Constant-Factor Approximation Algorithm for Binary Search in Trees with Monotonic Query Times

Dariusz Dereniowski
Faculty of Electronics, Telecommunications and Informatics,
Gdańsk University of Technology, Poland

Izajasz Wrosz
Faculty of Electronics, Telecommunications and Informatics,
Gdańsk University of Technology, Poland
Intel, Gdańsk, Poland

Abstract

We consider a generalization of binary search in linear orders to the domain of weighted trees. The goal is to design an adaptive search strategy whose aim is to locate an unknown target vertex of a given tree. Each query to a vertex v incurs a non-negative cost $\omega(v)$ (that can be interpreted as the duration of the query) and returns a feedback that either v is the target or the edge incident to v is given that is on the path towards the target. The goal of the algorithm is to find a strategy that minimizes the worst-case total cost. We propose a constant-factor approximation algorithm for trees with a monotonic cost function. Such function is defined as follows: there exists a vertex r such that for any two vertices u, v on any path connecting r with a leaf it holds that if u is closer to r than v, then $\omega(u) \geq \omega(v)$. The best known approximation algorithm for general weight functions has the ratio of $\mathcal{O}(\sqrt{\log n})$ [Dereniowski et al. ICALP 2017] and it remains as a challenging open question whether constant-factor approximation is achievable in such case. This gives our first motivation towards considering monotonic cost functions and the second one lies in the potential applications.

2012 ACM Subject Classification Theory of computation; Theory of computation → Design and analysis of algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases binary search, graph search, approximation algorithm, query complexity

Digital Object Identifier 10.4230/LIPIcs.MFCS.2022.42

Funding Work partially supported by National Science Centre (Poland) grant number 2018/31/B/ST6/00820.

1 Introduction

We study a natural generalization of the classical binary search problem, where the algorithm is asked to locate a specific element in a sorted array of keys with as few comparisons as possible [20]. We generalize the binary search in two dimensions. Instead of arrays, we consider trees, with the keys represented as vertices of the tree. Additionally, we allow the cost of comparisons to vary across the vertices of the tree. As a result, instead of minimizing the worst-case number of queries, our objective is to minimize the worst-case total cost of the search. This problem has been introduced for trees in [26] and for general graphs in [15]. There are two characteristics of such search algorithms: the computational complexity of calculating a search strategy and the worst-case cost called the *query complexity*.

As a fundamental problem in computer science, binary search in trees and graphs has numerous practical applications in data management systems or scheduling of parallel processes (through graph coloring), machine learning, and other fields. We discuss the relevant practical problems later on in more detail.

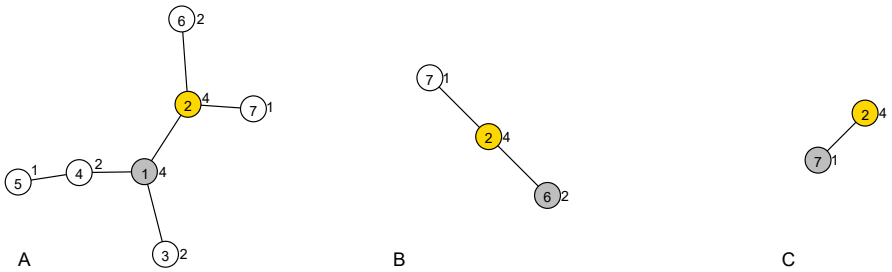

Figure 1 Binary search on weighted trees. The input tree A contains a target vertex (2), whose position is unknown to the algorithm. A series of queries is performed to the vertices 1, 6, and 7, which incurs a cost equal to 4, 2 and 1, for each query respectively. As a result of the subsequent queries, subtrees B and C are generated that consist of all vertices that may still contain the target, given the information revealed by previous queries.

1.1 Problem statement

An a priori unknown target vertex t in a known input tree $T = (V, E, \omega)$, with a query cost function $\omega\colon V(T) \to \mathbb{R}_+$, should be located by an algorithm via a series of queries. Each query selects a vertex v and as an answer receives information that either v is the target (which completes the search), or otherwise it is given an edge $\{v, u\}$ such that u lies on a path between v and t. (Note that in the case of a tree the path is unique but for general graphs, any of the shortest paths is provided, see [15].) The answer is generated at the cost of $\omega(v)$ and we want to design an algorithm that finds the target with the minimum cost in the worst case. More precisely, once the target has been returned by the algorithm, the cost of the search equals the sum of the costs of all queries that have been asked, and the performance of the algorithm is the maximum search cost taken over all vertices as potential targets. One might think about this search process that with each query the search is narrowed to a subtree of T that can still contain the target. The process is *adaptive* in the sense that the choice of the subsequent elements to be queried depends on the locations and answers of the previous queries. Also, we consider only *deterministic* algorithms, where given a fixed sequence of queries performed so far (and the information obtained), the algorithm selects always the same element as the next query for a given tree. E.g., the optimal algorithm for the classic binary search problem on paths that always queries the median element is adaptive and deterministic. It can be equivalently stated that the algorithm calculates a *strategy* for the given input tree, which encodes the queries to be performed for each potential target.

1.2 Related work and earlier techniques

While in this work we analyze queries that ask questions about vertices of a tree (vertex query model), an edge query model has been also studied. In the latter, the answer to each queried edge $e \in E(G)$ returns one of the two subgraphs in $G \setminus \{e\}$ which contains the target. The two search models are basically equivalent for paths, but in general, a query to a vertex with a high degree reveals more information than a query to an incident edge. For example, to locate the target in a star, one query is sufficient in the vertex model, while in the edge model all edges need to be queried in the worst case. The binary search can be further generalized to general graphs, where the answer to a vertex (or edge) query returns directional information with respect to the *shortest* path to the target [15].

It should be noted that other optimization criteria can be considered (e.g., the average search cost), as well as noisy search models where the response to a query can be incorrect with a certain probability.

For paths, a natural dynamic programming approach obtains an optimal algorithm for the weighted version of the problem, which runs in $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ space [19]. The cubic time can be improved by exploiting a monotonic structure of search costs of the sub-intervals, which yields an $\mathcal{O}(n^2)$ time solution [5], for any assignment of the query costs. A linear time $(2+\epsilon+o(1))$-approximation algorithm has been presented in [21], where the approach is based on a trade-off between querying the vertices with small costs and maintaining a balanced decision tree.

For unweighted trees, optimal search strategies can be computed in linear time by calculating vertex rankings of the input tree [23, 26], which is a type of graph coloring, where on a path between any two vertices with the same color there must be a vertex with a color of a higher value [17, 18]. Once the ranking is calculated, the search strategy is obtained by always querying the vertex with the highest color, considering the set of vertices that can still contain the target. Tree rankings have been studied as an independent line of research leading to the discovery of several algorithms, including linear time algorithms [22, 28].

For weighted trees, the binary search problem becomes NP-complete [12]. A quasi polynomial-time approximation scheme was obtained through a dynamic programming approach [8]. The QPTAS is then recursively applied to carefully selected subtrees of the input tree, which results in a polynomial-time algorithm with an approximation ratio of $\mathcal{O}(\sqrt{\log n})$.

The binary search problem has been extended to general graphs in [13, 15], where a function Φ is defined over the vertex set of the graph that can still contain the search target, given the responses to the queries performed so far. The Φ is defined in such a way that for each considered vertex it encodes the sum of distances to all other considered vertices. In each step, the vertex that minimizes Φ is selected as the next query, and the search space is reduced to the vertices consistent with the query response, i.e., the vertices with a shortest path from the queried vertex containing the edge returned by the query. The algorithm uses at most $\log_2 n$ queries to find arbitrary vertex in a graph with a uniform cost of queries. A further study of using graph median for queries can be found in [11].

In the context of practical applications, a related search model has been used to characterize the confidentiality of searching information in databases of genetics information [16].

1.3 Our contribution

An open question has been posed several times about whether a constant-factor approximation algorithm exists for arbitrary weight functions, see e.g. [2, 4, 5]. The main motivation of this work is to address this question by finding a natural input instance for which a constant-factor approximation is achievable. We define this instance as follows: we allow an input tree to have arbitrary structure but the weight function is assumed to be monotonic:

▶ **Definition 1.** *Given a tree $T=(V,E,\omega)$, we say that a cost function ω is* monotonic *if there exists a vertex $r\in V$ such that for any $u,v\in V$, if v lies on the path between r and u in T, then $\omega(u)\leq\omega(v)$.*

Our primary contribution is in providing a constant-factor approximation algorithm for the binary search problem in this subclass of trees with non-uniform weights. Our solution shows how a method developed for the problem with uniform costs [26] can be leveraged for the monotonic case. Our main result is the following.

▶ **Theorem 2.** *There exists an 8-approximate adaptive search algorithm for the search problem in weighted trees with monotonic cost functions. The algorithm runs in linear time.*

The organization of the remaining parts of the paper is as follows. Section 1.4 describes the motivation for our work, based on practical applications of the studied search model and several use cases of the monotonic structure of the cost function. Section 2 introduces our notation and essential concepts commonly used in the related body of research. Section 3 presents the class of decision trees (structured decision trees) that we restrict ourselves to when building the solution for the search problem. Section 4 introduces further concepts, which are needed to formulate our algorithm in the subsequent Section 5 and to prove the approximation factor in Section 6. Section 7 concludes the paper, where we suggest potential directions for future research.

1.4 Motivation and applications

Graphs form a natural abstraction for processes in many domains. Large graphs are becoming common in data management and processing systems [27]. Compared to data structures without order over its elements, it is known that maintaining at least a partial order improves the performance of fundamental operations like search, update or insert in terms of the number of comparisons needed [20]. At the same time, creating optimal strategies for searching in structures with a partial order, can be inherently hard [12]. We conclude that efficient approximation algorithms for searching in large graphs are important for the advancement of systems focused on large graph analysis.

The classic binary search and its generalizations is a basic problem in computer science that occurs in numerous practical problems. The binary search problem posed as a graph ranking problem can be used to model parallel Cholesky factorization of matrices [9], scheduling of parallel database queries [10], and VLSI layouts [29]. The binary search model for graphs has been used to build a general framework for interactive learning of classifiers, rankings, or clusterings [14].

A monotonic structure of the comparisons cost occurs naturally when considering data access times in computer systems, e.g., due to memory hierarchies of modern processors, characteristics of the storage devices, or distributed nature of data management systems. In the literature on the binary search problem, a Hierarchical Memory Model of computation has been studied [1], in which the memory access time is monotonic with respect to the location of a data element in the array. Another work was devoted to modeling the problem of text retrieval from magnetic or optical disks, where a cost model was such that the cost of a query was monotonic from the location of the previous query performed in the search process [24]. We also mention an application of tree domains in automated bug search in computer code [3]. In such a case naturally occurs a possibility that the tree to be searched has the monotonicity property. In particular, each query represents performing an automatic test that determines whether the part of the code that corresponds to the subtree under the tested vertex has an error. Thus, the vertices that are closer to the root represent larger parts of the code and thus may require more or longer lasting tests.

2 Preliminaries

The set of vertices of a tree T is denoted by $V(T)$. Given a tree $T = (V, E, w)$ and a connected subset of vertices $V' \subseteq V(T)$, we denote by $T[V']$ the induced subtree of T consisting of all vertices from V'. For a rooted T, we denote the root of T by $r(T)$, while for any $v \in V(T)$, T_v denotes the subtree of T rooted at v and consisting of v and all its descendants. For any two intervals $I, I' \in \{[a, b) \colon 0 \leq a < b\}$ we write $I > I'$ when $i > i'$ for each $i \in I$ and $i' \in I'$.

The *sequence of queries* is defined as a sequence of vertices queried by a search strategy $\mathcal{A}$ for a target t and is denoted by $\mathcal{Q}_{\mathcal{A}}(T,t)$. The vertex queried by $\mathcal{A}$ in the i-th step is denoted by $\mathcal{Q}_{\mathcal{A},i}(T,t)$. Given a search strategy for T, one can easily generate a search strategy for any subtree T' by simply discarding the queries to the vertices outside of T'. A useful way of encoding search strategies is by using decision trees.

▶ **Definition 3.** *We define a* decision tree *for $T=(V,E,w)$ generated by a search strategy $\mathcal{A}$ as a rooted tree $\mathcal{T}=(\mathcal{V},\mathcal{E})$, where $\mathcal{V}=V(T)$. The root of $\mathcal{T}$ is the first vertex queried by $\mathcal{A}$. For any $v\in\mathcal{V}$, v has a child v' if and only if v' is queried by $\mathcal{A}$ right after v for some choice of the target. Each $v\in\mathcal{V}$ corresponds to a subset $\mathcal{T}(v)\subset V(T)$ which contains all vertices that can still contain the target when $\mathcal{A}$ queries v.*

It can be seen that the root r of any decision tree corresponds to $\mathcal{T}(r)=V(T)$ and the leaves correspond to single vertices in $V(T)$.

Lemma 4 shows a property of the decision trees with respect to the positions of vertices that belong to some connected component of the input tree.

▶ **Lemma 4.** *Let $\mathcal{T}$ be a decision tree for T. If T' is a subtree of T, then there exists a vertex $u\in V(T')$, such that for every $v\in V(T')$ it holds $v\in V(\mathcal{T}_u)$.*

Proof. We traverse the decision tree $\mathcal{T}$ starting at the root of $\mathcal{T}$ and following a path to some leaf. If $r(\mathcal{T})\in V(T')$, then the lemma follows. Otherwise, consider children v_i of $r(\mathcal{T})$. Because in T there is only one edge incident to $r(\mathcal{T})$ that lies on a path to all vertices of T', all vertices of T' belong to exactly one subtree $\mathcal{T}_{v_i}$. We continue the traversal in $\mathcal{T}_{v_i}$ and set u as the first visited vertex in this process that belongs to T'. ◀

We now formally define the cost of a search process, which is the property the algorithm is trying to minimize. In terms of the search strategy $\mathcal{A}$ for an input tree $T=(V,E,\omega)$, the sum of costs of all queries performed when finding a target $t\in V(T)$ is denoted by $COST(\mathcal{A},t)$, while the *cost* of $\mathcal{A}$ is defined as $COST(\mathcal{A})=\max_{t\in V}COST(\mathcal{A},t)$.

Sometimes in our analysis, we will take one decision tree and measure its cost with different query times, i.e., for different weights. This approach will allow us to artificially increase the cost of some queries in the decision tree, which we formalize as follows. Let $\mathcal{T}$ be a decision tree for $T=(V,E,\omega)$ and ω' an arbitrary cost function defined on $V(T)$, potentially a different one than ω. We denote by $COST(\mathcal{T},\omega')$ the *cost of the decision tree according to the cost function ω'*:

$$COST(\mathcal{T},\omega')=\max\{\sum_{v\in P}\omega'(v)\mid P \text{ is a path from the root to a leaf in } \mathcal{T}\}.$$

The *cost* of $\mathcal{T}$ is then $COST(\mathcal{T},\omega)$ and we shorten it to $COST(\mathcal{T})$. (Note that one may equivalently define the cost of $\mathcal{T}$ by taking $COST(\mathcal{T})=COST(\mathcal{A})$ where $\mathcal{T}$ is generated by $\mathcal{A}$.) The minimum cost that is achievable for T is denoted by

$$OPT(T)=\min\{COST(\mathcal{T})\mid \mathcal{T} \text{ is a decision tree for } T\}.$$

We say that $\mathcal{T}$ *is optimal* if and only if $COST(\mathcal{T})=OPT(T)$.

We will use the following simple folklore lemma whose proof is given for completeness. Lemma 5 has been previously formulated in [5] in the context of a slightly different problem of edge search in weighted trees.

▶ **Lemma 5.** *If $\mathcal{T}$ is a decision tree for T, then for any subtree T' of T there exists a decision tree $\mathcal{T}'$, such that $COST(\mathcal{T}')\leq COST(\mathcal{T})$.*

Proof. We first apply Lemma 4 and reduce $\mathcal{T}$ to its subtree rooted at $u \in V(T')$. Then, we sequentially remove from $\mathcal{T}_u$ all nodes that do not belong to T'. For every removed node v, consider all subtrees rooted at the children of v in the current decision tree. When removing v we also remove all subtrees that do not contain any vertex from T' but we connect the roots of all other subtrees (note that there will be only one such subtree) directly under the parent of v. Hence, the reduced tree is still a decision tree. What is more, all paths from the root to a leaf in the obtained decision tree result from corresponding paths from $\mathcal{T}$ by removing zero or more nodes, which upperbounds the cost of $\mathcal{T}'$ by the cost of $\mathcal{T}$. ◀

We say that a cost function ω is *rounded* if the values taken by ω are powers of two (for any $v \in V(T)$, $\omega(v) = 2^k$ for some $k \in \mathbb{N}$). We also say that T' is the *rounding* of T if it is obtained from T by rounding the cost function to the closest, greater power of two. We obtain that $OPT(T') \leq 2\,OPT(T)$ because $\omega' \leq 2\omega$. On the other hand, Lemma 6 shows how large is the overhead from applying an optimal decision tree for the input with a rounded cost function to search through the input tree with the original weights.

▶ **Lemma 6.** *Let $T' = (V, E, \omega')$ be the rounding of $T = (V, E, \omega)$. Let $\mathcal{T}$ and $\mathcal{T}'$ be optimal decision trees respectively for T and T'. We have:*

$$COST(\mathcal{T}', \omega) \leq 2\,COST(\mathcal{T}) = 2OPT(T).$$

Proof. We have $COST(\mathcal{T}', \omega) \leq COST(\mathcal{T}', \omega') \leq COST(\mathcal{T}, \omega') \leq 2COST(\mathcal{T}, \omega)$. The first inequality holds because $\omega \leq \omega'$. The second inequality holds because $\mathcal{T}'$ is optimal for T'. The last inequality holds because $\omega' \leq 2\omega$. ◀

3 Structured decision trees

For a rooted tree $T = (V, E, \omega)$ with a monotonic ω, we assume that the root is selected as in Definition 1, that is informally speaking, while traversing any path from the root to a leaf, the weights of visited vertices are non-increasing. From now on we assume that the input tree is rooted. We define a *layer* of a tree T as a subgraph L of T such that all vertices in $V(L)$ have the same cost, $\omega(u) = \omega(v)$ for any $u, v \in V(L)$. A connected component of L is called a *layer component*. We also denote by $\omega(L)$ the cost of querying a vertex that belongs to L, i.e., $\omega(L) = \omega(v)$ for any $v \in V(L)$.

The *upper border* of layer L is the set of roots of its layer components. The subset of $V(L)$ consisting of vertices that have at least one child that belongs to a different layer is called the *lower border* of L. We will also say that a layer component L' is *directly below* a layer component L if and only if $r(L')$ has a parent in L. The *top* layer component is the one that contains the root of T and a *bottom* layer component is any L such that there is no component directly below L. We note that we will apply the above terms to a rounded ω.

▶ **Definition 7.** *Let T be a tree with a monotonic cost function and $\mathcal{T}$ a decision tree for T. Consider a layer component L of T. Let $v = r(L)$. We say that $\mathcal{T}$ is* structured with respect to L *if for every vertex $u \in V(T_v)$, it holds that $u \in V(\mathcal{T}_v)$. We say that a decision tree $\mathcal{T}$ for T is* structured *if and only if $\mathcal{T}$ is structured with respect to all layer components of T.*

Informally speaking, in a structured $\mathcal{T}$ we require that each vertex in the entire subtree T_v is below v in $\mathcal{T}$. This is one of the central ideas in our work for the following reason. While performing bottom-up processing of (an unweighted) T in [26] (and also other works e.g. [22]) each vertex v encodes which steps are already used for queries of the vertices from T_v. In our case, that is in a weighted T, we need to encode intervals and the technical problem with

that is that they have different durations while moving upwards from one layer component to the next. Thus, some intervals that are free to perform queries while moving to a parent of v may be too short due to the weights in the next layer component. To deal with that, we make $v = r(L)$ also the root of $\mathcal{T}_v$. In this way only one interval, i.e., the one assigned to v needs to be taken into account while moving upwards.

Our method requires the decision tree to be structured only when processing the internal components of T. By not enforcing structuring with respect to the top layer component, a potential additive cost of a single query can be avoided. See the *Process Component* procedure (Line 9, Algorithm 2). However, this optimization does not change the worst-case cost of the strategy. In result, the decision tree generated by our algorithm is not structured in general, and although $r(\mathcal{T})$, the first vertex queried by the strategy, always belongs to the top layer component of T, we may have that $r(\mathcal{T}) \neq r(T)$, while in a structured decision tree we always have $r(\mathcal{T}) = r(T)$, because $\mathcal{T}$ is structured with respect to the top layer component of T.

It turns out in our analysis that considering only structured decision trees introduces an overall multiplicative cost of 2 to the performance of the algorithm:

▶ **Lemma 8.** *Let $T = (V, E, \omega)$ be a tree with a monotonic and rounded cost function. There exists a structured decision tree $\mathcal{T}'$ for T, such that $COST(\mathcal{T}') \leq 2\, OPT(T)$.*

Proof. Let $\mathcal{T}$ be any decision tree for T. Since $\mathcal{T}$ is selected arbitrarily, it is enough to prove that $COST(\mathcal{T}') \leq 2\, COST(\mathcal{T})$.

In order to construct the structured decision tree, we traverse $\mathcal{T}$ in a breadth-first fashion starting at the root of $\mathcal{T}$. The $\mathcal{T}'$ is constructed in the process.

By Lemma 4, for an arbitrary layer component L of T, there exists a vertex $u' \in V(L)$ such that all vertices from L belong to $\mathcal{T}_{u'}$. Among all vertices of L, u' is visited first during the traversal.

Consider an arbitrary node u being accessed during the traversal. Let L be the layer component such that $u \in L$. If $\mathcal{T}$ is structured with respect to L we continue with the next node. Otherwise (which for a specific L will happen only once, when $u = u'$, as argued in the last paragraph of the proof), we modify $\mathcal{T}$ by performing the following steps.

Let $v = r(L)$. We want to replace the subtree $\mathcal{T}_u$ with a subtree consisting of the same nodes but rooted at v. Consider a subtree T' of the input T, which can still contain the target when the search process is about to query u according to $\mathcal{T}$. E.g., $V(T') = V(\mathcal{T}_u)$. Let v has k neighbors in T'. We attach under v the decision trees created with Lemma 5 for the k components obtained by removing v from T'. Lemma 5 shows that the cost of any of the k decision trees is not greater than $COST(\mathcal{T}_u)$. Accounting for the cost of v, structuring L increases $COST(\mathcal{T})$ no more than $\omega(L)$. Consider an arbitrary path P from root to a leaf in $\mathcal{T}$. For every node $u \in P$ that triggers structuring (informally, u is u' for some L) an additional node from L is inserted in P. Thus, in the worst case, the weighted length of arbitrary P increases 2 times when transforming $\mathcal{T}$ into structured $\mathcal{T}'$.

By starting the breadth-first traversal at the root of $\mathcal{T}$, structuring with regards to one layer component does not make $\mathcal{T}$ not structured with regards to any of the previously structured layer components. ◀

By combining Lemmas 6 and 8 one obtains Corollary 9, which relates the cost of an optimal decision tree for an input T with the cost of a structured decision tree obtained through the analysis of the rounding of T.

▶ **Corollary 9.** *For any tree T with a monotonic cost function there exists a structured decision tree with rounded weights, whose cost is at most $4 \cdot OPT(T)$.*

4 Bottom-up tree processing

Our method extends the ranking-based method for searching in trees with uniform costs of queries [26, 28]. The state-of-the-art algorithm calculates a function $f\colon V(T) \to \mathbb{N}$ called a *strategy function* (also called in the literature a *ranking* [22, 28], or tree-depth [25]). The strategy function is such that for each pair of distinct vertices $v_1, v_2 \in V(T)$, if $f(v_1) = f(v_2)$, then each path connecting v_1 and v_2 has a vertex v_3 such that $f(v_3) > f(v_1)$. One property of the strategy function is that it encodes the (reversed) order of the queries. E.g., in any sequence of queries, a vertex with a higher value of f is always queried before those with lower values of the strategy function. Thus, the maximal value of f over the vertices of a tree is the worst-case number of queries necessary to search the tree. In order to express the variable costs of queries we extend the strategy function into an *extended strategy function* (see also [7]).

▶ **Definition 10.** *A function $f\colon V(T) \to \{[a,b)\colon 0 \le a < b\}$, where $|[a,b)| \ge \omega(v)$ for each $v \in V(T)$, is an* extended strategy function *if for each pair of distinct vertices $v_1, v_2 \in V(T)$, if $f(v_1) \cap f(v_2) \neq \emptyset$, then the path connecting v_1 and v_2 has a v_3 such that $f(v_3) > f(v_1) \cup f(v_2)$. The length of an interval assigned to a vertex v encodes the cost corresponding to querying v.*

Keeping in mind that the cost function is the time needed to perform a query, the values of an extended strategy function indicate the time periods in which the respective vertices will be queried. Note that, e.g., due to the rounding, the time periods will be longer than the query durations and this is easily resolved either by introducing idle times or by following the actual query durations while performing a search based on a strategy produced by our algorithm.

We say that $u \in V(T)$ is *visible from* $v \in V(T)$ if on the path from v to u there is no vertex x such that $f(x) > f(u)$, where f is a strategy function defined on $V(T)$. If u is visible from v we also say that the value $f(u)$ is visible from v. The sequence of values visible from v in descending order is called a *visibility sequence* for v.

In our approach, informally speaking, we will partition the rounding of the input tree into several subtrees (the layer components) and for each of them, we will use the algorithm from [26]. To clearly describe the initialization required prior to triggering the leveraged subroutine, and also for completeness, we recall a *vertex extension operator* used in [26] for calculating a strategy function. Let f be a strategy function defined on all vertices below $v \in V(T)$. Let v_i, $1 \le i \le k$, be the children of v. The vertex extension operator attributes v with $f(v)$, based on the visibility sequences $S(v_i)$ of the children according to the following procedure. Consider a set M with values that belong to at least two distinct visibility sequences $S(v_i)$. Let m be the maximum value in M if $M \neq \emptyset$ or -1 otherwise. The $f(v)$ is set to the lowest integer greater than m that does not belong to any $S(v_i)$. The following lemma characterizes the vertex extension operator.

▶ **Lemma 11** ([26]). *Given the visibility sequences assigned to the children of $v \in V(T)$, the* vertex extension operator *assigns to v a visibility sequence that is lexicographically minimal (the vertex extension operator is minimizing). Moreover, (lexicographically) increasing the visibility sequence of a child of v does not decrease the visibility sequence calculated for v (the vertex extension operator is monotone).*

In our approach, we restrict ourselves to extended strategy functions which generate decision trees that are structured (see Definition 7).

▶ **Definition 12.** *We say that an extended strategy function f, defined on the vertices of a tree T with monotonic cost function, is* structured *if for any layer component L, $f(u) > f(v)$ for each $v \in V(T_u)$, where $u = r(L)$.*

It follows that a structured extended strategy function represents a structured decision tree.

As mentioned earlier, we will process each layer component L with the vertex extension operator but in order to do it correctly, we need to initialize appropriately the roots of the layer components below L. For that, we define the following operators. Let f be an extended strategy function defined on the vertices of T_v, where v is the root of a layer component L'. The *structuring operator* assigns to v the minimal interval so that f is a structured extended strategy function on T_v. (We note that in our algorithm v will have some interval assigned when the structuring operator is applied to v. Hence, the application will assign the new required interval.) The *cost scaling operator* aligns the interval attributed to v: if $f(v) = [a, b)$ and L is the layer component directly above L', the cost scaling operator assigns v a new interval $[\omega(L)\lceil \frac{b}{\omega(L)} \rceil - \omega(L), \omega(L)\lceil \frac{b}{\omega(L)} \rceil)$. We will say that the interval $f(v)$ is *aligned to* $\omega(L)$ after this modification. Informally speaking, this is done so that the children of the leaves in L have intervals whose endpoints are multiples of $\omega(L)$ so that they become "compatible" with the allowed placements for intervals of the vertices from L. Yet in other words, this allows us to translate the intervals of the children of the leaves in L into integers that can be treated during bottom-up processing of L in a uniform way.

▶ **Observation 13.** *Let a layer component L' be directly below a layer component L in $T = (V, E, \omega)$. If $r(L')$ is assigned an interval $[a, b)$, whose endpoints are consecutive multiples of $\omega(L')$ (i.e. $a = k\omega(L')$ and $b = (k+1)\omega(L')$ for some integer k), then the cost scaling operator assigns to $r(L')$ an interval $[a', b')$ such that $b' \leq b + \omega(L) - \omega(L')$.*

Proof. Consider a layer component L of $T = (V, E, \omega)$ and a structured extended strategy function f defined on the vertices below L. The cost scaling operator selects b' as the smallest multiple of $\omega(L)$ that is greater or equal than b. Since $a' = b' - \omega(L)$, we have that $a' < b$. On the other hand, because $b - a = \omega(L')$, a is the highest multiple of $\omega(L')$ that is less than b. Subsequently, because $w(L)$ is divisible by $w(L')$, we obtain that $a' \leq a$. What follows, $a' + \omega(L') \leq b$, and finally $b' \leq b + \omega(L) - \omega(L')$. ◀

5 Algorithm

We propose an algorithm *Structured Tree Search* (Algorithm 1). The algorithm starts with a pre-processing pass, which transforms the input tree to its rounding and selects a root arbitrarily from the set of vertices with the highest cost o query (cf. Definition 1). An extended strategy function f is then calculated during a bottom-up traversal over the layer components of the tree by applying the *Process Component* procedure (Algorithm 2). Hence, the extended strategy function for the vertices of a layer component L is calculated only after it is already defined for all vertices below the component. Moreover, since we use a structured f, only its values assigned to the roots of the layer components below L are important when calculating f for the vertices of L.

Given a layer component L, the procedure *Process Component* iterates over the vertices of L in a depth-first, postorder fashion. Suppose that a structured extended strategy function f is defined on all vertices below L and that f is aligned to $\omega(L)$ at all roots of layer components directly below L. To calculate f for the vertices of L, we first calculate the strategy function f' for each vertex of L and then obtain f from f' using the formula: $f(u) = [f'(u)\omega(L), (f'(u) + 1)\omega(L))$, where $u \in V(L)$. (Recall that this is a conversion

Algorithm 1 Structured Tree Search.

Let T be an input tree with a monotonic cost function, for which the extended strategy function f is calculated.
$T \leftarrow$ the rounding of T
for each layer component L in bottom-up fashion **do**
    ProcessComponent(L, f)
end for
return f

that takes us from the integer-valued strategy function f' to the interval-valued extended strategy function.) To provide an applicable input for the vertex extension operator at the leaves of L, for each child $u \in V(L')$ of such leaf, we calculate the integer $f'(u) = \frac{b}{\omega(L')}$ that is derived from the extended strategy function at u, i.e. from $f(u) = [a, b)$. Note that since our decision trees are structured, the visibility sequence corresponding to each root of a layer component below L (which includes u) consists of only one value, the one that equals the strategy function at the root. What follows, deriving the value of f' only for the roots of the layer components directly below L is sufficient to create a valid input for the vertex extension operator. In other words, we make these preparations to use the operator and the corresponding method from [26]. Once each vertex $v \in V(L)$ obtains the corresponding extended strategy function $f(v)$, we apply for the root of L the structuring operator followed by the cost scaling operator. This will close the entire "cycle" of processing one layer component.

Algorithm 2 Calculates extended strategy function f for vertices of a layer component L.

procedure ProcessComponent(L, f)
    for each root v of a layer component directly below L **do**
        $f'(v) \leftarrow \frac{b}{\omega(L)}$, where b is the right endpoint of the interval $f(v)$
    end for
    for each $u \in V(L)$ in a postorder fashion **do**
        Obtain $f'(u)$ by applying the vertex extension operator to v
        $f(u) \leftarrow$ the interval derived from $f'(u)$, see Section 4
    end for
    if the root v of L is not the root of T **then**
        Update $f(v)$ by applying the structuring operator to v
        Update $f(v)$ by applying the cost scaling operator to v
    end if
end procedure

6 Analysis

Lemma 14 characterizes the interval of the extended strategy function that the *Process Component* procedure assigns to the root of a layer component.

▶ **Lemma 14.** *Consider a layer component L of $T = (V, E, \omega)$ and a structured extended strategy function f defined on the vertices below L. Let the intervals of f assigned to the roots of the layer components directly below L be aligned to $\omega(L)$. Suppose that the values of f over $V(L)$ are obtained by a call to procedure* ProcessComponent(L,f)*. We have that f is a structured extended strategy function on $V(T_{r(L)})$, and the lowest possible interval I, such that $I > f(u)$, for each u below $r(L)$, is assigned to $r(L)$.*

Proof. Due to the alignment of f at the roots of the components directly below L, the strategy function f' derived from f for these roots is consistent with the query costs in L. That is, the construction of f' done at the beginning of the procedure gives an integer-valued strategy function. Since f is structured, among all vertices below L, the extension operator correctly needs to consider only the values of f' at the roots of components directly below L.

By Lemma 11, the visibility sequence calculated for the root of L is (lexicographically) minimized among all possible valid assignments of f'. Hence in particular, the value of $\max\{f'(v) \mid v \in V(T_{r(L)})\}$ is minimized. It follows that if there exists such an optimal assignment of f' that is structured, then this f' is calculated by the procedure is structured and consequently the f obtained from f' is minimal and structured. If there exists no optimal assignments of f' over $V(L)$ that is structured, the structuring operator modifies f and assigns to $r(L)$ the lowest interval that is greater than $f(u)$ for any u below $r(L)$. ◀

What follows from Lemma 14 and Observation 13 is that given an extended strategy function f fixed below a layer component L, Algorithm 1 extends f to the vertices of L in such a way that the interval assigned to $r(L)$ is not higher than the optimal interval (assuming that f is fixed below L) incremented by an additive factor $\omega(L') - \omega(L)$, where L' is a layer component directly above L. If L is the top component, the structuring transform is not applied and the additional cost is not incurred.

We now formulate a technical Lemma 15 that helps analyze the strategy generated by Algorithm 1.

▶ **Lemma 15.** *Let $T = (V, E, \omega)$ be a tree with a monotonic and rounded cost function. We denote by f_{opt} an optimal structured extended strategy function on $V(T)$. Let L be a layer component of T with k layer components L_j, $1 \leq j \leq k$, directly below L. Let $v_j = r(L_j), 1 \leq j \leq k$. We define f' as a function defined on $V(T)$ such that for all vertices below any v_j, f' is equal to f_{opt}, while for the vertices v_j and above, f' is equal to $f_{opt} + \omega(L)$. It holds that f' is a structured extended strategy function on $V(T)$.*

Proof. We first show that f' is an extended strategy function. We need to show that for any pair $x, y \in V(T)$, $x \neq y$, if $f'(x) = f'(y)$, then there is a vertex $z \in V(T)$ separating x, y such that $f'(z) > f'(x)$. (See Definition 10.)

We analyze the following cases with respect to the locations of the vertices x and y. In all cases we assume that $x \neq y$, $f'(x) = f'(y)$, and z is the vertex separating x and y according to f'.

- $x \in V(L_i)$, $y \in V(L_j)$ and $i = j$.
  Let $v = r(L_i)$. Because f_{opt} is structured, f' is also structured within the subtree T_v, since it assigns v a higher interval than f_{opt} and f' is otherwise identical to f_{opt}. What follows, because f' is structured and we assumed $f'(x) = f'(y)$, we have that $x \neq v$ and $y \neq v$.
  Consider the f_{opt} as defined on all vertices of L_i. Because $x, y \in V(L_i)$, we have $z \in V(L_i)$. If $z \neq v$, then $f'(z) = f_{opt}(z) > f_{opt}(x) = f'(x)$, where the last equation holds because $x \neq v$. If $z = v$, we also have $f'(z) > f'(x)$, because f' is structured in T_v.
- $x \in V(L_i)$, $y \in V(L_j)$ and $i \neq j$.
  Because f' is structured in L_i, we have $z = r(L_i)$.
- Both x and y belong to $V(L)$.
  Because f_{opt} is an extended strategy function in L, there is a $z \in V(L)$ such that $f_{opt}(z) > f_{opt}(x)$. Then, by adding $\omega(L)$ to both sides of the inequality one obtains $f'(z) > f'(x)$.

- $x \in V(L_i)$ for some i and $y \in V(L)$.
  If $x \neq r(L_i)$ then $z = r(L_i)$, because f' is structured in L_i.
  Otherwise, both x and y are in the subtree $T[V(L) \cup \{x\}]$. Recall that in this subtree f' is derived from f_{opt} by adding a constant offset, which gives an extended strategy function.

To show that f' is structured in the remaining cases, we observe that for all vertices above L, f' is derived from the structured f_{opt} by shifting the assigned intervals by a positive offset, which does not change the structural property of the layer components' roots. ◀

▶ **Lemma 16.** *The cost of the decision tree generated by Algorithm 1 is at most 2 times greater than the cost of an optimal structured decision tree.*

Proof. The proof is by induction, bottom-up over the layer components of the input tree. Take a tree $T = (V, E, \omega)$ with a monotonic and rounded cost function. We denote by $c(i), 1 \leq i \leq l$, the query cost to a vertex in the $i-th$ layer, where the layers are ordered according to their cost of query and l is the number of layers in T. Let f_{opt} be an optimal (structured) extended strategy function and f_{alg} be the extended strategy function generated by Algorithm 1. We denote by $z(v)$ the cost of querying the parent of v or the cost of v itself if there is no vertex above v, i.e. when v is the root of T.

We want to prove the following induction claim. For any layer component L it holds $f_{alg}(v) \leq f_{opt}(v) + z(v)$, where $v = r(L)$.

The cost of the decision tree generated by f_{alg} is equal to the supremum of $f_{alg}(r(T))$. Since the cost of a single query to the top layer is not more than the cost of an optimal structured decision tree, the lemma follows from the induction claim applied to the top layer component.

Let L be a bottom layer component. Based on Lemma 14 we know that by applying the vertex extension and structuring operators, Algorithm 1 generates an optimal structured assignment of the extended strategy function to the vertices of L. In a case where L is also the top component (which is the trivial case of T being a single layer component) we have f_{alg} is optimal and the induction claim holds. Otherwise, the cost scaling operator extends the interval assigned to $r(L)$, increasing the cost of L by (at most) an additive factor of $c(2) - c(1)$ over the optimal cost. Since the added factor is less than $z(r(L))$, the induction claim holds for the bottom layer component L.

Let L be a layer component from the i-th layer, with k layer components L_j, $1 \leq j \leq k$, directly below L. We denote the roots of the layer components L_j by v_j. Consider a strategy function f' which is defined as in Lemma 15, that is for all vertices below any v_j, f' is equal to f_{opt}, while for vertices v_j and all vertices above, f' is equal to $f_{opt} + \omega(L)$. Lemma 15 implies that f' is a structured extended strategy function on $V(T)$.

From the induction hypothesis we have that for any v_j, $f_{alg}(v_j) \leq f_{opt}(v_j) + z(v_j) = f_{opt}(v_j) + \omega(L)$. According to Lemma 14 and Observation 13, Algorithm 1 assigns intervals to the vertices of L such that the interval assigned to the root of L is the lowest possible (when L is the top component), or the lowest possible interval incremented by a positive offset of $\omega(L') - \omega(L)$, where L' is the layer component directly above L. If we sum up the additive increments due to cost scaling in all layer components below L, we obtain $\sum_{i<l} c(i) < c(l) = z(v)$ (recall that $c(i)$'s are powers of 2). Thus, $f_{alg}(v) \leq f_{opt}(v) + z(v)$. ◀

By combining Lemmas 6, 8, and 16 we obtain that Algorithm 1 generates a solution for the binary search problem with a cost at most 8 times greater than that of an optimal solution. We note that Algorithm 1 maintains the main structure of the linear time algorithm

from [26] (a single bottom-up pass over vertices of the tree.) Our method extends the state-of-the-art algorithm by adding a fixed number of $\mathcal{O}(1)$ steps, computed when visiting vertices during the bottom-up traversal. See the *Process Component* procedure, Algorithm 2. What follows, Algorithm 1 also runs in linear time. This proves Theorem 2.

7 Conclusions

We recall a related tree search problem, called *edge search*, in which one performs queries on edges: each reply provides information which endpoint of the queried edge is closer to the target [3]. One can similarly define a monotonic cost function $\omega\colon E(T) \to \mathbb{R}_+$ for the edge search by requiring that there exists a choice for the root r so that for any two edges $\{x, y\}$ and $\{y, z\}$, if x is closer to r than y, then $\omega(\{x, y\}) \geq \omega(\{y, z\})$.

Interestingly, the edge search problem is NP-complete for monotonic weight function which follows from [6], although the authors do not obtain this fact directly. More precisely, the reduction in [6] constructs a spider in which each leg has three edges with weights $2a, a$, and $\Theta(an)$ (listing them by following from the root to the leaf). Denote by e the latter edge incident to a leaf. It is not hard to see that e can be replaced by a star on $\Theta(n)$ edges, each of weight at most a, thus getting an instance with a monotonic cost function and the same search cost.

It turns out that the problem we study in this work is more general in the class of weighted trees – see the corresponding reduction in [8]. This reduction however does not preserve the property of monotonicity. In other words, a conversion from an instance of the edge search with a monotonic cost function produces an instance of the vertex search with a non-monotonic cost function. Thus, the fact that the edge search is NP-complete for weighted trees pointed above does not imply hardness in our case. We do believe that the argument of hardness cannot be easily obtained in a similar way as for the edge search in [6] and we leave as an open question whether the vertex search problem studied in this work is NP-complete in case of monotonic cost functions.

Also, we repeat the previously mentioned interesting and challenging open question whether a constant-factor approximation is possible for the vertex search in trees with general weight functions.

References

1 Alok Aggarwal, Bowen Alpern, Ashok K. Chandra, and Marc Snir. A model for hierarchical memory. In Alfred V. Aho, editor, *STOC 1987*, pages 305–314. ACM, 1987. doi:10.1145/28395.28428.

2 Haris Angelidakis. Shortest path queries, graph partitioning and covering problems in worst and beyond worst case settings. *CoRR*, abs/1807.09389, 2018. arXiv:1807.09389.

3 Yosi Ben-Asher, Eitan Farchi, and Ilan Newman. Optimal search in trees. *SIAM J. Comput.*, 28(6):2090–2102, 1999. doi:10.1137/S009753979731858X.

4 Piotr Borowiecki, Dariusz Dereniowski, and Dorota Osula. The complexity of bicriteria tree-depth. In Evripidis Bampis and Aris Pagourtzis, editors, *FCT 2021*, volume 12867 of *Lecture Notes in Computer Science*, pages 100–113. Springer, 2021. doi:10.1007/978-3-030-86593-1_7.

5 Ferdinando Cicalese, Tobias Jacobs, Eduardo Sany Laber, and Caio Dias Valentim. Binary identification problems for weighted trees. In Frank Dehne, John Iacono, and Jörg-Rüdiger Sack, editors, *WADS 2011*, volume 6844 of *Lecture Notes in Computer Science*, pages 255–266. Springer, 2011. doi:10.1007/978-3-642-22300-6_22.

6 Ferdinando Cicalese, Balázs Keszegh, Bernard Lidický, Dömötör Pálvölgyi, and Tomás Valla. On the tree search problem with non-uniform costs. *Theor. Comput. Sci.*, 647:22–32, 2016. doi:10.1016/j.tcs.2016.07.019.

7 Dariusz Dereniowski. Edge ranking of weighted trees. *Discret. Appl. Math.*, 154(8):1198–1209, 2006. doi:10.1016/j.dam.2005.11.005.

8 Dariusz Dereniowski, Adrian Kosowski, Przemyslaw Uznanski, and Mengchuan Zou. Approximation strategies for generalized binary search in weighted trees. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *ICALP 2017*, volume 80 of *LIPIcs*, pages 84:1–84:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ICALP.2017.84.

9 Dariusz Dereniowski and Marek Kubale. Cholesky factorization of matrices in parallel and ranking of graphs. In Roman Wyrzykowski, Jack J. Dongarra, Marcin Paprzycki, and Jerzy Wasniewski, editors, *PPAM 2003*, volume 3019 of *Lecture Notes in Computer Science*, pages 985–992. Springer, 2003. doi:10.1007/978-3-540-24669-5_127.

10 Dariusz Dereniowski and Marek Kubale. Efficient parallel query processing by graph ranking. *Fundam. Informaticae*, 69(3):273–285, 2006.

11 Dariusz Dereniowski, Aleksander Lukasiewicz, and Przemyslaw Uznanski. An efficient noisy binary search in graphs via median approximation. In Paola Flocchini and Lucia Moura, editors, *IWOCA 2021*, volume 12757 of *Lecture Notes in Computer Science*, pages 265–281. Springer, 2021. doi:10.1007/978-3-030-79987-8_19.

12 Dariusz Dereniowski and Adam Nadolski. Vertex rankings of chordal graphs and weighted trees. *Inf. Process. Lett.*, 98(3):96–100, 2006. doi:10.1016/j.ipl.2005.12.006.

13 Dariusz Dereniowski, Stefan Tiegel, Przemyslaw Uznanski, and Daniel Wolleb-Graf. A framework for searching in graphs in the presence of errors. In Jeremy T. Fineman and Michael Mitzenmacher, editors, *SOSA 2019*, volume 69 of *OASIcs*, pages 4:1–4:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/OASIcs.SOSA.2019.4.

14 Ehsan Emamjomeh-Zadeh, David Kempe, Mohammad Mahdian, and Robert E. Schapire. Interactive learning of a dynamic structure. In Aryeh Kontorovich and Gergely Neu, editors, *ALT 2020*, volume 117 of *Proceedings of Machine Learning Research*, pages 277–296. PMLR, 2020.

15 Ehsan Emamjomeh-Zadeh, David Kempe, and Vikrant Singhal. Deterministic and probabilistic binary search in graphs. In Daniel Wichs and Yishay Mansour, editors, *STOC 2016*, pages 519–532. ACM, 2016. doi:10.1145/2897518.2897656.

16 Mine Su Erturk and Kuang Xu. Private genetic geneaology search. Research papers, Stanford University, Graduate School of Business, 2021. URL: https://EconPapers.repec.org/RePEc:ecl:stabus:3973.

17 Ananth V. Iyer, H. Donald Ratliff, and Gopalakrishnan Vijayan. Optimal node ranking of trees. *Inf. Process. Lett.*, 28(5):225–229, 1988. doi:10.1016/0020-0190(88)90194-9.

18 Ananth V. Iyer, H. Donald Ratliff, and Gopalakrishnan Vijayan. On an edge ranking problem of trees and graphs. *Discret. Appl. Math.*, 30(1):43–52, 1991. doi:10.1016/0166-218X(91)90012-L.

19 William J. Knight. Search in an ordered array having variable probe cost. *SIAM J. Comput.*, 17(6):1203–1214, 1988. doi:10.1137/0217076.

20 Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

21 Eduardo Sany Laber, Ruy Luiz Milidiú, and Artur Alves Pessoa. On binary searching with non-uniform costs. In S. Rao Kosaraju, editor, *SODA 2001*, pages 855–864. ACM/SIAM, 2001.

22 Tak Wah Lam and Fung Ling Yue. Optimal edge ranking of trees in linear time. In Howard J. Karloff, editor, *SODA 1998*, pages 436–445. ACM/SIAM, 1998.

23 Shay Mozes, Krzysztof Onak, and Oren Weimann. Finding an optimal tree searching strategy in linear time. In Shang-Hua Teng, editor, *SODA 2008*, pages 1096–1105. SIAM, 2008.

24 Gonzalo Navarro, Ricardo A. Baeza-Yates, Eduardo F. Barbosa, Nivio Ziviani, and Walter Cunto. Binary searching with nonuniform costs and its application to text retrieval. *Algorithmica*, 27(2):145–169, 2000. doi:10.1007/s004530010010.

25 Jaroslav Nešetřil and Patrice Ossona de Mendez. Tree-depth, subgraph coloring and homomorphism bounds. *Eur. J. Comb.*, 27(6):1022–1041, 2006. doi:10.1016/j.ejc.2005.01.010.

26 Krzysztof Onak and Pawel Parys. Generalization of binary search: Searching in trees and forest-like partial orders. In *FOCS 2006*, pages 379–388. IEEE Computer Society, 2006. doi:10.1109/FOCS.2006.32.

27 Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid G. Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. The future is big graphs: a community view on graph processing systems. *Commun. ACM*, 64(9):62–71, 2021. doi:10.1145/3434642.

28 Alejandro A. Schäffer. Optimal node ranking of trees in linear time. *Inf. Process. Lett.*, 33(2):91–96, 1989. doi:10.1016/0020-0190(89)90161-0.

29 Arunabha Sen, Haiyong Deng, and Sumanta Guha. On a graph partition problem with application to VLSI layout. *Inf. Process. Lett.*, 43(2):87–94, 1992. doi:10.1016/0020-0190(92)90017-P.