A Decentralized Solution for Locating Mobile Agents

Paola Flocchini and Ming Xie

SITE, University of Ottawa, 800 King Edward, Ottawa, Canada, K1N 6N5
{flocchin,mxie}@site.uottawa.ca

Abstract. In this paper we propose a new strategy for tracking mobile agents in a network. Our proposal is based on a semi-cooperative approach: while performing its own prescribed task, a mobile agent moves keeping in mind that a searching agent might be looking for it. In doing so we want a fully distributed solution that does not rely on a central server, and we also want to avoid the use of long forwarding pointers. Our proposal is based on appropriate delays that the mobile agents must perform while moving on the network so to facilitate its tracking, should it be needed. The searching agent computes a particular searching path that will guarantee the tracking within one traversal of the network. The delays to be computed depend on structural properties of the network. We perform several experiments following different strategies for computing the searching path and we compare our results.

Keywords: Algorithms for mobile agents, agents location, tracking.

1 Introduction

In this paper we consider a classic problem for mobile agents: the *tracking* (or *locating*) problem. An agent, or a group of agents, is sent on a network to locate a particular agent that is instead moving to perform some tasks. Sometimes the tracking is necessary to communicate with the agent or to terminate its task (e.g., see [4,9]).

Other problems involving two mobile agents are somehow related to this one: *pursuit evasion*, and *rendezvous*. In pursuit evasion there is a *competitive* setting, where one agent tries to escape, while the other is chasing it. The problem has been extensively studied in deterministic and especially in randomized environments (e.g., see [1,5,11]). In rendezvous the two agents *cooperate* to find each other; in fact, their goal is to meet somewhere in the network and their actions go towards this common goal. Also rendezvous has been widely investigated, under different scenarios and different assumption (e.g., see [3] and, for a recent survey [10]). In our problem there is no competition, since the moving agent does not try to escape. On the contrary, there is cooperation, since the moving agent is willing to facilitate the task of the locating agent. However, the degree of cooperation is much weaker than in the rendezvous problem. In fact, the moving agent has other tasks to perform, and it does not even know if some agent has been

A.-M. Kermarrec, L. Bougé, and T. Priol (Eds.): Euro-Par 2007, LNCS 4641, pp. 618–628, 2007.

sent to locate it. While performing its primary tasks it can also perform some actions in order to help the tracking, in case it has to be performed. The agent is willing to do so at the expenses of its own performances, up to a certain degree.

For this purpose, typical solutions involve: 1) the existence of a central host where the moving agent constantly reports its position 2) having the moving agent leave a trace of its movements on its way (e.g., see [2,7]). Both solutions present obvious disadvantages. In the centralized solution the biggest problems are related to fault tolerance and security (problems that are present every time a centralized solution is employed). In fact, the central database could crash resulting in a complete loss of the information; moreover a third party accessing the central database could immediately determine the positions of all the agents at a given time. The second solution consists of leaving at each node the indication of the node where the agent has moved (*forwarding pointer*). This solution could result in a high space complexity to store all the information (especially considering that several moving agents are usually present in a network).

We propose a totally different approach. The idea is to pre-compute a particular *searching walk* that will be followed by the searching agent whenever the tracking is required. The moving agent moves autonomously and independently to perform its task, without reporting its location and without leaving long traces. The "speed" of its movement, however, is appropriately controlled in such a way that, should the searching agent look for the moving agent, it would locate it within one searching walk. By "controlling the speed" we mean that when the moving agent needs to move over a link, it cumulates a delay (proportional to some network parameter appropriately chosen) before performing the movement. In the paper the network is assumed to be synchronous; preliminary studies suggest that this assumption could be relaxed for a more realistic environment.

We describe how to compute this appropriate delay, we show that the amount is related to a network parameter called MinMax chord. The choice of the optimal search walk is conjectured to be an NP-complete problem. We describe several heuristics to compute various searching walks. We then run some experiments to see what are the performances of the heuristic algorithms in random graphs of various size and degree. The performances measure we consider are the maximum and average delay incurred by the moving, as well as the location time. The results are interesting and motivate further study.

2 Model and Terminology

Although the problem might involve several moving agents and several searching agents, without loss of generality we focus on the behavior of a single pair, so we have a *searching agent* SA and a *moving agent* MA. The algorithm we describe would apply to the case of more moving agents.

We assume the system is synchronous, that is, it takes one unit of time for an agent to traverse a link. The searching agent move at the maximum possible speed (1 time unit per link), while the moving agent "slows down" its movement by waiting an appropriate amount of time at the nodes it encounters on its way.

We assume that the moving agent is followed by a single forwarding pointer; in other words, a trace of the moving agent arrived at node y from link (x, y) is present at node x as long as the moving agent is in y. This fact guarantees that either the moving agent or its trace are always present on a node (even when the moving agent is in transit on a link). The locating problem is considered solved when SA resides on the same node as MA, or when it finds its forwarding pointer.

3 The Searching Walk

The general idea is to determine a traversal of the graph (called *searching walk*) for the searching agent. While the moving agent moves arbitrarily, the searching agent follows this predefined walk. Initially the searching agent is at the "beginning" of the searching walk and the moving agent is obviously "ahead".

The searching walk is a traversal of the graph. Let $T = [y_1, y_2, \ldots, y_k]$ be a traversal of $G = (V, E)$. An extended weighted graph $G' = (V', E')$, based on the traversal T can be constructed by aligning the traversal nodes (each y_i connected to y_{i+1}), and adding the other edges of E as chords, as shown in Figure 1. If a node is visited more than once, there is a chord in G' only between any two consecutive occurrences of the same node in T. Given an edge $(y_i, y_j) \in E'$, let $weight(y_i, y_j)$ denote its weight .

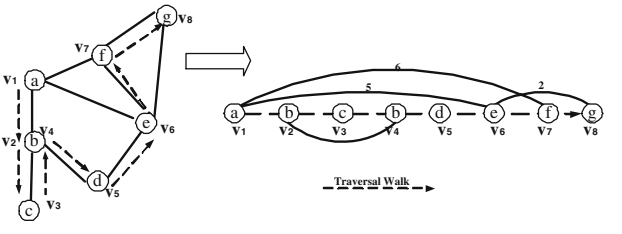

Fig. 1. Traversal Walk with Chords

More precisely, let $G = (V, E)$ be a graph with n nodes, and let $T = [y_1, y_2, \ldots, y_k]$ be a traversal walk of G (with $k \geq n$). Let $f : V' \Rightarrow V$ be a (non injective) function that, for each element y_i of the traversal, returns the corresponding node $f(y_i)$ of V. Let $E(x)$ denote the edges incident to node x in G. We now define a weighted graph $G' = (V', E')$ as follows:

- *Vertices:* $V' = \{y_1, \ldots, y_k\}$ contains one vertex per element of T;
- *Edges :* $(y_i, y_j) \in E'$ if:
  1) $((f(y_i), f(y_j)) \in E)$ AND (there exists no k $(i < k < j)$ such that $f(y_i) = f(y_k)$ or $f(y_k) = f(y_j)$).
  2) $f(y_i) = f(y_j)$ with $i \neq j$ AND there exists no k $(i < k < j)$ such that $f(y_i) = f(y_k)$.

  Weight: $weight(y_i, y_j) = i - j$

Let us call *physical chord* a chord in G' that corresponds to a link in G, and *virtual chord* a chord in G' that connect two occurrences of the same node of G (thus it does not correspond to any link of G). A virtual chord, in fact, corresponds to a cycle in the traversal walk. For each node $x \in V$, $F^{-1}(x) = \{f^{-1}(x)\}$ contains the corresponding occurrences of x in T.

We now define a node and an edge labeling for graph G. Let $\beta : V \rightarrow Z$ be a node labeling function that associate an integer to each node of the network, and $\lambda_x : E(x) \rightarrow Z$ be an edge labeling function that associate an integer to each edge incident to node We label the nodes and the edges of G as follows:

- *Vertices:*
  Let $x \in V$. If $|F^{-1}(x)| = 1$, then $\beta(x) = 0$. If $F^{-1}(x) = \{v_{i,1}, v_{i,2}, \ldots, v_{i,m}\}$ $m > 1$, then $\beta(x) = Max_j\{weight(v_{i,j+1}, v_{i,j})\}$ for $j = 1 \ldots m-1$.
- *Edges:*
  Let $(x, y) \in E$. We define $\lambda_x(x, y) = Max_{a,b,i,j}\{weight(v_{i,a}, v_{j,b})\}$ for $v_{i,a} \in F^{-1}(x)$, $v_{j,d} \in F^{-1}(y))$, and $(v_{i,a}, v_{j,b}) \in E'$.

We compute the delays that the moving agent has to introduce in such a way that the searching agent (SA) is always "behind" the moving agent (MA) along the searching walk, except when it locates it or its forward pointer (FP). With the delays we define below, in fact, it could happen that SA overpasses MA; in this case, however MA is behind the forwarding pointer and it is guaranteed to reach it before it expires.

Suppose the moving agent arrives at node x from link (w, x) and has to move to node y through link (x, y). If $\lambda_x(x, y) < 0$ the agent can move directly without adding any delay because it is moving "away" from the searching agent; if $\lambda_x(x, y) > 0$, it waits the following amount of time.

DELAY
Agent reaching x from (w, x), moving to y through (x, y) in G.

If $\lambda_x(x, y) < 0$
  move-to-y
If $\lambda_x(x, y) > 0$
  WAIT $Max\{\beta(x), \lambda_x(x, y) - 1, \lambda_w(w, x) - 1\}$
  move-to-y

Let $T = [y_1, y_2, \ldots, y_k]$ be the searching walk with SA initially in y_1 when the process starts at time $t = 1$.

Lemma 1. *Let MA arrives at node x at time t from node w. Let del be the time MA has to wait before moving to some node y. If by the time $t + del$ MA has not been located, then after the movement of MA towards y, SA is "before" the first occurrence of $\{f^{-1}(y)\}$ in T (i.e., $t + del + 1 < i \ \forall v_i \in F^{-1}(y)$).*

Proof. (Sketch) By induction on the movements of MA. It is true at time 1, when MA moves for the first time. Let the lemma be true when MA arrives to node w and let us prove it is true when MA reaches the next node x. The lemma

is trivially true if link (x, y) corresponds to a "forward link" in the traversal path, i.e., if $\lambda_x(x, y) < 0$. Let us then consider only "backward" links.

Consider time t when MA arrives in x. At this time, SA is in y_t and, by induction hypothesis, it is either ahead of MA of it will catch it before MA can move. Now MA waits for $Max\{\beta(v), \lambda_w(w, x) - 1, \lambda_x(x, y) - 1\}$ time units. Let m and M be smallest and the largest indices such that $y_m, y_M \in F^{-1}(x)$. We now consider three cases:

- Case $t < m$. If $m - t < \lambda_x(x, y)$, by definition of delay, SA will locate MA before it moves to y (because $del > \lambda_x(x, y)$). Otherwise, SA will be "before" or on y at time $t + del + 1$.
- Case $m < t < M$. Let $y_a, y_b \in F^{-1}(x)$ be the closest occurrences of node x to y_t. SA will reach node x before MA moves to y because, by definition of delay, $del > \beta(x)$ and β is the largest cycle containing x (which is greater than or equal to $b - a$).
- Case $t > M$. SA will reach FA in the next del time units because, by definition of delay, MA stays in x for at least $\lambda_w(w, x) - 1$ time units and thus, FA is in w when SA reaches it at time $t + \lambda_w(w, x) - 1$.

It follows from the previous Lemma that:

Theorem 1. *The searching agent locates the moving agents by the end of its traversal.*

Clearly, one would like to minimize both the location time (which depends on the length of the searching walk) and the delay incurred by the moving agent. In the following we are interested in searching walks of length $O(n)$ and we would like to minimize the maximum delay as well as the average delay incurred by the moving agent. The maximum delay corresponds to the longest (virtual or physical) chord; thus, we would need to find the traversal that minimizes such a chord. More precisely, we define the MinMax Traversal $\mathcal{T}_G$ of G as the traversal that minimizes the maximum weight in G': $\mathcal{T}_G = Min_T\{Max_{i,j}\{w(y_i, y_j)\}\}$ with $(y_i, y_j) \in E'$.

4 Building Good Searching Walks

We conjecture that finding the MinMax traversal of a graph is an NP-complete problem. In the following we propose several heuristic algorithms to construct "good" traversal and we compare them.

General Traversal Algorithm. Consider the following general traversal algorithm that visits the nodes in depth. If the node where the searching agent resides has unvisited neighbours, the agent moves to one of them; if it has no unvisited neighbours, but some nodes have not been visited, it moves through already visited nodes to reach one that has not been visited yet.

In the general traversal algorithm described above, there are two points where we can introduce some variations for obtaining a searching walk that is good for

our purposes: 1) How to choose the next node when the current node has several unvisited neighbours. 2) Where to move after visiting a node without unvisited neighbours. In the common *depth-first traversal* (DFT), for example, a random choice is performed for 1) and a backtrack for 2). Obviously an arbitrary DFT does not necessarily result in an efficient searching strategy.

Choosing an Unvisited Neighbouring Node. The following are different strategies to choose one among the unvisited neighboring nodes.

- **Random.** The agent randomly chooses an unvisited neighboring node.
- **Priority Queue.** When the agent moves to a new node, the unvisited neighbouring nodes of that node are stored in a priority queue. When the agent finds more than one unvisited neighboring nodes, it chooses the unvisited neighboring node with highest priority. If none of the unvisited nodes is in the queue:
  - **Basic:** The agent randomly chooses an unvisited neighboring node.
  - **Improved:** The agent checks the unvisited neighboring nodes' unvisited neighbors. If it finds at least one of the unvisited neighboring nodes at distance two in the priority queue, it chooses the one with highest priority and it moves there.
  - **Closest to the queue:** The agent chooses the one that is closest to a node in the priority queue (in case of ties it selects the one closest to a highest priority element in the queue).
- **Closest to start node.** When the agent has more than one unvisited neighboring nodes, it moves to the unvisited neighboring node that is closest to the start node.
- **Neighbour of least visited node.** When the agent has more than one unvisited neighboring nodes, it moves to the unvisited neighboring node whose neighbour has been visited least recently.

Moving to a Non neighbouring Unvisited Node. The following are different strategies to move to an unvisited node when there is no unvisited neighboring node from the current agent's location.

- **DFT.** The agent backtracks to the most recently visited node that has unvisited neighboring node, and then continues the traversal.
- **Greedy.** The agent moves to the nearest unvisited node.
- **BFT.** The agent moves to one of the unvisited node that is closest to the start node.
- **Hybrid.** The agent moves to the nearest unvisited node if there is only one such node; if there are more such nodes, the agent moves to the node among the nearest unvisited nodes that is closest to the starting node of the walk. This strategy combines the *greedy* and *BFT* strategies.

When considering a non-neighbouring unvisited node, we include in the walk the shortest path between the current node and the next. Notice that, in doing so we might visit new nodes.

The strategies described above (except for *Random* and *DFT*) are motivated by the empirical observation that it might be useful to fully visit an area around a visited node before moving to another area of the graph. In fact, visiting a neighbour of a node already visited creates a chord: since we would like to minimize the length of the chords, we would like to visit these nodes as soon as possible. On the other hand, we also want to maintain a walk of size $O(n)$ and to achieve this we cannot revisit nodes already visited too many times.

5 Experimental Results

5.1 Experimental Setup

We now combine the different strategies for choosing the next neighbouring unvisited node and choosing a non-neighbouring unvisited node. For each algorithm, we record the length of the traversal walk, the maximum and average delay for the moving agent. Notice that the maximum delay corresponds to the length of the longest chord.

We run the algorithms on different random topologies of various size and density. For each type of graph, we generated 20 graphs with the same parameters, and we averaged the obtained results. We utilize *Java Universal Network/Graph Framework* [8] library to generate the graphs.

- A1: *random + DFT* .
- A2: *improved priority queue + DFT* .
- A3: *improved priority queue + Greedy*.
- A4: *random + Greedy*.
- A5: *improved priority queue+ BFT*.
- A6: *random + BFT*.
- A7: *improved priority queue + Hybrid*.

We have observed all the results for all the combinations, we however show here only the ones from which we have obtained the most interesting results.

In this set of experiments we are interested in the maximum and average delay incurred by the moving agent, and in the location time, depending on the traversal strategy followed by the searching agent. In all cases, the length of the traversal is only slightly higher than the number of nodes.

In the experiments, we generated 10 random graphs for each type of graph, and obtained the average values for the results. We have run experiments for various graph sizes n ($n = 100, 200, 500, 800, 1000$) and levels of density m (number of edges) ($m = 4n, 5n, 6n, 7n, 8n, 9n, 10n$). For each choice of n and m we generated 10 random graphs to count the average values. In each case we randomly select the starting node.

The graphic in Figure 2 *a*) shows that, among our heuristics, the ones with the best performance in terms of the length of the maximum chord are A_3 and A_7; that is: the *improved priority queue* heuristic as the choice of an unvisited neighbouring node, and either the *Greedy* or the *Hybrid* heuristic for the choice of

a non-neighbouring unvisited node. This graph correspond to the case $n = 1000$; the results are quite consistent with different sizes of the graph.

Interestingly, for all heuristics, increasing the density of the network (i.e., its average degree), results in a decrease of the length of the traversal and, most of the times, also of the length of the maximum chord and average delay.

The graphic in Figure 2 *b*) shows the changes in average delay incurred by the moving agent as the number of edges increases, for the different strategies. Also in this case, the best performances are obtained by A_3 and A_7.

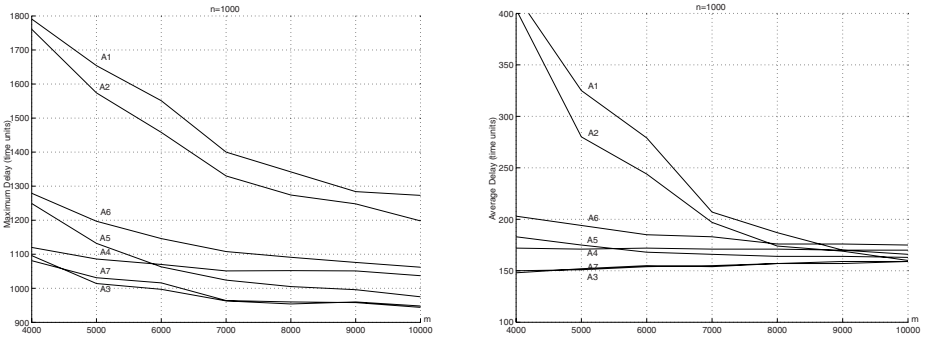

Fig. 2. *a*) Max delay *b*) Average delay

The various results for the plain *DFT* and some combinations of heuristics are reported in the tables below (again for $n = 1000$).

m	A_1: Random + DFT			
	Max. D.	Av. D.	Loc. T.	Length
4000	1791	418	791	1855
5000	1654	325	712	1705
6000	1551	279	669	1617
7000	1400	207	608	1464
8000	1342	187	587	1393
9000	1284	170	554	1319
10000	1273	166	536	1321

m	A_7: Improved Priority+ Hybrid			
	Max. D.	Av. D.	Loc. T.	Length
4000	1081	148	509	1163
5000	1031	152	501	1124
6000	1016	155	510	1098
7000	964	154	498	1075
8000	960	157	479	1067
9000	958	159	502	1060
10000	944	159	491	1049

m	A_4: Random + Greedy			
	Max. D.	Av. D.	Loc. T.	Length
4000	1120	172	506	1149
5000	1086	171	502	1110
6000	1070	172	502	1089
7000	1051	171	499	1076
8000	1052	171	486	1065
9000	1041	170	497	1057
10000	1037	170	492	1051

m	A_5: Improved+BFT			
	Max. D.	Av. D.	Loc. T.	Length
4000	1248	183	621	1329
5000	1132	175	609	1240
6000	1063	168	562	1178
7000	1024	166	556	1144
8000	1005	164	545	1116
9000	996	164	514	1098
10000	976	163	504	1082

Maximum delay, average delay, location time, length of the walk for $n = 1000$.

5.2 Observations

The experiments lead to the following observations.

1. With our best combinations of heuristics (*priority queue and hybrid* and *priority queue and greedy*), the maximum delay is reduced by approximately half compared to a Depth-first traversal; the average delay is reduced of 75 %. The reductions are more evident when the graph is sparse, they become less relevant when the graph is very dense (see Figure 2 and tables in the previous Section). Notice that there are graphs (for example the ring) where it is impossible to find a walk of size $O(n)$ with a maximum chord of length smaller than $O(n)$.
2. In general, with any heuristic, the maximum delay is slightly shorter than the length of the traversal walk.
3. The choice of the heuristic to move to a non-neighbouring unvisited node deeply affects the performances. The traversals with Greedy or Hybrid heuristics give generally the best results, followed by the BFT, while DFT has the worst performance. This confirms the intuition that it is more efficient to fully visit the neighbours of a visited node before moving to nodes that are further apart.
4. While the maximum delay always decreases with the increase of the number of edges, the average delay is more stable and does not display this strong behavior.
5. Combinations A_4 (Random+ Greedy) and A_5 (Improved+ BFT) behaves in a similar way and they both improve the plain depth-first walk. Interestingly, A_4 outperforms A_5 for sparse graphs, while it becomes less efficient then A_5 for more dense graphs. In our experiments this happens roughly when $m = 6n$ Our intuition for this behavior is that the BFT heuristic heavily depends on the density; in fact, when the graph is too sparse BFT gives little chance to find a "good" (close to the start) unvisited node; increasing the number of edges, however, it is more likely that such a "good" node is found thus discovering shorter chords. On the other hand, the Greedy heuristic does not depend on the density since it can always move to the nearest visited node.
6. An interesting observation is that the performances are proportional to the walk's length, in the sense that strategies with longer walks give generally worse performances than strategies with shorter walks. The only exception is A_7 when compared with A_4. The performances of A_7, in fact, are better than those of A_4 although the length of the traversal walk of A_7 is slightly higher (at least for densities $m < 10n$).
7. Fixing the average degree and changing the graph size, the average and maximum delay increases linearly with the graph size. Furthermore, the lower is the lower degree, the more the choice of the heuristic affects the performance (the graphics corresponding to this type of analysis are not shown for lack of space).

6 Conclusion

The contribution of this paper is the proposal of a new approach for locating mobile agents that involves neither forwarding pointers (only a single trace of size one) nor a central server. The approach is based on having the moving agent move at a variable speed, depending on the structural properties of the links it is traversing. At this stage the algorithm could not seem applicable in practice because it is based on very strong assumptions about the environment, which is assumed to be synchronous. Notice however that synchronicity could be relaxed by slightly increasing the length of the trace left by the agents; in fact, preliminary studies suggest that this approach would work also in environments that are not synchronized provided the moving agent leaves a short trace of length 2. Further notice that while computing the delay we have not taken into consideration the time the agent has to actually spend at each node. Depending on its tasks, the agents might have to spend a certain amount of time thus decreasing the forced delay.

We are currently working on several possible improvements that we believe will highly decrease the average delay. The employ of more than a single searching agent would considerably decrease the locating time. For example, static searching agents could be placed at crucial nodes (the ones which have long chords along the searching walk) while a single searching agent could traverse the walk; alternatively, several searching walk could be traversed concurrently thus reducing the locating time. This paper is just a first step towards exploiting the structure of the network for locating purposes, we are now working on improvements like the ones mentioned above to make the technique more applicable and efficient in a practical setting.

References

1. Adler, M., Racke, H., Sivadasan, N., Sohler, C., Vocking, B.: Randomized pursuit-evasion in graphs. In: Int. Colloquium on Automata, Languages and Programming, pp. 901–912 (2002)
2. Alouf, S., Huet, F., Nain, P.: Forwarders vs. centralized server: An evaluation of two approaches for locating mobile agents. Performance Evaluation 49(1-4), 299–319 (2002)
3. Alpern, S., Gal, S.: The theory of search games and rendezvous. Kluwer Academic Publishers, Dordrecht (2003)
4. Baumann, J.: Mobile Agents: Control Algorithms. LNCS, vol. 1658. Springer, Heidelberg (2000)
5. Demirbas, M., Arora, A., Gouda, M.G.: A pursuer-evader game for sensor networks. In: Pro. 6th Symposium on Self-Stabilizing Systems, pp. 1–16 (2003)
6. Diaz, J., Petit, B.J., Serna, M.: A survey of graph layout problems. ACM Computing Surveys 34(3), 313–356 (2002)
7. Fowler, R.J.: The complexity of using forwarding addresses for decentralized object finding. In: 5th ACM Symp. on Principles of Distributed Computing, pp. 108–120. ACM Press, New York (1986)

8. Jung: Java Universal Network/Graph Framework, http://jung.sourceforge.net/
9. Lien, Y., Leng, C.W.R.: On the search of mobile agents. In: 7th IEEE Int. Symposium on Personal, Indoor, and Mobile Radio Communications, pp. 703–707. IEEE Computer Society Press, Los Alamitos (1996)
10. Kranakis, E., Krizanc, D., Rajsbaum, S.: Mobile agent rendezvous. In: 13th Int. Coll. on Structural Information and Communication Complexity, pp. 1–9 (2006)
11. Parsons, T.D.: Pursuit-evasion in a graph. In: Jantke, K.P. (ed.) AII 1992. LNCS, vol. 642, pp. 426–441. Springer, Heidelberg (1992)