A Pursuer-Evader Game for Sensor Networks

Murat Demirbas†   *Anish Arora*†   *Mohamed Gouda*‡

† Department of
Computer & Info. Science
The Ohio State University
Columbus, OH 43210 USA

‡ Department of
Computer Sciences
The University of Texas at Austin
Austin, TX 78712 USA

Abstract

In this paper we present a self-stabilizing program for solving a pursuer-evader problem in sensor networks. The program can be tuned for convergence speed or energy efficiency. In the program, sensor motes close to the evader dynamically maintain a tree of depth r that is always rooted at the evader. The pursuer, on the other hand, searches the sensor network until it reaches the evader tree, and then follows this tree to its root in order to catch the evader.

Keywords : Sensor networks, pursuer-evader games, distributed algorithms, convergence rate, energy minimization.

1 Introduction

In this paper we present a tunable, energy-efficient, and self-stabilizing algorithm for solving a pursuer-evader problem for sensor networks. The goal of the pursuer is to catch the evader (despite the occurrence of faults) by means of information gathered by the sensor network. The pursuer can move faster than the evader. However, the evader is omniscient –it can see the state of the entire network– whereas the pursuer can only see the state of one sensor node (say the nearest one).

Before presenting our hybrid, tunable program for achieving this goal, we present two orthogonal approaches: an evader-centric program and a pursuer-centric program.

In the evader-centric program, motes communicate periodically with neighbors and dynamically maintain a tree structure that is always rooted at the evader. We prove that the pursuer eventually catches the evader by following this tree structure to the root: the pursuer asks the closest sensor node who its parent is, then proceeds to that node, and thus, reaches the root node (and hence the evader) eventually.

In the pursuer-centric program, motes communicate with neighbors only on the request of the pursuer: When the pursuer reaches a mote, the mote resets its recorded time of a detection of an evader to zero and directs the pursuer to a neighboring mote with the highest recorded time.

In the hybrid program we combine the evader-centric and pursuer-centric approaches:

1. We limit evader tree to a depth r instead of entire network to save energy.
2. We modify pursuer-centric program to exploit the evader tree structure.

Organization of the rest of this paper. After presenting the system and fault model, we present an evader-centric program in Section 3 and a pursuer-centric program in Section 4. In Section 5, we present a tunable, hybrid program combining the previous two programs.

2 The Problem

System model. A sensor network consists of a (potentially large) number of sensor nodes (called motes). Each mote is capable of receiving/transmitting messages within its field of communication. All motes within this communication field are its neighbors; we denote this set for mote j as $nbr.j$. We assume the nbr relation is symmetric and induces a connected graph. (Protocols for maintaining biconnectivity exist [?], which are transparent to the applications.)

Problem Statement. Given are two distinguished processes, the pursuer and the evader, that each reside at some mote in the sensor network. Each mote can immediately detect whether the pursuer and/or the evader are resident at that mote.

Both the pursuer and the evader are mobile: each can atomically move from one mote to another, but the speed of evader movement does not match or exceed the speed of the pursuer movement.

The strategy of evader movement is unknown to the network. The strategy could in particular be intelligent, with the evader omnisciently inspecting the entire network to decide whether and where to move. By way of contrast, the pursuer strategy is based only on the state of the mote at which it resides.

Required is to design a program for the motes and the pursuer so that the pursuer can "catch" the evader, i.e., to guarantee in every computation of the network it is eventually the case that both the pursuer and the evader reside at the same mote.

We assume partially synchronous semantics of computation. Each mote has a clock, that is in loose synchrony with the clocks of other motes. (This is a reasonable assumption at least for the case for Mica motes [1]. We note that this choice of semantics is for simplicity: elsewhere, we present an asynchronous version of the solution.)

Programming Model A program consists of a set of variables and set of actions at each mote. A state

We assume maximal parallel semantics of computation. At each state, each mote executes all actions that are enabled in that state. (Execution of multiple enabled actions in a modes is treated as executing them in some sequential order.)

We assume that each mote has a clock, that is in synchronized with the clocks of other motes. (This assumption is reasonably implemented at least for Mica motes [1].)

Fault model. Transient faults may corrupt the state of the motes. Transient faults may also fail-stop or restart motes (in a manner that is detectable to their neighbors); we assume that the connectivity of the graph is maintained despite these faults.

3 An Evader-centric Program

In this section we present an evader-centric solution to the pursuer-evader problem in sensor networks. In our program every sensor mote, j, maintains a value, $ts.j$, that denotes the latest timestamp that j knows for the detection of the evader. If j detects the evader, it sets $ts.j$ to its timer's value. Every mote j periodically broadcasts its $ts.j$ value and based on the information it receives from its neighbors, assigns the maximum timestamp value it is aware of as $ts.j$. We use $p.j$ (read parent of j) to record the mote that j received the maximum timestamp value. Note that the parent relation embeds a tree rooted at the evader on the sensor network. We refer to this tree as the *evader tree*.

Thus, the evader-centric program for mote j is as follows:

$$
\begin{array}{lll}
\{\text{ Evader detected at } j\ \} & \longrightarrow & p.j := j;\ \ ts.j := timer.j \\
\ \square & & \\
\{\text{ True }\} & \longrightarrow & p.j \in \{k \mid k \in nbr.j \ \ \wedge\ \ ts.k > ts.j\}; \\
 & & ts.j \ := \ ts.(p.j)
\end{array}
$$

We can improve this program by maintaining a variable $d.j$ at each mote j that denotes the distance of j from the evader.

$$
\begin{array}{lll}
\{\text{ Evader detected at } j\ \} & \longrightarrow & p.j := j;\ \ ts.j := timer.j;\ d.j := 0 \\
\ \square & & \\
\{\text{ True }\} & \longrightarrow & p.j \in \{k \mid k \in nbr.j \ \ \wedge\ \ ts.k > ts.j \\
 & & \qquad \vee\ (ts.k = ts.j \ \ \wedge\ \ d.k + 1 < d.j)\} \\
 & & ts.j \ := \ ts.(p.j);\ \ d.j := d.(p.j) + 1
\end{array}
$$

We assume maximum parallelism to guarantee that a minimum distance spanning tree is constructed. That is, every enabled action is executed instantaneously. Note that this is achieved in the sensor network model by event-driven message sends: When a mote receives a new value, it forwards it immediately.

Once an evader tree is formed, the pursuer follows this tree to reach the evader simply by querying its closest mote for its parent and proceeding to the parent mote.

3.1 Proof of correctness

For proof of catching, we only assume that $T_{exch} < T_{evader}$, that is, the speed of reading the state of the neighboring motes should be faster than the speed of the evader. The necessity of this assumption follows from our assumption that the pursuer is faster than the evader (i.e., $T_{pursuer} < T_{evader}$) and from the pursuer's program (i.e., $T_{exch} \leq T_{pursuer}$).

We define a logical "round" concept as follows. Within each round every mote is given a chance to execute one of (if any) its enabled actions. We use α to denote the ratio of the speed of the evader to that of the pursuer, i.e., $T_{pursuer}/T_{evader}$.

Lemma 0. The evader tree is always rooted at the mote that the evader is on. □

Lemma 1. A minimum spanning tree is constructed within at most D rounds where D denotes the diameter of the network. □

Lemma 2. The distance between the pursuer and evader does not increase after the tree is constructed.

Proof. We assume that a path $v'_1, v'_2, \ldots v'_x$ exists in round r.

1. If the evader has not moved before round $r+1$, then from the program it follows that there exists a path $v_1, v_2, \ldots v_l$ where $(\forall i : 1 \leq i \leq x-1 : v_i = v'_i)$. If the pursuer has moved to v_{x-1} before round $r+1$, then $v_l = v'_{x-1}$ and the length of the path is decreased by one. Otherwise, $v_l = v'_x$ and the length of the path stays the same.
2. If the evader has moved before round $r+1$, then there exists a path $v_1, v_2, \ldots v_x$ where $(\forall i : 1 < i \leq x : v_i = v'_{i-1})$. Even though the evader

and the pursuer does not move synchronously, we can argue that every time the evader makes a move and increases the distance by one, the pursuer will decrease the distance by one as explained in the previous case before the evader can move again. The net effect is that, since the pursuer is at least as fast as the evader, the pursuer also makes a move and the length of the path stays the same.

□

Theorem 3. After the evader tree is formed the pursuer catches the evader in at most $d/(1-\alpha)$ rounds where d is the distance between the pursuer and the evader. □

3.2 Proof of stabilization

Lemma 4. The evader tree is eventually rooted at the mote that the evader is on. □

Theorem 5. The evader tree stabilizes in at most $2 * D$ rounds. □

3.3 Discussion

Performance metrics. ??? Add message complexity here

Fading away of evader tree with the distance. For energy efficiency, one might choose to sacrifice from accuracy by updating the evader tree with less frequency as the depth increases. To this end, at each mote j, we maintain a variable $d.j$ that denotes the distance of j from the evader, and modify the update action in the evader-centric program as follows:

$$
\begin{array}{lll}
(\mathrm{rand}(1)^*D \;<\; D - d.j) & \longrightarrow & p.j \in \{k \mid k \in nbr.j \;\wedge\; ts.k > ts.j \\
 & & \quad \vee\; (ts.k = ts.j \;\wedge\; d.k + 1 < d.j)\} \\
 & & ts.j \;:=\; ts.(p.j);\;\; d.j := d.(p.j) + 1
\end{array}
$$

Above, rand(1) denotes a function that returns a random real number between 0 and 1, and D is the diameter of the network, that is, the maximum depth that the evader tree can grow.

4 A Pursuer-centric Program

In this section we present a pursuer-centric solution to the pursuer-evader problem in sensor networks. Here, similar to the evader-centric program, every sensor mote, j, maintains a value, $ts.j$, that denotes the latest timestamp that j knows for the detection of the evader. If j detects the evader, it sets $ts.j$ to its timer's value.

In this program, motes communicate with neighbors only on the request of the pursuer: When the pursuer reaches a mote, the mote resets its recorded time of a detection of an evader to zero and directs the pursuer to a neighboring mote with the highest recorded time.

Thus, the pursuer-centric program for mote j is as follows:

$$
\begin{array}{lll}
\{ \text{ Evader is at } j \ \} & \longrightarrow & ts.j := timer.j \\
\ [\!] & & \\
\{ \text{ Pursuer is at } j \ \} & \longrightarrow & \text{direct pursuer to } \{k \mid k \in nbr.j \\
 & & \qquad \wedge \ ts.k = max(\{ \ ts.i \mid i \in nbr.j \ \})\}; \\
 & & \text{ts.j:=0}
\end{array}
$$

Note that if all ts values of the neighbors are the same (e.g., zero), the pursuer is sent to a random direction.

4.1 Proof of correctness

4.2 Proof of stabilization

4.3 Discussion

Performance metrics.

5 A Hybrid Pursuer-Evader Program

In the hybrid program we combine the evader-centric and pursuer-centric approaches:

1. We limit evader tree to a depth r instead of entire network to save energy.

2. We modify pursuer-centric program to exploit the evader tree structure.

We limit the depth of the evader tree to r by means of the distance d variable. When a mote j discovers (via interaction with its neighbors) that its distance to the evader is more than r, j sets $d.j$ to ∞ and $p.j$ to self in order to dissociate from the evader tree.

$$
\begin{array}{l}
\{\text{ Evader detected at } j \;\} \;\longrightarrow\; p.j := j;\; ts.j := timer.j;\; d.j := 0 \\
\;[] \\
\{\text{ True }\} \;\longrightarrow\; p.j \in \{k \mid k \in nbr.j \;\wedge\; ts.k > ts.j \\
\qquad\qquad\qquad \vee\; (ts.k = ts.j \;\wedge\; d.k + 1 < d.j \;\wedge\; d.k + 1 < r)\} \\
\qquad\qquad\qquad ts.j \;:=\; ts.(p.j);\;\; d.j := d.(p.j) + 1
\end{array}
$$

By limiting the tree to a depth r we lose the advantages of soft-state stabilization: there is no more a flow of fresh information to correct the state of the motes that are outside the evader tree. To achive stabilization, we add explicit stabilization actions. Next we describe these two actions.

For the case where the initial graph has cycles, each cycle is detected and removed by using the bound on the length of the path from each process to its root process in the tree. To this end, we exploit the way that we maintain the d variable: j sets $d.j$ to be $d.(p.j) + 1$ whenever $p.j \in neighbor.j$ and $d.j < r$. The net effect of executing this action is that if a cycle exists then the $d.j$ value of each process j in the cycle gets "bumped up" repeatedly. Eventually, some $d.(p.j)$ reaches $r - 1$, and since the length of each path in the adjacency graph is bounded by $r - 1$, the cycle is detected. To remove a cycle that it has detected, j sets $p.j$ to point to j and $d.j$ to ∞. j also takes the same action, if $p.j$ is not a valid parent (e.g. $d.j \neq d.(p.j) + 1$ or $ts.j > ts.(p.j)$ or $(p.j = j \;\wedge\; d.j \neq 0)$).

We add another action to correct the fake tree roots. If a mote j is spuriously corrupted to $p.j = j \;\wedge\; d.j = 0$, this is detected by explicitly asking for a proof of the evader at j.

Thus the stabilization actions for the bounded length evader tree is as follows.

$$
\begin{array}{l}
(p.j = j \ \wedge \ d.j \neq 0) \ \vee \ ts.j > ts.(p.j) \\
\ \vee \ d.j \neq d.(p.j) + 1 \ \vee \ d.(p.j) \geq r - 1 \quad \longrightarrow \quad p.j := j;\ d.j := \infty \\
\ [] \\
(p.j = j \ \wedge \ d.j = 0) \quad \longrightarrow \quad \text{if (evader is not at } j) \\
\qquad\qquad\qquad\qquad\qquad\qquad p.j := j;\ d.j := \infty
\end{array}
$$

We modify the pursuer program only slightly so as to exploit the evader tree structure.

$$
\begin{array}{lll}
\{ \text{ Pursuer is at } j \ \} & \longrightarrow & \text{if } (p.j \neq j) \\
& & \quad \text{direct pursuer to } p.j \\
& & \text{else} \\
& & \quad \text{direct pursuer to } \{k \mid k \in nbr.j \ \wedge \\
& & \qquad\quad ts.k = max(\{ \ ts.i \mid i \in nbr.j \ \})\}; \\
& & \quad \text{ts.j:=0}
\end{array}
$$

5.1 Proof of correctness

5.2 Proof of stabilization

5.3 Discussion

Performance metrics.

6 Concluding Remarks

In this paper we have investigated a pursuer-evader game for sensor networks. More specifically, we have presented a hybrid, tunable, energy-efficient, and self-stabilizing program to solve this problem. We proved that the pursuer catches the evader even in the presence of faults by following this tree structure to the root.

We have found several variations of the problem to be worthy of study, where we change for instance the communication time between motes, the communication model to be message broadcast instead of shared memory, the numbers of pursuers and evaders, the range of a move.

References

[1] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. *ASPLOS*, 2000.