Online Graph Exploration with Advice*

Stefan Dobrev[1], Rastislav Královič[2], and Euripides Markou[3]

[1] Institute of Mathematics, Slovak Academy of Sciences, Bratislava, Slovakia
Stefan.Dobrev@savba.sk
[2] Department of Computer Science, Comenius University, Bratislava, Slovakia
kralovic@dcs.fmph.uniba.sk
[3] Department of Computer Science & Biomedical Informatics,
University of Central Greece, Lamia, Greece
emarkou@ucg.gr

Abstract. We study the problem of exploring an unknown undirected graph with non-negative edge weights. Starting at a distinguished initial vertex s, an agent must visit every vertex of the graph and return to s. Upon visiting a node, the agent learns all incident edges, their weights and endpoints. The goal is to find a tour with minimal cost of traversed edges. This variant of the exploration problem has been introduced by Kalyanasundaram and Pruhs in [18] and is known as a *fixed graph scenario*. There have been recent advances by Megow, Mehlhorn, and Schweitzer ([19]), however the main question whether there exists a deterministic algorithm with constant competitive ratio (w.r.t. to offline algorithm knowing the graph) working on all graphs and with arbitrary edge weights remains open. In this paper we study this problem in the context of *advice complexity*, investigating the tradeoff between the amount of advice available to the deterministic agent, and the quality of the solution. We show that $\Omega(n \log n)$ bits of advice are necessary to achieve a competitive ratio of 1 (w.r.t. an optimal algorithm knowing the graph topology). Furthermore, we give a deterministic algorithm which uses $O(n)$ bits of advice and achieves a constant competitive ratio on any graph with arbitrary weights. Finally, going back to the original problem, we prove a lower bound of $5/2 - \epsilon$ for deterministic algorithms working with no advice, improving the best previous lower bound of $2 - \epsilon$ of Miyazaki, Morimoto, and Okabe from [20]. In this case, significantly more elaborate technique was needed to achieve the result.

1 Introduction

The exploration of an unknown environment is a well studied problem under many different scenarios. This problem appears in many areas, such as terrain exploration by robots, network exploration by agents, maintaining security of large networks or searching for data in the internet and ad-hoc networks. As

* E. Markou was supported in part by a research grant offered by the Ministries of Education of Greece and Slovakia (Bilateral Exchange Programme for Researchers) and by THALES: ALGONOW project funded by the Greek Ministry of Education.

G. Even and M.M. Halldórsson (Eds.): SIROCCO 2012, LNCS 7355, pp. 267–278, 2012.
© Springer-Verlag Berlin Heidelberg 2012

the agent has initially limited knowledge about the network topology and this knowledge grows only by the agent observing its immediate neighbourhood, in order to perform exploration (and/or construct a complete map of the network) the agent has to visit each node.

In the online graph exploration problem an agent starts at a node s of an undirected labeled graph $G = (V, E)$, with $|V|$ denoted by n. Each edge $e \in E$ has a non-negative weight, also called length or cost of e. The agent has no knowledge about the topology of G. The task of the agent is to visit every node of the graph and return to s. The agent can move only along the edges of G, each time paying the respective edge cost. In the particular variant we consider in this paper, when the agent arrives at a node $u \in G$, it learns all incident edges, their weights and their endpoints. This scenario has been introduced by Kalyanasundaram and Pruhs in [18] and is known as a *fixed graph scenario*. While learning the endpoints of the incident edges is stronger than the typical exploration scenario, it does have justification (see [18] and [19]); it also corresponds to previously studied *neighbourhood sense of direction* [8].

The quality of an exploration algorithm under the above scenario is usually measured by a *competitive analysis* ([4]), which compares the solution of an algorithm with an optimal offline solution, i.e., the solution of an optimal algorithm which has access to a complete and accurate map of the network. This analysis is complicated by the fact that the underlying offline problem corresponds to the *Traveling Salesman Problem* (TSP), which is known to be NP-hard, even to get a constant-approximation (e.g., see [14]).

Related Work. A simple and fast heuristic for the traditional TSP offline setting which has been studied a lot is the greedy algorithm *Nearest Neighbor* (NN): Once at a node u, go to the closest yet unexplored vertex v and repeat the process until all vertices have been explored. This algorithm also applies in the online setting, achieving competitive ratio of $\Theta(\log n)$ ([21]), which is tight even on planar unit-weight graphs ([16]).

While NN is non-competitive on general graphs, it performs quite well (with competitive ratio of 3/2 ([1])) on simple cycles. A close lower bound of 5/4 was also proved in [1]. These results for cycles have been later improved to $\frac{1+\sqrt{3}}{2}$ matching lower and upper bound [20].

For graphs in which all edges have the same weight, a *Depth First Search* (DFS) is 2-competitive, as the weight of a Minimum Spanning Tree (MST) is a lower bound. This has been shown to be optimal in [20]. A sophisticated generalization of DFS (named `ShortCut`), introducing a parameterized condition which determines when to diverge from DFS, has been proposed in [18]. `ShortCut` has been shown to achieve competitive ratio of 16 in planar graphs; it has been longstanding hypothesis that it is in fact constant competitive. `ShortCut` has been reformulated in [19] and the upper bound has been generalized to $16(1 + 2g)$ for graphs of genus at most g. However, it has been shown in [19] that neither of these algorithms is constant competitive in general graphs with arbitrary weight. In fact there are classes of graphs for which their competitive ratio is arbitrarily large. Finally, a generalization of DFS that can be seen as a hierarchical DFS

was shown to be constant competitive on graphs with a bounded number of different edge weights. A slight generalization of this algorithm achieves $\Theta(\log n)$ competitive ratio for graphs with arbitrary weights.

Advice Complexity. The impact of additional (typically structural) information on complexity of algorithms has been a longstanding and rich field of study. Impact of various aspects of structural information (e.g. knowledge of network size and/or network topology, presence of *sense of direction*, availability of distinct node IDs) has been extensively studied for various problems. In general, this information has been of qualitative type, i.e. the questions asked were of the type "what is the impact of presence/absence of specific structural information?".

A new line of research focusing on quantitative aspects of such information has has recently become popular. The idea is to provide the algorithm/agent with some additional information (*advice*) given explicitly as a binary string, thus allowing to measure the information quantitatively. We use the model from [5,10] where the advice is given to the agent at the beginning of the algorithm. Alternatively, the advice could be stored in a distributed fashion in the nodes of the network (see e.g. [9,11,12,13,17]), and the maximum or average size of the advice per node be considered. The advice encodes problem-relevant information about unknown facts (i.e. topology in case of distributed algorithms) and can be seen as computed by an oracle (of unlimited power) that knows the missing information the algorithm wants to use. This approach allows to precisely measure the amount of additional information provided to the agent and facilitates study of the tradeoff between the size of the advice, and the quality of the solution.

In the context of online algorithms, analogous concept has been independently proposed in [6], and has been developed in two ways: the model from [7] considers that the algorithm receives, with each request, some fixed amount of b bits of advice. In the model from [3] (see also [2,15]), on the other hand, the whole advice is given to the algorithm at the beginning.

Our Results. Our primary interest is in the study of the tradeoff between the advice size and the quality of the solution for the case of general graphs with arbitrary weights. As a first step, we show (in Section2.1) that in order to have an optimal algorithm with competitive ratio strictly 1, advice of size $\Omega(n \log n)$ bits is needed.

The primary question we are interested in is what is the smallest advice with which there is a constant competitive algorithm. This can be seen as a relaxation of the original question whether there is a constant competitive algorithm with no advice at all. We provide an upper bound (in Section 3), presenting an algorithm that achieves constant competitive ratio $6 + \epsilon$ using $O(n)$ bits of advice. This result seems rather weak as we had to pay $O(n)$ bits of advice to reduce the competitive ratio by a factor of $O(\log n)$. However, it took a quite elaborate algorithm to achieve even this result. As the algorithm has to make many decisions over the whole network, using $o(n)$ bits of advice and still achieving constant competitive ratio would require significant new insight.

Going back to the original problem with no advice, we prove (in Section 2.2) that without advice the competitive ratio of any deterministic online algorithm cannot be less than $5/2 - \epsilon$. The best previous lower bound of $2 - \epsilon$ is from [20] and concerns graphs with unit weights (for which there is also a matching upper bound). Our lower bound is significantly more involved, necessarily employing edges of many different weights in an elaborate hierarchical structure. Due to space constraints the proofs of lemmas and theorems and some of the more technical parts will appear in the full version of the paper.

2 Lower Bounds

2.1 Advice Size for Optimal Solution

Let $\{v_0, v_1, \ldots, v_{n-1}\}$ be a set of vertices. Define $w(v_i) = n - i$. Denote by K_n^w the clique on vertices $v_0, v_1, \ldots, v_{n-1}$ in which the edge between vertices v_i and v_j is of weight $w(v_i, v_j) = \max(w(v_i), w(v_j))$.

Lemma 1. *There is a unique (up to reversal) walk π (visiting all nodes of K_n^w) of a minimum cost in K_n^w with endpoints v_0, v_{n-1}. Furthermore, $\pi = \{v_0, v_1, \ldots, v_{n-1}\}$.*

Consider now K_n^w in which the adversary assigns the IDs visible to the exploring agent. Hence, when at node v_i, the agent can from the weights of the incident edges deduce i and the IDs of the nodes v_j for $j < i$. However it cannot distinguish between the $n - i$ edges of weight $w(i)$ leading to not-yet-visited vertices. Therefore, in order to ensure the vertices are visited in the optimal order $v_0, v_1, \ldots, v_{n-1}$, the agent needs advice of size $\log(n - i)$ at vertex i. Assuming the agent starts at vertex v_0 and summing up over all vertices yields $\Omega(n \log n)$ bound on the advice.

However, since the reverse of π is also an optimal path, the adversary can give the advice (of size $\log n$) which edge leads to v_{n-1}. Once the agent is in v_{n-1}, it can from the weight of the incident edges deduce the remainder of the optimal traversal sequence. Hence an agent could complete a cycle visiting all nodes of K_n^w using just $\log n$ bits of advice. In order to prevent this exploit, consider the graph G consisting of two copies of K_n^w and two additional edges of unit weight. An example with two copies of K_8^w is shown in Figure 1. Using Lemma 1 we show that the optimal cycle including all nodes of G is:

$$\{v_0, v_1, \ldots, v_{n-1}, v'_{n-1}, v'_{n-2}, \ldots, v'_0, v_0\}$$

Hence, to traverse this cycle, the agent needs $\Omega(n \log n)$ bits of advice in at least one copy of K_n^w.

Theorem 1. *There is a family of graphs for which any optimal-cost algorithm solving the graph exploration problem needs $\Omega(n \log n)$ bits of advice.*

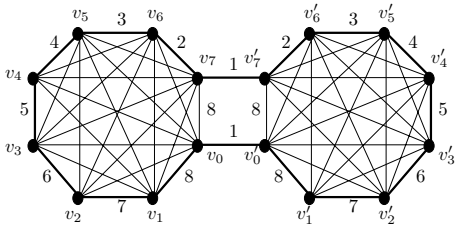

Fig. 1. The lower bound graph. The bold line is the optimal exploration path.

2.2 Lower Bound for the Case of No Advice

Theorem 2. *For any deterministic algorithm $\mathcal{A}$ and any $\frac{1}{2} > \varepsilon > 0$ there exists a graph G_{lb} such that the competitive ratio of $\mathcal{A}$ is at least $\frac{5}{2} - \varepsilon$. Moreover, G_{lb} has $r^{O(\log r)}$ vertices, where $r = 2/\varepsilon$.*

The lower bound graph G_{lb} consists of a *backbone* cycle and additional *return* and *skip* edges. At the highest level, G_{lb} includes x level-k (x and k will be determined later, x is odd) blocks connected in a backbone cycle, and $x - 1$ additional skip edges to be described later. Each block of level-i, where $i > 1$ consists of x sub-blocks of level-$(i-1)$ forming a line connected by level-$(i-1)$ backbone edges. The agent starts exploration in the middle of a level-k block. Each block has a right and left side; which side is which is decided based on algorithm's actions. In particular, the side whose endpoint is first reached by the agent is by definition the right side; an adversary decides how the left and right sides of neighbouring blocks align.

Let v be the rightmost vertex of a level-i block B such that B is the highest level block for which v is the rightmost vertex. Then v is connected to the next level-i block by two level-i edges: the backbone edge leading to the leftmost vertex of the next level-i block, and the skip edge leading to the middle of the next level-i block. There are two exceptions (the middle block has two skip edges, and one block has no skip edges). Additionally, there is a level-i return edge from v to the middle of the leftmost sub-block of B. The weights of these three level-i edges incident to v are the same and are equal to the cost of the minimum cost (we also call it shortest) path (by the cost of a path we mean the sum of the weights of the edges of the path) connecting the endpoints of the return edge and not using those three level-i edges. Note that if v is also the rightmost vertex of a lower-level block B' then B' does not contain a return edge of its level. When given a choice (i.e., the algorithm wants to traverse an edge whose endpoint's ID has not yet been seen) the adversary's order of preference is return edge, then skip edge, then backbone edge.

Level-1 block is analogous: It consists of x vertices connected in a line by backbone edges of weight 1. There is a difference, though: The return edge leads

to the second leftmost vertex[1]. As a consequence, the right side of a level-1 block is the side in which the third vertex from the end is visited first[2]. The structure of a level-i block is captured in Figure 2.

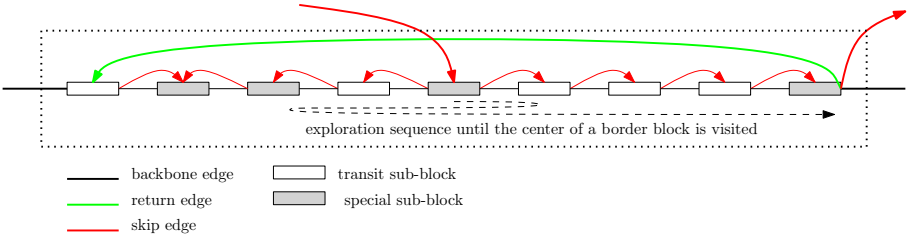

Fig. 2. The structure of a level i block (this one is transit). Fat edges are of level i.

In the following, by the cost of a path we mean the sum of the weights of the edges of the path. Let r_i denote the cost of the shortest path from the middle of a level-i block to its rightmost/leftmost vertex (note that these distances are the same) Let t_i denote the cost of the shortest path using edges of level at most $i-1$ and connecting the endpoints of a level-i return edge. Let o_i denote the cost of the backbone path from the left-most vertex to the right-most vertex of a level-i block.

Let us classify the blocks as *transit* and *special*. A block B of level i is *transit*, if and only if all of the following conditions hold:

- B is entered for the first time via the skip edge leading into its center
- at that time, exactly one neighbouring block of B has not been visited – the block B'
- B' is visited for the first time via a skip edge from B
- B contains a return edge of level i

All other blocks are *special*. Observe that in an optimal algorithm, a transit block has at most 4 special sub-blocks, while a special block may have up to 5 special sub-blocks.

Let us denote by e_i and $\tilde{e}_i$ the minimal cost (perhaps over several visits) incurred by the agent in a level-i transit and special block, respectively, until all vertices of the block have been visited. For the transit blocks, we charge to the block also the cost of arriving to and leaving the block (if there is such activity) in the time period between the first arrival of the agent to B and the first leaving of the agent towards B'. From the definition of G_{lb} we have

[1] The reason is that the ID of the leftmost vertex might be already known to the algorithm, allowing to distinguish the incident level-i edges at the rightmost vertex.

[2] Once this vertex is visited, it knows the ID of the second-from-end vertex which would allow to recognize the endpoint of the return edge.

$$
\begin{array}{ll}
r_1 = \frac{1}{2}(x-1) & r_{i+1} = \frac{1}{2}(x+1)r_i + \frac{1}{2}(x-1)t_i \\
t_1 = x-2 & t_{i+1} = (x-1)t_i + xr_i \\
o_1 = x-1 & o_{i+1} = xo_i + (x-1)t_i \\
e_1 \geq \frac{5}{2}(x-1) & e_{i+1} \geq (x-4)e_i + 4\tilde{e}_i + (x-1)t_i + t_{i+1} + r_{i+1} - r_i = \\
 & \quad (x-4)e_i + 4\tilde{e}_i + \frac{5}{2}(x-1)t_i + \frac{3x-1}{2}r_i \\
\tilde{e}_1 \geq x-1 & \tilde{e}_{i+1} \geq (x-5)e_i + 5\tilde{e}_i + (x-1)t_i
\end{array}
$$

Consider the highest-level ring of blocks of level k. As there are two special blocks in an optimal algorithm (the starting one, and its left neighbour), the exploration cost is at least $(x-2)e_k + 2\tilde{e}_k + xt_k$. The optimal traversal cost (of an algorithm using a map of the graph) is $xo_k + xt_k$. Putting this in ratio yields the approximation factor

$$\frac{(x-2)e_k + 2\tilde{e}_k + xt_k}{xo_k + xt_k} = \frac{5}{2}$$

Corollary 1. *The competitive ratio of any deterministic algorithm on n-vertex graphs is at least $\frac{5}{2} - 2^{-O(\sqrt{\log n})}$.*

3 Upper Bounds

Let M be the weight of a Minimum Spanning Tree (MST) of G. We design the exploration algorithm to incur cost $O(M)$ and hence achieve constant approximation ratio. Intuitively the algorithm classifies edges of G into groups depending on their weight and leads (by providing $O(n)-$ bits of advice) the agent to explore G by traversing components of an MST and some not very 'heavy-weight' edges connecting those components. All the advice described is stored in self-delimited way; this ensures that the cost of an advice of number s is $O(\log s)$ regardless of the potential range the value s is from. As a first step, the advice given is n and $l = \lceil \log(M/n) \rceil$. Although l can be unbounded w.r.t. n, it can be encoded in $O(\log n)$ bits in the following way: Advice (n', p, l') is given, interpreted as follows: Keep exploring the cheapest outgoing edge from the currently explored subgraph until n'-th vertex is encountered. Consider its p-th incident edge e, let $w(e)$ be its weight. Then $l = \lceil \log w(e) \rceil + l'$. (n', p, l') are chosen in such way that e is the first encountered edge of weight between M/n^2 and M. Note that such an edge must exists, otherwise the MST weight would not be M. Observe that $O(\log n)$ bits are sufficient to encode (n', p, l') and the total exploration cost until e is found is $O(M)$: a)$l' \leq \log n$, and b)the cost of reaching the cheapest outgoing edge is $O(M/n)$ since each so-far explored edge is of weight at most $O(M/n^2)$, and this has to be repeated for at most n times until a heavier edge is found.

Define for each edge e its level $l(e)$ as follows: If $\log(w(e)) < l$, then $l(e) = 0$, otherwise $l(e) = \lceil \log(w(e)) \rceil - l$. Note that in the MST there at most $n/2^i$ edges of level i. Define G_i to be the graph induced in G by the edges of level at most i. Denote by $G_i(v)$ the connected component of G_i containing vertex v.

From a high level view, the algorithm tries to mimic the MST of G, with the following modifications: a) The G_0 components are explored using DFS, as the total overhead in them w.r.t. to the MST is $O(M)$, and b) for each level-i MST edge connecting two G_0 components, the algorithm is able to identify (and traverse) a level i edge (let us call it a *tree edge*) connecting them, incurring cost of at most 6 level-i edge traversals (leading to $O(M)$ overall cost).

The main problem is that the tree edges cannot be encoded explicitly, as that might cost $O(n \log n)$ advice bits. Note that all special nodes that need advice to be stored in them can be encoded in $O(n)$ bits, by storing in each special node the number of newly visited nodes to skip until the next special node is encountered. Hence, it is sufficient to focus on how to efficiently (in terms of advice size and incurred traversal cost) identify the tree edges incident to a source vertex v. As there are at most $n/2^i$ tree edges of level i, we can afford to spend $O(i)$ bits per tree edge of level i; $O(\log i)$ bits are in fact sufficient for our algorithm.

Let us call an edge *unexplored* if one of its endpoints has not yet been visited by the agent. A level-i edge (u, v) is an out_i edge if $v \notin G_{i-1}(u)$, otherwise it is an in_i edge. In the easiest case, all level-i unexplored edges incident to v are out_i edges. This is indicated by advice $\{Out, i, 0\}$ at v. In such case, the algorithm can safely cross the incident level-i unexplored edges and recursively explore the corresponding components. However, it might be the case that there are in_i edges incident to v. In order to avoid taking them (and paying unnecessarily high cost), a $\{Wait, i, 0\}$ advice is given. In such case, the algorithm ignores for now the level-i edges incident to v, with the promise that it will return later when only out_i edges remain unexplored. The right moment to return to v is when the last in_i edge (v, w) incident to v becomes explored (i.e., when the agent arrives to w). In such case, we say that w is the trigger vertex at level-i for v. In fact, w can be a trigger vertex for several vertices. This is indicated by an advice tuple $\{Trigger, i, mult\}$ at w, where $mult$ is the number of vertices for which w is trigger. It is too costly to store explicitly for which of w's neighbours it is the trigger. Instead, w checks how many of its neighbours are waiting for trigger. If that number is equal to $mult$, then w knows that it can trigger all its level-i neighbours that are waiting for the trigger. However, it may be the case that w has more level-i neighbours that are waiting for level-i trigger. In such case, the exploration proceeds without triggering, with the promise that once w learns whom to trigger (we call it that w is *released*), it will do so. A vertex v being triggered (at level i) by w means that the agent travels from w to v, explores the level-i edges incident to v and returns to w. Before the return to w, the agent notifies[3] all level-i triggers incident to v that v is not waiting anymore for a trigger. This might release some triggers. In such case, the agent will visit the released triggers from v before returning to w. In fact, such triggering and releasing can cascade several levels. Nevertheless, we will show that the overall cost is still $O(M)$. The pseudocode of the algorithm is given in Algorithm 1 and Algorithm 2.

[3] Note that this is just an internal calculation in the agent's data structures, no actual traversal is needed.

Algorithm 1. Exploration with linear advice

```
1: procedure ExploreWithAdvice
2:     Initialize n, M and l as described above, finishing at vertex v
3:     Let next, Waiting(v, lvl) and Trigger(v, lvl) be global variables
4:     next ← ReadAdvice(integer)
5:     call Expand(v)
6:     return to the starting vertex
7: end procedure
1: procedure Expand(v)
2:     next ← next − 1
3:     if next = 0 then
4:         next ←ReadAdvice(integer)
5:         adviceList ←ReadAdvice(list of triples of integers)
6:         call ExpandNeighbours(v, 0)
7:         for every tuple {type, lvl, mult} in adviceList do
8:             if type = Out then
9:                 call ExpandNeighbours(v, lvl)
10:            else if type = Trigger then
11:                W ← {u : l((v, u)) = lvl ∧ Waiting(u, lvl) = True}
12:                if mult = |W| then
13:                    call TriggerNeighbours(v, lvl, W)
14:                else
15:                    Trigger(v, lvl) ← mult
16:                end if
17:            else                                          ▷ type = Wait
18:                Waiting(v, lvl) ← True
19:            end if
20:        end for
21:    else
22:        call ExpandNeighbours(v, 0)
23:    end if
24: end procedure
```

In order to complete the description, the advice given (i.e., which vertices are special, and for which levels) has to be specified. Unfortunately, it is not possible to reflect in a straightforward manner the structure of the MST edges connecting the G_0 components: Consider the scenario shown in Figure 3. In this case, both $G_{i-1}(w)$ and $G_{i-1}(w')$ will be reached from v, although (u, w) might be the MST edge. The solution is simple: drop u as a special vertex, the cost of reaching w from v is at most twice the cost of reaching it from u. More generally, the special vertices can be computed as follows:

Simulate the run of the exploration algorithm and whenever you come to a vertex v with out_i edges, mark it as special for level i and add the corresponding tuple to the advice. Note that this may create many connections to the same G_0 component C, potentially substantially increasing advice size. However, these connections will all stop to be out_i edges when C is visited and fully explored (note that the agent returns from a component only after it has been fully

Algorithm 2. Exploration with linear advice – helper procedures

```
 1: procedure ExpandNeighbours(v, lvl)
 2:     for all incident edges (v, u) of level lvl do
 3:         if u has not yet been expanded then
 4:             go to u
 5:             call Expand(u)
 6:             return to v
 7:         end if
 8:     end for
 9:     if lvl > 0 then
10:         Waiting(v, lvl) ← False
11:         T ← {u : l((u, v)) = lvl ∧ Trigger(u, lvl) > 0}
12:         for all u ∈ T do
13:             Trigger(u, lvl) ← Trigger(u, lvl) − 1
14:             W ← {w : l((u, w)) = lvl ∧ Waiting(u, lvl) = True}
15:             if Trigger(u, lvl) = |W| then
16:                 go to u
17:                 call TriggerNeighbours(u, lvl, W)
18:                 return to v
19:             end if
20:         end for
21:     end if
22: end procedure
 1: procedure TriggerNeighbours(v, lvl, W)
 2:     for all u ∈ W do
 3:         go to u
 4:         call ExpandNeighbours(u, lvl)
 5:         return to v
 6:     end for
 7:     Trigger(v, lvl) ← 0
 8: end procedure
```

explored). If it happens that all out_i edges of a special vertex v stop being out_i before v had a chance to explore them, the tuple for v, i (and possibly the corresponding Trigger tuple) are removed from the advice. This leaves at most $n-1$ Out/Wait tuples (plus corresponding Trigger tuples) in the advice. Let us classify the edges traversed by the algorithm as follows: i) *traverse edges*: the edges traversed on lines 4 and 6 of ExpandNeighbours, ii) *trigger edges*: the edges traversed on lines 3 and 5 of TriggerNeighbours, and iii) *release edges*: the edges traversed on lines 16 and 18 of ExpandNeighbours. Note that no other edges are traversed by the algorithm.

Lemma 2. *The traverse edges form a spanning tree of G of weight $O(M)$, while the total advice used by the algorithm is of size $O(n)$ bits.*

Note that each release edge can be charged to a corresponding trigger edge, which itself can be charged to the corresponding traversal edge (all of them of the same level). Combined with Lemma 2 this yields:

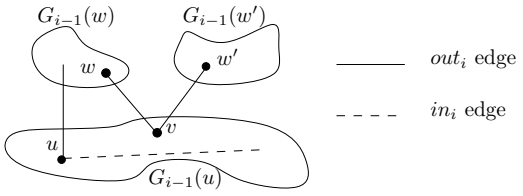

Fig. 3. u is visited before v, but v is the first one to expand its neighbours

Theorem 3. *Algorithm 1 explores an unknown n-node graph using advice of size $O(n)$ and incurring cost linear in the optimal exploration cost.*

A more careful analysis shows that exploration cost is at most $3W$ times the weight of the MST, where W is the ratio between the weights of the cheapest and the costliest edge of a level. Combined with the fast that the weight of the MST itself is a 2-approximation of the optimal cost, this yields approximation ratio of $6W$. In our case, we have chosen $W = 2$ for simplicity; W can be reduced to $1 + \epsilon$ by choosing narrower levels, at the expense of increasing the advice size (by increasing the number of levels). However, for any constant ϵ, the resulting advice size is still linear, providing a $6 + \epsilon'$ approximation bound with linear advice.

4 Conclusion

The question of whether there exists a constant competitive deterministic algorithm for the exploration problem on general graphs with arbitrary weights remains open. Hence adding to an algorithm the capability of accessing an advice seems a natural step for getting positive results. This is the case especially if it turns out that the answer to the above question is negative.

Our original aim was to come up with an algorithm using a small (polylogarithmic) advice and achieving constant competitive ratio. The hope was that such algorithm can perhaps be adapted to not need the advice at all. However, we did not succeed in this task and were only able to provide an algorithm using $O(n)$ bits of advice.

The principal problem lies in the fact that the algorithm has to make many decisions over the whole network. Hence, even reducing the advice to $o(n)$ requires new insight and would be a significant progress. From the lower bound side, we were able to raise the lower bound from $2-\epsilon$ to $5/2-\epsilon$, although breaking the barrier of 2 required use of many different edge weights and elaborate hierarchial construction. The difficulties in raising the lower bound give hope that perhaps the answer to the original question is positive. Aside from the very strict lower bound on advice size for optimal algorithms, the question of lower bound tradeoff between the advice size and competitive ratio is interesting on itself and remains widely open.

References

1. Asahiro, Y., Miyano, E., Miyazaki, S., Yoshimuta, T.: Weighted nearest neighbor algorithms for the graph exploration problem on cycles. Information Processing Letters 110(3), 93–98 (2010)
2. Böckenhauer, H.-J., Komm, D., Královič, R., Královič, R.: On the Advice Complexity of the k-Server Problem. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part I. LNCS, vol. 6755, pp. 207–218. Springer, Heidelberg (2011)
3. Böckenhauer, H.-J., Komm, D., Královič, R., Královič, R., Mömke, T.: On the Advice Complexity of Online Problems. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 331–340. Springer, Heidelberg (2009)
4. Borodin, A., El-Yaniv, R.: Online Computation and Competitive Analysis, vol. 2. Cambridge University Press (1998)
5. Dereniowski, D., Pelc, A.: Drawing maps with advice. J. Parallel Distrib. Comput. 72(2), 132–143 (2012)
6. Dobrev, S., Královič, R., Pardubská, D.: Measuring the problem-relevant information in input. ITA 43(3), 585–613 (2009)
7. Emek, Y., Fraigniaud, P., Korman, A., Rosén, A.: Online computation with advice. Theor. Comput. Sci. 412(24), 2642–2656 (2011)
8. Flocchini, P., Mans, B., Santoro, N.: Sense of direction in distributed computing. Theor. Comput. Sci. 291(1), 29–53 (2003)
9. Fraigniaud, P., Gavoille, C., Ilcinkas, D., Pelc, A.: Distributed computing with advice: information sensitivity of graph coloring. Distributed Computing 21(6), 395–403 (2009)
10. Fraigniaud, P., Ilcinkas, D., Pelc, A.: Impact of memory size on graph exploration capability. Discrete Applied Mathematics 156(12), 2310–2319 (2008)
11. Fraigniaud, P., Ilcinkas, D., Pelc, A.: Communication algorithms with advice. J. Comput. Syst. Sci. 76(3-4), 222–232 (2010)
12. Fraigniaud, P., Korman, A., Lebhar, E.: Local mst computation with short advice. Theory Comput. Syst. 47(4), 920–933 (2010)
13. Fusco, E.G., Pelc, A.: Trade-offs between the size of advice and broadcasting time in trees. Algorithmica 60(4), 719–734 (2011)
14. Gutin, G., Punnen, A.P.: The Traveling Salesman Problem and Its Variations. Springer, Heidelberg (2002)
15. Hromkovič, J., Královič, R., Královič, R.: Information Complexity of Online Problems. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 24–36. Springer, Heidelberg (2010)
16. Hurkens, C.A., Woeginger, G.J.: On the nearest neighbor rule for the traveling salesman problem. Operations Research Letters 32(1), 1–4 (2004)
17. Ilcinkas, D., Kowalski, D.R., Pelc, A.: Fast radio broadcasting with advice. Theor. Comput. Sci. 411(14-15), 1544–1557 (2010)
18. Kalyanasundaram, B., Pruhs, K.R.: Constructing competitive tours from local information. Theoretical Computer Science 130(1), 125–138 (1994)
19. Megow, N., Mehlhorn, K., Schweitzer, P.: Online Graph Exploration: New Results on Old and New Algorithms. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 478–489. Springer, Heidelberg (2011)
20. Miyazaki, S., Morimoto, N., Okabe, Y.: The online graph exploration problem on restricted graphs. IEICE Transactions on Information and Systems 92(9), 1620–1627 (2009)
21. Rosenkrantz, D.J., Stearns, R.E., Lewis II, P.M.: An analysis of several heuristics for the traveling salesman problem. SIAM J. Comput. 6(3), 563–581 (1977)