Faster Computation On Directed Networks of Automata

Rafail Ostrovsky* Daniel Wilkerson†

TR-94-037

August 1994

Abstract

We show how an arbitrary strongly-connected *directed* network of synchronous finite-state automata (with bounded in- and out-degree) can accomplish a number of basic distributed network tasks in $O(ND)$ time, where D is the diameter of the network and N is the number of processors. The tasks include (among others) the Firing Synchronization Problem; Network Search and Traversal; building outgoing and incoming Spanning Trees; Wake-up and Report When Done; and simulating a step of an undirected network protocol for the underlying graph of the directed network. Our approach compares favorably to the best previously known $O(N^2)$ algorithms of Even, Litman and Winkler [ELW-90] for all these problems.

* U.C. Berkeley Computer Science Division and ICSI. Supported by an NSF Postdoctoral Fellowship and ICSI. E-mail: `rafail@ICSI.Berkeley.EDU`

† U.C. Berkeley Mathematics Department. E-mail: `wilkerso@math.berkeley.edu`

1 Introduction

In this paper we consider *directed* strongly-connected networks of synchronous finite-state automata and consider the efficiency of basic network tasks in this setting. The tasks we focus on include the "Firing Synchronization Problem"; the "Network Search and Traversal Problems" and several others (i.e. DFS, BFS, simulating a step of an undirected graph protocol, etc.) We show $O(ND)$ time solutions to all these problems. Moreover, the techniques developed for these tasks are in fact more general in nature and can be used to solve in a more efficient manner other problems on directed networks.

1.1 The Model

In this paper, we consider strongly-connected *directed* networks. In directed networks, there may exist one-way communication links between pairs of processors without links in the opposite direction. In order to ensure that any processor can talk to any other processor, we require our network to be strongly-connected.

In addition to its theoretical importance, the uni-directional communication occurs in various practical settings as well. For example, unidirectional communication occurs in radio networks (due to different transmission strength), and in VLSI. It also allows us to model the situation in bi-directional networks when one of the communication links has failed.

In this paper we are also concerned with reducing the amount of memory required per each processor. The current technological trend is to implement network protocols in hardware, minimizing the amount of required memory (for further discussion, see [MOOY92]). In this paper, we model processors as identical deterministic finite-state machines (i.e. chips) of a constant size (that is, independent of the size of the network) with a constant number of input and output ports. A finite alphabet if signals is used. The network is constructed by connecting the output ports of automata to the input ports of other automata. Not all input/output ports need to be connected. Thus, each automaton has some of its input and output ports connected to other automata (active ports), and some ports which are not connected at all (inactive ports). We assume that for every pair of processors, in each direction there is a directed path through the network connecting them. That is, the resulting graph is strongly-connected.

We assume synchronization: automata make state transitions simultaneously. Thus, the computation is divided into *time-steps*, where at the beginning of each time-step, each automaton receives a vector of inputs from its input ports (in-ports), performs a state transition, and then sends a vector of outputs to its output ports (out-ports). Initially all automata are in an identical *quiescent* state, in which they send an "idle" signal on all active out-ports. They remain quiescent until receiving a non-"idle" signal on an in-port. In the problems we consider, a user puts one of the nodes into a special "start" state, after which various tasks (which we describe below) must be performed. Of course, the objective is to minimize the time (from

the initial "start" signal) to perform these tasks.

1.2 The Firing Synchronization Problem

The classic *Firing Synchronization Problem* (FSP) has a rich and colorful history (see, for example, overview of [M-86]). In essence, it is the problem of achieving micro-synchronization, given macro-synchronization. That is, given a global pulse, (i.e., given a pulse that all processors can hear at the same time, coming, say, from a satellite) the network wishes to establish a consistent time reference point for (some subset of) the processors of the network. The difficulty is that the network (or a subset of a larger network) may be such that it is not easily signaled separately by the source of the global pulse, and thus the source cannot be used to signal a common time reference point for the subset. More specifically, the problem is as follows: given a synchronous network, all processors start in an identical "quiescent" state, in which they remain until a user puts a single processor into a special "start" state. (Processors must remain in "quiescent" state as long they receive "idle" signals from their neighbors.) Subsequently, at some time after the initial "start" signal, all processors must *simultaneously* go into a special "fire" state (i.e. achieve micro-synchronization). (The two analogies used in the literature for the FSP are the "Simultaneous Neuron Firing" analogy and the somewhat grotessque "Firing Squad" analogy.) Anyway, let t be the number of time-steps after the first "start" signal until the processors "fire". The goal is to minimize this synchronization time t.

In addition to large body of work for undirected networks (see [M-86]), FSP problem was considered for *directed* networks of automata in [K-78, HN-81, ELW-90], where $O(N^2)$ was the best previously known running time [ELW-90]. In this paper, we show how to solve FSP in $O(ND)$ steps, where D is the diameter of the network and N the the number of processors:

Theorem 1 There exists an $O(ND)$ *firing synchronization* algorithm for strongly-connected directed networks of automata.

In order to construct our solution to the firing problem, we first construct an efficient solution to the *traversal* problem on directed networks, as explained below.

1.3 Backwards Communication and Network Traversal

We design a *network traversal algorithm* which can search and traverse an arbitrary strongly-connected synchronous network of automata in $O(ND)$ time. The network traversal algorithm is a fundamental primitive in its own right. For example, it can be used to convert protocols designed for a bidirectional network to run on the underlying undirected graph of the directed network.

The traversal problem is for the single *root* processor to create a token which then visits all the other processors in the network and returns to the root in minimal time

(i.e., minimizing the number of times edges are traversed to get to all the nodes). Our tokens are of constant-size (and can carry constant-size messages with them.) Our algorithm must work for arbitrary strongly-connected networks of automata. This problem is also known as the "Chinese postman problem", where one must completely explore an unknown city which has many one-way streets [AG-93].

In undirected networks, a simple DFS will suffice to traverse the network. However, in directed networks, one-way edges may prevent a token from going "back" on a directed edge, and a "detour" must be found instead. That is, in directed networks, the effort of traversing the network can be reduced to finding an efficient "backwards communication" procedure: for any directed edge $A \rightarrow B$, the procedure finds a short path from B to A, and uses it to simulate moving a message backwards on the edge. A DFS may then be performed using this procedure to traverse edges in the reverse direction [GA-84, Kut88, ELW-90, AG-93].

In this paper we show how a single token can go "backwards" on an edge in an unknown graph (i.e. without any pre-processing, such as the down-tree of [ELW-90].) We call such such an algorithm a *backwards communication* algorithm since the token can carry a constant-size message with it:

Theorem 2 There exists an $O(D)$ *backwards communication* algorithm for strongly-connected directed networks of synchronous automata.

We remark that our theorem is actually stronger than stated, and, in fact, is optimal: we can achieve backwards communication in time proportional to the length of the smallest directed cycle that includes the edge in question[1]. (Our solution should be compared to the best previous [ELW-90] solution for this task which takes $O(N^2)$ time.)

We use backward communication algorithm as a building block our traversal algorithm. Previously, (in the setting of strongly-connected bounded degree directed networks of synchronous automata) [GA-84, AG-93] exhibited a $O(N^2D)$ solution for the traversal problem, and [ELW-90] showed a $O(N^2)$ solution. Using backwards communication, in this paper we further reduce this time to $O(ND)$:

Theorem 3 There exists an $O(DN)$ *network traversal* algorithm for strongly-connected directed networks of synchronous automata.

It should be pointed out that our solution to the traversal problem extends and builds upon the best previously known solution of [ELW-90] for this problem.

1.4 Other Tasks

The traversal and backwards communication algorithms play a central role in the design of many other protocols for unidirectional networks, which we describe below.

[1] for further details, see remark 2 of section 4.

"Wake-up and report when done" is a task requiring a single "lead" processor to send a signal to all the other processors in the network and then go into a special "done" mode only after every processor in the network has received the signal. (There is no requirement of simultaneity as with firing synchronization.) The "up-tree" and a "down-tree" are rooted spanning trees, where in the "up-tree" edges lead away from the root and in the "down-tree" edges lead toward the root node. Each node in such a tree is required also to know exactly which of it's incoming and outgoing edges belong to the tree. The "long circuit — slow clock" is the task of finding an outgoing spanning tree and a cycle through the root of that tree such that the length of the cycle is longer than that of any path in the tree. Of course, the objective is to minimize the height of the tree and the diameter of the cycle. (The name comes from the fact that once set up, a slow clock can be constructed as a message cycling in the long circuit. Within each pass a message may be broadcast to the network that is guaranteed to reach everyone by the completion of the the pass.) In this paper we achieve the following:

Theorem 4 There exists $O(DN)$ algorithms to

- *simulate a step of computation on the underlying undirected graph* ;
- *wake-up the network and report when done* ;
- *construct an up-tree* ;
- *construct a down-tree* ;
- *search* ;
- *find an $O(D)$ long circuit — slow clock* .

on strongly-connected directed networks of synchronous automata.

Previously all these tasks required $O(N^2)$ time [ELW-90]. (Regarding "long circuit — slow clock" we mean that it takes $O(ND)$ time to setup and then $O(D)$ time for each cycle of the slow clock.)

1.5 Organization of the rest of the paper

In section 2 we describe an $O(ND)$ algorithm for the Firing Synchronization Problem, assuming an $O(D)$ backwards communication algorithm. In section 3 we present an $O(D)$ backwards communication algorithm. In section 4 we describe other applications of the techniques developed in sections 2 and 3. Section 5 contains conclusions and open problems.

2 The Firing Synchronization Algorithm

We give solutions to the Firing Synchronization Problem (FSP) for progressively more general families of graphs, ending with the general problem. We show how to solve the general problem in $O(ND)$ time assuming an $O(D)$ backwards communication algorithm, which we describe in the next section.

2.1 Firing on the Directed Ring

We start with the case of directed rings. In fact, for directed rings a solution already exists:

Lemma 1 [K-78, HN-81]: There exists an $O(D)$ FSP algorithm for the directed ring.

For the actual proof the reader is referred to [K-78, HN-81] and for an informal description of their algorithm to [ELW-90]. Here we describe the idea for rings whose diameter is a power of two (the algorithm can be easily extended to non-power-of-two diameter rings as well, see [K-78, HN-81]). First note that the network can simulate a finite number of messages that travel at different speeds relative to one another. Initially, the initiator node simultaneously sends four tokens, traveling at speeds 1, 2, 3, and 4 respectively. (That is, the speed-4 token travels four times faster than the speed-1 token.) Eventually 1 and 3 collide at a point on the opposite side of the ring from the initiator node. Simultaneously with that collision, 2 and 4 collide at the initiator node. Both of these nodes now act as initiator nodes and repeat the process. One may remark that the two halves are no longer rings. The computation of each is identical, however, so that messages coming off of one half are identical to those simultaneously coming on the other end from the other half. Thus each computes as a ring of half the size. Finally when all nodes become initiator nodes (i.e. when each node is an initiator node and its predecessor on the ring is also an initiator node) the ring "fires".

2.2 Firing on the Ring-of-Trees

The above solution can also be extended to the *ring-of-trees*:

Definition 1 A *ring-of-trees* is a ring with an initiator node on the ring and additional directed trees attached to the ring. Each tree has as its root a node on the ring, and the edges are directed away from the root. Further, a path from the initiator node to any leaf of any of the trees is shorter than the path from the initiator node to itself around the ring.

Definition 2 For any ring-of-trees, let D' denote the diameter of the ring.

Lemma 2 [K-78, HN-81]: There exists an $O(D')$ FSP algorithm for any ring-of-trees.

The proof is an extension of the solution to the ring (see, [K-78, HN-81, ELW-90]). The idea of the algorithm is that every processor executes the same ring algorithm, independent of whether it is on the ring or in one of the trees, sending its output to *all* output ports. It is not hard to see that all the nodes which are located at the same distance from the initiator node have the same computational transcript. Thus a node on a tree fires at the same time as the ring node which is at the same distance from the initiator as it is. As before, all ring nodes fire at the same time. Thus, all nodes in the network fire simultaneously.

2.3 Firing on an Arbitrary Network

Next, we reduce the problem of firing on an arbitrary strongly-connected directed graph to that of firing on a ring-of-trees.

One of the building blocks of our construction for the general directed graph is the Backwards Communication algorithm, which we describe in the next section (section 3). In this section, we will assume that the following two versions are possible to implement in $O(D)$ time and without side-effects (once the procedure is finished) for the rest of the network: The simplest version sends a message from node B to node A where there is an edge from $A \rightarrow B$. The second version can send a message from node B to node A where there is only a *marked* directed path from A to B. (A *marked path* is a path in which each node has distinguished the in-port and out-port that connect to the previous and next nodes in the path respectively. Thus messages which are labeled as being "on the path" can pass down the path without carrying with them any further routing information.) Additionally, the second variant finds a shortest directed path from B to A. We elaborate how this can be done in section 3.

With the above two variants of Backwards Communication algorithm we can now explain the general outline of our algorithm. The basic idea is to build a BFS tree and then to find a cycle (to a leaf of the BFS tree and back to the root) which is longer then a height of the BFS tree. The BFS tree together with this cycle cover the entire graph and in fact form a ring-of-trees. Hence we can fire. Below, we describe the actual algorithm, together with analysis of its running time:

Step 1 *A spanning BFS tree is built, with the root as the initiator node.* The initiator node releases a *wakeup message* that propagates in all directions. When a quiescent node first receives this message, the in-port by which it is received is designated as the *parent* of that node in the BFS tree. The node echos the wakeup message to all out-ports. All subsequent incoming copies of this message are ignored.

The edges in this tree point away from the root. Each node is aware of its parent in the tree, but not its children. In particular, no node at this stage of the algorithm knows if it is a leaf or not. (Notice that for the wakeup message to reach all nodes takes $O(D)$ time.)

Step 2 *The leaf with the longest* return cycle *is found.* The second stage starts one step after the first step starts (the messages of step two never "catch up" the messages of the first step). Now we define the procedure of the second step: Define the *return cycle* of a leaf in the BFS tree to be the directed cycle consisting of the path from the root to the leaf in the tree, together with the shortest path in the graph from the leaf back to the root. Recall that the idea is to find a return cycle which is longer then a height of the BFS tree. The longest return cycle will do, in fact.

The root node creates a token which performs a DFS traversal of the BFS tree created in step 1. Whenever the token must backtrack, the backwards communication algorithm is used. Using the backwards communication algorithm, a traversal token can detect if the current processor is a leaf: if none of the processors connected to out-port ports of the current node point to the current node as their parent in the BFS tree, then the current processor is a leaf. Hence, checking if a node is a leaf takes $O(D)$ steps (the token must check every out-port and come back to the current node).

When the traversal token comes to a leaf node, it establishes a marked path of length at most D to the root of the BFS tree. (This is done using the second version of the backwards communication algorithm mentioned above. The algorithm requires an already established marked path in the other direction, for which we use the current path from the root to the DFS token in the BFS tree.) Finding this path back to the root takes $O(D)$ steps.

The longest cycle found so far, the Current Longest Return Cycle (CLRC), is also kept marked. Each time the DFS token finds a new leaf, the length of its return cycle is compared with that of the CLRC and the longer cycle becomes the new CLRC. (The precise mechanism for this comparison is given below.) Notice that at the end of the DFS, the longest return cycle is the CLRC.

To compare the CLRC to the return cycle of the current leaf that the DFS token is occupying, we run a "race". That is, the DFS token (using the marked path to the root established above) signals the root to send send two tokens traveling at the same speed around the two cycles. The token which looses the race was on the longer of the two cycles. That cycle becomes the new CLRC.

Since each such race takes $O(D)$ time and there are at most N leaves, running races for all the leaves takes $O(ND)$ time. Thus the entire DFS, including the races, takes $O(ND)$ time.

Step 3 *An embedded spanning ring-of-trees is constructed and fired.* Notice that the longest return cycle found above together with the tree form a ring-of-trees. This ring-of-trees spans the graph, and the diameter of its ring is at most $2D$.

Using the algorithm of the previous subsection, the embedded ring of trees is fired in $O(D)$ steps.

Thus, we have proved the following:

Theorem 5 There exists an $O(ND)$ *firing synchronization* algorithm on strongly-connected directed networks.

Our theorem is in fact stronger then stated here due to efficiency of our backwards communication algorithm. (See remark 2 in section 4.)

3 The Backwards Communication Algorithm

Suppose we have two nodes, A and B, and we would like to send a message (or token) from B to A, such that only A gets the message, A knows that it is the intended recipient of the message, B knows when A has gotten the message, and at the end of the transaction, the rest of the graph is left undisturbed.

If there is an edge from B to A, this is trivial. Now suppose there is an edge from A to B. Can we somehow use this communication link to get information to move in the opposite direction? Recall that the graph is strongly-connected. So B can broadcast a message in all directions, and it will get to all nodes in $O(D)$ time. So the difficulty is not in getting the message to A, it is in getting the message *only* to A. This is accomplished by the Backwards Communication algorithm.

The types of messages employed by this algorithm, their uses, and their rules of propagation are detailed below. After that follows a top-down description of the algorithm itself. We employ the idea of snakes (and their propagation), introduced in [ELW-90]. A *snake* is a message consisting of many characters that follow each other through the graph. Many types of snakes can be constructed. We use two particular kinds, *growing* snakes and *dying* snakes.

Growing Snakes Our communication alphabet contains a finite number of *growing-snake characters* $g_1, \ldots, g_\delta$ (where δ is the degree of the network) and a tail character, t, distinct from all other characters. As we will see below, it is B that will release growing snakes in order to "find" A. B can do this by simultaneously outputting the character g_i through out-port i, for each out-port, and on the next time-step simultaneously outputting the tail character, t, through each out-port.

Growing snakes propagate as follows [ELW-90]: Upon receiving its first growing snake character, (Ties are broken by choosing the in-port of least index.) a quiescent processor changes to the *tree state*, setting its parent pointer (used in the next section) to the in-port by which the character was received. During the next time-step, the character is rebroadcast through all out-ports. The tree node continues to re-broadcast each character of this snake as they arrive, until the tail is received.

Instead of re-broadcasting the tail, for each out-port i, the node sends the character g_i (to record in the snake that at this node it was sent through out-port number i), thus appending one character to the end of the snake where the tail had been. The next time-step, a new tail is broadcast, completing the snake. All snakes other than the very first to arrive are ignored and thus vanish.

Abort messages To eliminate the growing snakes when they are no longer needed, we introduce a new kind of message which was not present in [ELW-90]: *abort* messages. That is, at some point we no longer need the growing snakes moving through the network, or the tree structure of parent pointers they have made. To do that, B releases *abort* messages. They are single characters that travel two times the speed of growing snakes. (As mentioned above, messages of different speeds are easily simulated.) On contact they eliminate growing snakes and return tree nodes to the quiescent state (also un-setting parent pointers).

The rules by which abort messages propagate are designed to follow growing snakes. Any tree node receiving an abort message by its parent in-port (from any other inport it will be ignored) broadcasts it to all out-ports, un-sets its parent pointer, and reverts to the quiescent state. Any snake character currently waiting to be broadcast is forgotten. Abort messages are ignored by quiescent nodes, so they vanish when the tree is gone. (We explain how and when abort messages are used below.)

Dying snakes Our alphabet also contains *dying-snake characters* $d_1, \ldots, d_\delta$. Upon receiving the first dying-snake character d_i, the node temporarily stores the value of i, but does not output the character (the character is lost and the value of i is forgotten when the snake passes). All subsequent characters of the snake are passed to out-port i. (Unlike the growing-snakes above, we will never use these snakes in a manner in which it will be possible for two dying snakes to collide. Notice that additional message characters may be appended to the end of the snake, and passed along with it. Also there is no need of a dying-snake tail character.) The head of this snake moves at half the speed of the tail, being delayed one time-step at each node while the lead character is consumed. The dying snake characters travel at the same speed as growing snake. For use of the timing message below, the dying snake leaves its path marked as it passes. Also, since the snake is getting shorter and shorter, there is a (unique) node where the snake vanishes completely.

The Timing message We will also need a timing signal (not used by [ELW-90]). As we will see below, with the natural timing of the release of messages in the algorithm, the dying snake arrives at A and delivers its token before the abort messages are done cleaning up the graph. We want A to wait until this cleanup is done before taking further action, such as initiating another backward traversal. By itself A does not know how long to wait. Thus, at the arrival of the dying snake and its token, A releases a *timing message* that travels at unit speed around the same cycle as did the

dying snake (which the dying snake marked for it, above). The message is times such that upon its return, A knows that the graph has been cleaned up. (The passing of the timing message "un-marks" (i.e., deletes the markings of) the cycle.)

The interaction of the two types of snakes, and the abort and timing messages The graph simulates two levels of message interaction. The propagation of growing snakes and abort messages occurs within the first level, whereas dying snakes and the timing message propagate in the second. Messages do not interact across the two levels. The following lemma about growing and dying snakes was used in [ELW-90]:

Lemma 3 *[ELW-90]:* If a growing snake happens to return to its point of origin and is than mutated into a dying snake, it will retrace its path again.

The first traversal of the path will be recorded in the growing snake and this information will be used by the dying snake to duplicate the path. We now have a way to send messages backward, since we have the "address" encoded into a snake. We are also going to use the following lemmas about growing and dying snakes. (These lemmas are new.)

Lemma 4 Suppose there is an edge from A to B and B is the initiator node of growing snakes. Then the first growing snake which returns to B via A traversed the smallest cycle in the graph which includes the edge from A to B.

Let us denote by $\lambda(AB)$ the length of the smallest directed cycle in the graph which includes an edge from A to B. Also, recall that we assume that the head of the growing snake travels at unit speed.

Lemma 5 Suppose there is an edge from A to B and B is the initiator node of growing snakes. Then by the time the tail of the growing snake returns to B via A, $2\lambda(AB)$ time-steps have elapsed and all growing snakes are within a distance of $2\lambda(AB)$ from B.

B releases the abort messages simultaneous with the arrival of the tail of the growing snake from A. Recall that, while the growing snakes have 2λ "head start", and the abort message travels twice as fast as the heads of growing snakes. It is not hard to see that:

Lemma 6 $2\lambda(AB)$ steps after the simultaneous release of the abort messages and the last character of the dying snake, all growing snakes, abort messages, and the BFS tree are gone.

Recall that the timing message is released at the arrival of the last character of the dying snake. Both move at unit speed. Thus, it is again not hard to see that:

Lemma 7 $\lambda(AB)$ steps after the simultaneous release of the first abort message and the last character of the dying snake, the dying snake arrives back A. After $\lambda(AB)$ additional steps, the timing message arrives. Thus when the timing message has arrived all growing snakes, abort messages, and the BFS tree are gone.

Thus our abort messages cleanup the graph in $O(D)$ time, and A and B know when this process is done (thanks to our timing message). The token may now travel backward again on some other edge without interference from the previous backwards communication process. This is the key point of our algorithm, which allows us to avoid a costly "down-tree" [GA-84, AG-93, ELW-90] construction, which was the bottleneck of the previous solutions.

The Algorithm: Consider a setting where B has a token M that it would like to communicate to A, and A has an edge to B, as above. The algorithm is as follows.

- First, B generates growing snakes out of all of its out-ports. Notice that as they spread throughout the graph, the tree created by their propagation is a BFS tree.
- When B receives a snake back again along the edge from A, it immediately mutates it into a dying snake and passes it on according to the dying-snake protocol. To the end of this snake, B appends the token, M.
- Simultaneous with this release of the token, B releases the abort messages through all out-ports. They eliminate the growing snakes and the BFS tree and return all nodes to the quiescent state before vanishing themselves.
- The dying snake returns to A by the shortest path and dies there. That is, when a node finds that the first character after the lead character (which it eliminates of course) of a dying snake is a token (that is, none of the snake is left), then this node knows that it is A, for which the token is intended. A can then keep the token M, and read any message that it is carrying. At this point the timing message is released.
- When the timing message returns to A, sufficient time have passed for the abort messages to clear out the graph. Hence we are done.

REMARK 1: Notice that the fact that A had an edge to B was unnecessarily strong. The algorithm could be easily adopted to the case when A has only a marked path to B: Upon reaching A, the growing snake turns into a dormant snake that simply passes down the marked path without changing at all. Upon reaching B, the snake becomes a dying snake, and proceeds as before.

4 Applications

We now describe how the other aforementioned tasks could be implemented. Simulating one step of an undirected graph protocol on the underlying undirected graph is achieved as follows. A lead node creates a BFS tree and performs a DFS traversal of it, as in the Firing Synchronization algorithm. When traversal token visits a node, it traverses all forward and backward edges adjacent to the node, simulating the necessary messages to be sent in each direction. This takes $O(D)$ time for each node, hence it takes $O(ND)$ time altogether.

The wake-up and report when done is also accomplished by building a BFS tree and doing a DFS traversal. An up-tree is basically a BFS tree and as such may be constructed identically. It is traversed later by a DFS so that each node may determine which of its out-ports are actually its children in the BFS tree. The down tree can be constructed by doing a DFS traversal of backward edges (of course using backwards communication algorithm). The long circuit — slow clock is equivalent to finding a cycle longer than the tree depth. Our FSP algorithm found such a cycle of length $O(D)$, hence we are done.

REMARK 2: For any directed edge e of the network, recall that $\lambda(e)$ denotes the length of the smallest directed cycle which includes e. Let $\Lambda(V, E)$ denote the maximum $\lambda(e)$ for all the edges in the network (V, E). Notice that $\Lambda \leq D$ on any strongly-connected di-graph. We remark that all our theorems can be restated in a stronger form, replacing D by Λ.

5 Conclusions and open problems

We show that by allowing $O(ND)$ time many basic uni-directional network problems can be solved. In this paper we did not address the question of fault-tolerance (i.e. self-stabilization). It would be interesting to address this question as well. Additionally, it is not clear if $N\Lambda$ (or ND) is the best possible running time for FSP and other problems mentioned above. The only known lower bound for any of the above problems is a trivial $\Omega(D)$. It would be interesting to close this gap.

References

[AG-93] Y. Afek and E. Gafni Distributed algorithms for Unidirectional networks. manuscript.

[ELW-90] S. Even, A. Litman and P. Winkler Computing with Snakes in Directed Networks of Automata. *FOCS 1990.*

[GA-84] E. Gafni and Y. Afek Election and Traversal in Unidirectional Networks *PODC 1984.*

[HN-81] N. Honda and Y. Nishitani The Firing Squad Synchronization Problem for Graphs *Theoretical Computer Sciences* Vol 14. 1981.

[K-78] K. Kobayashi The firing squad synchronization problem for a class of polyautomata networks. *Journal of Computer and System Science* 17:300-318, 1978.

[Kut88] S. Kutten Stepwise construction of an efficient distributed traversing algorithm for general strongly-connected networks. *Proceedings of the 9'th International Conference on Computer Communication* pp.446-452, October 1988.

[M-86] J. Mazoyer An overview on the Firing Squad Synchronization Problem. in *Automata Networks* Springer Verlag LNCS 316, 1986.

[MOOY92] A. Mayer, Y. Ofek, R. Ostrovsky, and M. Yung Self-Stabilizing Symmetry Breaking in Constant-Space *STOC 1992*.