A UNIFIED APPROACH TO THE CONSTRUCTION OF EFFICIENT DISTRIBUTED LEADER FINDING ALGORITHMS

by

Shay Kutten

Technical Report No. 327

July 1984
ABSTRACT

This study discusses an algorithm for finding a leader in certain sets of undirected graphs in $O(n \cdot \log n)$ messages. Santoro has shown that in a general graph at least $O(|E|+n \cdot \log n)$ messages are needed for finding a leader. However, several algorithms which work in $O(n \cdot \log n)$ messages for some restricted sets of graphs, do exist. Each of this algorithms specializes in one such set.

We introduce here the property all these known "n \cdot \log n graphs" have in common and suggest one algorithm which achieves $O(n \cdot \log n)$ messages leader finding in all the known special cases, as well in some new ones. The algorithm is the same for all those cases. The special features of the given case is taken advantage of in a lower layer (servant) protocol which is used to pass the messages (this layering is common in communication networks). As we will see, the construction of the lower layer protocol in the known special cases is straightforward. At last we generalize the result for $O(f(n) \cdot \log n)$ for graphs for which an $O(f(n))$ auxiliary algorithm (similar to the one we use) can be constructed. Then there exists a distributed leader finding algorithm.

1. INTRODUCTION

This study discusses a general distributed algorithm for finding a leader in certain sets of undirected graphs. (The directed case i.e. networks containing unidirectional links, is discussed elsewhere [1]). A corollary of our results enables us to find a leader in some cases in $O(n \cdot \log n)$ messages. Distributively finding a leader in computers communication networks is a problem which has lately achieved much attention [2,3,4,5]. One major use for the leader is the information routing [7]. Another use is the making of unambiguous decisions. Distributively finding the leader enables the network to perform these tasks without storing much information in each node. It is desir-
able for network's self overcoming topological changes [4]. A lower and an upper bound of \( O(|E| + n \log n) \) messages for leader finding in a general graph were found by Santoro [4] and by Gallager, Humblet, and Spira [3]. However, algorithms for \( O(n \log n) \) messages leader finding in restricted sets of graphs namely complete graphs and circle graphs [2,4]. The algorithm suggested in each case is different.

In this paper we define a property of sets of graphs through the existence (or non existence) of a certain general algorithm for them, with certain properties. As we shall see, the checking of this property (i.e. the construction of the auxiliary algorithm) in the known \( O(n \log n) \) graphs, as well as in some other sets of graphs, is straightforward.

We modelize each computer communication networks with \( n \) nodes by an undirected graph \( G(V,E) \) where \(|V|=n\), and the nodes in \( V \) correspond to computers, and the edges in \( E \) correspond to bi-directional (i.e. full duplex [5]) communication links. Each two endpoint nodes of an edge are neighbors. Each node can send messages on any of it's edges and be sure that the message will arrive at the neighbor at the other endpoint of the edge, after an unknown but finite time [4]. Messages arriving at a node on the same edge are recognized and handled according to the order they have been sent. No such order is specified concerning messages arriving on different edges. We distinguish between a message and the information it may carry. A message, \( m \), may be created by, say, node \( i \), sent to node \( j \) which may copy \( m \)'s information to node \( j \)'s memory, change the information in the \( m \), and send \( m \) (it is still \( m \)) to node \( k \).

Distributed algorithms are divided into layers. A higher layer algorithm executed in a node may call upon the services of a lower level algorithm service needed. The called lower level algorithm receives the call as an event and reports upward (on termination or on the completion of part of the service required) by causing the upper layer algorithm to receive an event.
A traversing algorithm is a distributed algorithm which can pass a message created by any node through all other nodes. At all times the message is either in transit over one edge or held by exactly one node. In any node the traversing algorithm can be told (by an upper layer algorithm) to send the message back along the same route (or pass) it traversed from node i. When the message has passed all nodes the traversing algorithm notifies an upper layer algorithm (by an event): In a linear traversing algorithm the traversing message passes the first k (distinct) nodes in O(k) hops. An example is the following algorithm performed in complete graphs (cliques): Node i (message initiator) sends the message to one of its neighbors, together with a request that it will be sent back. When receiving the message back, i, sends it to another neighbor which has not received it yet.

2. MAIN THEOREM

Theorem 1: Let a set S of graphs be f(k) minimum convexly traversable

1. There exists a distributed algorithm for finding a leader in the graphs of S in O(f(n)logn) messages.

2. There is no general algorithm which finds a leader in each graph in S in less than f(n) messages.

The proof of (b) is trivial. The proof of (a) will be done in the following sections by presenting the leader finding algorithm and proving it's properties.

3. ALGORITHM INFORMAL DESCRIPTION

Due to space limitation we give here only the informal description. A node, i, may enter the algorithm either spontaneously (i.e. awakened by an upper layer algorithm) or by receiving a message generated by the algorithm in a neighboring node. Either way once a node has entered the algorithm it
is assumed it can not be awakened (again) till the algorithm terminates. Each node has a variable containing it’s phase which is initially zero. Node’s phase is the phase of the domain to which the node belongs. Node’s phase is increased by joining another node’s domain (in this case it adapts that domain’s phase) or by having it’s own domain conquer another domain with the same phase. Both cases will be explained in the sequel.

The algorithm uses a lower layer traversing algorithm. The messages traversing the network, called tokens, are created, one by each spontaneously awakened node. Each token carries (among other information) the identification and the phase number of it’s originating node. This phase is at least one, as on token’s origination the originating node increases it’s phase from zero to one. Having been originated by node i \( \text{TOKEN}_i \) is send forward by the lower layer traversing algorithm along the serial broadcast route from node i. As we shall see, during algorithm execution \( \text{TOKEN}_i \) may return (by using the reversibility property of the traversing algorithm) several times to node i, increase it’s phase and be sent again forward. on each way forward it may visit certain nodes several times if it does not follow a Hamiltonian circle. Thus, when we mention forward (or backward) arrival to a node, we mean the first arrival of that token to that node on the way forward (or backward) with the same phase number.

When a token originated by node i, carrying phase x forward arrives at node k which belongs to a domain with phase y, the following happens. If \( y < x \), than node k joins i’s domain (by recording i’s identity), adapts i’s phase, and i’s token is sent forward along the serial broadcast from node i. If \( y > x \) than \( \text{TOKEN}_i \) is destroyed. If \( y = x \) than node k belongs to the domain of another token, say \( \text{TOKEN}_j \). We term node k the sensing node. \( \text{TOKEN}_i \) will try to meet \( \text{TOKEN}_j \) in order that only the one with higher (phase,identity) will survive, the ordering being done lexicographically from left to right. Such a meeting between two tokens with the same phase is called a combat and it increases the phase of the survivor. Arranging the meeting will be done as follows. If i
< j then \( \text{TOKEN}_i \) waits in the sensing node for \( \text{TOKEN}_j \)'s return. Else (\( i > j \)) \( \text{TOKEN}_i \) chases \( \text{TOKEN}_j \). The chase is performed by sending \( \text{TOKEN}_i \) along \( \text{TOKEN}_j \) route if \( \text{TOKEN}_j \)'s mark in the sensing node contains the value FORWARD. Otherwise the chase is performed by backtracking \( \text{TOKEN}_j \)'s retreating route. Indication about the chase is left along the route to avoid the possibility of the two tokens passing by it's other. In either direction the chasing token will meet the chased one in the part of the route the chased token has covered (in that phase), or it will run into a node covered with a higher phase token (possibly the chased token after a successful raising it's phase) or will arrive to a sensing node from which \( \text{TOKEN}_j \) has started to chase some \( \text{TOKEN}_i \) with \( 1 < i < j \). In The first case a meeting has occurred and only one of the two tokens has survived it. In the second case the chasing token will be destroyed. In the third case \( \text{TOKEN}_i \) can also be destroyed as the combat between \( \text{TOKEN}_j \) and \( \text{TOKEN}_i \) will yield a higher phase token (or the combat between \( \text{TOKEN}_i \) and some \( \text{TOKEN}_j \) with \( r < i \), and so on).

When two tokens with the same phase meet, the survivor (the one with the higher identity) returns to it's originating node, increases there it's phase by one, and is sent again forward. It may happen that when a meeting between the chasing and the chased tokens is possible, the phases of the two tokens are no longer equal (through phase increasing during other combats). In that case the token with the lesser phase number is destroyed. There may be other tokens waiting for the destroyed token, but it doesn't matter as their phase is no longer the highest phase and thus they are no longer needed for leader election. A token trying to chase the destroyed token will enter the territory of it's destroyer and will be destroyed too.

Finally we explain the case in which some \( \text{TOKEN}_k \) is chased by more than one token. Each node on \( \text{TOKEN}_k \)'s back route does not transfer forward or backward more than one token of a given phase. The other chasing tokens are not needed any more as when after the first chasing token meets the chased one- there is a token with a higher phase which is serially
broadcasted again from it's originating node.

4. PROOF OF MAIN THEOREM

Lemma 1: At least one Token terminates in it's originating node with true being the value of it's LEADER field.

Proof: Whenever a token terminates or is destroyed there is another token with a higher phase. Clearly this is true in a termination through a combat, when the winner increases it's phase. It is also true in the "on MeetingWithHigherPhase" routine. The lemma will follow when we show that there is one token among the undestroyed ones, which does not stay forever in a waiting state. Assume the contrary, i.e. the algorithm reaches a situation when all remaining tokens are waiting. Let x be the highest phase any of those tokens has got. Let TOKEN_i be the token with highest identity among the remaining tokens with phase x. Let us look at TOKEN_i before entering the waiting state - no token could have caused it to enter the wait state in the "on ForwardArrival" routine. Thus it enters the wait state in the "on chased" routine. This however can happen only in a node, j, in which a chaser with the same phase has visited. This chaser can not be destroyed according to the assumption that no higher phase exists when the algorithm halts. At least one chaser (there can be several) cannot also enter a wait state and each node on the chased route transfer one chaser with phase x) thus it must pass TOKEN_i even without changing it's direction along TOKEN_i's route. Upon such a passage TOKEN_i enters a wait state and thus on a full backtracking in one direction (along TOKEN_i's route) and one node on the opposite direction must lead to an encounter and the increase of the phase of TOKEN_i, contrary to the assumption that phase x is the highest phase when the algorithm halts.

Q.E.D.
Lemma 2: No more than one TOKEN terminates in it's originating node with true being the value of it's LEADER field.

Proof: Assume the contrary. A LEADER token, L, must pass all nodes (See BroadcastEnd and Home routines) before it can terminate in it's originating node— if only there is a higher phase token in the network than TOKEN is destroyed. Thus all the LEADER tokens has the same phase, x, upon termination of the broadcast. However TOKEN passes all nodes with phase x, starting from it's originating node (see Encounter routine: on increasing it's phase a token returns to it's originating node), and if there is another node with the same phase than TOKEN must arrive at a node of another such token's domain, and according to the proof of lemma 1 the final phase will not be x, contrary to the assumption that x is the highest phase reached in the algorithm.

Q.E.D.

The two following lemmas discuss algorithm complexity. Lemma 4: The total number of messages (token's transfers) in a phase is O(f(n)).

Proof: A token with k nodes in it's domain in phase x, makes O(f(k)) hops (messages sending) both in it's forward and backward way in it's domain. We charge it's way in other token's domain (during chasing) on the account of the other token. all the tokens chasing a token with m nodes in it's domain in phase x do not make more than O(f(k)) hops together, as nodes do not transfer more than one chaser (after the same node in the same phase). The lemma now follows from the fact that the sum of the numbers of nodes in all the domains in a phase x is bounded by n.

Lemma 5: The number of phases is bound by log₂n.

Proof: The lemma follows from the fact that when a token with phase x increases it's phase it destroys another token with the same phase.
5. THEOREM APPLICATIONS FOR SPECIAL CASES

We now give examples of sets of graphs in which it is easy construct a linear traversing algorithm (and thus, by main theorem, how to find a leader in $O(n \log n)$ messages). Some of them are the known results for $O(n \log n)$ messages leader finding [2,3,5] and some of them are new. As the linear traversing algorithm construction is usually quite straightforward (and due to space limitations) we leave the construction to the reader. The list of examples includes: Circles; Complete graphs; Trees; Graphs containing a star (note that in this new case we assume much less information known than in the case of complete graphs); Complete bi-partite graphs; k-neighbors dominating Graphs (i.e. graphs in which any node together with any k of its neighbors are a dominating set, where k is a constant); Cliques circles; E.g. A circle of $2x=O(\sqrt{n})$ subgraphs (even number) were every even placed (on the circle) subgraph is a clique of $x=O(\sqrt{n})$ nodes, and every odd placed (on the circle) subgraph is a single node.
REFERENCES


