Online Graph Exploration: New Results on Old and New Algorithms

Nicole Megow[1], Kurt Mehlhorn[1], and Pascal Schweitzer[2,⋆]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany.
{nmegow,mehlhorn}@mpi-inf.mpg.de
[2] The Australian National University, Canberra, Australia.
pascal.schweitzer@anu.edu.au

Abstract. We study the problem of exploring an unknown undirected connected graph. Beginning in some start vertex, a searcher must visit each node of the graph by traversing edges. Upon visiting a vertex for the first time, the searcher learns all incident edges and their respective traversal costs. The goal is to find a tour of minimum total cost. Kalyanasundaram and Pruhs [23] proposed a sophisticated generalization of a Depth First Search that is 16-competitive on planar graphs. While the algorithm is feasible on arbitrary graphs, the question whether it has constant competitive ratio in general has remained open. Our main result is an involved lower bound construction that answers this question negatively. On the positive side, we prove that the algorithm has constant competitive ratio on any class of graphs with bounded genus. Furthermore, we provide a constant competitive algorithm for general graphs with a bounded number of distinct weights.

1 Introduction

In an exploration problem an agent, or searcher, has to construct a complete map of an environment without any a priori knowledge of its topology. The searcher makes all its decisions based on partial local knowledge and gathers new information on its exploration tour. Exploration problems appear in various contexts, such as robot motion planning in hazardous or inaccessible terrain, maintaining security of large networks, and searching, indexing, and analyzing digital data in the internet [7, 20, 28].

We study the online graph exploration problem on undirected connected graphs $G = (V, E)$. We assume that the vertices are labeled so that the searcher is able to distinguish them. Each edge $e = (u, v) \in E$ has a non-negative real weight $|e|$, also called the length or the cost of the edge. Beginning in a distinguished start vertex $s \in V$, the searcher learns G according to the following online paradigm, also known as *fixed graph scenario* [23]: whenever the searcher visits a vertex, it learns all incident edges, their weights, and the labels of their end vertices. To explore a new vertex, the searcher traverses previously learned edges in the graph. For traversing an edge, the searcher has to pay the respective edge cost. The task is to find a tour that visits all vertices V and returns to the start vertex. The goal is to find a tour of minimum total length. An illustration of this model (see [23]) is the scenario where vertices correspond to cities and

⋆ Supported by the National Research Fund, Luxembourg, and cofunded under the Marie Curie Actions of the European Commission (FP7-COFUND).

upon arrival in a city the searcher sees the road signs of routes to other cities including distance information.

A standard technique to measure the quality of online algorithms is *competitive analysis* [9], which compares the outcome of an algorithm with an *optimal offline solution*. For our graph exploration problem, the corresponding offline problem is the fundamental *Traveling Salesman Problem* (TSP), one of the most studied optimization problems which is in general even NP-hard to approximate [21]. It asks for a shortest tour that visits every vertex of a graph known in advance. For a positive number c, we call an online exploration algorithm *c-competitive*, if it computes for any instance a tour of total length at most c times the optimal offline tour through all vertices. The *competitive ratio* of an algorithm is the infimum over all c such that it is c-competitive.

The greedy algorithm *Nearest Neighbor* (NN) is a simple and fast heuristic that has been studied intensively in the traditional offline TSP environment. It repeatedly chooses the next vertex to be visited as an unexplored vertex closest to the current location. The worst case ratio for this greedy algorithm, $\Theta(\log n)$ [29], also applies to our online scenario. It is tight even on planar unit-weight graphs, which follows from a nice and simple lower bound construction of particular graphical instances [22].

In case all edges have equal weight, a *Depth First Search* (DFS) is 2-competitive. It yields a total tour not larger than twice the size of a minimum spanning tree (MST), a lower bound on the optimal tour. This is optimal in the unit-weight case [25].

For general graphs with arbitrary weights no constant competitive algorithm is known. A promising candidate was introduced by Kalyanasundaram and Pruhs [23]. Their algorithm ShortCut is a sophisticated generalization of DFS obtained by introducing a parameterized *blocking condition* that determines when to diverge from DFS. They prove an upper bound of 16 on its competitive ratio in planar graphs. The algorithm itself is defined for general graphs. However, since its introduction almost two decades ago, it has been open if ShortCut has constant competitive ratio in general. There has been no progress on this question since then, and in fact, all subsequent results concerned with our graph exploration setting only apply to simple cycles: In [2], it is shown that NN yields a competitive ratio of $3/2$ on simple cycles. Additionally, a lower bound of $5/4$ for any deterministic online algorithm is proven. Both lower and upper bound are improved in [25]. There, the authors give a more sophisticated algorithm which takes additionally the current total tour length into account. They prove that, on simple cycles, this algorithm achieves the best possible competitive ratio of $1 + \sqrt{3}$. It is not clear how the algorithm can be generalized and applied to more complex graphs.

Our contribution. We revisit algorithm ShortCut proposed by Kalyanasundaram and Pruhs [23]. We elaborate on the sophistication of the underlying idea, but report also a precarious issue in the given formal implementation. We propose a reformulation, which we call Blocking, highlighting the elaborate idea from [23], and adapt the proof of [23] to assure that the reformulation has constant competitive ratio for planar graphs. Here, a concise observation allows us to simplify the proof and to generalize it to graphs of bounded genus. More precisely, we generalize the upper bound on the competitive ratio of 16 for planar graphs to a bound of $16(1 + 2g)$ for graphs of genus at most g.

As further contribution we give a constant competitive algorithm for general graphs with a bounded number of distinct weights. Our online algorithm generalizes DFS to an algorithm that hierarchically performs depth first searches on subgraphs induced by restricted weights. For arbitrary graphs with arbitrary weights we round weights up to the nearest power of 2 and apply the same algorithm. With this modification the algorithm has a competitive ratio of $\Theta(\log n)$ in general.

As our main result we show that Blocking does not have constant competitive ratio on general graphs. We use a classical construction of Erdős [15] of graphs with large girth and large minimum degree to construct complex graphs for which Blocking has arbitrarily large competitive ratio. Considering the fact that we have shown that Blocking is constant competitive for classes of graphs that have bounded genus, it seems plausible that similarly heavy machinery is indeed necessary for the lower bound construction.

Related work. Exploration problems have been studied extensively; see the surveys [7, 28]. In the sixties, such problems were addressed mainly from a game-theoretic perspective [19]. More recent research on online motion planning, aiming explicitly for worst-case performance guarantees on the total travel distance, was initiated in [5].

Generally, the geometry of the search environment can be arbitrary — a bounded or unbounded space, with or without obstacles, two, three, or higher dimensional. However, in many particular applications, it is possible to abstract from the geometry of the real environment and model the unknown search space as a graph, in which the searcher may only move along edges. First formal models for exploring an unknown graph were proposed in [27] in the context of finding a shortest path between two given points. Research on fully exploring a graph was initiated in [10]. In contrast to our problem, they consider the task of exploring all *edges* in a directed labeled unknown graph (with unit-weight edges). At any time, the searcher is given the number of unexplored edges leaving the vertex, but not their endpoints. Notice that the corresponding offline problem is the polynomially solvable Chinese postman problem, in contrast to the TSP in our setting. This exploration problem has been studied extensively in directed [1,16,24] and undirected graphs [11, 26]. Numerous variants were considered, e.g., routing multiple searchers [6, 14, 17], models that impose additional constraints on the searcher, such as being tethered [12], or having a tank of limited capacity [4,8], and exploration problems in graphs without unique labeling but with some other additional information [18, 20].

Our problem of exploring all *vertices* of a labeled undirected graph is in some sense also a variant of the initial problem in [10]. In particular, on trees the problem of exploring all vertices is equivalent to exploring all edges. Apart from the aforementioned previous work on our problem in planar graphs [23] and cycles [2, 25], it has been studied on un-weighted trees also for multiple synchronously moving searchers [13, 14, 17].

Even though our online graph exploration problem has the classical TSP as corresponding offline problem, another class of online problems is typically regarded as *Online TSP* in the literature. In [3] a model is introduced in which the graph is given in advance and the vertices to be visited appear online over time. This means that new vertices appear as the salesman proceeds, in contrast to our model, independently of his current position. The corresponding offline problem is a TSP with release dates.

2 The exploration algorithm of Kalyanasundaram and Pruhs

We discuss the algorithm ShortCut, that was proposed and analyzed in [23].

Algorithm 1 The exploration algorithm Blocking$_\delta(G, y)$

Input: A partially explored graph G, and a vertex y of G that is explored for the first time.

```
1: while there is an unblocked boundary edge $e = (u, v)$, with $u$ explored and $v$ unexplored,
   such that $u = y$ or such that $e$ had previously been blocked by some edge $(u', y)$ do
2:     walk a shortest known path from $y$ to $u$
3:     traverse $e = (u, v)$
4:     $\mathrm{Blocking}_\delta(G, v)$
5:     walk a shortest known path from $v$ to $y$
6: end while
```

Definition 1. *A vertex is* explored *once it has been visited by the searcher. An edge is* explored *once both endpoints are explored. A* boundary edge (u, v) *is an edge with an explored end vertex u and an unexplored end vertex v.*

We adopt the convention that for a boundary edge, the first entry is always vertex that has already been explored. For a set of edges E' we let $|E'| = \sum_{e \in E'} |e|$.

Algorithm ShortCut can be seen as a sophisticated variant of DFS. The crucial ingredient is a *blocking condition* depending on a fixed parameter $\delta > 0$, which determines when to diverge from DFS.

Definition 2. *At any point in time during the exploration of the graph, a boundary edge $e = (u, v)$ is said to be* blocked, *if there is a boundary edge $e' = (u', v')$ with u' explored and v' unexplored which is shorter than e (i.e, $|e'| < |e|$) and for which the length of any shortest known path from u to v' is at most $(1 + \delta) \cdot |e|$.*

Intuitively, the exploration algorithm ShortCut performs a standard DFS but traverses a boundary edge only if it is not blocked. Suppose the searcher is at a vertex u and considers traversing a boundary edge (u, v). If (u, v) is blocked then its traversal is postponed, possibly forever; otherwise the searcher traverses (u, v). Traversing (u, v) and exploring v may cause another edge (x, y), whose traversal was delayed earlier, to become unblocked. Then the shortest path from v to y is added as virtual edge (called jump edge in [23]) to the DFS-tree and can be traversed virtually like any real edge.

It is important to carefully update the blocking-state of edges as the algorithm proceeds. In particular, an edge which has become unblocked, after having previously been blocked, may become blocked again. This may be the case if a new shorter path from an unblocked edge to another boundary edge is revealed. In this case, the virtual edge must be removed again. Disregarding reblocking will cause an unbounded worst case ratio, even for planar graphs. This important issue is not explicitly addressed in the algorithm description in [23]. In particular, no means of removing virtual edges are provided therein.

In Algorithm 1, we formalize our interpretation of the algorithmic idea by Kalyanasundaram and Pruhs. To distinguish it from [23] and since the (parameterized) blocking condition is a very subtle and a key ingredient, we choose the name Blocking$_\delta$. To explore the entire graph starting in vertex s, we call Algorithm 1 as Blocking$_\delta(G_s, s)$, where G_s is the partially explored graph in which only s has been visited so far.

We prove that Blocking$_\delta$ has constant competitive ratio on graphs of bounded genus. Our proof not only extends the one in [23] for planar graphs (genus 0), but also differs in using an additional argument which allows an easier handling of the recurrence.

Theorem 1. Blocking$_\delta$ *is* $2(2+\delta)(1+2/\delta)(1+2g)$*-competitive on graphs of genus* g.

Proof. Since in every iteration of the while loop a new vertex is explored, the algorithm terminates. It is easy to see that all vertices are eventually explored.

We let P be the set of edges that are traversed during some execution of Line 3.

For each iteration of the while loop, we charge all costs that occur in Lines 2, 3 and 5 to the edge in P traversed in Line 3. Since any execution of Line 3 explores a new vertex, every edge is charged in at most one while-loop iteration.

The cost charged to any edge e is at most $2(2+\delta)|e|$: Indeed, either the edge had previously not been blocked, in which case the cost is simply $2|e|$, or the edge e had previously been blocked by some edge ending in y, and therefore (by the definition of blocking) the distance from y to the starting point of e is at most $(1+\delta)\cdot|e|$. Thus Lines 2, 3 and 5 provoke costs of at most $(1+\delta)\cdot|e|$, $|e|$, and $(2+\delta)\cdot|e|$, respectively.

Let MST be a minimum spanning trees that shares a maximum number of edges with P. It suffices now to show that $|P| \leq (1+2/\delta)(1+2g)|MST|$ in order to get an overall cost of at most $2(1+\delta)(1+2/\delta)(1+2g)|MST|$.

Claim. If an edge $e \in P \setminus MST$ is contained in a cycle C in $P \cup MST$, then the cycle C has length at least $(2+\delta)|e|$.

Proof. Suppose otherwise. On the cycle C, consider the edge $e' = (u,v) \in P \setminus MST$ with $|e'| \geq |e|$ that is charged the latest. W.l.o.g. suppose e' is traversed from u to v at the time it is charged. Due to the choice of MST, the edge e' is strictly larger than any edge in $C \cap MST$: Otherwise we could replace e' with an edge in MST to obtain a smaller minimum spanning tree or to obtain a minimum spanning tree that shares more edges with P. At the time e' is charged, e' is a boundary edge, and therefore not the whole cycle has been explored. Thus there is a boundary edge different from e' on the cycle. Moreover at this point in time e' is not blocked. Let e'' be the first boundary edge encountered when traversing $C - e'$ starting from u towards v. Since we assume the cycle has length less than $(2+\delta)|e| \leq (2+\delta)|e'|$ and e' is not blocked, we conclude that e'' is not smaller than e' and thus not in MST. This contradicts the fact that e' is the edge in $P \setminus MST$ with $|e'| \geq |e|$ that is charged the latest and shows the claim. $\square$

Consider an embedding of G on a surface of genus g. We choose $MST' \subseteq MST \cup P$ to be a maximal superset of MST obtained by repeatedly adding edges that do not separate two faces, i.e., are incident with only one face. (Topologically this can be viewed as adding a set of non-separating cycles, after contracting MST to a single point.) Since adding a non-separating edge increases the Euler characteristic of the surface bounded by the edges, and a surface of genus g has Euler characteristic $2-2g$, there are at most $2g$ edges in $MST' \setminus MST$. In case MST' does not bound a topological disk, we artificially add non-separating edges each of weight $|MST|$ to the graph induced by P, to obtain a superset MST'' of MST' that bounds a topological disk. These edges are artificial in the sense that they do not need to be edges of G. By the Euler characteristic argument above, there are at most $2g$ edges in $MST'' \setminus MST$ in total.

All edges in P, and thus, all edges in $MST'' \setminus MST$ have a weight not larger than $|MST|$, since otherwise they would be blocked until the whole minimum spanning tree has been explored. This implies $|MST''| \le |MST| + 2g|MST|$. Since every edge $e \in P \setminus MST''$ is contained in a cycle of length at most $|e| + |MST|$ and edges in $MST'' \setminus MST'$ are of length $|MST|$, we can extend the claim: If an edge $e \in P \setminus MST''$ is contained in a cycle C in $P \cup MST''$, then the cycle C has length at least $(2+\delta)|e|$. We iteratively define for every edge $e \in P \setminus MST''$ a cycle C_e in the following way: In each step we choose an edge that together with edges in MST'' and edges to which a cycle has already been assigned closes a face cycle. Note that for two distinct edges $e, e' \in P \setminus MST''$ the associated cycles C_e and $C_{e'}$ are different. Every edge in $MST'' \cup P$ is contained in at most two such cycles, since they form a set of distinct face cycles. For an edge in $P \setminus MST''$ one of these cycles is C_e. In fact these cycles are exactly all face boundaries apart from the boundary of the outer face. Therefore, $|P \setminus MST''| \le \frac{1}{1+\delta} \sum_{e \in P \setminus MST''} |C_e - e| \le \frac{1}{1+\delta}(2|MST''| + |P \setminus MST''|)$, and thus $|P \setminus (MST'')| \le (2/\delta)|MST''|$. We conclude that $|P| \le (1+2/\delta)|MST''| \le (1+2/\delta)(1+2g)|MST|$. Overall, we conclude that $\mathsf{Blocking}_\delta$ is $2(2+\delta)(1+2/\delta)(1+2g)$-competitive on graphs of genus g. □

Corollary 1. *Algorithm* $\mathsf{Blocking}_2$ *is* $16(1+2g)$*-competitive on graphs of genus* g*.*

3 A lower bound construction

Our lower bound construction for Algorithm $\mathsf{Blocking}_\delta$ relies on a *base graph* H with specific bounds on girth and degree. Its existence is guaranteed by the following lemma which extends a classical construction of Erdős [15].

Lemma 1. *For all* $\bar{d}, \delta \in \mathbb{N}$ *there exists a connected bipartite graph* H *with minimum degree at least* $\bar{d}$*, maximum degree at most* $2\bar{d}$*, and a girth of* $g \ge \delta + 2$*.*

Let H be a connected bipartite n-vertex graph with average degree at least $\bar{d}$, maximum degree at most $2\bar{d}$, and girth at least $\delta + 2$ as given by Lemma 1. Suppose the partition classes have size n_1 and n_2. We fix orders $(u_1, \dots, u_{n_1})$ and $(v_1, \dots, v_{n_2})$ for the vertices in each of the bipartition classes. We call the vertices in the bipartition classes in-vertices and out-vertices, respectively. We order the edges by the lexicographical ordering satisfying $\{u, v\} < \{u', v'\}$ if $v < v'$ or ($v = v'$ and $u < u'$).

In H we now replace each in-vertex and each out-vertex by a release gadget and collection gadget, respectively. Our final construction will have edges with two types of weights, namely 1 and $w > 1$. We call edges of weight w *heavy* edges.

Description of the release gadgets: Figure 1 depicts a release gadget of degree 4. In general a release gadget consists of two parts, which we call left and right part. A gadget replacing a vertex of degree d consists in the left part of d vertices forming a path. Each of these vertices is attached to a heavy (red) edge of weight w that has an endpoint in some collection gadget. The right part contains d vertices forming a path. Each of these vertices is incident with an attached *release path* of length $d-1$. The endpoints of these paths are incident with a (blue) edge that ends in a collection gadget. The two parts are

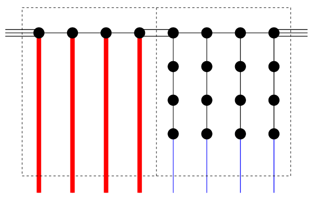

Fig. 1. Release gadget of a degree 4 vertex.

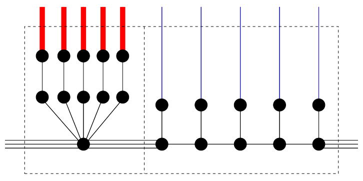

Fig. 2. Collection gadget of a degree 5 vertex.

joined by a *center path* of length $(\delta + 1)w - d$ (depicted by a double edge) with unit-weight edges. Finally, each part has a *blocking path* (depicted as triple edges) of length $(\delta + 1)w$ with unit-weight edges, by which it is connected to other release gadgets.

The crucial property of a release gadget is the following: The length of the center path is chosen such that the i-th heavy edge may be blocked by the first edge of the i-th release path, but not by the $(i + 1)$-st release path (both times counting from the left to right). Once the exploration of a release path has begun, the algorithm will finish the exploration of the entire release path before exploring any other edges.

Thus, the i-th heavy edge of the gadget is blocked, if one of the release paths $1, \ldots, i$ has not yet been explored. If a release path has been completely explored, we also say that the release has been triggered. Suppose in some release gadget all releases $1, \ldots, i$ have been triggered, but the i-th heavy edge is still blocked. This situation implies that there is a path to some unit-weight boundary edge which exits the gadget via another heavy edge: Indeed, the blocking paths are sufficiently long to prevent other release gadgets from interfering with this fact. We will show later that at the moment release i is triggered such paths exiting via heavy edges do not exist.

It will be clear later that when a release gadget is entered for the first time, this happens via the blocking path to the right of the gadget. Assuming this for now, we can require that the online algorithm traverses the gadget from right to left, without entering the release paths: Indeed, whenever there is a choice among edges of equal weight, we can adversarially choose the edge that is traversed next.

Description of the collection gadgets: Figure 2 depicts a collection gadget of degree 5. In general a collection gadget consists of a left and a right part. For a gadget replacing a vertex of degree d, the left part has one vertex of degree d incident with d paths of length 2. The ends of these paths are incident with heavy (red) edges emanating from release gadgets. The right part of a collection gadget contains d vertices inducing a path. Each vertex on the path is adjacent to another vertex which itself is incident with a (blue) edge emanating from a right part of a release gadget. Three blocking paths (triple edges) of length $(\delta + 1)w$ join the parts with each other and with other collection gadgets.

We will see that when a collection gadget is first entered, this happens via the blocking path to the left. We can then require adversarially that the online algorithm traverses from the left part directly to the right without exploring the left part. Then, on entering a vertex in the right, it deviates from the main path to explore the respective blue edge. We will argue that the algorithm will then return via a corresponding heavy edge. It

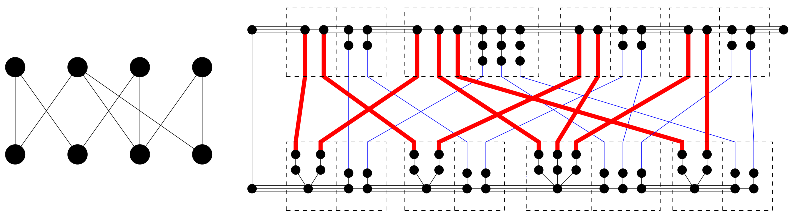

Fig. 3. The assembly of the gadgets: A base graph H (left), and the resulting graph with linked replacement gadgets (right) showing release gadgets on top and collection gadget at the bottom.

then backtracks and subsequently explores the next vertex of the right part, and so on. Before leaving the gadget to the right, the gadget has been completely explored.

Assembly of the gadgets according to the base graph H: To assemble the gadgets using the base graph H (see Figure 3), we join the release gadgets according to the order of in-vertices along the blocking paths (triple lines). The same is done with the collection gadgets, with respect to the order of out-vertices. The right blocking path of the last (rightmost) release gadget is connected to a single vertex, the starting vertex, that we add to the graph. The left blocking path of the first (leftmost) release gadget and the first (leftmost) collection gadget are joined by two added, adjacent vertices.

The (red and blue) edges that run between the gadgets correspond to the edges in H. Heavy (red) edges of weight w run from a left part of a release gadget to a left part of a collection gadget. Blue edges of weight 1 run from a right part of a release gadget to a right part of a collection gadget.

In the lexicographical order of the edges defined above, we insert for each edge of H a heavy (red) edge and a blue edge. To insert a heavy edge corresponding to the edge (u, v) of H, we connect the leftmost unused vertex in the left part of the release gadget corresponding to u with the leftmost unused vertex in the left part of the collection gadget of v. To insert the blue edge, we connect the leftmost unused vertex in the right part of the release gadget corresponding to u with the leftmost unused vertex of the right part of the collection gadget of v.

Inserting the heavy edges in this ordering has the consequence that the ordering of the edges is exactly the ordering of their end vertices in the collection gadgets from left to right. Furthermore, within each release gadget, the heavy edges from left to right are also in the lexicographic order.

The tour traversed by the algorithm: Beginning at the starting vertex, we may require adversarially that the algorithm first traverses all release gadgets without exploring any release path. Then, via the two additional vertices on the left, the leftmost collection gadget is entered from the left, and the exploration continues into its right part. Subsequently release paths are triggered, one at a time. In the following we prove that the algorithm traverses all heavy edges of H. The lexicographic order defined on the edges is the order in which these edges are traversed. All of them are traversed from a release gadget to a collection gadget. The blue edges, each used to trigger a traversal of a heavy edge, are traversed from a collection gadget to a release gadget. Recall that due to the length of the center path connecting the right and left part of a release gadget, a heavy edge is blocked, unless its corresponding release has been triggered.

Lemma 2. *The heavy (red) edges are traversed in the lexicographic order of the edges of the base graph. Whenever a release is triggered, the corresponding heavy edge e_r becomes unblocked and is explored subsequently.*

Proof. Inductively we assume that all release paths that correspond to edges that appear earlier than e_r in the ordering of edges have been completely explored, and all release paths that appear later than e_r are completely unexplored.

A heavy weight edge can only be blocked by an edge of weight 1. Thus, for a heavy edge to be blocked, there has to be a path of length at most $(\delta+1)w-1$ to a boundary edge of weight 1. To show the claim, we show that no such path exists for edge e_r.

To do so, we analyze where a hypothetical boundary edge of such a path may be situated in the graph. Observe that the length of blocking paths (triple edge) is chosen such that they cannot be traversed to reach a boundary edge within a distance of $(\delta+1)w-1$. Thus, only two possibilities have to be ruled out:

1. There is a path to a boundary edge that can be reached by a path of length $(1+\delta)w-1$, which traverses a center path (double edge).
2. There is a path to a boundary edge that uses heavy edges, but otherwise is completely contained in left parts of gadgets.

To rule out Possibility 1, observe that any path that uses a double line to cross from a left part of a release gadget to a right part, and then uses a complete release path is longer than $(\delta+1)w-1$. Moreover, since release paths are either completely explored or the corresponding heavy edge has by induction not been triggered, for every unexplored edge in the right part of a release gadget, all explored heavy edges in the left part are further away than $(\delta+1)w-1$.

To rule out possibility 2, note that the only boundary edges of weight 1 situated in the left part of a gadget are contained in the currently used collection gadget. All other left parts of gadgets have been completely explored or not explored at all. Thus, any path staying in the left parts of the gadgets that leads to a boundary edge in the left part of the currently used collection gadget will, together with e_r, project to a cycle in the graph H. Since the girth of H is at least $\delta+2$, the path has to use at least $\delta+1$ heavy edges and is thus of length more than $(\delta+1)\cdot w$.

We have shown that the heavy edge e_r becomes unblocked when its release is triggered. The algorithm thus explores e_r, returns to the release path of to e_r, backtracks, and continues to trigger the release corresponding to the next heavy edge. $\square$

Theorem 2. *For no $\delta \in \mathbb{R}$ does* $\mathsf{Blocking}_\delta$ *have constant competitive ratio.*

Proof. Consider a graph that is obtained from the replacement construction from a base graph H on n vertices with minimum degree $\bar{d}$, maximum degree at most $2\bar{d}$, and girth at least $\delta+2$ (Lemma 1). Including blocking paths, the number of unit-weight edges in a release gadget corresponding to a vertex v of degree $d(v)$ is $\mathcal{O}(d(v)^2)+\mathcal{O}(\delta w) \subseteq \mathcal{O}(\bar{d}^2)+\mathcal{O}(\delta w)$. This bound also holds for collection gadgets. Thus, for fixed δ, the resulting graph has a minimum spanning tree of size $\mathcal{O}(n\bar{d}^2)+\mathcal{O}(nw)$. Since $\mathsf{Blocking}_\delta$

Algorithm 2 Exploration algorithm hDFS(G, u, w)

Input: A partially explored graph G, a vertex u of G that is visited for the first time, and a weight $w \in \mathbb{R}_{\geq 0} \cup \{\infty\}$.

```
1: while there is a $w' < w$ such that $w'$ occurs in $\mathrm{comp}(G_{\leq w'}, u)$ but $\mathrm{comp}(G_{\leq w'}, u)$ is not completely explored do
2:     hDFS$(G, u, w')$
3: end while
4: choose a minimal spanning tree of $\mathrm{comp}(G_{<w}, u)$ and order all vertices according to a depth first search in this spanning tree
5: while there is a boundary edge $(u', y')$ of weight $w$ with $u' \in \mathrm{comp}(G_{<w}, u)$ do
6:     let $(u', y')$ be a boundary edge of weight $w$ with $u' \in \mathrm{comp}(G_{<w}, u)$ such that $u'$ is minimal with respect to the ordering of $\mathrm{comp}(G_{<w}, u)$
7:     traverse a shortest path to $y'$
8:     hDFS$(G, y', w)$
9: end while
10: traverse a shortest path to $u$
```

traverses all heavy edges (Lemma 2), it incurs a cost of $\Omega(\bar{d}nw)$. By choosing $\bar{d}$ large in comparison to all constants involved and then choosing w large in comparison to the constants and $\bar{d}^2$ the competitive ratio becomes arbitrarily large. □

4 Graphs with a bounded number of distinct weights

We describe a constant competitive algorithm for a bounded number of distinct weights.

Definition 3. *For any graph G, weight w, and vertex u, let $\mathrm{comp}(G_{\leq w}, u)$ be the connected component of the subgraph of G comprised of all edges of weight at most w containing u. The graph $\mathrm{comp}(G_{<w}, u)$ is defined similarly.*

Our algorithm *hierarchical depth first search* (hDFS), defined in Algorithm 2, explores $\mathrm{comp}(G_{\leq w}, u)$ for any weight $w \in \mathbb{R}_{\geq 0} \cup \{\infty\}$, which is provided as a parameter. The algorithm is based on a DFS in the graph $\mathrm{comp}(G_{\leq w}, u)$. However, whenever a new vertex of this component is encountered, it first explores $\mathrm{comp}(G_{<w}, u)$. The algorithm then intuitively simulates DFS in the graph $G/\mathrm{comp}(G_{<w}, u)$. Here G/H denotes the graph obtained from G by contracting the subgraph H of G to a single point. To ensure that the total length traversed within $H = \mathrm{comp}(G_{<w}, u)$ is not too large, the boundary edges leaving H are explored according to a specific order. This order is obtained by computing a depth first search on a minimum spanning tree of $\mathrm{comp}(G_{<w}, u)$.

The computation of $\mathrm{comp}(G_{<w}, u)$ can be reduced to recursive calls of the algorithm itself with parameters smaller than w due to the following basic observation:

Lemma 3. *The component $\mathrm{comp}(G_{<w}, u)$ is completely explored if and only if there is no boundary edge of weight smaller than w with an end-vertex in $\mathrm{comp}(G_{<w}, u)$.*

To explore the entire graph starting in vertex s, we call Algorithm 2 as hDFS(G_s, s, ∞), where G_s is the partially explored graph in which only s has been visited so far.

Theorem 3. hDFS *is $2k$-competitive on graphs with at most k distinct weights.*

Proof. We first prove that all vertices are explored. To prove this it suffices to show that hDFS(G, s, w) explores $\mathrm{comp}(G_{\leq w}, s)$. We show this by induction. Suppose there remains a boundary edge (u, v) with v unexplored after the call hDFS(G, s, w), and suppose this boundary edge is contained in $\mathrm{comp}(G_{\leq w}, s)$. By induction (u, v) has weight w. But u has been explored, thus there is a vertex y which was explored in a call with parameter w, such that this call caused u to be explored. But then the call hDFS(G, y, w) causes v to be explored, which gives a contradiction.

Let MST be a minimum spanning tree of G. To show $2k$-competitiveness, we show that for each $w < \infty$ the sum of all traversals made in calls with parameter w is at most $2|MST|$. For this it suffices to show: If F is a sub-forest of G that contains edges of weight at most w such that for each vertex u the graph $\mathrm{comp}(F_{<w}, u)$ is a minimum spanning tree of $\mathrm{comp}(G_{<w}, u)$, then F is contained in a minimum spanning tree of G. Finally note that the outer call with parameter $w = \infty$ does not incur any costs. □

For graphs with arbitrary weights, we adapt the algorithm by rounding each edge weight to the nearest power of 2 and simulating the exploration on this altered graph. This yields a competitive ratio of $\Theta(\log(n))$ for graphs with n vertices.

5 Concluding remarks

Our main result is a non-trivial graph construction which proves that Algorithm Blocking does not have constant competitive ratio on arbitrary graphs. This answers a long-standing open question. Nevertheless, the result does not generally rule out online algorithms with constant competitive ratio. In particular, our construction involves only two distinct types of weights, and thus, our new Algorithm hDFS has constant competitive ratio. However, at present, there is no candidate for an algorithm that may achieve a constant competitive ratio on general graphs. Of course showing that no such algorithm exists might require a construction even more complicated than the one presented in this paper. For such result it might be helpful to use the fact that one can equivalently consider the exploration model in which the label of a vertex is only revealed upon arrival at the vertex. This can be seen by replacing each vertex by a star with edges of small weight, and linking the previous neighbors to the outer vertices of the star.

References

1. S. Albers and M. R. Henzinger. Exploring unknown environments. *SIAM J. Comput.*, 29(4):1164–1188, 2000.
2. Y. Asahiro, E. Miyano, S. Miyazaki, and T. Yoshimuta. Weighted nearest neighbor algorithms for the graph exploration problem on cycles. *Inf. Process. Lett.*, 110(3):93–98, 2010.
3. G. Ausiello, E. Feuerstein, S. Leonardi, L. Stougie, and M. Talamo. Algorithms for the on-line travelling salesman. *Algorithmica*, 29(4):560–581, 2001.
4. B. Awerbuch, M. Betke, R. L. Rivest, and M. Singh. Piecemeal graph exploration by a mobile robot. *Inf. Comput.*, 152(2):155–172, 1999.

5. R. A. Baeza-Yates, J. C. Culberson, and G. J. E. Rawlins. Searching in the plane. *Information and Computation*, 106(2):234–252, 1993.
6. M. A. Bender and D. K. Slonim. The power of team exploration: Two robots can learn unlabeled directed graphs. In *Proceedings of FOCS*, pages 75–85, 1994.
7. P. Berman. On-line searching and navigation. In *Online Algorithms: The State of the Art*, volume 1442 of *Lecture Notes in Computer Science*, pages 232–241. Springer, 1998.
8. M. Betke, R. L. Rivest, and M. Singh. Piecemeal learning of an unknown environment. *Machine Learning*, 18:231–254, 1995.
9. A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
10. X. Deng and C. H. Papadimitriou. Exploring an unknown graph (extended abstract). In *Proceedings of FOCS*, pages 355–361, 1990.
11. A. Dessmark and A. Pelc. Optimal graph exploration without good maps. *Theoretical Computer Science*, 326(1-3):343–362, 2004.
12. C. A. Duncan, S. G. Kobourov, and V. S. A. Kumar. Optimal constrained graph exploration. *ACM Trans. Algorithms*, 2:380–402, July 2006.
13. M. Dynia, J. Kutylowski, F. der Heide, and C. Schindelhauer. Smart robot teams exploring sparse trees. In *Proceedings of MFCS*, volume 4162 of *LNCS*, pages 327–338, 2006.
14. M. Dynia, J. Lopuszanski, and C. Schindelhauer. Why robots need maps. In *Proceedings of SIROCCO*, volume 4474 of *LNCS*, pages 41–50, 2007.
15. P. Erdős. Graph theory and probability. *Canad. J. Math.*, 11:34–38, 1959.
16. R. Fleischer and G. Trippen. Exploring an unknown graph efficiently. In *Proceedings of ESA*, volume 3669 of *LNCS*, pages 11–22, 2005.
17. P. Fraigniaud, L. Gąsieniec, D. R. Kowalski, and A. Pelc. Collective tree exploration. *Netw.*, 48:166–177, October 2006.
18. P. Fraigniaud, D. Ilcinkas, and A. Pelc. Impact of memory size on graph exploration capability. *Discrete Applied Mathematics*, 156(12):2310–2319, 2008.
19. S. Gal. *Search Games*. Academic Press, 1980.
20. L. Gasieniec and T. Radzik. Memory efficient anonymous graph exploration. In *Proceedings of WG*, volume 5344 of *LNCS*, pages 14–29, 2008.
21. G. Gutin and A. P. Punnen. *The Traveling Salesman Problem and Its Variations*. Springer, 2002.
22. C. A. J. Hurkens and G. J. Woeginger. On the nearest neighbor rule for the traveling salesman problem. *Operations Research Letters*, 32(1):1–4, 2004.
23. B. Kalyanasundaram and K. Pruhs. Constructing competitive tours from local information. *Theor. Comput. Sci.*, 130(1):125–138, 1994.
24. S. Kwek. On a simple depth-first search strategy for exploring unknown graphs. In *Proceedings WADS*, volume 1272 of *LNCS*, pages 345–353, 1997.
25. S. Miyazaki, N. Morimoto, and Y. Okabe. The online graph exploration problem on restricted graphs. *IEICE Transactions on Information and Systems*, E92.D(9):1620–1627, 2009.
26. P. Panaite and A. Pelc. Exploring unknown undirected graphs. *Journal of Algorithms*, 33(2):281–295, 1999.
27. C. Papadimitriou and M. Yannakakis. Shortest paths without a map. *Theoretical Computer Science*, 84(1):127–150, 1991.
28. N. Rao, S. Kareti, W. Shi, and S. Iyengar. Robot navigation in unknown terrains: Introductory survey of nonheuristic algorithms. Report ORNL/TM-12410, Oak Ridge Nat. Lab., 1993.
29. D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II. An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comput.*, 6(3):563–581, 1977.