Improved Distributed Exploration of Anonymous Networks

Shantanu Das[1], Shay Kutten[2], and Ayelet Yifrach[2]

[1] SITE, University of Ottawa, Ottawa ON K1N6N5 Canada
shantdas@site.uottawa.ca
[2] Faculty of Industrial Engineering and Management, Technion,
Israel Institute of Technology, Haifa, Israel
Kutten@ie.technion.ac.il, ayifrach@univ.haifa.ac.il

Abstract. The problem of constructing a labeled map of an anonymous and asynchronous network is addressed. We present an algorithm that explores and maps the network by using k identical agents that have no prior knowledge of the network topology. An algorithm of Das, Flocchini, Nayak and Santoro for mapping of the network requires that n and k are co-prime. Our improved algorithm, presented here, requires at most $O(m \cdot \log k)$ edge traversals, while theirs uses $O(m \cdot k)$ edge traversals (m is the number of edges in the network). The size of the whiteboard memory needed in our algorithm is the same as that used in DFNS algorithm $O(\log n)$. We employ techniques utilized in solutions to the Leader Election task, and introduce a modification to resolve issues of electing first "local leaders" among adjacent candidates, which otherwise may deadlock the process.

Keywords: anonymous network, unlabeled nodes, asynchronous distributed leader election, k agents, map construction.

1 Introduction

1.1 Labeled Map Construction and Related Work

Problems of exploring an anonymous and asynchronous network have been addressed extensively [3, 8, 11-16, 23]. Mapping an anonymous network and labeling its nodes by multiple agents was presented as the *Labeled Map Construction* (*LMC*) problem by Das *et al*. in [12]. The exploration of anonymous graphs requires the agents to label the nodes. We follow earlier works [12, 14, 20] in utilizing the whiteboard model (e.g. for labeling) and introduce improvements to the process by reducing the number of edge traversals.

In exploring the network by more than one agent (e.g. see [8, 12, 15, 16]) Das *et al* employ a group of k identical agents having no knowledge of the network's topology or of one another. The agents explore the network (consisting of n nodes), each starting from an arbitrary node, and executing identical algorithms. The objectives are to construct a map of the graph, and to label each node by a unique label, both map

S. Chaudhuri et al. (Eds.): ICDCN 2006, LNCS 4308, pp. 306 – 318, 2006.
© Springer-Verlag Berlin Heidelberg 2006

and labels agreed upon by all the agents. The nodes in the graph have no identities[1]. The process eliminates all the agents but one, the elected *leader*, who maps the graph and labels the nodes. Based on [4, 6, 7, 19, 24], Das *et al.*[12] show that it is not possible, in general, to solve the *LMC* problem when n (the number of nodes) and k (the number of agents) are not co-prime, i.e. $\gcd(n,k)\neq 1$. By introducing the requirement that an agent has the knowledge of either n or k and also that n is co-prime to k we ensure in our algorithm that an agent would always terminate successfully, solving the *LMC* problem. The *LMC* problem is closely related to other problems, such as *Leader Election* ([1, 2, 18, 20, 22, 28]), *Rendezvous* ([21, 23]) and *Labeling* ([17]). Solving one of these problems leads to solving all the others as well.

We consider a distributed solution to the *LMC* problem, as in [12], using k asynchronous agents exploring an undirected simple graph. A lower layer service (*traversal algorithm*) is used to transfer agents from one node to another. The traversal algorithm has rules for transfering the agent to another node, once called upon by the agent. Section 3.1 and [20] describe the interface between the agent algorithm and the lower layer service.

1.2 Our Results

Our solution of the *LMC* problem is deterministic and requires at most $O(m\cdot\log k)$ edge traversals compared to $O(m\cdot k)$ edge traversals in the algorithm of [12].

We modify techniques from solutions to leader election problem to resolve mistaken identification between neighboring candidates. Previous algorithms achieve efficiency by competing between neighboring candidates first, and between the far-away candidates only when a few remain. Such methods may deadlock in anonymous networks as nearby candidates may be indistinguishable. The main modification presented here is the addition of two search rounds beyond the per-phase used by the previous algorithm [20]. We show that this ensures that an agent encounters all nearby distinguishable agents and as a result, the algorithm solves the problem using at most $O(m\cdot\log k)$ edge traversals. We assume that an agent has the knowledge of either n or k and it also knows that n is co-prime to k. This ensures that an agent in our algorithm always detects termination.

The rest of the paper is organized as follow: In section 2 we describe the problem and the model for the algorithm, in section 3 we describe the full algorithm and finally, section 4 we present the proof of correctness and complexity analysis.

2 Model and Problem

The network is modeled as a graph $G=(V, E)$ with $|V|=n$ and $|E|=m$. The network is asynchronous, i.e. it takes a finite but unpredictable time to traverse an edge of the network. We assume that the edges of the network obey the FIFO discipline.

There are k mobile agents located in distinct nodes of the network. An agent is a mobile entity that can execute an algorithm and move through the edges of the graph. The operations of an agent at a node are atomic. Thus, only one agent can be active

[1] A similar situation that is solvable by this algorithm is one where the nodes do have identities but the identities are not unique. This case is not explicitly discussed.

(operating) at a node at any given time. An agent also has a storage that moves with it. Initially, the agents do not have any knowledge about the graph or its topology (except for the knowledge of n or k).

Each node contains a *whiteboard* - a memory area of the node used by agents to communicate with each other. An agent visiting a node v can write to the whiteboard of node v and also read any information written previously by another agent that visited node v.

The nodes and the agents in the graph are anonymous: they have no distinct identities. The edges incident to the nodes are labeled with port numbers providing a local orientation among the edges incident at a node. This allows the traversal algorithm at a node to distinguish between the edges incident to the node.

Formally, we re-state the *LMC* problem as follows: given an instance (G, λ, p) where $G(V,E)$ is a graph, λ is an edge-labeling defined on G, and p: $V \rightarrow \{0,1\}$ is a placement function defining the initial location of the $k = |\{v \in V : p(v) = 1\}|$ agents, the *LMC* problem is said to have been solved when one of the agents, designated as the "elected" agent, obtains a uniquely labeled map of the graph.

3 Presentation of the Algorithm

Our *LMC* algorithm proceeds in phases. The traversal algorithm is invoked by the agents several times in each phase. It moves an agent from one node to another, based on the agent's label. All agents' labels are initially identical but are later refined to distinguish agents from one another. Nodes and edges visited by an agent are marked with the agent's label (unless already marked by the same or larger label). An agent *territory* is the sub graph of G consisting of the nodes and edges marked by the agent.

3.1 Traversal Algorithm

The traversal algorithm used by the agents is a distributed version of the depth-first-search (*DFS*) algorithm ([25, 26] and its (serial but) distributed version [5, 9, 20]).

Each invocation of the DFS algorithm consists of the label of the agent, which is used for marking the ports of the nodes during the execution of the algorithm. Simultaneous invocations occur when more than one agent traverses the graph at a given time. As opposed to [20], not all are distinguishable. Hence, when an agent A enters a node v through an unvisited edge and one or more ports of node v are marked with A's label the DFS algorithm acts as if agent A visited this node. These ports may have been marked by the execution invoked by agent A or by an execution invoked by a different agent B with an identical label.

The operations agent A can execute at a node v (as a lower-level system call):

Go-To-Next(): Sends the agent to the next node to be visited.
Go-Territory(): Sends the agent to the next node to be visited in the A's territory.
Chase(): Sends agent A through the last port from which an agent (in the same phase) was sent (excluding a port which an agent in the same phase marked as *back edge*).

3.2 Informal Overview of the Algorithm

The message complexity improvement achieved over [12] is due to the usage of phases. This bounds the number of graph traversals through node v to O(log k). In each phase, an agent searches to find another agent in the same phase so that they can merge and become one agent in a higher phase. As we go to higher phases, fewer agents remain (there are at most $k \cdot 2^{-P+1}$ agents in phase $p>0$). The search is performed by agents, that annex nodes by writing the label of the agent to the node's whiteboard.

Phase 0 is different than all the phases that follow. Its goal is to initialize a label for the agent. An agent A wakes up and traverses the graph marking every unvisited node by turning on a "visited" flag in the node and destroying any sleeping agent in the node. Agent A maintains a node-counter which is increased by 1 after each node flagging. By the end of phase 0, each node of the graph would have counted by exactly one agent. After phase 0, agent A constructs a label out of its node-counter, its phase and the number of agents A annexed (at this point the phase equals 0 and the last field equals 1). Agent A then raises its phase to 1. Due to the assumption that gcd(n,k)=1, there must be at least two agents with different labels after phase 0.

In all higher phases, the status of agent A can be in one of the following:

(i) *Annexing status*: agent A tries to annex to its territory all the nodes (in lower phases) or to find another agent in the same phase.
(ii) *Chasing status*: agent A chases some other agent B in the same phase but with a lower label. Agent A attempts to reach B and merge, creating a single agent in the next phase.
(iii) *Candidate status*: agent A is waiting at a node to be merged or annexed.

Agent A in *annexing* status traverses the graph looking to encounter other agents. During the traversal, agent A annexes nodes by writing its label to the node's whiteboard. When agent A traverses a node that was traversed by another agent, agent A compares the label written in the node with its own and decides how to act. Agent A ignores labels of lower phase but becomes a candidate when the label is of a higher phase. However, if the label L is of the same phase then agent A will chase the agent that labeled the node if L is lexicographically smaller than its own label. On the other hand, agent A becomes a candidate if label L in the node is lexicographically bigger than its own label. When agent A encounters an agent B waiting in a node (B is in a *candidate* status), A will annex B if agent B is in a lower phase or *merge* with B if agent B is in the same phase. Agent A raises its phase only by a *merging*. When an agent recognizes it annexed all n nodes (or all k agents, if it knows k), it is the only agent left. At that time, it declares itself a leader. The leader traverses the graph, labeling the nodes with unique names and constructing a labeled map of the graph.

Definition. *Agents* A *and* B *are adjacent agents if there exists an edge connecting a node* v *labeled by* A *and a node* u *labeled by* B.

Our Modification. We now outline the modification (compared to [20]) that we introduce in order to deal with adjacent agents with the same label. This is the action an agent takes when it fails to merge with an agent in its own phase during its annexing traversal. First, if agent A completes an annexing traversal without encountering any agent at all (even in lower phases) then it enters the *candidate* status. Note that in [20] each agent has a unique label so when agent A completes an

annexing traversal it either becomes a leader, or gets destroyed (at a node labeled with a higher phase or during the processes of creating an agent in a higher phase).

However, in our algorithm, if agent *A* encountered (and thus annexed) one or more agents in lower phases (thus *A*'s label changed) *A* should take an action. If agent *A* updates its label during the traversal the nodes it already traversed would not be updated. If one of those nodes *v* is adjacent to a node marked by an agent *B* with the same label *A* had when it traversed *v* then *A* would not notice *B* (although the updated label of *A* is now different than that of *B*). Furthermore, when agent *A* during the current round enters a node labeled by it with the former label it would act as if it entered a node of another agent and start a chase. Hence, agent *A* cannot update its label during the traversal and should take an action after the termination of the current traversal. Notice that if agent A took any of the following options, that would fail to solve the problem:

- If agent *A* enters *candidate* status then this might have ended in a deadlock (see figure 1).
- If agent *A* raises its phase (without merging with an agent in the same phase) then the number of phases could be as high as (*k*-1), thereby increasing the complexity of our algorithm.
- If agent *A* starts another annexing traversal (without raising its phase) then this might again end-up in a deadlock (see figure 2).

We solved the problem by the following mechanism, for which we prove that no deadlock arises and the complexity is still $O(m \cdot \log k)$. Let round *I* be the annexing traversal described above for a phase. We add two additional traversals per phase, termed *round II* and *round III*.

Round II: Agent *A* informs the nodes it annexed in *round I* about *A*'s new label.
Round III: Agent *A* starts a new annexing traversal (the same as in *round I*). We will show that in this round the agent can only encounter agents from the same phase. An agent ends this round either by merging with another agent in the same phase (and thus starting a new phase) or, by becoming a candidate at a node.

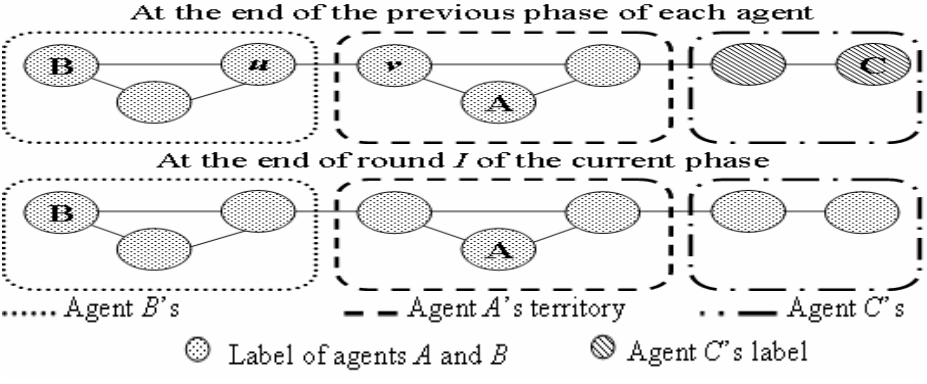

Fig. 1. Agents *A* and *B* have the same label at the beginning of phase *p,* hence when they visit nodes *u* and *v* during round *I* they do not know of one another. Agent *C* in phase *q* ($q<p$) is annexed by agent *A* during round *I*. After round *I*, agents *A* and *B* have different labels. Thus, had *A* changed its status to candidate by the end of round *I* the algorithm would have had entered a deadlock (agent *A* waits for *B* and vice versa).

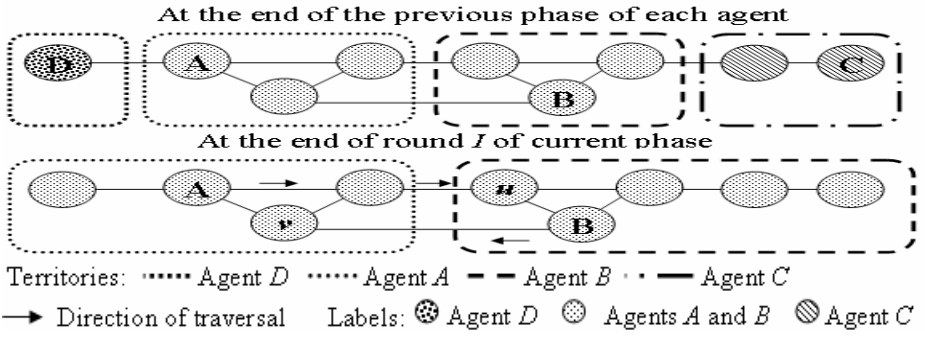

Fig. 2. Agents *A* and *B* have the same label at the beginning of phase *p*. Hence, during round *I* they cannot distinguish between themselves, and thus do not know of one another. During round *I*, agents *C* and *D* in phase *q* (*q*<*p*) are annexed by agent *B* and agent *A* respectively. After round *I*, agents *A* and *B* have different labels. Thus, had *A* and *B* started another annexing traversal without performing round *II* before that, then *B* would have had started chasing *A* from node *v* and *A* would have had started chasing *B* from node *u*. This would have had caused the algorithm to deadlock.

3.3 A More Formal Description

The data held by the agent
The label of the agent:

- *phase(A)* – The phase of agent *A*.
- *nodesAnnexed(A)* – The number of nodes marked at phase 0 by agent *A* itself and by the agents that were annexed by agent *A*.
- *agentsAnnexed(A)* – The number of agents annexed by agent *A*, including itself.

The status of the agent:

- *Status* – One of the following: *annexing, chasing, or candidate.*

Temporary variables:

- *tname(A)* – The value of *tname(A)* accumulates the values of *tname* of the agents that were annexed by agent *A* (during *round I* of the current phase). This value is initialized to *nodesAnnexed(A)* at the beginning of each phase.
- *agentsCounter(A)* – The value of *agentsCounter(A)* accumulates the values of *agentsCounter* of the agents that were annexed by agent *A* (during *round I* of the current phase). This value is initialized to 1 at the beginning of phase 0 and to *agentsAnnexed(A)* at the beginning of every other phase.
- *nodesCounter(A)* – The number of nodes annexed by agent *A* (initialized to 1).

Data written to a node v. The label of the node is equal to the label and the status of the agent that annexed it (*phase(v)* , *nodesAnnexed(v)*, *agentsAnnexed(v)*).

The name of the node:

- *nodeName(v)* – The sequence number of the node (assigned by the last annexing agent).
- *visitFlag(v)* – Indicates whether the node was marked by an agent in phase 0.

The status of the node:

- *Status* – One of the following *annexed, chased,* or *round2* (assigned by the agent that last traversed the node).

Comparing variables in labels. During the execution, the agent compares its own label with the label of the node lexicographically.

The Algorithm. Initially, all the agents are asleep and on waking up (spontaneously), an agent *A* starts executing Phase 0.

3.3.1 Phase 0

On waking-up in some node (say *u*),[2] agent *A* sets its label to (*phase(A)* =0, *nodesAnnexed(A)*=0, *agentsAnnexed(A)* =1) and sets *agentsCounter(A)* to 1. Agent *A* marks node *u* as visited by turning on *visitFlag(u)*. Agent *A* then starts traversing the graph, using the traversal algorithm. When *A* reaches a node *v* it acts as follow:

- If *v* is unmarked then *(1) A* increments its *nodesCounter(A)* by 1; *(2)* marks node *v* as visited by turning on *visitFlag(v)*; (*3*) if a sleeping agent exists in the node then agent *A* destroys the agent; and *(4)* continues the graph traversal.
- If *v* is marked *visited* then *A* continues the traversal. Recall that the traversal algorithm acts as if *A* has been in node *v* before (which may or may not be the case), so it returns to the node from which *A* was last sent to node *v* and continues the traversal.

Upon termination of the traversal of phase 0, *A* raises its phase to 1 (*phase(A)* ← 1), sets values of nodesAnnexed (*nodesAnnexed(A)*←*nodesCounter(A))*, agentsAnnexed (*agentsAnnexed(A)*← *agentsCounter(A))* and its status to *annexing*. If *n* is known and *nodesAnnexed(A)*=*n* or *k* is known and *agentsAnnexed(A)*= *k* (*k*=1) then agent *A* detects the successful termination and performs the *leader procedure* (described below). Otherwise, agent *A* proceeds to *Round I*.

3.3.2 Round I

Agent *A* starts an annexing process by starting a graph traversal with the label (*phase(A)*, *nodesAnnexed(A)*, *agentsAnnexed(A)*) and the *annexing* status. The following rules apply:

Whenever an annexing or a chasing agent *A* reaches some node *v* or raises its phase at some node *v*, agent *A* acts according to its label as follows:

Either annexing or chasing status

- If there is an agent *C* labeled (*phase(C)*, *nodesAnnexed(C)*, *agentsAnnexed(C)*) waiting in node *v* (as a candidate) then,
  - If *phase(A)* > *phase(C)* then agent *A* annexes agent *C* by adding the value of *C*'s *tname* and *agentsCounter* to its own *tname* and *agentsCounter* respectively. Agent *A* then continues the traversal. Agent *A* doesn't update its label till the end of the traversal. Thus agent *A* uses the variables *tname* and *agentsCounter* to collect the data from the annexed agents.

[2] We use a name such as *u* for the convenience of a description; note that the algorithm does not have an access to unique names such as *u* or *A*.

- If *phase(A) = phase(C)* then agent *A* merges with agent *C* to create a single agent *B* such that *phase(B) = phase(A)*+1, *nodesAnnexed(B) = tname(A) + tname(C)* and *agentsAnnexed(B) = agentsCounter(A) + agentsCounter(C)*, with temporary variables *tname(B) = nodesAnnexed(B)*, *agentsCounter(B) = agentsAnnexed(B)*.
  - If *n* is known and *nodesAnnexed(B)=n* or *k* is known and *agentsAnnexed(B)= k* then *B* detects the successful termination and performs the *leader procedure* (described below).
  - Else: Agent *B* aborts its current traversal and starts *Round I*.
- If *phase(A) < phase(C)* then *A* changes its status to *candidate* and waits at node *v* (aborting its traversal algorithm).

- Otherwise agent *A* acts according to its status and label as follow:

Annexing status

- If the traversal has terminated and *v* is in the *annexed* status:
  - If *A* annexed one or more agents during the traversal (that is *tname(A)*> *nodesAnnexed(A)*) then agent *A* updates its label: *nodesAnnexed(A)*← *tname(A)* and *agentsAnnexed(A)*← *agentsCounter(A)*.(This can happen only in round *I*)
    - If *n* is known and *nodesAnnexed(A)=n* or *k* is known and *agentsAnnexed(A)= k* then agent *A* detects the successful termination and performs the *leader procedure* (described below).
    - Else agent *A* proceeds to *Round II* (described below).
  - In all the other cases *A* changes its status to *candidate* and waits at node *v*.
- If *phase(A) > phase(v)* then *A* performs the *Annexing procedure* (described below) and then continues the graph traversal.
- If *phase(A) < phase(v)* then *A* changes its status to *candidate* and waits at node *v* (aborting *A*'s traversal algorithm).
- If *phase(A) = phase(v)*
  - If *v* is in the *chased* status, agent *A* changes its status to *candidate* and waits at node *v* (aborting *A*'s traversal algorithm).
  - If *v* is in the *annexed* or *round2* status, agent *A* acts as follows:
    - If (*nodesAnnexed(A)*,*agentsAnnexed(A)*)=(*nodesAnnexed(v)*,*agentsAnnexed(v)*) and node *v* is with status *round2* then *A* performs the *Annexing procedure* (described below) and then continues the graph traversal.
    - If (*nodesAnnexed(A)*,*agentsAnnexed(A)*)=(*nodesAnnexed(v)*,*agentsAnnexed(v)*) and node *v* is with status *annexed* then agent *A* continues the traversal. Note that during this phase either agent *A* or an identical agent to *A* has visited node *v* before; in both cases, the edge through which agent *A* entered node *v* is now marked *back edge* for the label of agent *A*.
    - If (*nodesAnnexed(A)*,*agentsAnnexed(A)*)<(*nodesAnnexed(v)*,*agentsAnnexed(v)*) then agent *A* changes its status to *candidate* and waits at node *v* (aborting *A*'s traversal algorithm). (Note that in [20] the agent is destroyed, here the label of the candidate is important to maintain at least one distinguishable agent).
    - If (*nodesAnnexed(A)*,*agentsAnnexed(A)*)>(*nodesAnnexed(v)*,*agentsAnnexed(v)*) then agent *A (1)* changes its status to *chasing*; *(2)* aborts the current traversal; *(3)* starts a new graph traversal in which it acts as described in (see section "chasing status").

Chasing status: When agent *A* labeled (*phase(A)*, *nodesAnnexed(A)*, *agentsAnnexed(A)*) in the *chasing* status, enters a node *v* labeled (*phase(v)*, *nodesAnnexed(v)*, *agentsAnnexed(v)*) with no agent in status *candidate* waiting at node *v*, agent *A* acts as follows:

- If *v* is in the *annexed* status and *phase(A)* = *phase(v)* then *A* (*1*) changes node *v*'s status to *chased*; (2) *A* then continues the graph traversal.
- In all the other cases: *A* changes its status to *candidate* and waits at node *v* (aborting *A*'s traversal algorithm).

Note that the case where an agent in status *candidate* is waiting at node *v* was handled in the beginning of this paragraph in section "Either annexing or chasing status".

3.3.3 Round II

Agent *A* starts a traversal of its own territory (using operation *Go-Territory()* as explained earlier). Whenever agent *A* reaches node *v*:

- If node *v* is in status *annexed* and in the same phase as *A* then agent *A* assigns its own label to the label of the node and assigns the node the status *round2*.
- Else agent *A* changes its status to *candidate* and waits at node *v* (aborting *A*'s traversal algorithm).

At the end of the territory traversal (if not aborted), agent *A* proceeds to round *III*.

3.3.4 Round III

Agent *A* starts an annexing process by starting a graph traversal with the label (*phase(A)*, *nodesAnnexed(A)*, *agentsAnnexed(A)*) and the *annexing* status and follows the rules applied in *Round I*. However, as we show later, an agent at the end of this round will not re-enter *Round II*.

Annexing procedure: Agent *A* annexes node *v* by assigning *A*'s label to node *v*'s label and changing node *v*'s status to *annexed*.

Leader procedure: When *A* detects a successful termination, agent *A* acts as follows:

(1) Agent *A* sets *nodesCounter* ←0

(2) Agent *A* invokes an execution of the graph traversal (using operation *Go-To-Next()* as explained in 3.1) to label every node with a unique name and to construct a map of the graph. Agent *A* labels node *v* by: *(a)* Incrementing the *nodesCounter* by 1; *(b)* Naming node *v* by setting *nodeName(v)*←*nodesCounter*.

The agent uses the labels of the nodes and the marks on the edges (the ports of the nodes) to construct a map of the graph.

4 Complexity and Correctness Proofs

Due to lack of space the proofs were omitted and can be found in http://iew3.technion.ac.il/Home/Users/kutten.html#part4

4.1 Correctness

All the lemmas concerning correctness are based on the assumption that gcd(n,k) =1 and the FIFO discipline behavior of the messages sent over the edges.

Definitions

(1) A *root* is the node from which an agent started the current graph traversal, if it is in the *annexing* mode.
(2) A *chase* is the action where an agent A in phase p changes its status to *chasing* as a result of entering a node labeled by some agent B, or agent A is in a *chasing* status, and continues to follow the traversal of agent B. We call agent A the *chaser* and agent B the *chased*.
(3) A *chase-route* is a sub-graph that consists of the nodes and edges traversed by the same chaser. The node at which the chaser ended the chase is termed as the *end* of the chase-route and the other brink of the chase-route is termed as the *beginning* of the chase-route. The chase-route is a directed path from the *beginning* to the *end*.
(4) A *sequence of chase-routes* is a maximal directed path in a sub-graph formed by the union of sub-graphs of chase-routes from the same phase, where the last node of each chase-route is a node that also belongs to another chase-route in the sequence.

Lemma 1. At any time after phase 0, when there are more than one agent, there are at least two agents with different labels (*phase*, *nodesAnnexed*, *agentsAnnexed*).

Lemma 2. The number of agents that ever reach phase $p > 0$ in an execution of the algorithm is at most $k \cdot 2^{-P+1}$.

Lemma 3. When agent A starts to chase agent B, agent A can enter only nodes labeled with the same phase of A or higher.

Lemma 4. When agent A chases agent B, agent A traverses a simple path in the graph, except possibly for the last traversed node.

Observation 1. If the last node v of the chase-route created by some chaser A creates a simple circle in the chase-route then the chased agent B became a candidate at v or merged with another candidate agent that waited at node v.

Lemma 5. A maximal sequence of chase-routes forms a simple path except possibly for the last traversed node.

Lemma 6. When agent A starts a chase after agent B in phase p, eventually, an agent in phase p+1 exists.

Lemma 7. In rounds *II* and *III*, agent A finds only nodes labeled with the same phase p or higher.

Lemma 8. If there is more than one agent at the highest phase p, then eventually there will be a chase in phase p.

Lemma 9. If there is more than one agent at a certain phase p, then an agent eventually raises its phase from p to p+1.

Lemma 10. In every execution of the algorithm, one and only one agent is left.

4.2 Complexity

The message complexity of an algorithm L acting on a graph G is the maximum number of edge traversals over all executions of L on G.

Lemma 11. The total number of edge traversals made by all agents in a given phase is $O(m)$.

Lemma 12. The total number of edge traversals performed in an execution is $O(m \cdot logk)$.

Bit Complexity: In our algorithm, the graph map is not carried by the agents but is constructed by the elected agent. The information carried by the agents during the election process consists of *phase(A)*, *agentsAnnexed(A)*, and *agentsCounter(A)*, each of size $\log_2 k$, and *nodesAnnexed(A), tname(A), nodesCounter(A)* of size $O(\log n)$. The size of *status* is a constant. The total number of edge traversals during the algorithm is $O(m \cdot \log k)$. Thus, the bit complexity during the election is $O(m \cdot \log k \cdot \log n)$.

In [12] the map is carried by each agent thus the message size is $O(m \cdot \log n)$ and $\log n$ is the size needed to store the identity of a node or an edge. The agent traverses the graph using the map it carries. The total number of edge traversals during the algorithm is $O(m \cdot k)$. Thus, the bit complexity during the execution of the algorithm is $O(m^2 \cdot k \cdot \log n)$. This bit complexity can be reduced to $O(m \cdot k)$ by marking the ports (with 'T' and 'NT') instead of carrying a map. The method of carrying the map by the agents at all times is more robust than the method of marking the port of the node.

Time Complexity: The time complexity in our algorithm is measured by the time it took the leader agent to traverse the graph in each one of the phases it passed. Thus, the time complexity of the algorithm is $O(m \cdot \log k)$.

In the algorithm of Das *et al*, the time complexity is given by the total number of edge-traversals made by all agents together and this was shown to be $O(m \cdot k)$.

5 Conclusion

We used the method of separating the algorithm into two layers - the agent algorithm and the lower layer traversal. The traversal algorithm used here is the DFS algorithm intended for the family of undirected networks. It is possible to generalize the results to other families of networks by using other traversal algorithms. As we showed here for DFS- the fact that two identical agents may visit the same node does not cause the traversal to get "confused". If the algorithm is used for other families of graphs by using other traversals, care must be taken to make sure these traversals do not get "confused" when the routes of identical agents collide.

We addressed instances that are solvable because n and k are co-prime. Recall that there are other solvable cases [19, 24]. The algorithm can be used, with some modifications, to follow the ideas of [24] and solve some other possible cases.

Our algorithm constructs a map only when it can detect a successful termination since we included the map construction in the leader procedure. If desired, it is possible to have every agent construct a map as it goes. This will increase the bit complexity. However, it will ensure that a map is constructed by the last remaining agent also in the case that n and k are co-prime, but neither n nor k is known.

Finally, the model of the algorithm is based on the FIFO behavior of the edges. Communication protocols that do not guarantee the FIFO discipline do exist. Hence, it may be interesting to adapt our algorithm to deal with non-FIFO behavior.

References

1. Afek, Y., Gafni, E.: Time and Message bounds for Election in Synchronous and Asynchronous Complete Networks. SICOMP, Vol.20 No.2 (1991) 376-394.
2. Afek, Y., Matias, Y.: Elections in Anonymous Networks. Information and Computation, Vol. 113, Issue 2 (1994) 312 – 330.
3. Albers, S., Henzinger, M.R.: Exploring Unknown Environments. SIAM Journal on Computing, Vol. 29, No. 4 (2000) 1164-1188.
4. Angluin, D.: Local and global properties in networks of processors. Proc. ACM STOC (1980) 82–93.
5. Awerbuch, B.: A new distributed depth-first-search algorithm. Information Processing Letters, Vol. 20, No. 3 (1985) 147-150.
6. Barriere, L., Flocchini, P., Fraigniaud, P., Santoro, N.: Electing a leader among anonymous mobile agents in anonymous networks with sense-of-direction. Technical Report LRI-1310, Univ. Paris-Sud, France (2002).
7. Barriere, L., Flocchini, P., Fraigniaud, P., Santoro, N.: Rendezvous and Election of Mobile Agents: Impact of Sense of Direction. Theory of Computing Systems (2005).
8. Bender, M.A., Slonim, D.K.: The Power of Team Exploration: Two Robots Can Learn Unlabeled Directed Graphs. Proc. FOCS'94 (1994) 75-85.
9. Chlamtac, I., Kutten, S.: Tree-based broadcasting in multihop radio networks. IEEE Transactions on Computers, Vol. 36, No. 10 (1987) 1209 - 1223.
10. Cidon, I.: Yet another distributed depth-first-search algorithm. Information Processing Letters, Vol. 26, Issue 6 (1988) 301-305.
11. Cohen, R., Fraigniaud, P., Ilcinkas, D., Korman, A., Peleg, D.: Label-Guided Graph Exploration by a Finite Automaton. Proc. ICALP 2005 (2005) 335-346.
12. Das, S., Flocchini, P., Nayak, A., Santoro, N.: Distributed Exploration of an Unknown Graph. Proc. SIROCCO 2005 (2005) 99-114.
13. Dessmark, A., Pelc, A.: Optimal graph exploration without good maps. Theoretical Computer Science, Vol. 326 (2004) 343-362.
14. Diks, K., Fraigniaud, P., Kranakis, E., Pelc, A.: Tree exploration with little memory. Journal of Algorithms, Vol. 51 (2004) 38-63.
15. Fraigniaud, P., Gasieniec, L., Kowalski, D., Pelc, A.: Collective tree exploration. Proc. 6th Latin American Theoretical Informatics Symp. (2004) 141-151.
16. Fraigniaud, P., Ilcinkas, D., Rajsbaum, S., Tixeuil, S.: Space Lower Bounds for Graph Exploration via Reduced Automata. Proc. SIROCCO 2005 (2005) 140-154.
17. Fraigniaud, P., Pelc, A., Peleg, D., Perennes, S.: Assigning labels in unknown anonymous networks. Proc. ACM PODC 2000 (2000) 101–111.
18. Gallager, R.G., Humblet, P.M., Spira, P.M.: A Distributed Algorithm for Minimum-Weight Spanning Trees. ACM TOPLAS, Vol. 5 No. 1 (1983) 66-77.
19. Kameda, T., Yamashita, M.: Computing on Anonymous Networks: Part I-Characterizing the Solvable Cases. IEEE TPDS, Vol. 7, No. 1 (1996) 69-89.
20. Korach, E., Kutten, S., Moran, S.: A Modular Technique for the Design of Efficient Distributed Leader Finding Algorithms. ACM TOPLAS, Vol. 12, No. 1 (1990) 84-101.

21. Kranakis, E., Santoro, N., Sawchuk, C., Krizanc, D.: Mobile Agent Rendezvous in a Ring. Proc. 23rd IEEE ICDCS'03 (2003) 592.
22. LeLann, G.: Distributed Systems - Towards a Formal Approach. Proc. IFIP Congress (1977) 155-160.
23. Panaite, P., Pelc, A.: Exploring unknown undirected graphs. Proc. 9th ACM-SIAM symposium on Discrete algorithms (1998) 316-322.
24. Sakamoto, N.: Comparison of initial conditions for distributed algorithms on anonymous networks. Proc. ACM PODC 1999 (1999) 173-179.
25. Sharir, M.: A strong-connectivity algorithm and its application in data flow analysis. Computers and Mathematics with Applications, Vol.7, No.1 (1981) 67-72.
26. Tarjan, R.E.: Depth first search and linear graph algorithms. SICOMP, Vol. 1, No. 2 (1972) 146-160.
27. Tel, G.: Introduction to Distributed Algorithms. Cambridge University Press (1994) 198-213.
28. Villadangos, J., Cordoba, A., Farina, F., Prieto, M.: Efficient Leader Election in Complete Networks. Proc. 13th PDP'05 (2005) 138-143.
29. Yifrach A.: Improved Distributed Exploration of Anonymous Networks. MSc Thesis, Faculty of Industrial Engineering and Management, Technion Israel Institute of Technology, Haifa, Israel (2006).