Distrib. Comput. (2015) 28:155–170
DOI 10.1007/s00446-014-0236-1

Distributed graph searching with a sense of direction

Piotr Borowiecki · Dariusz Dereniowski · Łukasz Kuszner

Received: 21 October 2013 / Accepted: 20 October 2014 / Published online: 19 November 2014

Abstract In this work we consider the edge searching problem for vertex-weighted graphs with arbitrarily fast and invisible fugitive. The weight function ω provides for each vertex v the minimum number of searchers required to guard v, i.e., the fugitive may not pass through v without being detected only if at least $\omega(v)$ searchers are present at v. This problem is a generalization of the classical edge searching problem, in which one has $\omega \equiv 1$. We assume that with a graph G to be searched, there is associated a partition $(V_1, \ldots, V_t)$ of its vertex set such that edges are allowed only within each V_i and between two consecutive V_i's. We provide an algorithm for distributed monotone connected edge searching of such graphs, where the searchers are initially placed on an arbitrary vertex of G and have no a priori knowledge on G, but they have a sense of direction that lets them recognize whether an edge incident to already explored vertex in V_i leads to a vertex in one of V_{i-1}, V_i or V_{i+1}. Starting from any vertex the algorithm uses at most $3 \cdot \max_{i=1,\ldots,t} \omega(V_i)+1$ searchers, where $\omega(V_i) = \sum_{v \in V_i} \omega(v)$. We also prove that this algorithm is best possible up to a small additive constant, that is, each distributed searching algorithm in worst case must use $3 \cdot \max_{i=1,\ldots,t} \omega(V_i) - 1$ searchers for some graphs.

This work was partially supported by Polish National Science Center under Contract DEC-2011/02/A/ST6/00201. Dariusz Dereniowski was partially supported by a scholarship for outstanding young researchers founded by the Polish Ministry of Science and Higher Education.

P. Borowiecki · D. Dereniowski (✉) · Ł. Kuszner
Faculty of Electronics, Telecommunications and Informatics, Gdansk University of Technology, Narutowicza 11/12, 80-233 Gdańsk, Poland
e-mail: deren@eti.pg.gda.pl

P. Borowiecki
e-mail: pborowie@eti.pg.gda.pl

Ł. Kuszner
e-mail: kuszner@eti.pg.gda.pl

Keywords Connected searching · Graph exploration · Distributed algorithm · Online algorithm · Fugitive search games · Pathwidth

1 Introduction

A team of mobile agents explores an unknown environment modeled as a graph in order to accomplish a selected task. The agents start from their initial configuration and may gain some knowledge on the topology of the graph only as a result of its path traversals. The task to be performed varies and may include exploration [2,12,20,26], map construction [10,14,18], gathering [13,19,32,40] or leader election [5,6,28]. In this work we focus on the task of capturing a fast fugitive hiding in the graph. The fugitive may represent a virus spreading in the network, a hostile agent that needs to be caught, or a mobile entity that is lost and needs to be found. Moreover, we are interested in *guaranteed* capture, which from the point of view of the agents implies that their movement must lead to capturing the fugitive regardless of its actions. The optimization criterion mostly considered for such problems is the minimum number of agents (called usually searchers in the graph searching terminology) required to complete this task.

We assume that the searchers are initially located on an arbitrarily selected vertex called the *homebase*. If we imposed no additional restrictions on the strategy that the searchers have to form, then the searching problem could be solved by first using any exploration algorithm that allows to determine the structure of the underlying graph (by one or more searchers). Then, once the structure of the graph is known, a search strategy could be computed (by any offline algorithm) and finally the strategy could be implemented by the searchers. In such a case the problem reduces to map

construction. However, one may have some restrictions on the search strategy that makes such an approach invalid. A typical restriction is that the strategy needs to be *monotone*, that is, each edge or vertex that has been cleared (i.e., is guaranteed to be free of the fugitive) must remain clear forever, see e.g. [3,38]. In such a setting, the computation of the strategy cannot be preceded by an exploration stage, and thus the strategy is formed online.

It has been proved that if the searchers posses no additional knowledge on the structure of the graph to be searched, then for some graphs every online algorithm that the searchers use, leads to arbitrarily bad strategies. In particular, even if the graph is a tree on n vertices, then in the worst case every online[1] search strategy of such a type needs $\Theta(n)$ searchers [31], while $O(\log n)$ searchers are enough in an offline solution [35]. This negative result justifies an assumption that the searchers have some additional ability that allows them to construct a strategy more efficiently. The selection of additional conditions that realize such an assumption is a separate and interesting problem (see e.g., the notion of online/distributed computation with advice [21,27]). In this work we assume that the vertex set of a graph is partitioned into sets $V_1, \ldots, V_t$ such that the neighborhood of a vertex in V_i is contained in $V_{i-1} \cup V_i \cup V_{i+1}$ (we take $V_0 = V_{t+1} = \emptyset$), and each searcher has a sense of direction that allows it to decide if an edge outgoing from a currently occupied vertex in V_i leads to a vertex either in V_{i-1}, V_i or V_{i+1}. The $(V_1, \ldots, V_t)$ is called the *grid partition* of G.

As an example that illustrates the applicability of our model we refer to the seminal paper of Breisch [9]. Consider the system of caves depicted in Fig. 1a. The searchers may agree on a set of virtual parallel lines (called *scanlines* for brevity) overlapping the environment. Then, let each intersection of a tunnel and a scanline be modeled as a graph vertex, and two vertices being adjacent if they belong to two consecutive scanlines or to the same scanline and are connected by a direct tunnel. Hence, a set V_i may be used to denote all vertices implied by ith scanline (see Fig. 1b). The searchers have to agree on the placement of scanlines prior to the beginning of exploration and this model is valid if there are no tunnel intersections 'between' the scanlines—one way of overcoming this issue is to take small distance between consecutive scanlines (this may lead to many degree two vertices in the graph but such vertices introduce no difficulty while constructing edge search strategies). The terrain modeling that leads from an actual environment to a graph is a separate and interesting problem that we do not consider in this work. If the searchers are equipped with a compass and are able to measure the distance traveled, then each of them is able to translate an edge traversal in the graph into the corresponding movement in the terrain and vice versa. Hence, the online graph searching algorithm presented in this work can be directly used to obtain a search strategy for the system of tunnels.

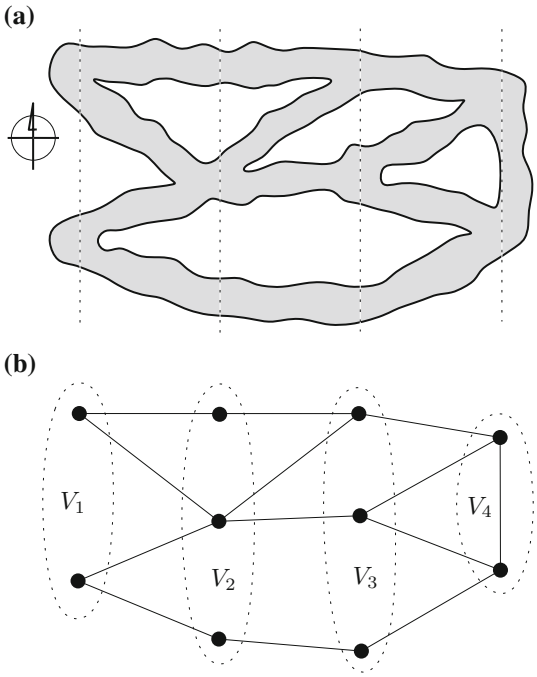

Fig. 1 A system of caves with scanlines and the corresponding graph that models the environment

1.1 Graph searching problem

In this section we discuss the edge searching problem introduced by Parsons in [39]. In fact, we recall a little bit more general definition of the problem, using a simple, undirected, *weighted graph* $G = (V, E, \omega)$ with the vertex set V, edge set E and the *weight function* $\omega\colon V \to \mathbb{N}$ that assigns to every vertex v in V a positive integer called its *weight*. For a set $U \subseteq V$ we write $\omega(U)$ to denote the sum of the weights of all vertices in U, i.e., $\omega(U) = \sum_{v \in U} \omega(v)$. The goal is to capture an omniscient, invisible and fast fugitive in a weighted graph. The fact that the fugitive is *omniscient* implies that it will avoid the capture as long as possible, or in other words, it knows in advance the strategy used by the searchers and therefore it will always choose the most advantageous position for itself. The consequence of the *invisibility* is that the searchers do not know the location of the fugitive and can only compute its possible locations using the history of their moves. The weight of a vertex v is the number of searchers required to *guard* v against the fugitive. In other words, the fugitive cannot pass without being detected through a vertex v holding at least $\omega(v)$ searchers. The fact that the fugitive is *fast* implies that at any moment it can traverse a path of an

[1] In view of the terms used in the literature on the subject, the terms online and distributed are used interchangeably in this work.

arbitrary length provided that no vertex of the path is guarded. In this work we assume that the graphs we consider are *connected*, i.e., for every two vertices u, v of the graph there exists a path between u and v. This does not lead to a loss of generality. Indeed, in general, an input graph for the edge searching problem does not have to be connected; having a clearing procedure for connected graphs we are able to clear every graph applying the procedure to each connected component independently. However, in such case it is required that each connected component has its own homebase with sufficient number of searchers initially placed on it.

1.2 Search strategies

An *edge search strategy* $\mathcal{S}$ (or *search strategy* for short) for a graph G is a sequence of moves $\mathcal{S}_1, \ldots, \mathcal{S}_l$ such that for each $i \in \{1, \ldots, l\}$ the move $\mathcal{S}_i$ is one of the following:

(m1) placing any number of searchers on a vertex of G,
(m2) removing any number of searchers from a vertex of G,
(m3) sliding any number of searchers that occupy a vertex u along an edge $\{u, v\}$ from u to v.

We say that an edge is *clear* at the end of a particular move of $\mathcal{S}$ if it is guaranteed to be free of the fugitive. Otherwise, an edge is said to be *contaminated*. A contaminated edge $\{u, v\}$ becomes clear in a move $\mathcal{S}_i$ if the following conditions hold:

(a) at the beginning of $\mathcal{S}_i$ at least $\omega(u)$ searchers occupy u,
(b) at least one searcher slides along $\{u, v\}$ from u to v in $\mathcal{S}_i$ (hence $\mathcal{S}_i$ must be of type (m3)),
(c) if at least two edges incident to u (respectively v) are contaminated at the beginning of $\mathcal{S}_i$, then at least $\omega(u)$ ($\omega(v)$, respectively) searchers are at u (v, respectively) at the end of $\mathcal{S}_i$.

For convenience, we use the terms clear and contaminated when referring to vertices as well. A vertex v is clear if it is guarded or all edges incident to v are clear. This may occur as a result of a move of type (m1) or (m3). Thus, a guarded vertex is clear.

A vertex or an edge becomes *recontaminated* in a move $\mathcal{S}_i$, $i \in \{2, \ldots, l\}$, if it is clear at the end of move $\mathcal{S}_{i-1}$ but may hold the fugitive at the end of move $\mathcal{S}_i$. We say that a search strategy $\mathcal{S}$ is *monotone* if no move of $\mathcal{S}$ results in recontamination.

A search strategy $\mathcal{S}$ is *connected* if the subgraph of G consisting of all currently clear vertices and edges, called the *cleared subgraph*, is connected at the end of each move of $\mathcal{S}$. Note that the 'classical' edge searching and connected edge searching problems for unweighted graphs are equivalent to our problem with the weight of each vertex equal to 1. Given a search strategy $\mathcal{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_l)$, if $|\mathcal{S}_i|$ is the number of searchers present in a graph G in the move $\mathcal{S}_i$, then

$$\mathtt{s}(\mathcal{S}) = \max\big\{|\mathcal{S}_i| \;\big|\; i \in \{1, \ldots, l\}\big\}$$

is the least number of searchers required by $\mathcal{S}$. Define

$$\mathtt{cs}(G) = \min\big\{\mathtt{s}(\mathcal{S}) \;\big|\; \mathcal{S} \text{ is a connected search strategy of } G\big\}$$

called the *connected search number of* G.

In this work we are interested in computing monotone connected search strategies. The monotonicity is required since we consider the online version of the searching problem: if a search strategy needs not to be monotone, then one searcher can explore the graph G (causing recontaminations in the process) and once it learns the structure of G, the searchers may compute a search strategy (in an offline manner) and use the strategy on G; in such a case the problem reduces to its offline version.

Hence, in view of those assumptions, we can ensure a simpler structure of every strategy $\mathcal{S}$ we consider. Namely, we assume that in every strategy $\mathcal{S}$, exactly $\mathtt{s}(\mathcal{S})$ searchers are initially placed on a single vertex h called the *homebase* and every move of $\mathcal{S}$ is of type (m3). Therefore, for simplicity, we omit the move of type (m1) that initially places all searchers on h. For any $h \in V$, define

$$\mathtt{mcs}(G, h) = \min\{\mathtt{s}(\mathcal{S}) \;\big|\; \mathcal{S} \text{ is a monotone connected search strategy of } G \text{ with homebase } h\},$$

and

$$\mathtt{mcs}(G) = \min\big\{\mathtt{mcs}(G, h) \;\big|\; h \in V\big\}.$$

The latter graph parameter is called the *monotone connected search number of* G.

1.3 The distributed/online graph searching model

In Sect. 1.1 and 1.2 one might assume that the graph to be searched was known to the searchers in advance. The purpose of this section is to formally describe our distributed model, whose main principle is that the searchers do not know a priori the structure of the graph, and need to learn it by performing the search moves.

For each vertex v and each edge $\{u, v\}$ there exists a positive integer denoted by $\rho(v, u)$ and called the *port number of* $\{u, v\}$ *at* v. Moreover, it is assumed that the set $P(v) = \{\rho(v, u) \;|\; \{u, v\} \in E\}$, consisting of the *port numbers at* v, equals $\{1, \ldots, \deg(v)\}$. Hence, any two edges incident to the same vertex v have different port numbers at v. Consequently, any searcher occupying v can use the port numbers to distinguish between the incident edges. If a searcher occupying a vertex v decides to slide along an edge $\{u, v\}$, then we say that the searcher *leaves* v *via port* $\rho(v, u)$ and *enters* u *via port* $\rho(u, v)$.

We now define the *sense of direction*, informally introduced in Sect. 1, that lets the searchers distinguish the three

directions *left*, *right* and *straight* so that any port number at the currently occupied vertex v can be uniquely classified as a port that leads to a vertex that is either to the left, to the right or in front of v. The sense of direction is realized as an *oracle function* ϕ that assigns a number in $\{-1, 0, 1\}$ to each pair (v, p), where v is a vertex and $p \in P(v)$. Hence, v and p are mapped to one of the three numbers that correspond to the left, straight and right direction, respectively. In other words, when a searcher leaves a vertex v via its port p, then $\phi(v, p)$ determines the direction of searcher's move. It is worth noting, that the above notion of sense of direction does not require usage of an external source of knowledge represented by an oracle. Instead, one might as well assume that the directions are given at vertices together with port numbers.

In order for the 'relative positioning' to be consistent, we need to impose some additional restrictions that ϕ needs to satisfy. More formally, we require that there exists a grid partition $(V_1, \ldots, V_t)$ of G such that for each $\{u, v\} \in E$ it holds:

(1) $u \in V_i$ and $v \in V_j, i, j \in \{1, \ldots, t\}$, implies $|i - j| \leq 1$,
(2) $u, v \in V_i, i \in \{1, \ldots, t\}$ if and only if $\phi(v, \rho(v, u)) = 0$ and $\phi(u, \rho(u, v)) = 0$; in such a case we say that u is *in front of* v and vice versa,
(3) $u \in V_i$ and $v \in V_{i+1}$, $i \in \{1, \ldots, t-1\}$ if and only if $\phi(u, \rho(u, v)) = 1$ and $\phi(v, \rho(v, u)) = -1$; in such case we say that u is *to the left of* v and v is *to the right of* u.

Consequently, whenever a searcher that occupies a vertex $v \in V_i$ leaves v via port p, then it reaches some vertex u in $V_{i+\phi(v,p)}$. The vertex u is called the *goal of the move* from v via port p and it is denoted by $g(v, p)$.

Recall that searchers have no a priori knowledge on the unexplored vertices of G. If a searcher occupies a vertex v, then it learns all port numbers at v (or equivalently, the degree of v) and for each $p \in P(v)$ the searcher learns $\phi(v, p)$. An important assumption is that given two explored vertices $v_1, v_2 \in V_i$ such that $\phi(v_1, p_1) = \phi(v_2, p_2)$ for some port numbers p_1, p_2 the searchers do not know in advance whether $g(v_1, p_1) = g(v_2, p_2)$, i.e., the goal remains unknown as long as the appropriate moves are performed.

We assume that the searchers have unique identifiers but the nodes are anonymous. Our algorithm is described as if there existed a global process that at each point knows the locations of all searchers and knows the structure of the graph explored to date. This process makes a decision regarding the next move that is then performed by the searchers. However, the algorithm is described in such a way only to simplify its pseudocode, and it can be easily turned into an algorithm in which the searchers communicate locally and learn the structure of the explored part of the graph by exchanging messages during meetings. This can be achieved as follows: after each clearing move any free searcher is responsible for computing the next clearing move and 'coordinating' its execution (thanks to unique identifiers the searchers can select the free searcher with minimum identifier for this task). This searcher first learns the cleared subgraph of G (e.g., by traversing all clear edges; see e.g. [11] for some methods for exploring a port-labeled network by a single searcher) and then computes the next move (in an offline fashion). Once the move is computed, the searcher informs all other searchers (by visiting each of them and assigning tasks according to their identifiers) what is the next move to be done. Then, the next clearing move follows.

We finish this section with a few remarks considering our model choice. We mention several possible extensions of our model to point out that our algorithm can be turned into a fully distributed one. However, we do not encode those extensions directly into our pseudocode in order to present the combinatorial structure of our method in a more transparent way.

As to the communication model, our centralized algorithm can be turned into a distributed one in which searchers can communicate directly when occupying the same vertex, or searchers can communicate through whiteboards. Indeed, this is straightforward when the unique searcher mentioned above is responsible for computing the next move. We assume that several searchers move along an edge simultaneously to avoid immediate recontamination, which occurs whenever the number of searchers reaching a newly explored vertex is less than its weight. However, if searchers travel with different speeds at any moment (the speed is chosen by an adversary) our algorithm produces a search strategy that clears the graph causing only 'local' recontaminations (recontamination of one edge only). Indeed, if several searchers slide from a vertex u to a contaminated vertex v which clears the edge in the synchronous model, then in the asynchronous one the edge is guaranteed to be cleared when the last searcher reaches v. An additional technical issue that needs to be resolved in the asynchronous model is the one of determining whether a move is completed, i.e., whether all searchers that started traversing an edge have finished it. This can be done thanks to the unique identifiers of the searchers.

Finally, we note that the algorithm we propose requires that the size of memory used by searchers is polynomial in the size of the graph to be searched. It is an interesting research direction to develop algorithms for searchers with stronger restrictions on the memory size, e.g., independent of the size of the graph to be explored.

1.4 Our results

The *width* $\mathrm{w}(G)$ of a weighted graph $G = (V, E, \omega)$ with a grid partition $(V_1, \ldots, V_t)$ is defined as

$$\mathrm{w}(G) = \max_{i \in \{1,\ldots,t\}} \omega(V_i).$$

Our main result is the following theorem:

Theorem 1 *Let $G = (V, E, \omega)$ be a weighted graph with a fixed grid partition and let ϕ be an oracle function defined on $V(G)$. For every homebase vertex h of G there exists a distributed algorithm that has no a priori knowledge on G and guarantees that whenever $3\mathrm{w}(G) + 1$ searchers are initially present at h, then the execution of the algorithm by the searchers results in a monotone connected search strategy for G with homebase h.*

This theorem in particular implies that there exists a monotone connected searching strategy that uses at most $3\mathrm{w}(G) + 1$ searchers for G, which gives the following.

Corollary 1 *If G is a weighted graph with a grid partition, then*

$$\mathrm{mcs}(G) \leq 3\mathrm{w}(G) + 1.$$

We also prove that the algorithm from Theorem 1 is best possible up to an additive constant of 2, i.e., there exists an infinite class of graphs with grid partitions such that each graph G in the class has a vertex h such that any monotone connected search strategy with homebase h uses at least $3\mathrm{w}(G) - 1$ searchers.

In Sect. 2 we formulate our algorithm. In Sect. 3 we give a formal analysis of the algorithm and a proof of Theorem 1. Then, in Sect. 4 we prove the lower bound. We finish with some conclusions in Sect. 5.

1.5 Related work

The problem of connected searching of a graph has been first studied in [4]. One of the interesting topics regarding this model of graph searching is the *price of connectivity*, that is, the ratio of the connected search number and the classical search number. For results and discussions on this issue see, e.g., [3,4,7,17,37]. For surveys on connected graph searching we refer the reader to [1,25].

In [17], it is proved that there exists an algorithm that converts any (monotone) search strategy using k searchers for a graph G into a (monotone) connected search strategy using $2k + 3$ searchers for G. This algorithm works on an auxiliary graph constructed on the basis of G which shares some properties with our weighted graphs with grid partitions. This relationship allows the key algorithmic idea used in the algorithm presented in this work to obtain results from [17]. However, this is a one-way implication since the algorithm in [17] is a strictly offline one.

In this work we discuss the generalization of the searching problem, where the graph to be searched has weights assigned to vertices. The version if the problem in which weights are assigned both to vertices and edges has been first studied in [4]. (In such case clearing an edge e requires that at least $\omega(e)$ searchers simultaneously slide along e, where $\omega(e)$ is the weight of e). As shortly discussed in [15], if edge weights are present, then in case of trees a 'reduction' to node-weighted graphs that 'preserves' the problem is possible for monotone connected searching. For algorithmic and complexity results on weighted graph searching see [15,16,36].

An 'intermediate' setting between offline algorithms and fully distributed solutions with no prior knowledge of the graph is a setting in which the searchers know in advance the structure of the graph, but they operate 'locally'—see, e.g., [22–24,42].

There exists a distributed algorithm (in which no prior knowledge of the graph is assumed) that finds a connected search strategy (using the minimum number of searchers) of any graph and arbitrary homebase; however, the algorithm is not monotone and its cost (the number of moves) is exponential in the size of the graph [8]. If one requires a distributed search strategy to be monotone, then it is known that the competitive ratio (the ratio of the number of searchers it uses in the worst case and the optimal offline number of searchers needed) of such a strategy is $\omega(n/\log n)$ [31]. The above worst case can occur even when the graph to be searched is a tree: an online strategy may use $\Theta(n)$ searchers while $O(\log n)$ searchers are enough [35]. Those results suggest a natural question about additional information provided to the searchers that would allow them to operate more efficiently either in terms of time (i.e., cost) or team size. Authors in [38] determine the (asymptotically) minimum number of bits of advice that needs to be provided when one requires a distributed searching algorithm to use the minimum number of searchers and operate in a monotone fashion.

Finally, we refer the reader interested in more practical aspects of connected graph searching (including distributed computations) to works on algorithms and applications in the field of robotics [29,30,33,34,41].

2 The algorithm

In this section we formally describe our distributed algorithm that the searchers use to obtain a monotone connected search strategy. This algorithm is divided into several subroutines; for each of them we give its pseudocode and an intuitive description. The section is finished with an example of the execution of the algorithm.

We start by introducing some additional notions. For a subgraph H of G and $d \in \{-1, 0, 1\}$ representing one of the three possible directions, let $\Gamma_d(H, v)$ be defined as follows

$$\Gamma_d(H, v) = \{(v, p) \mid p \in P(v), \{v, g(v, p)\} \notin E(H) \text{ and } \phi(v, p) = d\}.$$

Informally speaking, if H is the currently explored subgraph of G at some point of a search strategy, then $\Gamma_d(H, v)$ provides the set of unexplored ports at vertex v for which an oracle 'detects' that they lead in direction d. For a set $U \subseteq V$, we also define

$$\Gamma_d(H, U) = \bigcup_{v \in U} \Gamma_d(H, v).$$

Note that any vertex v such that $\Gamma_d(H, v) = \emptyset$ for all $d \in \{-1, 0, 1\}$ is clear and does not have to be guarded.

Our search strategy highly depends on careful maintenance of a border B, that is, the set of all vertices that have to be guarded in order to prevent recontamination. More formally, for the currently explored subgraph H of G, *the border B of H* is defined as the set of all vertices $v \in V(H)$ for which $\bigcup_{d \in \{-1,0,1\}} \Gamma_d(H, v) \neq \emptyset$. For precise control of border's extent we use its extremities that are closely related to a grid partition $(V_1, \ldots, V_t)$ of a graph. Namely, for a nonempty subset $U \subseteq V$,

$$l(U) = \min\{i \mid V_i \cap U \neq \emptyset, i \in \{1, \ldots, t\}\}$$

and

$$r(U) = \max\{i \mid V_i \cap U \neq \emptyset, i \in \{1, \ldots, t\}\}$$

are called the *left extremity* and the *right extremity* of U, respectively. We also denote $r(\emptyset) = 0$ and $l(\emptyset) = t + 1$.

When searching a graph, the algorithm works in a sequence of stages. Within each stage one can distinguish the two phases: an *expansion phase* and *border maintenance phase*. In the first phase the algorithm explores new vertices and edges of the graph. Then in the second phase it classifies vertices of the current border and all newly explored vertices as those that have to be guarded (either remain or must be added to the border) and those that have just became clear, and as we prove later, by monotonicity of the strategy, will always remain clear.

Now, we describe several short procedures that are the building blocks of the algorithm. Although, we do not describe these procedures from the perspective of a searcher, they can be used by each searcher to decide its individual actions. All variables used by the procedures are assumed to be global.

For the description of procedures of the expansion phase recall that in Sect. 1.3 we assumed a global communication model for the searchers. Consequently, at any moment each searcher knows the currently explored subgraph and the procedure is able to determine the state of each searcher to be either *free* or *guarding*. Those states are determined as follows: if x searchers are present on a vertex v that belongs to the current border, then the searchers choose arbitrarily $\omega(v)$ of them to be guarding (the selection can be made using their unique identifiers), while $x - \omega(v)$ remaining searchers at v, if any, are free. As we prove later, our search strategy guarantees that at any moment we have $x \geq \omega(v)$ for any guarded vertex v. Naturally, any searcher that is present at a vertex that does not have to be guarded, i.e., a vertex whose all incident edges are clear, is free. In contrast to the guarding searchers that cannot move until they become free, a searcher that is free can move to an arbitrarily selected vertex of the currently explored graph. We use this property to clear subsequent edges. Namely, before our algorithm performs a clearing move, all free searchers are gathered at an appropriate vertex and then the move is performed by *all* free searchers. In what follows, for simplicity, we skip the processing of states of the searchers in the pseudocodes of our procedures.

Procedure Expand realizes the most basic step of our algorithm, i.e., it clears a single edge of a graph by sliding all free searchers along that edge. The edge to be cleared is determined based on the input parameters v and p, where v is a vertex of the currently explored subgraph and $p \in P(v)$. The cleared edge is $\{v, g(v, p)\}$, i.e., the one 'outgoing' from v via port p.

Procedure Expand(v, p)

gather all free searchers at v
slide all free searchers from v to $g(v, p)$ via port p
$V(H) \leftarrow V(H) \cup \{g(v, p)\}$
$E(H) \leftarrow E(H) \cup \{\{v, g(v, p)\}\}$
$X_V \leftarrow X_V \cup \{g(v, p)\}$

As we will see later, it is possible that $g(v, p)$ is already occupied by searchers. However, since in general the weight of $g(v, p)$ is unknown, all free searchers slide from v to $g(v, p)$ to clear the edge $\{v, g(v, p)\}$. The searchers behave in such a way because if there were less than $\omega(g(v, p))$ searchers on $g(v, p)$ and there is a contaminated edge, different than $\{v, g(v, p)\}$, incident to $g(v, p)$ at the end of the sliding move, then $\{v, g(v, p)\}$, and possibly other edges, would become recontaminated. In the sequel, we prove that whenever $3\mathrm{w}(G) + 1$ searchers are available, then at least $\omega(g(v, p))$ searchers are present on $g(v, p)$ at the end of such move and hence no recontamination occurs. The procedure also updates the structure of the currently explored graph H. The variable X_V 'collects' vertices reached by the searchers performing the clearing moves in actual expansion phase, and it is also used by other procedures of our algorithm.

Procedure Expand is extensively used by the two other procedures of the expansion stage, i.e., procedures ExpandLaterally and ExpandStraight. The first one selects a set V_i and a direction $d \in \{-1, 1\}$, and then executes procedure Expand for each vertex $v \in V_i$ with an unexplored port p leading to a vertex in V_{i+d}. When ExpandLaterally finishes

its work, the set X_V contains all vertices that were reached during recent expansions. Then, procedure ExpandStraight uses X_V to perform searching moves entirely within V_{i+d}. Now, let us describe both procedures in a more detailed manner.

Procedure ExpandLaterally is strongly dependent on the two vertex sets L and R called *left* and *right borders*. Intuitively, the procedure extends L (respectively, R) by adding vertices in V_i reached for the first time by searchers coming from V_{i+1} (respectively, V_{i-1}). As we prove later, it always holds that $r(L) < l(R)$ (and thus L and R are disjoint) and these sets form a partition of the border B of the currently explored subgraph, i.e., $B = L \cup R$. First, the procedure computes a set U of pairs (v, p) such that v is a vertex of the border and $p \in P(v)$ is an unexplored port at v. Then, procedure Expand(v, p) is called for each $(v, p) \in U$. Note that the computation of the set U may require the knowledge of Γ_d, for each $d \in \{-1, 1\}$ which is computed based on the currently explored subgraph and the values of $\phi(v, p)$ provided to all ports p of each vertex v of the graph.

Procedure ExpandLaterally

```
if ω(L) ≥ ω(R) then
    if Γ+1(H, V_r(L) ∩ L) = ∅ then
        right ← false
        U ← Γ−1(H, V_r(L) ∩ L)
    else
        right ← true
        U ← Γ+1(H, V_r(L) ∩ L)
else
    if Γ−1(H, V_l(R) ∩ R) = ∅ then
        right ← true
        U ← Γ+1(H, V_l(R) ∩ R)
    else
        right ← false
        U ← Γ−1(H, V_l(R) ∩ R)
for all (v, p) in U do
    Expand(v, p)
```

In order to simplify the description we shortly say that procedure ExpandLaterally *expands from i to $i + d$*, where $d \in \{-1, 1\}$, when for each $(v, p) \in U$ it holds that $v \in V_i$ and $g(v, p) \in V_{i+d}$. Since the cases when $\omega(L) \geq \omega(R)$ and $\omega(L) < \omega(R)$ are symmetric, we describe here only the former one. Namely, procedure ExpandLaterally first tries to expand from $r(L)$ to $r(L) + 1$, if this is possible, that is, if there exists at least one pair (v, p) such that $v \in V_{r(L)} \cap L$ and the port p at vertex v is unexplored and leads from v to a vertex in $V_{r(L)+1}$. Otherwise, the procedure expands from $r(L)$ to $r(L)-1$. It follows from our analysis that one of those two actions is always possible. Depending on the direction of expansion, the procedure sets the Boolean variable $right$ that is further used by procedure UpdateBorders described later.

Procedure ExpandStraight is executed immediately after ExpandLaterally, except for a single execution in the initialization stage of the main procedure. Its main purpose is to clear the edges that correspond to unexplored ports in direction $d = 0$ at vertices in the set X_V. Note that the set X_V is formed either by the recent execution of procedure ExpandLaterally or it contains only the homebase h. Also note that subsequent calls of Expand in the 'while' loop can add new vertices to X_V (recall that X_V is a global variable) and hence ExpandStraight clears *all edges* between the vertices that are reached by the searchers (added to X_V) during the actual expansion phase. Naturally, $d = 0$ implies that the edges cleared by the procedure have both endvertices in the same set V_i for some $i \in \{1, \ldots, t\}$.

Procedure ExpandStraight

```
while Γ0(H, X_V) ≠ ∅ do
    select an arbitrary (v, p) from Γ0(H, X_V)
    Expand(v, p)
```

It is also worth pointing out that only those vertices in X_V that were explored in the current expansion phase may have unexplored ports in direction 0. If ExpandLaterally added to X_V a vertex u that has already been explored (we have already discussed this possibility in the description of procedure Expand), then its ports in direction 0, if any, have already been explored by an earlier execution of ExpandStraight; the one that took place just after the first exploration of the vertex u.

The border maintenance phase is realized by procedure UpdateBorders.

Procedure UpdateBorders is used to update the contents of variables L and R, that represent the left and the right borders, respectively. Depending on the value of variable $right$, which is set by ExpandLaterally, the procedure starts by adding all vertices in X_V either to the left or to the right border. Note that X_V may contain vertices whose all incident edges have been cleared in the expansion phase. Therefore, the 'for' loop aims at removing from L and R all vertices that do not have to be guarded in the currently explored subgraph.

Procedure UpdateBorders

```
if right = false then
    L ← L ∪ X_V
else
    R ← R ∪ X_V
for all v in L ∪ R do
    if Γ−1(H, v) ∪ Γ+1(H, v) = ∅ then
        L ← L \ {v}
        R ← R \ {v}
```

Finally, we give the details of our main procedure ConnectedSearching.

Procedure ConnectedSearching (CS for short) is the main procedure of our algorithm. The search strategy constructed by procedure CS can be partitioned into several *stages*. For $i \geq 1$ the ith stage consists of all steps that took place during the ith iteration of the 'while' loop of the procedure or equivalently, the steps necessary to realize the expansion and border maintenance phase. Additionally, we distinguish the 0th stage, called the *initialization stage*, consisting of all steps performed before the 'while' loop. Their purpose is to properly initialize the variables, e.g., a vertex $h \in V_i$, $i \in \{1, \ldots, t\}$, being the homebase of the search strategy to be computed is assigned to the set X_V. Since the homebase may have neither left nor right ports, CS has to execute procedure ExpandStraight and then properly set the current border using UpdateBorders. Setting the flag *right* to be `false` results in adding the vertices in X_V to the left border L during the first execution of UpdateBorders. Also note that each stage starts with an initialization of the set X_V, and that the 'while' loop of procedure CS executes as long as there exists at least one unexplored port in left or right direction, which is equivalent to the border of the currently explored subgraph being nonempty.

Procedure ConnectedSearching – CS

Require: $3\mathrm{w}(G)+1$ searchers are initially present on the homebase h
Ensure: A monotone connected search strategy for G
  $L \leftarrow \emptyset$ and $R \leftarrow \emptyset$
  $X_V \leftarrow \{h\}$
  ExpandStraight
  right $\leftarrow$ `false`
  UpdateBorders
  while $\Gamma_{-1}(H, V(H)) \cup \Gamma_{+1}(H, V(H)) \neq \emptyset$ **do**
    $X_V \leftarrow \emptyset$
    ExpandLaterally
    ExpandStraight
    UpdateBorders

In what follows we introduce some notation that is necessary for the presentation of an example and the analysis of the algorithm in Sect. 3.

The graph that is explored at the end of the kth stage is denoted by G_k, $k \geq 0$. To denote the left and right borders obtained at the end of the kth stage we use L_k, R_k, respectively, while B_k denotes the border of G_k. The kth stage, $k > 0$, is called:

(i) an LL-*expansion* if $\omega(L_{k-1}) \geq \omega(R_{k-1})$ and $\Gamma_{+1}(G_{k-1}, V_{r(L_{k-1})} \cap L_{k-1}) = \emptyset$,
(ii) an LR-*expansion* if $\omega(L_{k-1}) \geq \omega(R_{k-1})$ and $\Gamma_{+1}(G_{k-1}, V_{r(L_{k-1})} \cap L_{k-1}) \neq \emptyset$,
(iii) an RR-*expansion* if $\omega(L_{k-1}) < \omega(R_{k-1})$ and $\Gamma_{-1}(G_{k-1}, V_{l(R_{k-1})} \cap R_{k-1}) = \emptyset$
(iv) an RL-*expansion* if $\omega(L_{k-1}) < \omega(R_{k-1})$ and $\Gamma_{-1}(G_{k-1}, V_{l(R_{k-1})} \cap R_{k-1}) \neq \emptyset$.

Note that all conditions in the above definition directly correspond to those checked by ExpandLaterally, and that the first letter in the name of an expansion says which border (left or right) is tested for an existence of unexplored ports, while the second letter reflects the direction of an expansion. For example, an LL-expansion is a stage in which the searchers made sliding moves from *all* vertices in $V_{r(L_{k-1})} \cap L_{k-1}$ via ports leading to vertices in $V_{r(L_{k-1})-1}$, with the latter ones possibly added to the new left border L_k (as we prove later, for an LL-expansion, $r(L_k) < r(L_{k-1})$). On the other hand, if the stage is an LR-expansion, then the searchers successively explore the ports at vertices in $V_{r(L_{k-1})} \cap L_{k-1}$ but only those that lead to vertices in $V_{r(L_{k-1})+1}$, and the goal vertices that still have to be guarded in G_k are included in the new right border R_k. Note that LL-expansion and RR-expansion as well as LR-expansion and RL-expansion are symmetric. Therefore, in most proofs in Sect. 3 it is enough to focus on LL- and LR-expansions, and use the symmetry to obtain a general assertion for all types of expansions.

We finish this section with an example that demonstrates the execution of our algorithm.

Example 1 Figure 2 shows the explored subgraph together with unexplored edges outgoing from the guarded vertices at the end of each stage of a search strategy computed by procedure CS. All vertices are assumed to have the same weight equal to 1. Figure 2a depicts the explored subgraph at the end of the initialization stage. Note that at that point it holds that $\omega(L_0) = 2$, $R_0 = \emptyset$ and hence in the 1-st stage ExpandLaterally will test the left border for the existence of unexplored ports. Since there is a vertex in L_0 with an unexplored port leading to the right (see Fig. 2a), the first stage is an LR-expansion that results in $\omega(R_1) = 1$ (see Fig. 2b). Also, observe that in the 1-st stage no ports are explored by ExpandStraight. Clearly, since $\omega(L_1) \geq \omega(R_1)$, in the 2-nd stage ports at vertices in L_1 are tested and the two ports leading to the left are explored. This results in an LL-expansion depicted in Fig. 2c. Now, in the 3-rd stage, being an LL-expansion, a single port that leads to the left is explored, and then ExpandStraight reveals the second vertex. Finally observe that the 4th stage is an LR-expansion, while the two last stages are RR-expansions. Concerning the number of searchers, note that $\mathrm{w}(G) = 3$ and hence 10 searchers are initially placed on h; one can check that 5 searchers are sufficient for this particular search strategy.

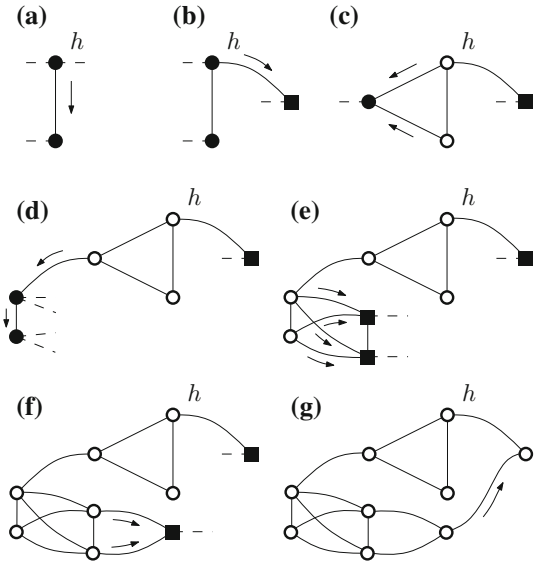

Fig. 2 **a–g** The explored subgraph at the end of stages 0–6; *dark circles* denote vertices in L, *dark squares* denote vertices in R and *empty circles* denote clear unguarded vertices

3 Analysis of the algorithm

In this section we prove an upper bound on the number of searchers required by our algorithm, and that the algorithm produces a monotone connected search strategy. We introduce the concept of a successful move. A move performed by searchers is called *successful* if it does not lead to recontamination. We will also say that a stage of the search strategy computed by procedure CS is *successful* if each move in this stage is successful.

Lemma 1 *If each move of the searchers is successful, then procedure CS produces a monotone connected search strategy for a graph G.*

Proof We first prove that the execution of procedure CS always stops. This in particular proves that the number of moves in the strategy produced by procedure CS is finite.

The execution of procedure ExpandStraight always stops because if $(v, p) \in \Gamma_0(G, X_V)$, then $\{v, g(v, p)\}$ is a contaminated edge. By assumption, this edge is cleared by procedure Expand(v, p) and no recontamination occurs, and hence the execution of the loop in procedure ExpandStraight finishes. By similar arguments, the execution of procedure ExpandLaterally always stops.

It remains to argue that procedure CS clears G. To that end we prove that if G is not entirely cleared at the end of the kth stage, then at least one edge becomes clear in the $(k+1)$-st stage. Note that this is sufficient, since no edge becomes recontaminated and, by assumption, each move is successful. The former follows from the fact that each vertex that belongs to a border is guarded (see procedures Expand and UpdateBorders). Therefore, after finite number of stages all edges of G are clear.

Due to the execution of procedure UpdateBorders at the end of kth stage, we have that if $u \in B_k$, then there exists at least one contaminated edge incident to u. Note that the cases when $\omega(L_k) \geq \omega(R_k)$ and $\omega(L_k) < \omega(R_k)$ are analogous and hence we consider only the former one. Since G is connected, $B_k \neq \emptyset$ and hence $L_k \neq \emptyset$. Therefore, there exists at least one contaminated edge $\{u, v\}$ such that one of its endvertices, say u, belongs to $V_{r(L_k)} \cap L_k$. Let $d = \phi(u, \rho(u, v))$. Note that $d \neq 0$ because otherwise $\{u, v\}$ would be cleared during the execution of procedure ExpandStraight in the same iteration in which u has been reached by searchers. Moreover, $(u, \rho(u, v)) \in \Gamma_d(G_k, u)$ and hence $\Gamma_d(G_k, V_{r(L_k)} \cap L_k)$ is nonempty. Since $d \in \{-1, 1\}$, we obtain

$$\Gamma_{-1}(G_k, V_{r(L_k)} \cap L_k) \cup \Gamma_1(G_k, V_{r(L_k)} \cap L_k) \neq \emptyset,$$

which implies that $U \neq \emptyset$. Thus, there exists a contaminated edge $\{u', v'\}$ such that $u' \in B_k$ and $(u', \rho(u', v')) \in U$. Therefore, by assumption and by the formulation of procedure Expand, the edge $\{u', v'\}$ becomes clear during the execution of the 'for all' loop in procedure ExpandLaterally. □

In the next two lemmas we prove several properties of graph G_k and its borders. This leads to Lemma 4 that bounds the size of B_k. This finally lets us use induction to prove Theorem 1. Recall that we focus on LL- and LR-expansions, since the results for other types of expansions follow by symmetry.

We continue our analysis with an assumption that each move performed by the searchers is successful, which holds provided that enough searchers are initially present at the homebase. This assumption is justified in the inductive proof of Theorem 1 given at the end of this section. Informally speaking, one could rephrase our algorithm as one that 'calls' for new searchers if required to avoid recontamination in the next move and then one could bound the number of searchers called.

Let $m \geq 1$ denote the number of iterations of the 'while' loop of procedure CS. We denote by C_k the set of all vertices cleared by searchers till the end of the kth stage, $k \in \{0, \ldots, m\}$. Note that, if each move is successful, then the explored vertices and cleared vertices are the same, i.e., $V(G_k) = C_k$ for each $k \in \{0, \ldots, m\}$. For this reason, in what follows we use C_k in place of $V(G_k)$.

We start with simple observations regarding the behavior of procedure CS. Informally speaking, the equality in Lemma 2(a) implies that the border of G_k is indeed 'protected' by searchers at the end of the kth stage. Lemma 2(b) says that, in case when the $(k+1)$-st stage is an LL-expansion, the borders do not change except within $V_{r(L_k)}$ and $V_{r(L_k)-1}$ and the clear (explored) part itself changes only within

$V_{r(L_k)-1}$. An analogous characterization for LR-expansions is given by Lemma 2(c).

Lemma 2 *Let $k \in \{0, \ldots, m-1\}$ and $j = r(L_k)$. Then,*

(a) $B_{k+1} = L_{k+1} \cup R_{k+1}$.
If $(k+1)$-st stage is an LL-expansion, then
(b) $L_{k+1} \setminus L_k \subseteq C_{k+1} \setminus C_k \subseteq V_{j-1}$, $R_{k+1} \subseteq R_k$ *and* $V_i \cap B_{k+1} = V_i \cap B_k$ *for each* $i \notin \{j, j-1\}$.
If $(k+1)$-st stage is an LR-expansion, then
(c) $R_{k+1} \setminus R_k \subseteq C_{k+1} \setminus C_k \subseteq V_{j+1}$, $L_{k+1} \subseteq L_k$ *and* $V_i \cap B_{k+1} = V_i \cap B_k$ *for each* $i \notin \{j, j+1\}$.

Proof Note that all vertices explored in the $(k+1)$-st stage are added to X_V and hence

$$C_{k+1} \setminus C_k \subseteq X_V. \tag{1}$$

(a) Recall that both borders are updated by procedure UpdateBorders. First, either L_k or R_k is extended with all vertices explored during an expansion phase. Then, the vertices that do not have to be guarded (to be skipped in B_{k+1}) are removed from the left and right borders. Thus, $B_{k+1} = L_{k+1} \cup R_{k+1}$ follows from a simple inductive argument on the number of stages and from $R_0 = \emptyset$ and $L_0 = B_0$. The latter is enforced by the execution of procedures ExpandStraight and UpdateBorders in the initialization stage of procedure CS.

(b) Since ExpandLaterally sets the variable *right* to `false` in an LL-expansion, procedure UpdateBorders will not add any vertices to the right border and hence $R_{k+1} \subseteq R_k$. Moreover, $X_V \subseteq V_{r(L_k)-1}$ which, by (1), gives $C_{k+1} \setminus C_k \subseteq V_{j-1}$. Also, the border vertices in X_V, i.e., the vertices in $B_{k+1} \cap X_V$, are added to the left border by procedure UpdateBorders in an LL-expansion. The latter in particular implies $L_{k+1} \setminus L_k \subseteq C_{k+1} \setminus C_k$. For $V_i \cap B_{k+1} = V_i \cap B_k$ for each $i \notin \{j, j-1\}$, observe that the only cleared edges are the ones with one endpoint in V_j and the other endpoint in V_{j-1} or both endpoints in V_{j-1}.

(c) Since ExpandLaterally sets variable *right* to `true`, no vertex will be added to the left border by procedure UpdateBorders and hence $L_{k+1} \subseteq L_k$. Similarly as in (b), $R_{k+1} \setminus R_k \subseteq C_{k+1} \setminus C_k \subseteq V_{j+1}$ follows from (1) and the facts that $X_V \subseteq V_{r(L_k)+1}$ and the border vertices in X_V are added to the right border by procedure UpdateBorders in an LR-expansion. Also, the edges cleared in $(k+1)$-st stage are the ones with one endpoint in V_j and the other endpoint in V_{j+1} or both endpoints in V_{j+1}, which proves the claim. □

See Fig. 3 for an example showing that $L_{k+1} \setminus L_k$ may be a proper subset of $C_{k+1} \setminus C_k$ in an LL-expansion. However, as we prove in the next lemma, $R_{k+1} = R_k$ always holds for LL-expansions, which strengthens part of Lemma 2(b). As to the claim in Lemma 2(c), see Fig. 4 for a case when $R_{k+1} \setminus R_k$ is a proper subset of $C_{k+1} \setminus C_k$ and L_{k+1} is a proper subset of L_k.

In the next lemma we analyze the behavior of borders' extremities, which depends on the type of expansion.

Lemma 3 *Let $k \in \{0, \ldots, m-1\}$. Then,*

(a) $r(L_k) < l(R_k)$ *if* $L_k \neq \emptyset$ *and* $R_k \neq \emptyset$.
If the $(k+1)$-st stage is an LL-expansion, then
(b) $r(L_{k+1}) < r(L_k)$, *if* $L_{k+1} \neq \emptyset$,
(c) $l(L_{k+1}) = l(L_k)$, *if L_{k+1} is not contained in a single set V_i and* $L_{k+1} \neq \emptyset$,
(d) $R_{k+1} = R_k$.
If the $(k+1)$-st stage is an LR-expansion, and $R_{k+1} \neq \emptyset$, *then*
(e) $r(L_k) < l(R_{k+1})$.

Proof The proof is by induction on k. Assume that (a)–(e) hold for some $k-1 \in \{0, \ldots, m-1\}$ and we prove them for k by analyzing the kth stage. Note that in case of LL- or LR-expansions it holds that $L_k \neq \emptyset$.

(a) If the kth stage is an LL-expansion, then we use induction hypothesis (a) and (b), and inclusion $R_k \subseteq R_{k-1}$ of Lemma 2(b), to obtain

$$r(L_k) \overset{(b)}{<} r(L_{k-1}) \overset{(a)}{<} l(R_{k-1}) \leq l(R_k).$$

In case of LR-expansion, by Lemma 2(c), $L_k \subseteq L_{k-1}$. This, and induction hypothesis (e) give

$$r(L_k) \leq r(L_{k-1}) < l(R_k)$$

This completes the proof of (a).

(b) By Lemma 2(b), $L_{k+1} \setminus L_k \subseteq V_{r(L_k)-1}$ and hence $r(L_{k+1}) \leq r(L_k)$. Moreover, an LL-expansion takes place only when no vertex in $V_{r(L_k)} \cap L_k$ has an unexplored right port in G_k. Hence, no vertex in $V_{r(L_k)} \cap L_{k+1}$ has an unexplored port in G_{k+1} and consequently $V_{r(L_k)} \cap L_{k+1} = \emptyset$. This implies that $r(L_{k+1}) < r(L_k)$ (see Fig. 3).

(c) First, observe that $L_k \not\subseteq V_{r(L_k)}$, for otherwise, by Lemma 2(b) and induction hypothesis (b), L_{k+1} would be a subset of $V_{r(L_k)-1}$, violating our assumption. Thus, $l(L_k) \leq r(L_k) - 1$. By Lemma 2(b), $V_i \cap L_k = V_i \cap L_{k+1}$ for $i < r(L_k) - 1$, and $L_{k+1} \setminus L_k \subseteq V_{r(L_k)-1}$. Thus, (c) follows.

(d) By induction hypothesis (a), $r(L_k) < l(R_k)$ and hence, by Lemma 2(a), $V_i \cap B_k = V_i \cap R_k$ for each $i \geq l(R_k)$. By

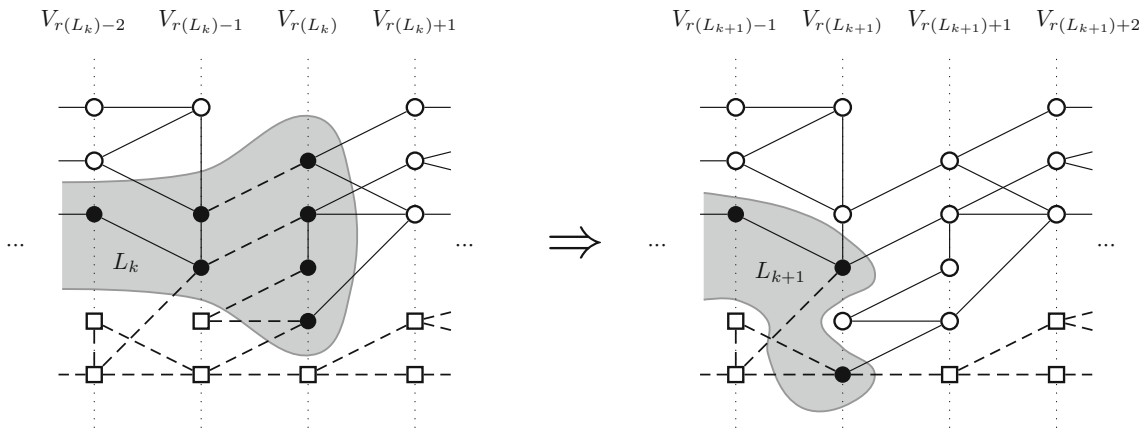

Fig. 3 A transition from L_k to L_{k+1} in an LL-expansion; the *dark vertices* are the ones in the left border, *white circles* denote clear unguarded vertices and the remaining ones are unexplored; *dashed edges* are contaminated

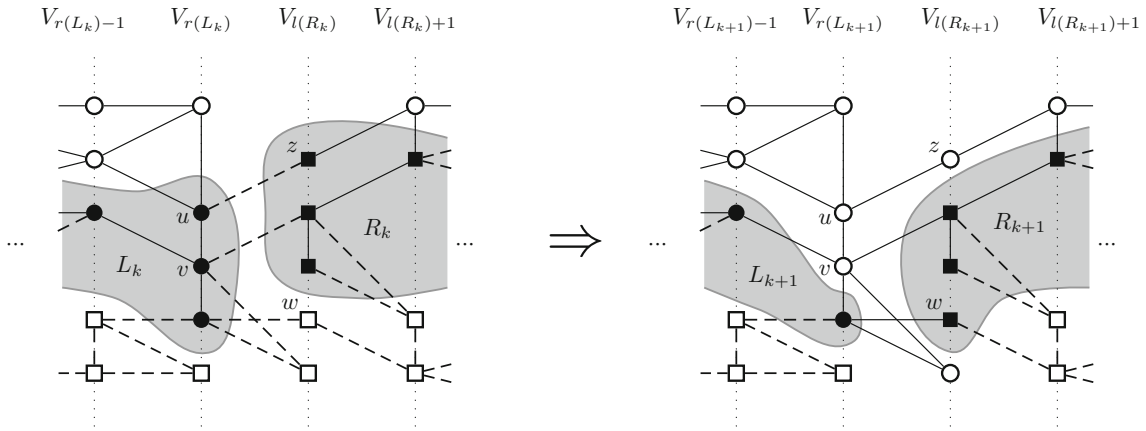

Fig. 4 A transition from L_k to L_{k+1} in an LR-expansion; the *dark circles* denote vertices in the left border, while *dark squares* denote vertices in the right border. *White circles* are clear unguarded vertices and the remaining ones are unexplored; *dashed edges* are contaminated

Lemma 2(b), $V_i \cap B_k = V_i \cap B_{k+1}$ for each $i > r(L_k)$, and $R_{k+1} \subseteq R_k$. Thus, $R_{k+1} = R_k$.

(e) Consider vertices that can be added to the right border in LR-expansion. According to Lemma 2(c), only some (possibly empty) subset of $V_{r(L_k)+1}$ can be added (see e.g., vertex w in Fig. 4) and hence $R_{k+1} \subseteq R_k \cup V_{r(L_k)+1}$. Note that procedure UpdateBorders can also remove vertices from R_k but only those in $R_k \cap V_{r(L_k)+1}$ (see e.g., vertex z in Fig. 4). Since $R_{k+1} \neq \emptyset$, we get $l(R_k) \leq l(R_{k+1})$ (also $r(R_k) = r(R_{k+1})$ holds). By induction hypothesis (a), $r(L_k) < l(R_k)$ and hence by the above inequality it follows that $r(L_k) < l(R_{k+1})$.

Though not necessary for the proof it is worth mentioning that some vertices may be also removed from L_k in an LR-expansion (see e.g., vertices u and v in Fig. 4). In such case, inclusion $L_{k+1} \subseteq L_k$ in Lemma 2(c) may be proper. □

Assume that $k \in \{0, \ldots, m-1\}$. Let $D_{k+1} = L_{k+1} \setminus L_k$ if the $(k+1)$-st stage is an LL-expansion and let $D_{k+1} = R_{k+1} \setminus R_k$ if the $(k+1)$-st stage is an LR-expansion. By Lemma 2, $D_{k+1} \subseteq V_{i'}$ for some $i' \in \{1, \ldots, t\}$, i.e., all vertices that become clear in the $(k+1)$-st stage are added to the same border and belong to a single set $V_{i'}$ of the grid partition, with $i' = r(L_k) - 1$ when LL-expansion took place and $i' = r(L_k) + 1$ for an LR-expansion. Note that Lemma 2 also implies that for every $i \neq i'$ it holds that $\omega(V_i \cap C_{k+1}) = \omega(V_i \cap C_k)$. In the sequel we use the following inequality that holds for all $i \in \{1, \ldots, t\}$:

$$\omega(V_i \cap C_{k+1}) \geq \omega(V_i \cap C_k) + \omega(D_{k+1} \cap V_i). \tag{2}$$

Lemma 4 *For each $k \in \{0, \ldots, m\}$, $\omega(B_k) \leq 2 \cdot \mathrm{w}(G)$.*

Proof In view of Lemma 3(a) we prove by induction on $k \in \{0, \ldots, m\}$ that the following conditions are satisfied:

(a) for every $i \in \{l(L_k), \ldots, r(L_k)\}$,

$$\omega(V_i \cap C_k) \geq \omega\big(L_k \cap (V_{l(L_k)} \cup \cdots \cup V_i)\big),$$

(b) for every $i \in \{l(R_k), \ldots, r(R_k)\}$,

$$\omega(V_i \cap C_k) \geq \omega\big(R_k \cap (V_i \cup \cdots \cup V_{r(R_k)})\big),$$

(c) for every $i \in \{r(L_k), \ldots, l(R_k)\}$,

$$\omega(V_i \cap C_k) \geq \min\big\{\omega(L_k), \omega(R_k)\big\}.$$

Observe that whenever L_k is empty, conditions (a) and (c) are clearly satisfied. Similarly, if $R_k = \emptyset$, then (b) and (c) hold.

For the base step it is enough to consider the execution of procedure ExpandStraight in the initialization of procedure CS. Let $V_s, s \in \{1, \ldots, t\}$, be the set of the grid partition that contains h. By the formulation of procedure ExpandStraight, $C_0 \subseteq V_s$. Since $R_0 = \emptyset$, conditions (b) and (c) follow. Moreover, $L_0 \subseteq V_s$, which implies (a).

For the induction step assume that conditions (a)–(c) are satisfied for some $k \in \{0, \ldots, m-1\}$.

Now we prove two claims; in the first one we consider an LL-expansion and in the second one we consider an LR-expansion. Then, as mentioned before, we conclude (see Claim 3) that (a)–(c) hold for $k+1$ for all types of expansions. The following, that is due to (2) and condition (a) for k, holds for each $i \leq r(L_k)$ when L_k is nonempty:

$$\begin{aligned}\omega(V_i \cap C_{k+1}) &\geq \omega\big(L_k \cap (V_{l(L_k)} \cup \cdots \cup V_i)\big) \\ &\quad + \omega\big((L_{k+1} \setminus L_k) \cap V_i\big) \\ &\geq \omega\big(L_k \cap (V_{l(L_k)} \cup \cdots \cup V_{i-1})\big) \\ &\quad + \omega\big(L_{k+1} \cap V_i\big). \end{aligned} \tag{3}$$

Before proving Claim 1 we give an intuition on the construction of its proof. In an LL-expansion, $R_{k+1} = R_k$ and hence condition (b) is obtained immediately from the induction hypothesis. For condition (a) we first observe that it follows when L_{k+1} is contained in a single set V_i. Otherwise, using Lemma 3 we argue that $\{l(L_{k+1}), \ldots, r(L_{k+1})\}$ is a subset of $\{l(L_k), \ldots, r(L_k)\}$, which with a proper use of the induction hypothesis (a) gives us (a) for $k+1$. Finally, condition (c) for $k+1$ is proven by observing that $\{r(L_{k+1}), \ldots, l(R_{k+1})\}$ is contained in the union of $\{r(L_k), \ldots, l(R_k)\}$ (in this case we use induction hypothesis (c)) and $\{l(L_k), \ldots, r(L_k)\}$ (in this case we use induction hypothesis (a)).

Claim 1 *If the $(k+1)$-st stage is an LL-expansion, then conditions* (a)–(c) *are satisfied for $k+1$.*

Proof By Lemma 3(d), $R_{k+1} = R_k$ and hence condition (b) for $k+1$ follows directly from the induction hypothesis. Since the $(k+1)$-st stage is an LL-expansion, $L_k \neq \emptyset$.

We now prove condition (a) for $k+1$. Recall that (a) trivially holds when L_{k+1} is empty. Hence, let $L_{k+1} \neq \emptyset$.

If $l(L_{k+1}) = r(L_{k+1})$, i.e., L_{k+1} is contained in a single set V_i, then (a) immediately follows for $k+1$.

Thus, let $l(L_{k+1}) \neq r(L_{k+1})$, i.e., $l(L_{k+1}) < r(L_{k+1})$. Let $i \in \{l(L_{k+1}), \ldots, r(L_{k+1})\}$ be selected arbitrarily. By Lemma 3(b) and 3(c), $i \in \{l(L_k), \ldots, r(L_k) - 1\}$. Hence, by (3), Lemma 2 and Lemma 3(a),

$$\omega(V_i \cap C_{k+1}) \geq \omega\big(L_{k+1} \cap (V_{l(L_{k+1})} \cup \cdots \cup V_i)\big)$$

as required by condition (a) for $k+1$.

It remains to prove condition (c) for $k+1$. Recall that if $L_{k+1} = \emptyset$ or $R_{k+1} = \emptyset$, then (c) holds for $k+1$. Hence, we assume that $L_{k+1} \neq \emptyset$ and $R_{k+1} \neq \emptyset$. Select an index $i \in \{r(L_{k+1}), \ldots, l(R_{k+1})\}$ arbitrarily. By Lemma 3(d), $R_{k+1} = R_k$. Thus, in particular, we have $R_k \neq \emptyset$ and $l(R_{k+1}) = l(R_k)$.

If $i \in \{r(L_k), \ldots, l(R_{k+1})\}$, then Lemma 2(b) implies $V_i \cap C_{k+1} = V_i \cap C_k$. This, induction hypothesis (c), the fact that $\omega(L_k) \geq \omega(R_k)$ in an LL-expansion and Lemma 3(d) (used in this order) give

$$\begin{aligned}\omega(V_i \cap C_{k+1}) = \omega(V_i \cap C_k) &\overset{(c)}{\geq} \min\{\omega(L_k), \omega(R_k)\} \\ &= \omega(R_k) = \omega(R_{k+1}). \end{aligned} \tag{4}$$

If $i \in \{r(L_{k+1}), \ldots, r(L_k) - 1\}$, then by Lemma 3(b) and 3(c), $i \in \{l(L_k), \ldots, r(L_k)\}$ and hence (3) holds. Then, by Lemmas 2(b), 3(a) and 3(c),

$$L_k \cap (V_{l(L_k)} \cup \cdots \cup V_{i-1}) = L_{k+1} \cap (V_{l(L_{k+1})} \cup \cdots \cup V_{i-1}).$$

Thus, for our choice of i, the right hand side of (3) equals $\omega(L_{k+1})$ and therefore

$$\omega(V_i \cap C_{k+1}) \geq \omega(L_{k+1}). \tag{5}$$

Then, from (4) and (5) it follows that condition (c) is satisfied for $k+1$. This completes the proof of Claim 1. □

Again, we precede the proof of Claim 2 with informal comments that provide some intuition. First, condition (a) for $k+1$ immediately follows from (a) for k, since $L_{k+1} \subseteq L_k$ in an LR-expansion. In order to prove (b) for $k+1$ we take any $i \in \{l(R_{k+1}), \ldots, r(R_{k+1})\}$ and we consider two subcases: if $i \leq l(R_k)$, then i is between $r(L_k)$ and $l(R_k)$ and induction hypothesis (c) helps us to obtain (b) for $k+1$; if $i > l(R_k)$, then i is between the 'extremities' of R_k and therefore (b) for k immediately gives (b) for $k+1$. The proof of (c) for $k+1$ is done by considering three subcases for $i \in \{r(L_{k+1}), \ldots, l(R_{k+1})\}$. The first subcase covers the situation in which i is between $r(L_{k+1})$ and $r(L_k)$. Note that this case is nontrivial as an LR-expansion may result in $r(L_{k+1}) < r(L_k)$. The second subcase deals with $i > l(R_k)$, where we use induction hypothesis (b) for k to obtain our claim. In the third subcase i is between $r(L_k)$ and $l(R_k)$ which allows us to use induction hypothesis (c) for k to prove (b) for $k+1$.

Claim 2 *If the* $(k+1)$*-st stage is an* LR*-expansion, then conditions* (a)–(c) *are satisfied for* $k+1$.

Proof We first prove condition (a) for $k+1$. The fact that the $(k+1)$-st stage is an LR-expansion implies $L_k \neq \emptyset$. Note that (a) trivially holds when $L_{k+1} = \emptyset$. Hence, let L_{k+1} be nonempty. By Lemma 2(c), $L_{k+1} \subseteq L_k$. The induction hypothesis (a) gives condition (a) for $k+1$.

Inequalities (b) and (c) are trivially satisfied when R_{k+1} is empty. Hence, assume in the remaining part of this proof that $R_{k+1} \neq \emptyset$. Also, since the $(k+1)$-st stage is an LR-expansion, $L_k \neq \emptyset$. For $i \in \{r(L_k), \ldots, l(R_k)\}$, by induction hypothesis (c) and by $\omega(L_k) \geq \omega(R_k)$ we obtain

$$\begin{aligned}\omega(V_i \cap C_k) &\geq \min\{\omega(L_k), \omega(R_k)\}\\ &= \omega(R_k)\\ &= \omega\big(R_k \cap (V_i \cup \cdots \cup V_{r(R_k)})\big). \end{aligned} \tag{6}$$

In order to prove (b) for $k+1$ take any index i in $\{l(R_{k+1}), \ldots, r(R_{k+1})\}$. Suppose first that $R_k = \emptyset$. Then, $R_{k+1} \subseteq V_{r(L_k)+1}$ by Lemma 2(c). By Lemma 3(a), $R_{k+1} \subseteq B_{k+1} \subseteq C_{k+1}$ which immediately gives

$$\omega\big(V_{r(L_k)+1} \cap C_{k+1}\big) \geq \omega\big(V_{r(L_k)+1} \cap R_{k+1}\big)$$

that proves (b) for $k+1$. Suppose now that $R_k \neq \emptyset$. Then, by Lemmas 2(c) and 3(a), $r(R_{k+1}) = r(R_k)$ and by Lemma 3(e), it holds that $l(R_{k+1}) > r(L_k)$. Therefore, if $i \in \{l(R_{k+1}), \ldots, l(R_k)\}$, then by Lemma 3(a), $D_{k+1} = R_{k+1} \backslash R_k$ in (2) and hence, by (2), (6) and Lemma 2(c), we obtain (b) for $k+1$ as follows:

$$\begin{aligned}\omega(V_i \cap C_{k+1}) &\geq \omega\big(R_k \cap (V_i \cup \cdots \cup V_{r(R_k)})\big)\\ &\quad + \omega\big((R_{k+1} \backslash R_k) \cap V_i\big)\\ &= \omega(R_k \cap V_i) + \omega\big(R_k \cap (V_{i+1} \cup \cdots\\ &\quad \cup V_{r(R_k)})\big)\\ &\quad + \omega\big((R_{k+1} \backslash R_k) \cap V_i\big)\\ &\geq \omega(R_{k+1} \cap V_i) + \omega\big(R_{k+1} \cap (V_{i+1} \cup \cdots\\ &\quad \cup V_{r(R_{k+1})})\big)\\ &= \omega\big(R_{k+1} \cap (V_i \cup \cdots \cup V_{r(R_{k+1})})\big). \end{aligned}$$

If $i \in \{l(R_k)+1, \ldots, r(R_{k+1})\}$, then by Lemma 3(a), induction hypothesis (b) and Lemma 2(c),

$$\begin{aligned}\omega(V_i \cap C_{k+1}) &= \omega(V_i \cap C_k)\\ &\geq \omega\big(R_k \cap (V_i \cup \cdots \cup V_{r(R_k)})\big)\\ &= \omega\big(R_{k+1} \cap (V_i \cup \cdots \cup V_{r(R_{k+1})})\big). \end{aligned}$$

This proves (b) for $k+1$.

Now we prove (c) for $k+1$. Recall that the condition follows when $L_{k+1} = \emptyset$. Hence, let L_{k+1} be nonempty. Let first $i \in \{r(L_{k+1}), \ldots, r(L_k)\}$. This set is nonempty because $L_{k+1} \subseteq L_k$ by Lemma 2(c). Moreover, $L_{k+1} \subseteq L_k$ and (3) imply

$$\omega(V_i \cap C_{k+1}) \geq \omega\big(L_{k+1} \cap (V_{l(L_{k+1})} \cup \cdots \cup V_i)\big).$$

This, together with $i \geq r(L_{k+1})$, gives

$$\omega(V_i \cap C_{k+1}) \geq \omega(L_{k+1}). \tag{7}$$

For $i > r(L_k)$ we consider two cases. In the first case let $R_{k+1} \backslash R_k \neq \emptyset$. This implies by Lemma 2(c) that $l(R_{k+1}) = r(L_k)+1$ and hence it is enough to take $i = r(L_k)+1$. Then, however, (c) for $k+1$ follows from (b) for $k+1$.

In the second case let $R_{k+1} \backslash R_k = \emptyset$. This implies that

$$R_{k+1} \subseteq R_k \text{ and } r(R_k) = r(R_{k+1}) \tag{8}$$

and

$$R_{k+1} = R_{k+1} \cap (V_i \cup \cdots \cup V_{r(R_{k+1})}) \tag{9}$$

for each $i \in \{r(L_k)+1, \ldots, l(R_{k+1})\}$. If $i \in \{l(R_k)+1, \ldots, l(R_{k+1})\}$, then the induction hypothesis (b) and (2) imply (note that $D_k = \emptyset$ in (2))

$$\omega(V_i \cap C_{k+1}) \geq \omega\big(R_k \cap (V_i \cup \cdots \cup V_{r(R_k)})\big). \tag{10}$$

If $i \in \{r(L_k)+1, \ldots, l(R_k)\}$, then (6) and (2) (where $D_k = \emptyset$) imply (10). Hence, for each $i \in \{r(L_k)+1, \ldots, l(R_{k+1})\}$ by (8), (9) and (10) we obtain

$$\begin{aligned}\omega(V_i \cap C_{k+1}) &\geq \omega\big(R_{k+1} \cap (V_i \cup \cdots \cup V_{r(R_{k+1})})\big)\\ &= \omega(R_{k+1}). \end{aligned} \tag{11}$$

Thus, from (7) and (11) it follows that (c) holds for $k+1$. This completes the proof of Claim 2. □

By symmetry, analogous arguments can be used to prove the following claim.

Claim 3 *If the* $(k+1)$*-st stage is an* RR*-expansion or an* RL*-expansion, then conditions* (a)–(c) *are satisfied for* $k+1$. □

Claims 1, 2 and 3 imply that conditions (a)–(c) hold for each $k \in \{0, \ldots, m\}$. Hence, using inequality (a) we can easily argue that the weight of the left border is always bounded by $\mathrm{w}(G)$. Indeed, for each $k \in \{0, \ldots, m\}$ we obtain

$$\begin{aligned}\omega(L_k) &= \omega\big(L_k \cap (V_{l(L_k)} \cup \cdots \cup V_{r(L_k)})\big)\\ &\overset{(a)}{\leq} \omega(V_{r(L_k)} \cap C_k) \leq \omega(V_{r(L_k)})\\ &\leq \mathrm{w}(G). \end{aligned}$$

Analogously, (b) implies $\omega(R_k) \leq \mathrm{w}(G)$ for each $k \in \{0, \ldots, m\}$. By Lemma 2(a), the proof is completed. □

Proof of Theorem 1 Due to Lemma 1, it is enough to argue that each move of the searchers is successful during execution of procedure CS.

We prove by induction on $k \in \{0, \ldots, m\}$ that the kth stage is successful. Note that the 0th stage (i.e., the initialization stage) is successful because, according to procedure ExpandStraight, all vertices reached by searchers belong to V_i, $i \in \{1, \ldots, t\}$, such that $h \in V_i$. Thus, in each move of the 0th stage, among $3\mathrm{w}(G) + 1$ available searchers at most $\mathrm{w}(G)$ of them are guarding and thus the number of free searchers reaching a new vertex is at least $2\mathrm{w}(G) + 1$, and hence no recontamination occurs.

Suppose now that kth stage is successful for some $k \in \{0, \ldots, m-1\}$ and we prove that the $(k+1)$-st stage is successful. Note that $V(G_{k+1})\backslash V(G_k)$ is the set of vertices reached for the first time in the $(k+1)$-st stage. (This set may be empty.) We have that

$$V(G_{k+1})\backslash V(G_k) \subseteq V_i, \tag{12}$$

where $i \in \{r(L_k) - 1, r(L_k) + 1, l(R_k) - 1, l(R_k) + 1\}$; see procedures ExpandLaterally and ExpandStraight. Since the first k stages are successful, the cleared subgraph equals the explored subgraph, $C_k = V(G_k)$. By Lemma 2(a) and Lemma 4,

$$\omega(B_k) = \omega(L_k \cup R_k) \leq 2 \cdot \mathrm{w}(G). \tag{13}$$

Thus, at the beginning of the $(k+1)$-st stage at most $2 \cdot \mathrm{w}(G)$ searchers are guarding the vertices in B_k.

Suppose that l clearing moves occur in the $(k+1)$-st stage, and let the rth of those moves slide some searcher along an edge $\{u_r, v_r\}$ from u_r to v_r, $r \in \{1, \ldots, l\}$.

For each $r \in \{1, \ldots, l\}$ we consider two cases: v_r has not been reached by a searcher prior to the move that we consider, or it has been reached before. In the former case $v_r \in V(G_{k+1})\backslash V(G_k)$ and hence the number of guarding searchers at *the beginning* of this rth move is, by (12) and (13), at most

$$\begin{aligned}\omega(B_k) + \omega(V(G_{k+1})\backslash V(G_k)) - \omega(v_r) \\ \leq 2 \cdot \mathrm{w}(G) + \omega(V_i) - \omega(v_r) \leq 3 \cdot \mathrm{w}(G) - \omega(v_r).\end{aligned}$$

Therefore, the number of searchers reaching v_r is at least $\omega(v_r)$. In the latter case, the number of guarding searchers at the beginning of the rth move is, again by (12) and (13), at most

$$\omega(B_k) + \omega(V(G_{k+1})\backslash V(G_k)) \leq 3 \cdot \mathrm{w}(G).$$

Thus, one free searcher is available and it slides from u_r to v_r, and v_r is already occupied by $\omega(v_r)$ guarding searchers. Therefore, by a simple inductive argument, in both cases the rth move is successful, and hence the $(k+1)$-st stage is successful as required. □

4 Lower bound

Let k and t be positive integers. Define a graph $G_{k,t}$ with grid partition $(V_1, \ldots, V_t)$ such that $V_i \cup V_{i+1}$ induces a complete subgraph in $G_{k,t}$ for each $i \in \{1, \ldots, t-1\}$ and $|V_i| = k$ for each $i \in \{1, \ldots, t-1\}$. In other words, $v \in V_i$ is adjacent to the vertices in $V_{i-1} \cup (V_i \backslash \{v\}) \cup V_{i+1}$ for each $i \in \{1, \ldots, t\}$, where we take $V_0 = V_{t+1} = \emptyset$. We take $\omega \equiv 1$ in our example.

We prove the following lemma.

Lemma 5 *Let k and $(V_1, \ldots, V_{6k-1})$ be any positive integer and the grid partition of $G_{k,6k-1}$, respectively. If $h \in V_{3k}$, then* $\mathrm{mcs}(G_{k,6k-1}, h) \geq 3k - 1$.

Proof Consider a monotone connected search strategy $\mathcal{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_l)$ of $G_{k,6k-1}$ that uses the minimum number of searchers. Take the minimum integer $j \in \{1, \ldots, l\}$ such that the set $X \subseteq V_1 \cup \cdots \cup V_{6k-1}$, consisting of vertices that have been reached by a searcher in one of the moves $\mathcal{S}_1, \ldots, \mathcal{S}_j$, is of size $3k - 1$. We argue that $|\mathcal{S}_j| \geq 3k - 1$.

Suppose for a contradiction that $|\mathcal{S}_j| < 3k-1$. Thus, there exists $v \in X$ such that v is not occupied by a searcher at the end of $\mathcal{S}_j$. Since $\mathcal{S}$ is monotone, all edges incident to v are clear at the end of $\mathcal{S}_j$. Moreover, $V_1 \cap X = \emptyset$ and $V_{6k-1} \cap X = \emptyset$ because $h \in V_{3k}$. Thus, $v \in V_i$ for some $i \in \{2, \ldots, 6k-2\}$ and therefore there are $3k-1$ edges incident to v in $G_{k,6k-1}$. This in particular means that $V_{i-1} \cup V_i \cup V_{i+1} \subseteq X$, which contradicts $|X| = 3k - 1$. □

Since each search strategy with homebase in V_{3k} in $G_{k,6k-1}$ (regardless if it is computed in a distributed or offline setting) must use at least $3k - 1$ searchers, and by construction $\mathrm{w}(G_{k,6k-1}) = k$, we obtain the following.

Theorem 2 *For each positive integer k there exists an infinite class $\mathcal{G}$ of graphs G with grid partitions for which $\mathrm{w}(G) = k$ and such that every $G \in \mathcal{G}$ has a vertex h for which $\mathrm{mcs}(G, h) \geq 3k - 1$.* □

5 Summary

5.1 Relations to connected path decompositions

In this section we discuss an application of the algorithm presented in this work to computation of connected path decompositions. For the definitions of (connected) pathwidth and (connected) path decomposition reader is referred e.g. to [17]. More precisely, our algorithm can be turned into a procedure that converts a given path decomposition of width k

into a connected one. The conversion ensures that the width of the resulting connected path decomposition is at most $2k+1$.

To state our claim more formally, we recall the following notion of *derived graph* used in [17].

Definition 1 Given a graph G and its path decomposition $\mathcal{P} = (X_1, \ldots, X_d)$, a node-weighted graph $F = (V, E, \omega)$ *derived* from G and $\mathcal{P}$ is the graph with vertex set

$$V = U_1 \cup \cdots \cup U_d,$$

where $U_i = \{v_i(H) \mid H$ is a connected component of the subgraph of G induced by $X_i\}$, $i \in \{1, \ldots, d\}$, and edge set

$$E = \{\{v_i(H), v_{i+1}(H')\} \mid V(H) \cap V(H') \neq \emptyset, \\ v_i(H) \in U_i, v_{i+1}(H') \in U_{i+1}, i \in \{1, \ldots, d-1\}\}.$$

The weight of a vertex $v_i(H) \in V$, $i \in \{1, \ldots, d\}$, is $\omega(v_i(H)) = |V(H)|$.

In other words, we construct U_i by taking the subgraph of G induced by the vertices in X_i and for each connected component H of this subgraph we add to U_i a vertex, denoted $v_i(H)$. In this construction, for the same subgraph H, we may have different vertices $v_i(H)$ and $v_{i'}(H)$ in different sets U_i and $U_{i'}$.

Suppose we are given a path decomposition $\mathcal{P}$ of a graph G. We first construct the derived graph F from G and $\mathcal{P}$. Note that it is easy to fix the port numbers in F so that $(U_1, \ldots, U_d)$ becomes its grid partition. Then, we slightly modify F by adding for each vertex $v \in U_i$ a new neighbor $v' \in U_{i+1}$ with $\omega(v') = 0$. Denote the new graph by F'. Note that $\mathrm{w}(F) = \mathrm{w}(F')$. Then, we select any vertex of F' that also belongs to F to be the homebase h, we place $3\mathrm{w}(F)+1$ searchers on h and we let the searchers execute procedure CS. According to Theorem 1, this leads to a monotone connected search strategy for F' with homebase h. Define $\mathcal{C} = (Z_0, \ldots, Z_m)$, where

$$Z_k = \bigcup_{v(H) \in B_k} V(H), \quad k \in \{0, \ldots, m\}.$$

It follows from our algorithm that $(V(F') \setminus V(F)) \cap B_k = \emptyset$ for each stage $k \in \{0, \ldots, m\}$. Thus, informally speaking, Z_k consists of vertices of G that 'correspond' to all vertices of F that belong to the border B_k. The reason for executing procedure CS on F' rather than on F is the property that each vertex of F belongs to B_k for some $k \in \{0, \ldots, m\}$. This property, by using similar arguments as in [17], allows us to conclude that $\mathcal{C}$ is a connected path decomposition of G. Thus, by Lemma 4, we obtain that for each graph G it holds:

$$\mathrm{cpw}(G) \leq 2\mathrm{pw}(G) + 1,$$

where $\mathrm{pw}(G)$ and $\mathrm{cpw}(G)$ denote the pathwidth and the connected pathwidth of G, respectively.

5.2 Computations with advice

Nisse and Soguet proved in [38] that the size of advice for monotone and connected distributed searching of a graph is $\Theta(n \log n)$, that is, $O(n \log n)$ bits are always enough and $\Omega(n \log n)$ are required for some graphs. To use the same terminology, we described in this work an algorithm that requires advice of size $O(|E|)$ for the input graph $G = (V, E)$ with a grid partition. The main advantage, however, of our approach is the structure of the advice we require. Namely, it is a strong assumption, especially in more practical situations, that an advice that requires a priori knowledge on the graph can be provided. Our advice, as suggested in Sect. 1, can be potentially available in some practical scenarios without any preprocessing of the graph. It is an interesting research direction on possible types of advice that could be practically useful in distributed agent algorithms.

Acknowledgments The authors would like to thank Dietmar Berwanger for an inspiring discussion on the subject considered in this paper.

References

1. Alspach, B.: Searching and sweeping graphs: a brief survey. Le Matematiche (Catania) **59**, 5–37 (2004)
2. Ambühl, C., Gąsieniec, L., Pelc, A., Radzik, T., Zhang, X.: Tree exploration with logarithmic memory. ACM Trans. Algorithms **7**(2), 17 (2011)
3. Barrière, L., Flocchini, P., Fomin, F.V., Fraigniaud, P., Nisse, N., Santoro, N., Thilikos, D.M.: Connected graph searching. Inf. Comput. **219**, 1–16 (2012)
4. Barrière, L., Flocchini, P., Fraigniaud, P., Santoro, N.: Capture of an intruder by mobile agents. In: SPAA'02: Proceedings of the Fourteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp. 200–209. ACM, New York, NY, USA (2002)
5. Barrière, L., Flocchini, P., Fraigniaud, P., Santoro, N.: Can we elect if we cannot compare? In: SPAA'03: Proceedings of the Fifteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp. 324–332 (2003)
6. Barrière, L., Flocchini, P., Fraigniaud, P., Santoro, N.: Rendezvous and election of mobile agents: impact of sense of direction. Theory Comput. Syst. **40**(2), 143–162 (2007)
7. Barrière, L., Fraigniaud, P., Santoro, N., Thilikos, D.M.: Searching is not jumping. In: WG '03: Proceedings of the 29th International Workshop on Graph-Theoretic Concepts in Computer Science, pp. 34–45 (2003)
8. Blin, L., Fraigniaud, P., Nisse, N., Vial, S.: Distributed chasing of network intruders. Theor. Comput. Sci. **399**(1–2), 12–37 (2008)
9. Breisch, R.L.: An intuitive approach to speleotopology. Southwest. Cavers **6**, 72–78 (1967)
10. Chalopin, J., Das, S., Disser, Y., Mihalák, M., Widmayer, P.: Simple agents learn to find their way: an introduction on mapping polygons. Discret. Appl. Math. **161**(10–11), 1287–1307 (2013)

11. Chalopin, J., Das, S., Kosowski, A.: Constructing a map of an anonymous graph: applications of universal sequences. In: OPODIS, pp. 119–134 (2010)
12. Cohen, R., Fraigniaud, P., Ilcinkas, D., Korman, A., Peleg, D.: Label-guided graph exploration by a finite automaton. ACM Trans. Algorithms **4**(4), 1–18 (2008)
13. Czyżowicz, J., Pelc, A., Labourel, A.: How to meet asynchronously (almost) everywhere. ACM Trans. Algorithms **8**(4), 37 (2012)
14. Das, S., Flocchini, P., Kutten, S., Nayak, A., Santoro, N.: Map construction of unknown graphs by multiple agents. Theor. Comput. Sci. **385**(1–3), 34–48 (2007)
15. Dereniowski, D.: Connected searching of weighted trees. Theor. Comput. Sci. **412**, 5700–5713 (2011)
16. Dereniowski, D.: Approximate search strategies for weighted trees. Theor. Comput. Sci. **463**, 96–113 (2012)
17. Dereniowski, D.: From pathwidth to connected pathwidth. SIAM J. Discret. Math. **26**(4), 1709–1732 (2012)
18. Dereniowski, D., Pelc, A.: Drawing maps with advice. J. Parallel Distrib. Comput. **72**(2), 132–143 (2012)
19. Dessmark, A., Fraigniaud, P., Kowalski, D.R., Pelc, A.: Deterministic rendezvous in graphs. Algorithmica **46**(1), 69–96 (2006)
20. Dessmark, A., Pelc, A.: Optimal graph exploration without good maps. Theor. Comput. Sci. **326**(1–3), 343–362 (2004)
21. Emek, Y., Fraigniaud, P., Korman, A., Rosén, A.: Online computation with advice. Theor. Comput. Sci. **412**(24), 2642–2656 (2011)
22. Flocchini, P., Huang, M.J., Luccio, F.L.: Decontaminating chordal rings and tori using mobile agents. Int. J. Found. Comput. Sci. **18**(3), 547–563 (2007)
23. Flocchini, P., Huang, M.J., Luccio, F.L.: Decontamination of hypercubes by mobile agents. Networks **52**(3), 167–178 (2008)
24. Flocchini, P., Santoro, N., Song, L.X.: On the complexity of decontaminating an hexagonal mesh network. In: ICCGI '07: Proceedings of the International Multi-Conference on Computing in the Global Information Technology, p. 21 (2007)
25. Fomin, F.V., Thilikos, D.M.: An annotated bibliography on guaranteed graph searching. Theor. Comput. Sci. **399**(3), 236–245 (2008)
26. Fraigniaud, P., Ilcinkas, D., Peer, G., Pelc, A., Peleg, D.: Graph exploration by a finite automaton. Theor. Comput. Sci. **345**(2–3), 331–344 (2005)
27. Fraigniaud, P., Ilcinkas, D., Pelc, A.: Communication algorithms with advice. J. Comput. Syst. Sci. **76**(3–4), 222–232 (2010)
28. Haddar, M.A., Hadj Kacem, A., Métivier, Y., Mosbah, M., Jmaiel, M.: Electing a leader in the local computation model using mobile agents. In: AICCSA'08: Proceedings of the 6th ACS/IEEE International Conference on Computer Systems and Applications, pp. 473–480 (2008)
29. Hollinger, G.A., Singh, S., Djugash, J., Kehagias, A.: Efficient multi-robot search for a moving target. Int. J. Robot. Res. **28**(2), 201–219 (2009)
30. Hollinger, G.A., Singh, S., Kehagias, A.: Improving the efficiency of clearing with multi-agent teams. Int. J. Robot. Res. **29**(8), 1088–1105 (2010)
31. Ilcinkas, D., Nisse, N., Soguet, D.: The cost of monotonicity in distributed graph searching. Distrib. Comput. **22**(2), 117–127 (2009)
32. Klasing, R., Kosowski, A., Navarra, A.: Taking advantage of symmetries: gathering of many asynchronous oblivious robots on a ring. Theor. Comput. Sci. **411**(34–36), 3235–3246 (2010)
33. Kleiner, A., Kolling, A., Lewis, M., Sycara, K.P.: Hierarchical visibility for guaranteed search in large-scale outdoor terrain. Auton. Agents Multi-Agent Syst. **26**(1), 1–36 (2013)
34. Kolling, A., Carpin, S.: Multi-robot pursuit-evasion without maps. In: ICRA'10: Proceedings of IEEE International Conference on Robotics and Automation, pp. 3045–3051 (2010)
35. Megiddo, N., Hakimi, S.L., Garey, M.R., Johnson, D.S., Papadimitriou, C.H.: The complexity of searching a graph. J. ACM **35**(1), 18–44 (1988)
36. Mihai, R., Todinca, I.: Pathwidth is NP-hard for weighted trees. In: FAW '09: Proceedings of the 3rd International Workshop on Frontiers in Algorithmics, pp. 181–195. Springer, Berlin, Heidelberg (2009)
37. Nisse, N.: Connected graph searching in chordal graphs. Discret. Appl. Math. **157**(12), 2603–2610 (2008)
38. Nisse, N., Soguet, D.: Graph searching with advice. Theor. Comput. Sci. **410**(14), 1307–1318 (2009)
39. Parsons, T.D.: Pursuit-Evasion in a Graph. In Theory and Applications of Graphs, Lecture Notes in Mathematics. Springer, Berlin (1978)
40. Pelc, A.: Deterministic rendezvous in networks: a comprehensive survey. Networks **59**(3), 331–347 (2012)
41. Sachs, S., LaValle, S.M., Rajko, S.: Visibility-based pursuit-evasion in an unknown planar environment. Int. J. Robot. Res. **23**(1), 3–26 (2004)
42. Shareghi, P., Imani, N., Sarbazi-Azad, H.: Capturing an intruder in the pyramid. In: CSR'06: Proceedings of the First International Computer Science Symposium in Russia, pp. 580–590 (2006)