
        if {k : k ∈ N̄(i), k.status=FREE} = φ then TERMINATE;
    end
end


Event OneThreeAway:
executed when Rec(Type2 from j; l is leader)
begin
if (l ∈ N(i)) OR (l ∈ N̄(N(i))) then (*i.e. if i is 1 or 2 hops away from l *)
    begin
        if j.status=FREE then
            Replace the value of j.vote in Votelist;
        Uplist=Uplist ∪{(l, j)};
        if ∃k ∈ N(i) − {l} s.t. (l,k) ∉ Uplist then
        begin
            WAIT= WAIT - {l};
            TryHead;
        end ;
    end
    else begin (*i is 3 hops away *)
        E_l = {k : k ∈ N(i), i.status≠HEAD s.t. ∃p ∈ {N(k) ∩ N(l)}
        AND p.status≠HEAD};
        if i.status=FREE then Recalculate Myvote;
        for all p ∈ N(l) ∩ N(N(i)) s.t. p.status≠HEAD do
            p.status=COVERED;
        if not l ∈ 3Sent then (*Only one broadcast per head *)
        begin
            Send (l, and [Myvote if i.status=FREE]; [type3 message]
            3Sent=3Sent ∪{l}
        end ;
    TryHead;
```

```
    end
end .

Event TwoFourReceive:
executed when Rec(Type3 from j; l is leader);
begin
    if l ∈ N̄(N(i)) then
    begin
        Uplist=Uplist ∪{(l, j)};
        Replace j.vote in Votelist
        if ∃k ∈ N(i) such that (l, k) ∉ Uplist then
        begin
            WAIT = WAIT −{l};
            TryHead;
        end
        else
            WAIT = WAIT ∪{l};
    end
    else
    begin
        Replace j.vote in Votelist;
        TryHead;
    end
end .
```