Rendezvous of Mobile Agents in Unknown Graphs with Faulty Links

Jérémie Chalopin[*] Shantanu Das[†] Nicola Santoro[‡]

Abstract

A group of mobile agents wandering among the nodes of a network have to gather together in a single node of the graph; This problem known as the Rendezvous problem has been studied extensively but only for networks that are safe or fault-free. In this paper, we consider the case when some of the edges in the network are dangerous or faulty such that any agent entering one of these nodes would be destroyed. Our objective is to minimize the number of agents that are destroyed and achieve rendezvous of all the surviving agents. We determine under what conditions this is possible and present algorithms for achieving rendezvous in such cases. Our algorithms are for arbitrary networks with an arbitrary number of dangerous channels; thus our model is a generalization of the case where all the dangerous channels lead to single node, called the *Black Hole*. We do not assume prior knowledge of the network topology; In fact, we show that knowledge of only a "tight" bound on the network size is sufficient for solving the problem.

1 Introduction

We consider distributed systems consisting of a set of mobile entities called agents that are moving in an environment simply modelled as a bidirectional graph, that could represent for example, a communication network. We are interested in the central problem in such systems, called the Rendezvous problem, which requires all the agents to meet together in a single node of the graph. Unlike previous studies on the rendezvous problem, we consider the case when the environment where the agents are moving, is not fault-free. In our model, some of the edges in the graph are dangerous for the agents such that any agent that attempts to traverse such an edge (from either direction) simply disappears, without leaving any trace. Notice that if all the edges incident to a node u are faulty, then node u can never be reached by any agent. Such a node is equivalent to a *black hole* as in the model of Dobrev et al.[11]. Thus, the black hole model is a specific case of the model considered in this paper.

We assume that the faults are permanent i.e. all the faulty edges remain faulty duration of execution of the rendezvous algorithm and no new faults may develop during this interval. The location of the faulty links are initially unknown to the agents. The task of the agents is to meet at a safe node, while avoiding the faulty links. Obviously, not all agents would be able to meet, because some of the agents may die while traversing the faulty edges. So, our objective is to minimize the number of agents that die and achieve the rendezvous of all the surviving agents whenever possible. If there are τ faulty links in the network then no more than $k - \tau$ agents may rendezvous, in the worst case. In this paper we show under what conditions rendezvous of the $(k - \tau)$ surviving agents is possible and then present an algorithm for achieving rendezvous under such conditions. Our algorithm works in any network topology even when the exact size of the network is unknown. We only assume the knowledge of an upper bound on the network size, which is shown to be necessary for achieving rendezvous under such conditions.

The only previous result for rendezvous in faulty networks was in the case of the ring network containing two faulty links leading to a single node—the black hole [13]. Our results can be seen as a generalization of these results to networks of arbitrary topology that contain faults at arbitrary locations.

1.1 Our Results:

We present the following results in this paper:

[*]LaBRI Université Bordeaux 1, Talence, France. Email: chalopin@labri.fr
[†]School of Information Technology and Engineering, University of Ottawa, Canada. Email:{shantdas}@site.uottawa.ca
[‡]School of Computer Science, Carleton University, Canada. Email: santoro@scs.carleton.ca

- If there are τ dangerous links in the graph G, and k agents are initially present, then at most $k-\tau$ agents can rendezvous in G.
- The Rendezvous of $k-\tau$ agents is possible only when the extended-view of the network is covering minimal. Even in this case, rendezvous is not possible if the agents do not know the size of the network or at least a tight upper bound. A loose upper bound $n \leq B_n \leq 2n$ is not sufficient.
- We present an algorithm *RDV* for solving rendezvous of $k-\tau$ agents in networks where it is solvable. Our algorithm requires only the knowledge of a tight upper bound B on the number of nodes n, such that $n \leq B < 2n$.
- The algorithm *RDV* correctly solves rendezvous of $k-\tau$ agents with explicit termination for each surviving agent and the total number of moves made by the agents during the algorithm is $O(m(m+k))$ where m is the number of edges in G.
- We show that solving rendezvous of $k-\tau$ agents in networks where it is solvable, requires at least $\Omega(m(m+k))$ moves, even when n, k and q are all known. Thus we prove the optimality of our algorithm.
- Finally we show that, there exists no *effective* algorithm for *maximal* rendezvous, i.e. an algorithm that when executed on any network achieves the rendezvous of as many agents as deterministically possible on that network.

1.2 Related Results

The problem of *Rendezvous* has been extensively studied mostly using randomized methods (see [2] for a survey). Among deterministic solutions to rendezvous, Yu and Yung [28] and Dessmark et al. [10] presented algorithm for agents with distinct labels. In the anonymous setting, the problem has been studied under different models (synchronous or asynchronous), using either whiteboards[4] or, pebbles/tokens [21]. Most of these solutions are designed for anonymous graphs (i.e. graphs where nodes do not have distinct identities) which present the most challenging (i.e. computationally difficult) situations. The issue of computability in anonymous graphs, have been studied by many authors including Angluin [3], Yamashita and Kameda [27], Mazurkiewicz [23], Sakamoto [26], and Boldi and Vigna [5]. Most of these studies have concentrated on the problem of symmetry-breaking or leader election which is in fact, closely related (and sometimes equivalent [7]) to the rendezvous problem for mobile agents.

However, all the above results are restricted to safe or fault-free networks. Computations in presence of either node-faults or link-faults has been studied extensively in the case of message-passing systems, mostly in relation to the consensus or agreement problem. For crash failures, Fischer et al.[16] showed that it is impossible to solve consensus in asynchronous systems even if a single node crashes. Pease et al.[24] studied the consensus problem in presence of Byzantine faults and showed that consensus is not possible even if one-third of the nodes are faulty. They gave an algorithm for reaching agreement with lesser number of faults, which was improved later by Dolev and Strong [15]. In the case of faulty links, the problems of leader election [1], broadcast [17] and arbitrary function computation [19] have been studied in presence of either permanent or dynamic faults. Santoro and Widmayer[25] studied the effects of ubiquitous link-faults on achieving agreement in asynchronous networks.

Recently attention has focused on designing mobile agent protocols for networks which are faulty, in particular, where there is a *black hole*. The research on such networks have concentrated on locating the black hole. This has been studied under two different methods—using whiteboards[11, 12] or using tokens[14] to mark edges. The objective here is minimizing the number of agents that fall into the black hole and the time taken by the surviving agents to locate the black hole [20]. The general case of multiple black holes has been considered only by Cooper et al. [8]. All these problems assume that the team of agents start from the same node, i.e. they are co-located. When the agents start from distinct nodes, it is very difficult to gather the agents while avoiding the black hole nodes. This has been studied earlier only in the case of ring networks containing a single black hole, by Dobrev et al. [13], where the authors give solutions to rendezvous and near-gathering assuming the knowledge of topology and the size of the network.

2 The Model and Definitions

2.1 The Model

The environment is modelled by the tuple $(G, \xi, p, \lambda, \eta)$ where G is an undirected connected graph, ξ is a set of agents and p specifies the initial placement of the agents in the graph G (i.e. $\forall A \in \xi, p(A) = v : v \in V(G)$). The number of nodes is denoted by $n = |V(G)|$ and the number of agents is denoted by $k = |\xi|$. The agents can move from one node to its adjacent node by traversing the edge connecting them. The edges incident to a node v are locally oriented i.e. they are labelled as $1, 2, \ldots, d(v)$, where $d(v)$ is the degree of node v. Notice that each edge $e = (u, v)$ has two labels, one for the link or port at node u and another for the link at node v. The edge labelling of the graph G is specified by $\lambda = \{\lambda_v : v \in V\}$, where for each vertex u, $\lambda_u : \{(u, v) \in E : v \in V\} \to \{1, 2, 3, \ldots, d(u)\}$ defines the labelling on its incident edges. For any edge (u, v) we use $\lambda(u, v)$ to denote the pair $(\lambda_u(u, v), \lambda_v(u, v))$.

The function $\eta : E(G) \to \{0, 1\}$ denotes which edges are safe/faulty. An edge $e \in E(G)$ is safe if $\eta(e) = 1$ and faulty otherwise. The faults are permanent, so any edge that is faulty at the start of the algorithm remains so until the end and no new faulty edge appears during the execution of the algorithm.

The node where an agent A is initially located is called the *homebase* of agent A. The agents are all identical (i.e. they do not have distinct names or labels) and they execute the same algorithm. An agent may wake-up at any time and start executing the algorithm. The system is totally asynchronous, such that every action performed by an agent takes a finite but otherwise unpredictable amount of time. As in previous papers on the subject, we assume that the agents communicate by reading and writing information on public whiteboards locally available at the nodes of the network. Thus, each node $v \in G$ has a whiteboard (which is a shared region of its memory) and any agent visiting node v can read or write to the whiteboard. Access to the whiteboard is restricted by fair mutual exclusion, so that, at most one agent can access the whiteboard of a node at the same time, and any requesting agent will be granted access within finite time. An agent that is granted access to the whiteboard at node v, is allowed to complete its activity at that node before relinquishing access to the whiteboard (i.e. access control is non preemptive).

Note that it is not necessary for two agents A and B traversing the same edge $e = (u, v)$ of the graph, to arrive at node v in the same order in which they left node u. However, using the whiteboards at the nodes, it is easy to implement a first-in first-out (FIFO) strategy such that agents traversing an edge can be assumed to have reach their destination in order (i.e. an agent cannot overtake another while traversing an edge). For the rest of this paper, we shall assume this FIFO property; this will simplify the description of our algorithms.

Unlike previous results, we do put the restriction that every agent should start from a distinct homebase. Indeed, some of the agents may be already be in the same node at the start of the algorithm.

2.2 Directed Graphs and Coverings

In this section, we present some definitions and results related to directed graphs(digraphs) which we use to represent the structure of a network. A directed graph(digraph) $D = (V(D), A(D), s_D, t_D)$ possibly having parallel arcs and self-loops, is defined by a set $V(D)$ of vertices, a set $A(D)$ of arcs and by two maps s_D and t_D that assign to each arc two elements of $V(D)$: a source and a target (in general, the subscripts will be omitted). A digraph D is strongly connected if for all vertices $u, v \in V(D)$, there exists a path between u and v. A *symmetric* digraph D is a digraph endowed with a symmetry, that is, an involution $Sym : A(D) \to A(D)$ such that for every $a \in A(D), s(a) = t(Sym(a))$. In this paper, we will only consider strongly connected symmetric digraphs. In particular, we consider digraphs where the vertices and the arcs are labelled with labels from a recursive label set L and such digraphs will be denoted by (D, μ_D), where $\mu_D \colon V(D) \cup A(D) \to L$ is the labelling function. In general, the label on an arc a would be a pair (x, y) and the labelling μ_D should satisfy the property that if $\mu_D(a) = (x, y)$ then $\mu_D(Sym(a)) = (y, x)$, for every arc $a \in D$.

A self-loop in the digraph D is an arc a, such that $s(a) = t(a)$. A self loop is called symmetric if $Sym(a) = a$ and otherwise it is asymmetric. Notice that for a symmetric self-loop a, $\mu_D(a)$ is of the form (x, x) but for asymmetric self-loop a', $\mu_D(a')$ is of the form (x', y') with $x' \neq y'$. A pair of parallel arcs are two arcs a and a', such that $s(a) = s(a')$ and $t(a) = t(a')$.

A digraph homomorphism γ between the digraph D and the digraph D' is a mapping $\gamma \colon V(D) \cup A(D) \to V(D') \cup A(D')$ such that if u, v are vertices of D and a is an arc such that $u = s(a)$ and $v = t(a)$ then

$\gamma(u) = s(\gamma(a))$ and $\gamma(v) = t(\gamma(a))$. A homomorphism from (D, μ_D) to (D', μ'_D) is a digraph homomorphism from D to D' which preserves the labelling, i.e., such that $\mu'_D(\gamma(x)) = \mu_D(x)$ for every $x \in V(D) \cup A(D)$.

We now define the notion of graph coverings, borrowing the terminology of Boldi and Vigna[6]. A *covering projection* is a homomorphism φ from D to D' satisfying the following: (i) For each arc a' of $A(D')$ and for each vertex v of $V(D)$ such that $\varphi(v) = v' = t(a')$ there exists a unique arc a in $A(D)$ such that $t(a) = v$ and $\varphi(a) = a'$. (ii) For each arc a' of $A(D')$ and for each vertex v of $V(D)$ such that $\varphi(v) = v' = s(a')$ there exists a unique arc a in $A(D)$ such that $s(a) = v$ and $\varphi(a) = a'$.

If a covering projection $\varphi : D \to D'$ exists, D is said to be a *covering* of D' via φ and D' is called the base of φ. A symmetric digraph D is a *symmetric covering* of a symmetric digraph D' via a homomorphism φ if D is a covering of D' via φ such that $\forall a \in A(D), \varphi(Sym(a)) = Sym(\varphi(a))$. A digraph D is *symmetric-covering-minimal* if there does not exist any graph D' not isomorphic to D such that D is a symmetric covering of D'.

Property 2.1 ([6]) *Given two non-empty strongly connected digraphs D, D', each covering projection φ from D to D' is surjective; moreover, all the fibres have the same cardinality. This cardinality is called the number of sheets of the covering.*

The notions of coverings extend to labelled digraphs in an obvious way: the homomorphisms must preserve the labelling. Given a labelled symmetric digraph (H, μ_H), the minimum base of (H, μ_H) is defined to be the labelled digraph (D, μ_D) such that (i) (H, μ_H) is a symmetric covering of (D, μ_D) and (ii) (D, μ_D) is symmetric covering minimal.

2.3 Definitions and Properties

Given any deterministic (distributed) algorithm $\mathcal{P}$ and a network $(G, \xi, p, \lambda, \eta)$, the order in which the various actions are performed by the agents defines an *execution* of the algorithm on the network $(G, \xi, p, \lambda, \eta)$. We define the *synchronous* execution of an algorithm $\mathcal{P}$ to be the particular execution where all agents start executing at exactly the same time and every action taken by any agent takes exactly one unit of time.

Property 2.2 ([6]) *If the digraph (H, μ_H) is a covering of (D, μ_D) via φ, then any execution of an algorithm $\mathcal{P}$ on (D, μ_D) can be lifted up to an execution on (H, μ_H), such that at the end of the execution, for any $v \in V(H)$, v would be in the same state as $\varphi(v)$.*

We define the *extended-view* of the network $(G, \xi, p, \lambda, \eta)$ as the labelled digraph (H, μ_H) such that, H consists of two disjoint vertex sets V_1 and V_2 and a set of arcs $\mathcal{A}$ as defined below:

- $V_1 = V(G)$;
- $\mu_H(v) = |\{A \in \xi : p(A) = v\}|$, $\forall v \in V_1$;
- For every safe edge $e = (u, v) \in E(G)$, there are two arcs $a_1, a_2 \in \mathcal{A}$ such that $s(a_1) = t(a_2) = u$, $s(a_2) = t(a_1) = v$, and $\mu_H(a_1) = (\lambda_u(e), \lambda_v(e))$, $\mu_H(a_2) = (\lambda_v(e), \lambda_u(e))$.
- For every faulty edge $e = (u, v)$, there are vertices u' and $v' \in V_2$ with $\mu_H(u') = \mu_H(v') = -1$ and arcs $(u, u'), (u', u), (v, v')$ and $(v', v) \in \mathcal{A}$ with labels $(\lambda_e(u), 0)$, $(0, \lambda_e(u))$, $(\lambda_e(v), 0)$, and $(0, \lambda_e(u))$ respectively;

Here, the vertices in V_1 represent safe nodes and the vertices in V_2 represent Black-Holes. Intuitively, the *extended-view* represents the all the information that an agent may obtain about the network.

Lemma 2.1 *For any deterministic algorithm $\mathcal{P}$, a synchronous execution of $\mathcal{P}$ on the network $(G, \xi, p, \lambda, \eta)$ is equivalent to a synchronous execution of algorithm $\mathcal{P}$ on the extended-view (H, μ_H), such that the final state of any node in G is exactly same as the state of the corresponding vertex in H.*

The proof follows from the definition of the *extended-view* of a network.

Lemma 2.2 *If the extended view of two networks have same minimum-base (D, μ_D) then all nodes in the two networks which belong to the pre-image of a vertex $v \in D$ would always be in the same state, during a synchronous execution of any algorithm $\mathcal{P}$.*

The proof follows from Lemma 2.1 and Property 2.2.

In the following, whenever the extended-view of a network is symmetric-covering minimal, we shall say the network is *minimal.*

3 Impossibility Results

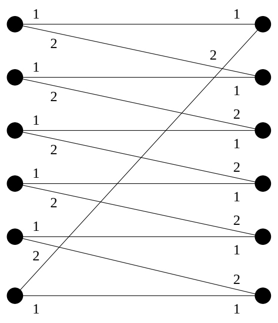


Figure 1: A network with n=k=12 and three faulty edges(not shown here)

Lemma 3.1 *In a network containing τ dangerous links and k dispersed agents, τ agents may die while executing any algorithm for rendezvous. (It is not always possible to rendezvous more than $k-\tau$ agents even if the network topology is known to the agents).*

Proof : Consider the network shown in figure 1, where each node contains one agent initially. There are exactly three faulty edges (i.e. $\tau = 6$) in the network but the location of the faulty edges is unknown to the agents (thus an adversary can decide which edges are faulty). Suppose all the agents wake-up at the same time; Since all these agents are in the same state initially, they would take the same action. So, either each agent would traverse the incident edge labelled 1 or each agent would traverse the edge labelled 2 (They cannot just wait because then we would have a deadlock!). In the first case, we assume all the faulty edges are labelled $(1, 1)$; and in the second case, we assume the faulty edges are labelled $(2, 2)$. Thus in both cases, at least τ agents would die. ■

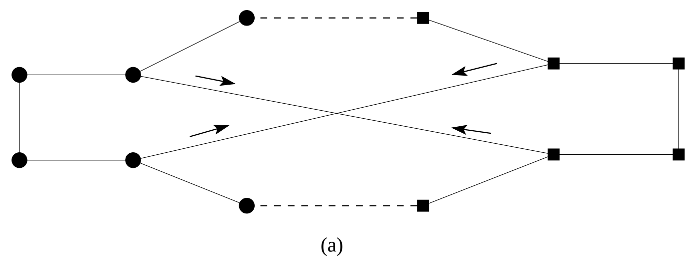
Lemma 3.2 *It is impossible to rendezvous $k - \tau$ agents in a network whose extended-view is not symmetric-covering minimal.*

Proof : Assuming the contrary, suppose $k - \tau$ agents are able to gather at a node v of the network, using an algorithm $\mathcal{P}$. Thus, in the equivalent execution of the algorithm on the extended-view (H, μ_H), $k-\tau$ agents must have gathered at the corresponding node v'. Let (D, μ_D) be the minimum-base of (H, μ_H). Due to property 2.1, there must be q vertices in H which map to each vertex $u \in D$, where $q = |H|/|D|$. Notice that $q \geq 2$, because (H, μ_H) is not covering-minimal. If node v' maps to node u, then all the $q-1$ other nodes that map to u, must also contain $k - \tau$ agents at the end of algorithm $\mathcal{P}$. However this is impossible because there are only $k - \tau$ surviving agents, due to Lemma 3.1. ■

Observation 3.1 *A faulty edge can not be distinguished from a slow edge, i.e. one which the agents take a long time to traverse.*

Lemma 3.3 *It is impossible to solve rendezvous (with termination detection) of $k - \tau$ agents even in minimal networks if the agents know only an upper bound B on the number of nodes n such that $n \leq B \leq 2n$.*

Proof : Consider the networks shown in Figure 2, where each node contains a single agent. In the first network (Figure 2(a)) the edges marked by arrows are slow edges. If the only prior knowledge available to the agents is the value of a bound $B = 12$ (which works for all the three networks), then the agents can not distinguish



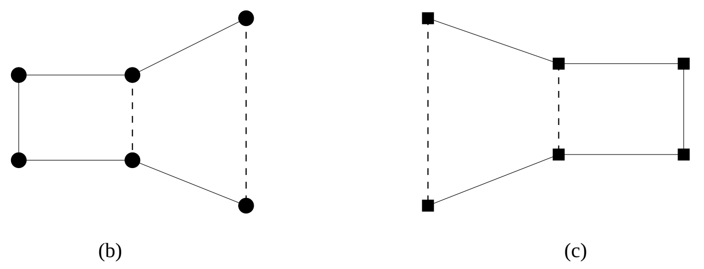


Figure 2: The network in (a) cannot be distinguished from the networks in (b) and (c) due the slow edges (marked by arrows)

between the three networks. In networks (b) and (c), the rendezvous of only two agents is possible, so in network (a) too, the algorithm terminates after rendezvous of only two agents even though the rendezvous of $(k-\tau)=8$ agents is possible here. ■

Definition 3.1 *An algorithm $\mathcal{P}$ is said to be an* effective *algorithm for rendezvous of $w>1$ agents, if the algorithm when executed on a network where rendezvous of w agents is possible, always succeeds in achieving rendezvous within a finite time.*

Lemma 3.4 *There does not exist any effective algorithm for rendezvous of $k-\tau$ agents even if the network topology is known a-priori to the agents.*

Proof : Consider the two networks shown in Figure 3(a) and (b). Each network has the same topology and there are three faulty edges in each (but their locations are different). In the network of Figure 3(b) there are two edges (marked by arrows) which are very slow. Since slow edges edges can not be distinguished from faulty edges, the agents would not be able to determine whether they are in the first network or the second. Thus any rendezvous algorithm $\mathcal{P}$ (that terminates within a finite time) must achieve the same result in both networks. Notice that the first network has an extended view which is not minimal (the minimum base is shown in Figure 3(c)). Since algorithm $\mathcal{P}$ must fail to achieve rendezvous in the first network, it must also fail in the second one, even though it is possible to rendezvous in the second network. Thus algorithm $\mathcal{P}$ is not *effective.* ■

Lemma 3.5 *It is possible to solve rendezvous of $(k-\tau)$ agents in any network that is covering minimal if the agents know only an upper bound B on the number of nodes n such that $n \leq B < 2n$.*

In the next section, we present an algorithm that achieves this. But first we show the following lower bound on the number of moves required to be made by such an algorithm.

Theorem 3.1 *For solving rendezvous of $(k-\tau)$ agents in an arbitrary network (G,ξ,p,λ,η) without any knowledge other than the size of the network, the agents need to make at least $\Omega(m(m+k))$ moves in total.*

4 Solution Protocol

In this section we present an algorithm for solving Rendezvous in faulty networks, using the knowledge of only an upper bound B on the network size, such that $n \leq B < 2n$. As shown in the previous section, there is

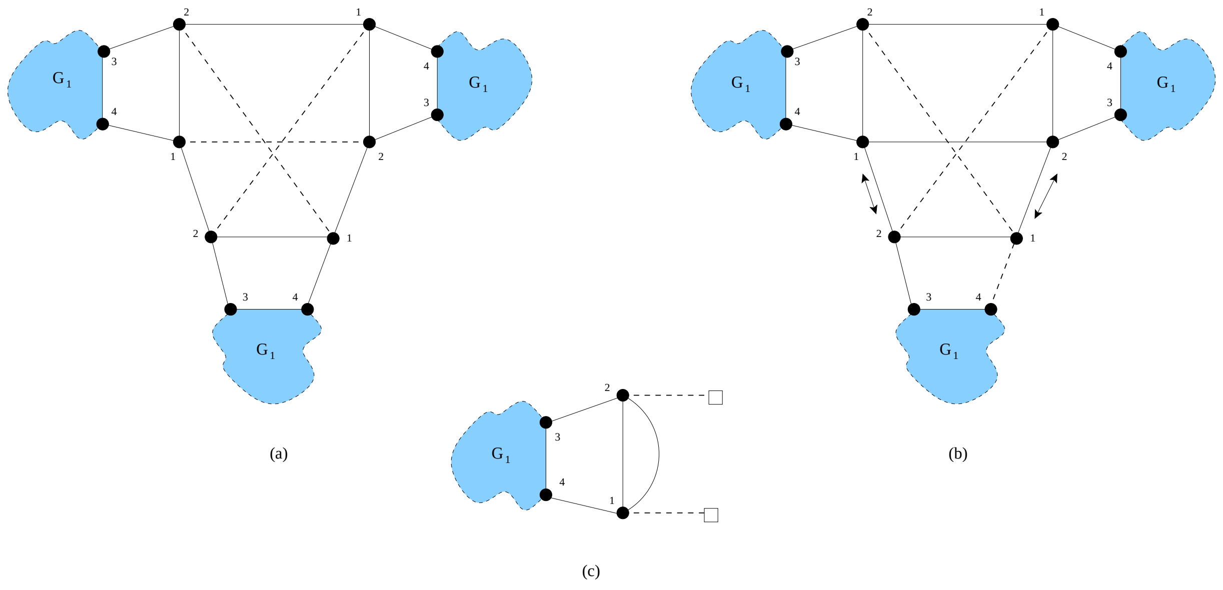


Figure 3: The networks in (a) and (b) have same topology but differ in the location of faulty edges (shown by dashed lines). (c) The minimum base for the network in (a).

no effective algorithm for Rendezvous in faulty networks. Our algorithm always works for any network whose extended-view is covering minimal, achieving the rendezvous of the maximum number of agents possible (i.e. $k - \tau$ agents). We also analyze the complexity of our algorithm and show that it is optimal in terms of the number of moves made by the agents.

4.1 The Algorithm for Rendezvous

We can ensure that no more than one agent dies while traversing the same link, using the *cautious walk* technique as in [13]. At each node, all the incident edges are considered to be unexplored in the beginning. Whenever an agent A at a node u has to traverse an unexplored edge $e = (u, v)$, agent A first marks link $\lambda_u(e)$ as "Being Explored" and if it is able to reach the other end v successfully, it immediately returns to node u and re-marks the link $\lambda_u(e)$ as "safe". During the algorithm we follow the rule that no agent ever traverses a link that is marked "Being Explored". This ensures no more than τ agents may die during the algorithm.

We now briefly describe our algorithm for rendezvous (Algorithm RDV). Due to the space constraint, we present only an oversimplified version of the algorithm without the minor technical details. The interested reader is referred to the complete pseudo-code for the algorithm that can be found in the appendix.

At any stage of the algorithm, there are teams of agents, each team possessing a *territory* which is a connected acyclic subgraph of G (disjoint from other territories). Each team of agents tries to expand its territory until it spans a majority of the nodes. Once a team is able to acquire more than half the nodes of the network, it wins and agents from all other teams join the winning team to achieve rendezvous.

Initially each territory consists of only the starting node(homebase) of an agent (all agents that start from that node are in this team). Note that if an agent on start-up, finds that its homebase has already been acquired by some other team, it simply joins this team. The algorithm proceeds in a series of exploring and competing rounds. In an exploring round, the team of agents try to expand its territory by exploring new edges and acquiring new nodes. On the other hand, in the competing round a team tries to defeat another team and conquering their territory. The competition between two teams occurs by comparison using the tuple $(j, Code)$ where j is the round number and Code is an encoding of the territory(and its immediate neighborhood).

The territory of each team is a rooted tree and the information stored in the root defines the status of the agents in the team (e.g. whether they are in an exploring round or competing round). Every other node in the tree stores a pointer to its parent in the tree. The status of the root can either INIT-EXPLORE, INIT-COMPETE, COMPETE, LOST, or END.

Only one agent in a team can be competing state (this is called the active agent and all others are passive). When a team is in exploring round, the active agent initiates the exploration and thereafter every agent may participate in the exploration. Each agent explores one unexplored edge and if it succeeds in reaching the other side, it reports this to the root of the tree and the new edge is added to the tree. If the edge connects to a node that is already explored (by another agent or the same agent) then it is marked as tree edge(T-edge) and the new node becomes part of the tree. Otherwise it is marked as non-tree edge(NT-edge).

In the algorithm below, T refers to the tree representing the territory to which the agent belongs. The root of T is denoted by r and Root_Status(T) is the status of the root as written on the whiteboard of r (A copy of T is also stored on the whiteboard). A node u is called a *neighbor* of T if there is a NT-edge between u and any node in T. Any agent can be in state *active*, *passive* or *finished*. The following actions are taken by an agent A:

Algorithm RDV

```
rNum := **Initialization**;
While (Status(A) $\neq$ finished ) {
    Case(Status(A) is active and Root_Status(T) is INIT-EXPLORE) {
        If ($|T| > B/2$) **terminate**;
        **InitExplore**;
        If (Root_Status(T) = LOST ) become passive and exit Case;
        Set Root_Status(T) to EXPLORE and become Passive;
    }
    Case( Status(A) is passive ) {
        While(Root_Status(T) $\notin$ { EXPLORE, INIT-EXPLORE, END } )
            Sleep until woken up;
        If(Root_Status(T) is END) become finished and exit;
        Explore an unexplored edge and go back to $r$ to update $T$;
        If(Root_Status(T) is EXPLORE) // i.e. no active agents
          become active and set Root_Status(T) to INIT-COMPETE
    }
    Case(Status(A) is active and Root_Status(T) is INIT-COMPETE) {
        do{ $T_{OLD} := T$ ;
          **InitCompete**($rNum$);
          If (Root_Status(T) = LOST ) become passive and exit Case;
          Result = **Compete**($rNum$); $rNum := rNum + 1$;
        }While( $T_{OLD} \neq T$ );
        If(Root_Status(T) $\neq$ LOST)  Root_Status(T):= INIT-EXPLORE;
    }
}
```

PROCEDURE *Initialization*: If the homebase of the agent is already part of a tree, then the agent joins this team in passive state. Otherwise, it initializes the tree with only the homebase node and then starts the algorithm in active state with $rNum = 1$.

PROCEDURE *InitExplore*: The agent initiates a new exploring round by traversing T writing "EXPLORE" on every node and waking up as many agents as needed for exploration (more precisely, it wakes-up x agents where x is the number of unexplored edges currently incident to T, unless less than x agents are available in T—in which case it wakes-up everyone).

PROCEDURE *InitCompete*(j): The agent initiates the competing round j by traversing T, writing $(COMPETE, j)$ on each node also assigning labels to the nodes in T. Next, it reads the labels written on the neighboring nodes and constructs an encoding of T and its neighbors, called CODE (this is used for comparisons). Finally it writes the CODE on every node in T.

PROCEDURE *Compete*(j): During this procedure, agent A competes with each node u that is neighbor of T, until it wins or loses. If u is marked END, agent A terminates after writing END on every node of T, waking up any sleeping agents and merging with the tree containing u. Else, if u is in a bigger competing round or in the same competing round with a larger Code, then agent A loses (i.e. it becomes passive and goes back to its root to sleep). Otherwise if u is in exploring round or in a smaller competing round or same round but has smaller Code, then Invade(u) is invoked to determine if the result of the competition is win or loss.

PROCEDURE *Invade*(u): The agent attempts to acquire the tree containing node u. The agent follows the father-links from u to reach the current root r_u of u. At node r_u it uses the usual comparison criteria and if r_u is bigger or equal, then it loses (i.e. it becomes passive and goes back to its root to sleep). Otherwise it wins and it acquires the Tree rooted at r_u, by reversing the father-links in the path from r_u to v where v is the node in the agent's territory from where it started the Invade procedure.

PROCEDURE *terminate*: The agent writes END on all nodes in T and then writes Root_Status(T)=END at the root of T; The agent now becomes finished and locally terminates.

4.2 Analysis of the Algorithm

Theorem 4.1 *Algorithm RDV correctly solves Rendezvous for $k - \tau$ agents in any network whose extended view is symmetric-covering minimal.*

This result follows from the following lemmas:

Lemma 4.1 *During the algorithm RDV, the following holds: (i) Each territory is a Tree. (ii) The territories are disjoint. (iii) There is at most one active agent in each territory.*

Lemma 4.2 *There is no deadlock in the algorithm RDV.*

Lemma 4.3 *(i) If the network is minimal, then exactly one node has Root_Status = END. (ii) When one node has Root_Status = END, every alive agent in G eventually joins this tree. (iii) (k-τ) agents eventually become finished and reach the node having Root_Status="END".*

We now analyze the cost of algorithm RDV.

Lemma 4.4 *There are $O(m + k)$ competing rounds. The number of exploring rounds is at most the number of competing rounds.*

Proof : A new competing round is started whenever T is expanded, i.e. the new territory T contains one more edge or one more agent than the previous territory T_{OLD}. Thus there can be at most (m+k) competing rounds. After every exploring round completes, there is one competing round. So, the number of exploring rounds can not be more than the number of competing rounds. ■

Lemma 4.5 *The agents make at most $O(n(m + k))$ moves in the all the exploring rounds combined.*

Proof : The Procedure InitExplore() makes $|T|$ moves in a tree T. Only one agent (the active one) in every tree executes this procedure. So, this accounts for $O(n)$ moves per exploring round and $O(n(m + k))$ moves in total. Other than that, every passive agent that is woken up makes $O(n)$ moves to go to an unexplored edge, explore it and report it to the root. Since each unexplored edge will be explored once (or at most twice), this cost can be counted per newly explored edge. Thus, this accounts for $O(n.m)$ moves. ■

Lemma 4.6 *The agents make at most $O(m(m + k))$ moves in all the competing rounds combined.*

Proof : Only the active agent in a tree participates in the competing round. During procedure InitCompete every edge in G is traversed a constant number of times; this accounts for $O(m)$ moves per round. Similarly O(m) moves are made per round during procedure Compete, for the comparisons with each neighbor. Each execution of Invade() takes $O(n)$ moves, because only tree-edges are traversed and no edge is traversed twice. Whenever an agent execute Invade(), it either wins or loses. A losing agent never competes again, so the total contribution from losing agents is $O(k.n)$. Edges traversed by the wining agents are disjoint, so this accounts for O(n) moves per round. ■

Theorem 4.2 *The moves complexity of algorithm RDV is $O(m(m + k))$. Thus algorithm RDV is optimal.*

5 Conclusions

We considered the problem of rendezvous of mobile agent in a faulty network and showed that it is possible to rendezvous at most $k - \tau$ in any network containing k dispersed agents and $\tau/2$ faulty edges. We determined the condition under which this is possible and gave an algorithm for solving the problem under this condition. The algorithm we presented is optimal in terms of the total number of moves made by the agents and requires no prior information about the network topology (except the size). Moreover, we showed that it is impossible to have an *effective* algorithm for rendezvous, one that always achieves the rendezvous of as many agents as possible in any given network.

Notice that the only information needed by our algorithm is a strict upper bound on the number of nodes. We assumed that the faulty links do not disconnect the network. In the case of a disconnected network, we can still rendezvous the agents in a connected component if we know a good bound on its size. For example, if the component contains a majority of the nodes (i.e. more than half of them), then original network size can be used as the bound. In this case, τ is equal to the number of outgoing edges from the component and we can rendezvous $k' - \tau$ agents where k' agents are initially located in this component.

In the particular case of a single black hole in the network, we can rendezvous $k - d_{BH}$ agents where d_{BH} is the degree of the black-hole (provided that the black hole is not a cut-vertex). If there are $r < n/2$ black holes and the rest of network is still connected, then we can rendezvous $k - \tau$ agents (using $B = n$) where τ is the number of edges which have a safe node on one end and a black-hole on the other end.

References

[1] H. Abu-Amara and J. Lokre. Election in Asynchronous Complete Networks with Intermittent Link Failures. *IEEE Trans. Comput.*, 43(7): 778-788, 1994.

[2] S. Alpern and S. Gal. *The Theory of Search Games and Rendezvous.* Kluwer, 2003.

[3] D. Angluin. Local and global properties in networks of processors. In *Proc. 12th ACM Symp. on Theory of Computing* (STOC '80), 82–93, 1980.

[4] L. Barrière, P. Flocchini, P. Fraigniaud, and N. Santoro. Election and rendezvous in fully anonymous networks with sense of direction. *Theory of Computing Systems*, 2007 (to appear).

[5] P. Boldi and S. Vigna. An effective characterization of computability in anonymous networks. In *Proc. 15th Int. Conference on Distributed Computing (DISC'01)*, 33–47, 2001.

[6] P. Boldi and S. Vigna. Fibrations of graphs. *Discrete Math.*, 243:21–66, 2002.

[7] J. Chalopin, E. Godard, Y. Métivier, and R. Ossamy. Mobile agents algorithms versus message passing algorithms. In *Proceedings of the 10th International Conference on Principles of Distributed Systems (OPODIS)*, 2006.

[8] C. Cooper, R. Klasing, and T. Radzik. Searching for Black-Hole Faults in a Network Using Multiple Agents. In *Proc. 10th International Conference on Principles of Distributed Systems (OPODIS)*, 320-332, 2006.

[9] S. Das, P. Flocchini, A. Nayak, and N. Santoro. Effective elections for anonymous mobile agents. In *Proc. of 17th Int. Symposium on Algorithms and Computation (ISAAC'06)*, 2006.

[10] A. Dessmark, P. Fraigniaud, D. Kowalski, A. Pelc. Deterministic rendezvous in graphs, *Algorithmica*, 46:69-96, 2006.

[11] S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Mobile agents searching for a black hole in an anonymous ring. In *Proc. of 15th Int. Symposium on Distr. Computing (DISC 2001)*, pages 166–179, 2001.

[12] S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Finding a black hole in an arbitrary network: optimal mobile agents protocols. In *Proc. of 21st ACM Symposium on Principles of Distributed Computing (PODC 2002)*, pages 153–162, 2002.

[13] Stefan Dobrev, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Multiple agents rendezvous in a ring in spite of a black hole. In *Proc. 7th Int. Conf. on Principles of Distributed Systems(OPODIS)*, pages 34–46. Springer, 2003.

[14] S. Dobrev, P. Flocchini, R. Kralovic, N. Santoro. Exploring a dangerous unknown graph using tokens. In *Proc. of 5th IFIP International Conference on Theoretical Computer Science (TCS)*, 2006.

[15] D. Dolev, and H.R. Strong. Polynomial algorithms for multiple processor agreement. In Proc. Fourteenth Annual ACM Symposium on theory of Computing (STOC '82), 401-407, 1982.

[16] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2): 374-382, April 1985.

[17] L. Gasieniec, A. Pelc. Broadcasting with linearly bounded transmission faults. *Discrete Applied Mathematics*, 83(1-3): 121-133, 1998.

[18] E. Godard, Y. Métivier, and A. Muscholl. Characterization of classes of graphs recognizable by local computations. *Theory of Computing Systems*, 37(2):249–293, 2004.

[19] Goldreich, O. and Shrira, L. 1986. The effects of link failures on computations in asynchronous rings. In Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing (Calgary, Alberta, Canada, August 11 - 13, 1986). PODC '86. ACM Press, New York, NY, 174-185.

[20] R. Klasing, E. Markou, T. Radzik, F. Sarracco. Hardness and approximation results for black hole search in arbitrary networks. *Theoretical Computer Science* (to appear), 2007.

[21] E. Kranakis, D. Krizanc, and E. Markou. Mobile agent rendezvous in a synchronous torus. In *Proc. 7th Latin American Theoretical Informatics Symposium* (LATIN' 06), SVLNCS 3887, pages 653-664, 2006.

[22] F. T. Leighton. Finite common coverings of graphs. *J. Combin. Theory, Ser. B*, 33:231–238, 1982.

[23] A. Mazurkiewicz. Distributed enumeration. *Inf. Processing Letters*, 61(5):233–239, 1997.

[24] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2): 228-234, 1980.

[25] N. Santoro, and P. Widmayer. Time is Not a Healer. In Proc. 6th Ann. Sym. on Theoretical Aspects of Computer Science (STACS'89), SVLNCS vol. 349, 304–313. 1989

[26] N. Sakamoto. Comparison of initial conditions for distributed algorithms on anonymous networks. In *Proc. 18th ACM Symposium on Principles of Distributed Computing* (PODC '99), 173–179, 1999.

[27] M. Yamashita and T. Kameda. Computing on anonymous networks: Parts I and II. *IEEE Trans. Parallel and Distributed Systems*, 7(1):69–96, 1996.

[28] X. Yu and M. Yung. Agent rendezvous: A dynamic symmetry-breaking problem. In *Int. Coll. on Automata Languages and Programming (ICALP'96)*, pages 610–621, 1996.

APPENDIX

A1. PROOFS:

Proof of Theorem 3.1

Given any graph G with n vertices and m edges, we can construct a family of networks each having $2n+2$ vertices and $4m+2n$ edges, such that any algorithm that solves rendezvous in this family of networks would perform $\Omega(m^2)$ moves in at least one of these networks.

Consider the graph G' in figure **??** consisting of two stars centered at nodes u and v each of which has n neighbors, corresponding to the n vertices of the graph G. Suppose the vertices in G are labelled 1 to n; for each $i \in [1,n]$, there are nodes u_i and v_i that are neighbors of u and v respectively. For each edge $(i,j) \in E(G)$, there are edges $(u_i,u_j),(v_i,v_j),(u_i,v_j),(v_i,u_j)$ in G'. We call the edges $(u_i,u_j),(v_i,v_j)$ as *in-edges* and the edges $(u_i,v_j),(v_i,u_j)$ as *out-edges*. (Intuitively, in-edges connect vertices in the same star, while out-edges connect vertices in two distinct stars. The edges incident to u and v, in G' are called star-edges. Thus, there are $2m$ in-edges, $2m$ out-edges and $2n$ star-edges in G').

Notice that it is possible to assign an edge-labelling λ to the graph in such a way such that the labelled graph (G',λ) covers a graph (H,λ) of size $n+1$, where nodes u and v are mapped to the same vertex and for each $1 \le i \le n$, u_i and v_i are mapped to the same vertex. We can define ξ, p, η in such a way that there is one agent in every node and all the edges are safe. In this network $N = (G', \xi, p, \lambda, \eta)$, the agents cannot distinguish between in-edges and the out-edges.

Now, consider a family $\mathcal{F}$ of networks, each similar to N but having an odd number of edge as faulty, in such a way that it does not disconnect the network. Notice that each of these networks is covering-minimal (since the location of the unsafe edges would break the symmetry between the left and right parts of the network) and thus it is possible to solve rendezvous of $k-\tau$ agents for each network in $\mathcal{F}$. Suppose $\mathcal{P}$ be an algorithm that solves rendezvous of $k-\tau$ agents in each network in the family. Consider a network $N' \in \mathcal{F}$ where for each pair (i,j) either both (u_i,u_j) and (v_i,v_j) or both (u_i,v_j) and (v_i,u_j) are faulty; except for one pair (x,y) where all the four edges (u_x,u_y), (v_x,v_y), (u_x,v_y) and (v_x,u_y) are safe edges. Consider an execution of an algorithm $\mathcal{P}$ on this network, controlled by an adversary. During this execution the following happens: all the in-edges and out-edges (except the unsafe ones) are very slow, i.e. any agents traversing these edges are blocked by the adversary. At this stage of the execution, the symmetry between the two parts is still maintained and thus, the *in-edges* are indistinguishable from the *out-edges*. Suppose the adversary unblocks the edges one by one, (while maintaining the symmetry between the two parts until the very end i.e., as soon as an edge is unblocked the corresponding symmetric edge must be unblocked immediately).
CLAIM: Whenever one edge is unblocked, all the safe and unblocked edges (except the star-edges) must be traversed by some agent!!

Suppose some edge e is not traversed, this could be the only out-edge that is slow and all the other out-edges could be unsafe (note that such a network belongs to the family). In this case, the algorithm would not achieve rendezvous because the agent in the left and right parts remain separated.

Due to the above claim, all the in-edges and the out-edges except ones that are blocked must be traversed every time one pair of edges is unblocked. There are originally $2m+1$ blocked edges and $2m-1$ unsafe ones among the in-edges and out-edges. So, the total number of edge-traversals made during the algorithm is $\Omega(m^2)$.

A similar argument can be used to show that the number of moves made by the algorithm must be $\Omega(m.k)$. In this case, we consider the family of networks with same G', λ, η and $k = 4p+1$ agents. Initially there are p agents in each star and the rest of the agents appear one by one, such that every time an agent appears, all the in-edges and out-edges must be traversed. ■

Proof of Lemma 4.1

(i) Initially each territory is a tree by construction; addition of a new edge does not create cycles because only tree edges are added. When two trees are merged, one is bigger than other and the active agent of the bigger tree performs the merging by changing the father-links and updating the root. Note that there can not be two active agents in a territory. Thus the merged territory is a tree. (ii) Each tree is a rooted tree and every node contains a pointer to its parent in the tree. Thus, a node cannot belong to two trees. (iii) Initially an agent becomes active only when it successfully constructs T and writes it at the root node. Due to the mutual exclusion property of the whiteboards, there is only one active agent initially in the tree (all other agents in the same tree must start in passive state). A passive agent can become active only when there are no active agents in the tree. ■

Proof of Lemma 4.2
Notice that as long as there is some active agent, the algorithm progresses. In each tree T, there is at least one active agent unless root-status is EXPLORE. Suppose the root-status of every tree is EXPLORE, then each agent is currently exploring some unexplored edge. There are only τ faulty links in the network and $k > \tau$, so at least one of the exploring agents must return safely from the exploration. This agent now becomes active. ■

Proof of Lemma 4.3
(i) First notice that due to the constraint on the bound B, only one Tree T can have a size greater than $B/2$. Due to Lemma 4.1, the trees are disjoint and each has a unique root. Thus, at most one node may have Root_Status=END. We now show that at least one node eventually reaches Root_Status=END.

A team of agents in tree T stops expanding its Tree only when either Root_Status(T)=END or, all the trees neighboring T have the same CODE and round number. In the later case, the team starts an exploring round. Notice that there must exist a Tree T which has some non-faulty edges incident to it that are either unexplored or being explored (otherwise the network is either disconnected or, not minimal). So one of these edges would be added to T. Thus the size of T would keep increasing until it contains more than $B/2$ nodes and Root_Status(T) = END.

(ii) Every agent either dies or reaches a node labelled "END". All nodes that are labelled END are part of the same tree.

(iii) Due to the use of *cautious walk*, at most τ agents may die. Thus each of the surviving agent eventually reaches a node v labelled END and goes to the root of this tree which is the node having Root_Status=END. ■

A2. Algorithms and Procedures:

```
Procedure initExplore($T, na$)
    $count := 0$;
    // traverse territory $T$ and do the following:
    foreach $v \in T$ do
        status($v$) := EXPLORE ;
        $sla := |\{$ agents in sleeping at $v\}|$ ;
        wake up $\max(na - count, sla)$ agents sleeping at $v$ ;
        $count := count + \max(na - count, sla)$ ;
    goToRoot ;
    return($count + 1$);
```

```
Procedure explore
    $(v, p) :=$ next-edge($\mathsf{T}(r)$) ;
    // returns a vertex $v \in \mathsf{T}(r)$ and the port-number $p$ of an unexplored edge incident to $v$
       after marking it as BEING EXPLOREDin $\mathsf{T}(r)$
    go to node $v$ ;
    port-status($v$)[$p$] := BEING EXPLORED ;
    leave $v$ through port $p$ to arrive in a vertex $u$ trough port $q$ ;
    if in-a-tree($u$) then
        leave $u$ through port $q$ ;
        port-status($v$)[$p$] := $NT$ ;
        goToRootToReport ;
        update $T$ by marking $(v, p)$ as $NT$ ;
    else
        in-a-tree($u$) := $true$ ;
        AgentCount($u$) := 0 ;
        father($u$) := $q$ ;
        port-status($v$)[$q$] := $T$ ;
        foreach $p \in [1, \deg(u)]$ where $p \neq q$ do
            port-status($u$)[$p$] := UNEXPLORED ;
        status($u$) := EXPLORE ;
        leave $u$ through port $q$ ;
        port-status($v$)[$p$] := $T$ ;
        root-status($v$) := $\perp$ ;
        goToRootToReport ;
        update $\mathsf{T}(r)$ by adding $u$ and marking $(v, p)$ and $(u, q)$ as $T$ and marking the other ports of $u$ as
        UNEXPLORED;
```

Algorithm 3: rdv

```
init ;
while $mystatus \neq finished$ do
    case $\textit{root-status}(v) = $ INIT-EXPLORE $and\ mystatus = active$
        $need := |\{(u,p) \mid u \in T$ and $\textsf{port-status}(u)[p] = $ UNEXPLORED$\}|$ ;
        $exploringAgents(v) := 0$ ;
        $available(v) :=$ initExplore($\mathsf{T}(v), need - 1$) ;
        $mystatus = passive$ ;
    case $\textit{root-status}(v) = $ INIT-EXPLORE $and\ mystatus = passive$
        if there exists an unexplored edge incident to a vertex of $\mathsf{T}(v)$ then
            $exploringAgents(v) := exploringAgents(v) + 1$ ;
            if $exploringAgents(v) = available(v)$ then root-status($v$) := EXPLORE ;
            explore ;
            if $\textit{root-status}(v) = $ INIT-EXPLORE then
                wait until root-status($v$) $\neq$ INIT-EXPLORE ;
            if $\textit{root-status}(v) = $ EXPLORE then
                root-status($v$) := INIT-COMPETE ;
                round($v$) := round($v$) + 1 ;
                $mystatus := active$ ;
            else if $\textit{root-status}(v) = $ END then
                $mystatus := finished$ ;
            else
                sleep ;
        else if $|\mathsf{T}(v)| > B/2$ then
            root-status($v$) := END ;
            foreach $v \in \mathsf{T}(v)$ do
                status($v$) := END ;
                wake up all sleeping agents ;
            $mystatus := finished$ ;
        else
            sleep ;
    case $\textit{root-status}(v) = $ INIT-COMPETE $and\ mystatus = active$
        $T_{OLD} := \mathsf{T}(v)$ ;
        $code :=$ initCompete($T_{OLD}$, round($v$));
        if $\textit{root-status}(v) \neq$ LOST then
            root-status($v$) := COMPETE ;
            $mystatus := active$ ;
    case $\textit{root-status}(v) = $ COMPETE $and\ mystatus = active$
        $result :=$ COMPETE(round($v$), $code, T_{OLD}$) ;
        if $result = finished$ and $\textit{root-status}(v) \neq$ LOST then
            $mystatus := finished$ ;
        else if $result = winner$ then
            update $\mathsf{T}(v)$ ; // Add the newly acquired territory to $T$.
            if $\textit{root-status}(v) \neq$ LOST then
                root-status($v$) := INIT-COMPETE ;
                round($v$) := round($v$) + 1 ;
        else if $result = equal$ and $\textit{root-status}(v) \neq$ LOST then
            if $\mathsf{T}(v) \neq T_{OLD}$ then
                root-status($v$) := INIT-COMPETE ;
                round($v$) := round($v$) + 1 ;
            else
                root-status($v$) := INIT-EXPLORE ;
        else
            wait until root-status($v$) = LOST;
    case $\textit{root-status}(v) = $ LOST $and\ mystatus = active$
        root-status($r$) := GRAB ;
        sleep ;
```

Procedure goToRoot

```
// enable an agent located at a node $v$ to go to the root of the tree currently
   containing $v$.
if $father(v) \neq 0$ then
    leave $v$ through port $father(v)$ ;
    goToRoot ;
```

Procedure goToRootToInvade

```
// enable an agent located at a node $v$ to go to the root of the tree currently
   containing $v$ when it wants to grab this tree.
if $wait\text{-}status(v) = \text{YES}$ then
    wait until $wait\text{-}status = \text{NO}$ ;
$wait\text{-}status(v) := \text{YES}$ ;
if $father(v) \neq 0$ then
    leave $v$ through port $father(v)$ ;
    goToRootToInvade ;
```

Procedure goToRootToReport

```
// enable an agent located at a node $v$ to go to the root of the tree currently
   containing $v$ when it wants to report that an edge is safe.
if $wait\text{-}status(v) = \text{YES}$ then
    wait until $wait\text{-}status = \text{NO}$ ;
if $father(v) \neq 0$ then
    leave $v$ through port $father(v)$ ;
    goToRootToReport ;
```

Procedure goToRootForEnding(u)

```
// enable an agent located at a node $v$ to go to the root of the tree currently
   containing $v$ when it knows that the state of one of the neighbor of the node $u$ is
   END.
if $wait\text{-}status(v) = \text{YES}$ then
    go to $u$ and for each $w$ on the path leading to $u$, $wait\text{-}status(w) := \text{NO}$ ;
    goToRoot ;
else
    $wait\text{-}status(v) := \text{YES}$ ;
    if $father(v) \neq 0$ then
        leave $v$ through port $father(v)$ ;
        goToRootForEnding($u$) ;
```

Procedure `initCompete(`T, j`)`

```
$count := 1$ ;
// traverse territory $T$ and do the following
**foreach** $v \in T$ **do**
    round($v$) := $j$ ;
    status($v$) := INIT-COMPETE ;
    Label($v$) := $count$ ;
    $count := count + 1$ ;
goToRoot ;
**if** root-status($r$) = LOST **then return**($\perp$) ;
// traverse territory $T$ and do the following
**foreach** $v \in T$ **do**
    N($v$) := $\emptyset$ ;
    **foreach** port $p$ at $v$ **do**
        **if** port-status($v$)[$p$] = UNEXPLORED **then**
            N($v$) := N($v$) $\cup \{(p, -1, \perp, \perp, \perp)\}$
        **else if** port-status($v$)[$p$] = BEING EXPLORED **then**
            N($v$) := N($v$) $\cup \{(p, 0, \perp, \perp, \perp)\}$
        **else**
            **leave** $v$ through port $p$ to reach a node $u$ through port $q$ ;
            N($v$) := N($v$) $\cup$ {($p, q$, Label($u$), AgentCount($u$), round($u$))}
    $code$[Label($v$)] := (AgentCount($v$), father($v$), N($v$)) ; ;
// $code$ encodes information about every node $v \in T$ and its neighborhood.
goToRoot ;
**if** root-status($r$) = LOST **then return**($\perp$) ;
// traverse territory $T$ and do the following
**foreach** $v \in T$ **do**
    code($v$) := $code$ ;
    status($v$) := COMPETE ;
goToRoot ;
**return**($code$) ;
```

Procedure `flipEdge(`q`)`

```
**if** father($v$) $\neq 0$ **then**
    $p$ := father($v$) ;
    father($v$) := $q$ ;
    wait-status := NO ;
    **leave** $v$ through port $p$ to reach a node $u$ through port $q$ ;
    flipEdge($q$) ;
**else**
    wait-status := NO ;
    father($v$) := $q$ ;
```

Procedure compete(j,$code$,T)

```
// traverse territory $T$ and do the following.
foreach $v \in T$ do
    foreach $p \in [1, \deg(v)]$ with port-status$(v)[p] = NT$ do
        leave $v$ through port $p$ to reach a node $u$ through port $q$ ;
        if status$(u) =$ COMPETE and round$(u) = j$ and code$(u) = code$ then
            go to $v$ ;
        else if status$(u) =$ END then
            go to $v$ ;
            goToRootForEnding$(v)$ ;
            Ending$(v, p)$;
            return (finished) ;
        else if status$(u) =$ COMPETE and (round$(u) > j$ or (round$(u) = j$ and code$(u) > code$)) then
            go to $v$ ;
            goToRoot ;
            return (loser) ;
        else
            goToRootToInvade; // The root reached must be active(not GRAB/LOST).
            // In the following $r$ denotes the root of the tree containing $u$
            if root-status$(r) =$ INIT-EXPLORE then
                wait until root-status$(r) \neq$ INIT-EXPLORE
            if root-status$(r) =$ END then
                go to $v$ and for each $w$ on the path leading to $v$, wait-status$(w) :=$ NO ;
                goToRootForEnding$(v)$ ;
                Ending$(v, p)$;
                return (finished) ;
            else if root-status$(r) =$ COMPETE and (round$(r) > j$ or (round$(r) = j$ and code$(r) \geq code$))
            then
                go to $v$ and for each $w$ on the path leading to $v$, wait-status$(w) :=$ NO ;
                goToRoot ;
                return (loser) ;
            else
                if root-status$(r) \neq$ EXPLORE then
                    root-status$(r) :=$ LOST ;
                    round$(r) := j$ ;
                    wait until root-status$(r) =$ GRAB ;
                else
                    root-status$(r) :=$ GRAB ;
                go to $u$ ;
                flipEdge $(q)$ ;
                root-status$(r) := \perp$ ;
                goToRoot ;
                return (winner) ;
return($equal$) ;
```

```
Procedure init
if in-a-tree($v$) then
    AgentCount($v$) := AgentCount($v$) + 1 ;
    goToRoot ;
    update T($r$) ;
    if root-status($v$) = INIT-EXPLORE then
        wait until root-status($v$) $\neq$ INIT-EXPLORE ;
    if root-status($v$) = EXPLORE then
        root-status($v$) := INIT-COMPETE ;
        $mystatus := active$ ;
    else if root-status($v$) = END then
        $mystatus := finished$ ;
    else
        sleep ;
else
    in-a-tree($v$) := $true$ ;
    AgentCount($v$) := 1 ;
    father($v$) := 0 ;
    foreach $p \in [1, \deg(v)]$ where $p \neq q$ do
        port-status($v$)[$p$] := UNEXPLORED ;
    T($v$) := $\{v\}$ ;
    status($v$) := EXPLORE ;
    root-status($v$) := INIT-EXPLORE ;
    round($v$) := 0 ;
    Label($v$) := 0 ;
    $mystatus := active$ ;
```

```
Procedure sleep
$mystatus := passive$ ;
wait until being woken up by another agent ;
goToRoot ;
if root-status($v$) = END then
    $mystatus := finished$ ;
else if root-status($v$) $\notin$ {INIT-EXPLORE, EXPLORE} then
    sleep ;
```

```
Procedure Ending($v_{start}, p$)
// Let $r$ be the current node ( must be a root node ).
if root-status($r$) $\neq$ LOST then
    wait-status($r$) := YES ;
    root-status($r$) := GRAB ;
    // traverse territory $T$ and do the following.
    foreach $w \in T$ do
        status($w$) := END ;
        wake up all sleeping agents ;
    go to $v_{start}$ ;
    flipEdge ($p$) ;
    root-status($r$) := $\perp$ ;
    wait-status := NO ;
    goToRoot ;
```