# Computing with Snakes in Directed Networks of Automata

*Shimon Even*, Ami Litman* and Peter Winkler**

*ABSTRACT*

We consider unidirectional, strongly connected networks of identical finite-state automata, of bounded in- and out-degree but unknown topology and unbounded size $n$. Protocols which are quadratic or linear in $n$ are provided which accomplish the following tasks: wake-up and report when done; construct spanning trees out from the root and in to the root; conduct breadth-first and depth-first searches; send a message from the end-point of an (unidirectional) arc to its start-point; run a slow clock; and solve the firing squad synchronization problem. Our protocols are highly parallel and use a new technique — a special kind of moving data structures we call *snakes.*

---

* Part of this work was done while the authors were with Bellcore, Morristown, NJ.

Shimon Even is a faculty member in the Comp. Sci. Dept. of the Technion -- Israel Institute of Technology, Haifa, Israel. His work there was supported by the Fund for the Promotion of Research at the Technion. He is now visiting Bell Laboratories, Lucent Technologies, 700 Mountain Ave., Murray Hill, NJ. E-address: even@cs.technion.ac.il

Ami litman is a faculty member in the Comp. Sci. Dept. of the Technion -- Israel Institute of Technology, Haifa, Israel. E-address: litman@cs.technion.ac.il

Peter Winkler is with Bell Laboratories, Lucent Technologies, 700 Mountain Ave., Murray Hill, NJ. E-address: pw@bell-labs.com

# 1. Introduction

## The Model

We consider unidirectional networks which are *strongly connected*, that is, there is a directed path from any node to any other node. The number $n$ of nodes in the network will be unknown and unbounded; the topology of the network will also be unknown. The network may have parallel arcs, anti-parallel arcs and self-loops.

The nodes will consist of identical finite-state automata, with $d$ in-ports and $d$ out-ports, each of which *may* be connected to another node. Each such connection is an (uni-directional) arc from an out-port of one node into an in-port of another (no fanout). We call the connected ports *live ports* and the unconnected ones *dead ports*.

A fixed finite alphabet of signals is used. One of these signals, $b$ (blank), is received by dead in-ports; in an application $b$ might correspond, e.g., to absence of an electrical pulse. A node responds deterministically to a vector of input signals (one from each in-port) and to its current state, by entering its next state and outputting, simultaneously, one signal through each out-port.

The network is synchronous – there is a global clock whose beats are heard by all automata at the same time. All state transitions and output transmissions occur on the beat. It is assumed that the time interval between beats is sufficient for all messages to arrive at their destination, and for all necessary computations to be completed. Thus, the network changes its global state in discrete steps.

Initially, each automaton is assumed to be in a *quiescent* state, during which time it produces $b$'s on all its out-ports. It remains in that state as long as all its in-ports deliver $b$'s. One of the nodes, called the *lead node*, may receive instructions from the outside world (say, a user), which instruct it to start a certain computation in the network; clearly, such an instruction will cause the lead node to leave its quiescent state.

We will consider two variants of the model:

(1) **In-port awareness:** Each node "knows" (by virtue of its initial state) which of its in-ports are dead.

(2) **Out-port awareness:** Each node knows which of its out-ports are dead.

We will accomplish all the following tasks with either in-port awareness or out-port awareness; neither of these task is possible without any awareness.

## The tasks

Following are several objective tasks for the directed network. Some are "ends in themselves" but most are useful primarily as tools for computing. Some of the tasks are closely related.

**Wake-up and report when done:** The lead node is contacted by an outside agent and asked to signal the others, then report to the agent when all the nodes in the network have received the signal. This task was considered in [S83] where it was called "PIF" (propagation of information with feedback).

**Construct an up-tree:** The lead node must initiate construction of a spanning tree with itself as root, and all arcs leading away from the root. Each node of the network must know the in-port leading from its parent and the out-ports leading to its children (if any) on the tree.

**Construct a down-tree:** Same as above, but the arcs of the spanning tree must lead in toward the root.

**Conduct a search:** The lead node must initiate a depth-first search of the up-tree.

**Traversal by a token:** A set $T$ of "token" states is designated for the automata. The lead node must initiate a process by means of which every node enters, at least once, a state in $T$, but at no instant are two nodes permitted to be simultaneously in states of $T$.

**Backwards communication:** Any node must be able to send a message specifically to the node connected to any one of its live in-ports.

Protocols for the last two tasks allow one to run, on a unidirectional network, many distributed algorithms which are designed for bidirectional networks. (Usually there is a cost in time-complexity.) For example, one can find a BFS spanning tree of the underlying undirected graph.

**Long circuit - Slow clock:** A directed circuit through the root must be established such that it uses any node or arc at most twice and its length is at most $2n$ and at least half of the directed diameter of the network.

This circuit can then be used to establish a slow clock – a free running clock whose period is the length of the circuit. Such a clock can be used by the nodes to learn that a certain operation is over.

**Detect dead ports:** A node must be able to determine whether a particular in-port or out-port is dead.

**Firing squad synchronization problem (FSSP):** Let $F$ be a special "firing" state. The lead node initiates a process which will result in all nodes entering state $F$ at precisely the same instant.

## Our Results

**Theorem 1.** The tasks "wake up and report when done", "construct an up-tree", "construct a down-tree", "traversal by a token" and "detect dead in-ports" can be done in time $O(n^2)$ on a synchronous, strongly connected, unidirectional network of $n$ identical automata, assuming either in-port or out-port awareness.

Without either in-port awareness or out-port awareness, no protocol exists which can accomplish the "wake up and report when done" task (nor, therefore, any of the others).

*Proof.* The positive results will follow from Protocol "A" and its variants, described below. The last statement follows by considering any network in which the lead node itself has undetected dead in-ports and out-ports. A protocol which purports to have finished the "wake up and report when done" task in time $t$ would have to give the same result erroneously if faced with the same network, but with a directed cycle of length $t+1$ attached to the lead node. □

**Theorem 2.** The tasks "construct a slow clock", "firing squad synchronization", "detect dead out-ports", and "backwards communication" can all be done in quadratic time on a synchronous, strongly connected, directed network of $n$ identical automata with either in-port or out-port awareness; and even in linear time, if the tasks of Theorem 1 have been accomplished.

*Proof.* See Protocols "B" and "C" below. □

**Theorem 3.** Any subtree of a synchronous, strongly connected, directed network of automata can be searched by a token, either in depth-first fashion (in quadratic time) or breadth-first (cubic time). Each edge of the subtree must be "known" by one of its endpoints to be in the subtree, but may be oriented in either direction, and as usual we assume in-port or out-port awareness.

*Proof.* This is fairly easy using the backwards communication of Theorem 2. □

## The Technique

The major limitation of our model is the bounded memory of the network nodes. We overcome this limitation via a new mechanism called *snakes.* A snake is a sequence of symbols that travels through the network as a unit. One may consider a snake to be a traveling data structure whose size is not bounded by the memory capacity of any single node.

We employ three species of snake: *adders* (which grow and reproduce as they move up the up-tree), *moccasins* (which run down the down-tree without changing), and *rattlers* (which shake off symbols, shrinking as they proceed up the up-tree.) Enough different signals are available so that one snake of each type may simultaneously be traveling on the same arc, along with certain other messages.

A snake will have an identifying *head:* adders will begin with the symbol *A*, moccasins with *M*, and rattlers with *R*. All snakes will terminate with a *tail T*. Between the head and tail will be the *body* – a string of integers, each between 1 and *d*, where *d* is the fixed bound on in- and out-degree. To simplify our accounting, the head and the tail are not separate symbols but are superimposed on the first and last symbol of the body, respectively; hence, the length of a snake is the length of his body.

**Prior Work**

When this work was presented in [ELW90], it accomplished all tasks listed above in less time than it was previously known. The firing squad problem, in particular, was previously solvable only in exponential time; and backwards communication, by means of which undirected computations can be simulated, was not known to be possible at all. In addition, we had protocols which solve these problems for the first time in the case where the automata have no prior knowledge of which of their out-ports are connected to other automata and which are dead, provided they have this knowledge about their in-ports.

Most of the literature concerning the tasks listed above assumes bidirectional networks; however, Gafni and Afek [GA84] have presented serial protocols for unidirectional networks which construct up- and down-trees, accomplish "wake up and report when done", and "traversal by a token" in time $O(n^2 \log n)$. Their protocols assume outport awareness, as does Kobayashi's below.

The firing squad synchronization problem has a colorful history, detailed in the excellent survey of Mazoyer [M86]. In his efforts to solve the FSSP for bidirectional networks, Nishitani invented a special structure, which consists of one directed ring with directed trees (directed from their roots outward) whose roots are on the ring, and such that the distance from the lead node (which is assumed to be on the ring) to any node is shorter than the length of the ring. He was able to solve the FSSP for such directed networks in linear time (see [NH81]). Kobayashi [K78] considered the firing squad problem for general directed, strongly connected networks, and found an exponential-time solution, ending with Nishitani's trick. Prior to [ELW90], Kobayashi's protocol was the only FSSP protocol available for general directed networks.

Recently, Afek and Gafni [AG94] have presented new protocols for constructing up- and down-trees, and for traversal by a token. The time complexity of their protocols is $O(n^2)$ – the same as ours.

## 2. Protocol A: Construction of up and down trees.

In order to keep the presentation simple, we first describe a protocol for a network with in-port awareness. A variant for a network with out-port awareness will be indicated in the remarks.

This protocol entails the construction of two trees, an "up-tree" and a "down-tree", each with root at the lead node. The arcs of the up-tree will lead away from the root, while those of the down-tree will lead toward the root. Nodes will eventually join both trees, so to avoid confusion we will refer to "father" and "sons" when speaking of the up-tree and "mother" and "daughters" for the down-tree. In both cases the root plays the role of the unique common ancestor, and, initially, the sole member.

The up-tree will initially be dumb, in that each node will know its father but not its sons; a node will broadcast to all its neighbors without knowing which, if any, are its sons. The up-tree is built quickly by broadcasting and is constant thereafter.

The down-tree will be smart, each node knowing its mother and its daughters; in particular each node of the network will know at every instant whether it is on the down-tree and if so whether it is a leaf. The down tree is built more slowly, but as in the up-tree, links once made are never broken.

The lead node begins by sending, repeatedly, the adder $AiT$ (of length one) to out-port $i$, for each $i$. Adders which reach the lead node are "killed", i.e. are ignored. Each other node will behave in the following way with respect to adders:

(1) The first time it receives an adder, it designates the corresponding in-port (breaking ties arbitrarily) as the port leading from its permanent father in the up-tree. Subsequently, all adders coming in at other in-ports are killed.

(2) After receiving the head of an adder from its father in the up-tree, the node begins broadcasting the adder to all of its out-ports. To the end of the body of each emanating adder, however, the node appends its number for the corresponding out-port. Thus as many as $d$ new adders are created, each one longer by one than the incoming adder.

(3) If another adder follows on the heels of the one being input, i.e. $T$ followed immediately by $A$, that new adder is killed. (Otherwise a node might be forced to remember an unbounded amount of information.)

The node has additional responsibilities depending on whether it is a member of the down-tree.

If the node is not in the down-tree, then whenever it sends out an adder on an arc, it also *generates* a moccasin on the same arc; that moccasin is identical to the adder except for the head. Thus moccasins are being sent through every out-port of each node not on the down-tree. The node kills all arriving moccasins.

If the node is in the down-tree, then it does not generate moccasins but it forwards all incoming moccasins (from any in-port) to its mother in the down-tree. However, any moccasin which arrives while another is passing through the node is killed; also, if several moccasins arrive at exactly the same time, one is chosen arbitrarily for forwarding and the others are killed.

Moccasins which survive all the way down to and through the root are transformed into rattlers, i.e. the $M$ head is changed to an $R$. The first number in the rattler's body tells the root to which out-port to send the rattler; that number is eliminated by the root, shortening the snake by one unit. Each subsequent node, when it receives a rattler, eliminates the first number in the body and sends the diminished rattler to the out-port corresponding to that number. (Note that to shorten the rattler in this manner, each node must delay the transmission of the head by one beat.)

In this manner the rattler retraces the precise steps of the adder which spawned it. When the rattler is down to only one integer in its body, it is at the node which earlier had generated a moccasin from that adder; the remaining integer thus indicates an out-port which leads to a member of the down-tree.

Consequently, when a node receives a rattler with only one integer left in it, it behaves as follows:

(1) If it is already a member of the down-tree (having joined since the original adder was changed to a moccasin), it ignores the rattler.

(2) If it is not yet a member of the down-tree, it joins the down-tree, using the rattler's last integer to identify the out-port which leads to its mother. A signal is sent through this out-port saying "I am now one of your daughters in the down-tree." In addition, a special signal $S$ is sent to all the *other* out-ports saying in effect "I have joined the down-tree, do not wait for me." (The lead node sends its $S$ signals out in the first round.)

As a member of the down-tree, a node contracts yet another duty: as soon as it has received an $S$ from every live in-port, it sends an $S$ to its mother in the down-tree. (It is at this point when we use the fact that each node knows which of its in-ports are actually connected to something; see Remark 1 below concerning dead ports.)

When the lead node reaches the point where it qualifies for sending an $S$, that is, when it has received an $S$ from every in-port, it instead signals that the protocol is terminated, meaning that every node has been reached and is in fact now on the down-tree. To see that this is really so, assume otherwise; then by strong connectivity, there is a node $x$ not on the down-tree with an out-port leading to a node $y$ which is on the down-tree. Since $x$ never sent an $S$ to $y$, some node on the down-path from $y$ to the root must have sent an $S$ to its mother without having received one from every in-port, contradicting its instructions.

To see that this protocol terminates, and that it does so in time $O(n^2)$, we investigate the snakes traffic.

First, let us consider the sizes of gaps between adders. The lead node is continuously sending out adders of length one with a zero gap. We claim that the nodes at level $k$ of the up-tree will be transmitting gaps of a fixed length $g(k)$ which is at most $k$. For, if $g(k-1) > 0$, we will have $g(k) = g(k-1) - 1$; if $g(k-1) = 0$ then every other snake will be killed and $g(k)$ will be one less than the length of an incoming snake, that length being $k$. It follows that moccasins (of length at most $n$) will be generated with gaps of at most $n-1$.

Next, we consider moccasins. Let $e$ be an arc and $B = [b, b']$ a time interval. We use the following terminology.

- Arc $e$ is *clear during $B$* if no symbol of a moccasin moves on $e$ during the interval.
- Arc $e$ is *heavy during $B$* if $e$ is not clear during any subinterval of $B$ of the form $[c, c+n]$.
- Arc $e$ is *steady during $B$* if either $e$ is clear during $B$ or $e$ is heavy during the interval $[b+n, b']$.

The following "moccasin lemma" is critical.

**Lemma.** Let $B = [b, b']$ be a time interval and $x$ a node that joined the down-tree before time $b$. Assume every arc into $x$ is steady during $B$. Then $e$, the down-tree arc out of $x$, is steady during the interval $B+1=[b+1, b'+1]$; moreover, the arc is clear during $B+1$ if, and only if, every arc into $x$ is clear during $B$.

*Proof.* The statement is obvious if all the incoming arcs are clear, so let us assume otherwise. Let $C = [c, c+n]$ be a subinterval of $[b+n, b']$. We need to show that $e$ is not clear during $C+1$.

Say first that at time $c$ a moccasin is moving on each heavy incoming arc. Then one of these moccasins must be passing out of $x$ at time $c+1$, since they can't all have killed one another, and no moccasin has passed sufficiently recently on a clear arc to have killed any of them.

Say next that at time $c$ one of the heavy incoming arcs is not sending a moccasin. Since the arc is heavy, a new head will come from this arc at time $t \in C$. At time $t+1$, node $x$ will send on $e$ either this head or a symbol of another moccasin that killed this head.

In both cases, arc $e$ is not clear during $C+1$, proving the lemma. □

To prove that Protocol "A" terminates in time $O(n^2)$, it suffices to show that for any positive integer $s$, either all nodes are on the down-tree by time $s$ or some new node joins the down-tree during the time interval $[s, s+6n]$. Accordingly, let us suppose otherwise, and let $D$ be the set of nodes in the down-tree at time $s$.

Let $v$ be a leaf of $D$, $e$ an arc from $u$ to $v$ and $B=[s+n, s+6n]$. If $u$ is in $D$ then $e$ is clear during $B$. If $u$ is not in $D$ then $u$ is generating moccasins with gaps of at most $n-1$ on all its out-ports; hence the arc $e$ is heavy during $B$.

We conclude that every leaf of the down-tree will satisfy the conditions of the lemma for the interval $B$. Inductive applications of the lemma tell us that the root itself will satisfy the conditions of the lemma for the interval $[s+2n, s+6n]$. Accordingly, some part of some *rattler* will leave the root during the interval $[s+3n, s+4n]$. That rattler was transformed from a moccasin that was generated by a node $v$ after time $s$; hence, $v$ is not in $D$. The tail of that rattler will leave the root before time $s+5n$ and before time $s+6n$ will have caused the node $v$ to enter the down-tree, creating the desired contradiction.

While Protocol A runs, each node learns which of its out-ports lead to its sons in the up-tree; they are precisely those out-ports to which the node has passed rattlers. Thus, upon termination the up-tree is smart as well.

**Remark 1: Protocol A under Out-port Awareness.**

As noted, Protocol A assumes in-port awareness but not out-port awareness. The following modification works with the reverse assumption.

In the modified version, there are no *S* signals. When a node joins the down-tree, it continues to generate moccasins on all its live out-ports but the one leading to its mother. When a rattler arrives indicating that a particular out-port leads to a node of the down-tree, that out-port is so marked and moccasins are no longer generated on this out-port. When all live out-ports have been marked, the node no longer generate any moccasins; we say then that the node is *complete.*

Now, every node, as soon as it has been reached by an adder, begins sending repeated *Z*'s via all of its live out-ports, and it continues to do so until it becomes complete. Even then, however, it continues passing received *Z*'s to its mother, every time it receives a *Z* from any in-port. Thus, conceivably a complete node may stop sending *Z*'s down the down-tree and then start again when a previously inactive in-port sends its first *Z*.

The proof for Protocol A now shows only that at least every $6n$ steps some node gets to mark an out-port, but this does mean that after at most $6dn^2$ steps every node will be complete; after at most another $n$ steps no further *Z*'s will be sent.

When the root itself becomes complete and also stops receiving *Z*'s, it calls a halt to the protocol. At this point every node is complete, for if not, there is an arc $x \rightarrow y$, leading from a complete node $x$ to an incomplete node $y$. At the instant when $x$ became complete $y$ must already have been on the down-tree; but at the instant $y$ joined the down-tree, the root was *already* receiving *Z*'s initiated by $y$, since $y$ began producing them together with its first moccasins, and one of its moccasins must have reached the root before $y$ joined the down-tree. Since $y$ never became complete that stream of *Z*'s was uninterrupted, contradicting the assumption that the root has stopped hearing them.

Note that upon completion of the protocol, each node has in fact identified its live in-ports; in contrast to the fact that after running the original version of Protocol A it is *not* the case that each node has identified its live out-ports (only its sons in the up-tree and mother in the down-tree). However, using the token system discussed in Section 3, and the slow clock constructed in Section 4, any node may subsequently find all of its live out-ports in linear time.

## 3. Protocol B: Depth-First Search of the up-tree.

Once the two trees have been constructed by Protocol A, our next goal is to show how to move a token over all the nodes, in a DFS manner, on the up-tree. Each node can hold the token as long as it wishes.

The *token* is born at the root. As the token moves around, it is either *active* or *passive*. If the token is active, let us call the node at which it is currently located the Center of Activity (CA).

For each node, let us regard the out-ports leading to sons as an ordered set. Each time a node becomes the CA it passes the active token upward to the first son who has not yet received it.

If the CA has no son, or if the token has returned (backtracked) from its last son, the CA has to backtrack the token to its father. Here is how this is achieved:

The CA sends to the root, via the down-tree, a request to send out one wave of adders. Nodes which are not the CA do not produce moccasins at all. When an adder reaches the CA, it converts it to a *Token Carrying Moccasin (TCM)*, without adding the symbol for the out-port, and sends it to its mother. The TCM carries the token, in passive form, at its head; since the token is not now active, there is currently no CA.

TCM's are passed from daughter to mother by each node other than the root. The root converts it into a *Token Carrying Rattler (TCR)*. When a node gets a TCR with a single symbol, $i$, it records $i$ and kills the TCR. The node recognizes that it has just received the active token, thus becomes the new CA.

When the lead node has received the token from its last son, the process is terminated.

This completes the description of backtracking and DFS is accomplished.

**Remark 2: Backward communication.**

In a similar manner, any message can be sent by a CA to any neighbor. Sending a message via an out-port is trivial. Sending a message via an in-port is conducted as follows:

Again, CA requests a wave of adders, but waits to get one through the designated in-port, instead of its father. The remainder of the protocol is similar to the description above, except that the snakes carry a message which is not necessarily the passive token.

This technique allows one to move a token around in any fashion, not necessarily DFS. For example, one can perform (a slow) BFS on a subnetwork, and the arcs of the subnetwork need not be oriented in any particular way.

## 4. Protocol C: Long circuit - Slow clock.

This protocol uses the up- and down-trees created above to find in linear time a longest directed circuit through the root, subject to the condition that it goes up the up-tree and then down the down-tree, with at most one non-tree edge in between. Let $c$ be such a circuit. Clearly, $c$ uses any node and any arc at most twice, hence its length is at most $2n$.

The length of $c$ is at least half the directed diameter of the network. To show that, let $u$ and $v$ be nodes such that the length of the shortest directed path from $u$ to $v$ is the network diameter. Let $p_1$ be the directed path from $u$ to the root in the down-tree, and $p_2$ the directed path from the root to $v$ in the up-tree. The length of either $p_1$ or $p_2$ is at least half the diameter, while the length of $c$ is no less than the lengths of $p_1$ and $p_2$.

The long circuit is established as follows. The root sends an adder simultaneously to each out-port, as in Protocol A, but only once. Moreover, the root generates one moccasin on each of its out-ports. The adders are passed up the up-tree, as before, and again moccasins are generated by each node on all its out-ports *except* the one leading to the node's mother.

Arriving moccasins are at first killed, but the in-ports at which they arrive are marked. When just one live in-port is still unmarked, the moccasin arriving at that in-port is *not* killed but instead passed on to the node's mother in the down-tree. Ties are broken arbitrarily.

It is apparent that precisely one moccasin will pass along each arc of the network; eventually one moccasin will survive the trip down the down-tree and not be killed by the root. That moccasin, which has taken a longest path, is changed to a rattler and sent up the same path, then back down the down-tree in a body-less *MT* form. As it passes, each node places a marker on the in-port and out-port used, thus etching the circuit into the network's memory. (If a node is used both going up and going down it employs two sets of markers.)

To generate a slow clock via this circuit, each node on the circuit (including the root) is responsible to pass an incoming *tick* signal to its successor on the circuit. The root starts the clock by sending an initial tick down the circuit. This establishes a free running clock whose period is at most $2n$ and at least half of the directed diameter of the network. This clock is available only to nodes on the circuit. To make it available to all nodes, the root broadcasts a *tack* signal on the up-tree whenever it hears a tick. (Note that the nodes do not hear the tack signal simultaneously; however, even a synchronous slow clock is possible – via the firing squad discussed below.)

Once the slow clock is available, nodes can use it to learn that a certain operation is over. For example, if a node broadcasts a message, by the time *three* tacks have passed, it can be sure that every node has got the message.

If a node "wishes" to determine whether a particular out-port is alive (see Remark 1 above), it waits until it is the CA. It sends a test signal $J$ through the suspect out-port. If the out-port is alive, the receiving node broadcasts $J$. If the testing node has heard three tacks without getting a $J$, the node correctly concludes that the out-port is dead. Clearly the node maintains the token, and thus, its status as the CA, as long as the test is not terminated.

## 5. Solving the firing squad problem in O($n^2$) time.

Once a long circuit has been found, as in Section 4, it is not hard to solve the firing squad synchronization problem (FSSP) in linear additional time. We use Nishitani's idea.

In order to keep this exposition self contained we shall first explain a simple linear time solution to the FSSP in a directed ring. This solution takes time $3n$, while $2n$ solutions exist. (See [NH81]; It is not difficult to show that $2n$ is the best possible.) For further simplicity, assume $n$, the number of nodes in the ring is a power of 2.

The lead node is the first "general". He begins the process by sending three types of signals: $\alpha$, which travels at full speed; $\beta$, which travels at two-thirds of the full speed; and $\gamma$, which travels at one-third speed. An additional signal, $\delta$, travels at full speed, but is killed by the first soldier (non-general) to receive it. Here are the rules of behavior:

- A new general, or an old general upon receiving $\beta$, emits all four signals at the same time.
- A soldier becomes a general if $\alpha$ and $\gamma$ reach him simultaneously.

- A general "fires" if δ is received.

To see that all nodes will "fire" at the same time, consider the beginning of the process. After $3n/2$ beats, α will arrive simultaneously with γ at the soldier diametrically opposite the original general. This new general, as well as the original one, having received β at the same time, send all four signals. After $3n/4$ additional beats, the two soldiers midway between the previous two generals will become generals, while all four send simultaneously a new wave of signals. Four beats before the "execution", half the nodes will be generals, with single soldiers separating them. Within three more beats, all these soldiers become generals, and one beat later, all fire.

Nishitani's idea, used by him to solve the FSSP in undirected networks, is as follows: Assume there is one directed circuit (the ring), which includes the lead node. All nodes, which are not on the ring, are on directed up-trees whose roots are on the ring. Also, the unique simple directed path from the lead-node to any node is shorter than the length of the ring. Now each node runs the program for solution on a ring, whether it is on it or not. (It does not even need to know.) Signals are passed to all sons, as if each were the successor on the ring. It is not hard to see that the nodes off the ring will fire at the same time as those on it.

The long circuit created by protocol B serves as our ring, and the up-tree takes care of all the others. Clearly, this is the structure required by Nishitani, hence our solution.

It takes $O(n^2)$ to prepare the up-tree and down-tree (via Protocol A). Then one can find a long circuit in $O(n)$ time, using Protocol C, and solve FSSP in additional time $O(n)$. Thus, the whole process runs in time $O(n^2)$, in contrast to $O((2d^2+1)^n)$ for Kobayashi's protocol [K78].

## 6. References

[AG94] Afek, Y., and Gafni, E., "Distributed Algorithms for Unidirectional Networks", *SIAM J. Comput.,* Vol. 23, 1994, pp. 1152-1178.

[ELW90] Even, S., Litman, A., and Winkler, P., "Computing with Snakes in Directed Networks of Automata", *Proceedings of the 31th Annual Symposium On Foundations of Computer Science,* 1990, pp. 740-745.

[GA84] Gafni, E., and Afek, Y., "Election and Traversal in Unidirectional Networks", *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, 1984, pp. 190-198.

[K78] Kobayashi, K., "The Firing Squad Synchronization Problem for a Class of Polyautomata Networks", *Journal of Computer and System Sciences,* Vol. 17, 1978, pp. 300-318.

[M86] Mazoyer, J., "An Overview of the Firing Squad Synchronization Problem", in *Automata Networks* (C. Choffrut, Ed.), *LNCS* 316, Springer Verlag, 1986, pp. 82-94.

[NH81] Nishitani, Y., and Honda, N.,"The Firing Squad Synchronization Problem for Graphs", *Theoretical Computer Science,* Vol. 14, 1981, pp. 39-61.

[S83] Segall, A., "Distributed Network Protocols", *IEEE Transactions on Information Theory* Vol. 29, January 1983.