# Approximation Strategies for Generalized Binary Search in Weighted Trees*†

**Dariusz Dereniowski[1], Adrian Kosowski[2], Przemysław Uznański[3], and Mengchuan Zou[4]**

1 **Faculty of Electronics, Telecommunications and Informatics, Gdańsk University of Technology, Gdańsk, Poland**
deren@eti.pg.gda.pl
2 **Inria Paris and IRIF, Université Paris Diderot, Paris, France**
adrian.kosowski@inria.fr
3 **Department of Computer Science, ETH Zürich, Zürich, Switzerland**
przemyslaw.uznanski@inf.ethz.ch
4 **Inria Paris and IRIF, Université Paris Diderot, Paris, France**
mengchuan.zou@inria.fr

## Abstract

We consider the following generalization of the binary search problem. A search strategy is required to locate an unknown target node $t$ in a given tree $T$. Upon querying a node $v$ of the tree, the strategy receives as a reply an indication of the connected component of $T \setminus \{v\}$ containing the target $t$. The cost of querying each node is given by a known non-negative weight function, and the considered objective is to minimize the total query cost for a worst-case choice of the target. Designing an optimal strategy for a weighted tree search instance is known to be strongly NP-hard, in contrast to the unweighted variant of the problem which can be solved optimally in linear time. Here, we show that weighted tree search admits a quasi-polynomial time approximation scheme (QPTAS): for any $0 < \varepsilon < 1$, there exists a $(1+\varepsilon)$-approximation strategy with a computation time of $n^{O(\log n/\varepsilon^2)}$. Thus, the problem is not APX-hard, unless NP $\subseteq$ DTIME$(n^{O(\log n)})$. By applying a generic reduction, we obtain as a corollary that the studied problem admits a polynomial-time $O(\sqrt{\log n})$-approximation. This improves previous $\widetilde{O}(\log n)$-approximation approaches, where the $\widetilde{O}$-notation disregards $O(\text{poly}\log\log n)$-factors.

**1998 ACM Subject Classification** G.2.2 Graph Theory, F.2.2 Nonnumerical Algorithms

**Keywords and phrases** Approximation Algorithm, Adaptive Algorithm, Graph Search, Binary Search, Vertex Ranking, Trees

**Digital Object Identifier** 10.4230/LIPIcs.ICALP.2017.84

## 1 Introduction

In this work we consider a generalization of the fundamental problem of searching for an element in a sorted array. This problem can be seen, using graph-theoretic terms, as a problem of searching for a target node in a path, where each query reveals on which 'side' of the queried node the target node lies. The generalization we study is two-fold: a more general structure of a tree is considered and we assume non-uniform query times. Thus,

* The full version can be found at https://arxiv.org/abs/1702.08207.
† Partially supported by ANR DESCARTES and by National Science Centre (Poland) grant number 2015/17/B/ST6/01887.

44th International Colloquium on Automata, Languages, and Programming (ICALP 2017).
Editors: Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl;
Article No. 84; pp. 84:1–84:14


Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

**Table 1** Computational complexity of the search problem in different graph classes, including our results for weighted trees. Completeness results refer to the decision version of the problem. In the case of unweighted paths, the solution is the classical binary search algorithm.

| *Graph class* | *Unweighted* | *Weighted* |
|---|---|---|
| Paths: | exact in $O(n)$ time | exact in $O(n^2)$ time [4] |
| | | strongly NP-complete [9] |
| Trees: | exact in $O(n)$ time [26, 28] | $(1+\varepsilon)$-approx. in $n^{O(\log n/\varepsilon)}$ time (Thm. 3.3) |
| | | $O(\sqrt{\log n})$-approx. in poly-time (Thm. 3.4) |
| Undirected: | exact in $n^{O(\log n)}$ time [10] | PSPACE-complete [10] |
| | $O(\log n)$-approx. in poly-time [10] | $O(\log n)$-approx. in poly-time [10] |
| Directed: | PSPACE-complete [10] | PSPACE-complete [10] |

our problem can be stated as follows. Given a node-weighted input tree $T$ (in which the query time of a node is provided as its weight), design a search strategy (sometimes called a decision tree) that locates a hidden *target node* $x$ by asking *queries*. Each query selects a node $v$ in $T$ and after the time that equals the weight of the selected node, a reply is given: the reply is either 'yes' which implies that $v$ is the target node and thus the search terminates, or it is 'no' in which case the search strategy receives the edge outgoing from $v$ that belongs to the shortest path between $u$ and $v$. The goal is to design a search strategy that locates the target node and minimizes the search time in the worst case.

The vertex search problem is more general than its 'edge variant' that has been more extensively studied. In the latter problem one selects an edge $e$ of an edge-weighted tree $T = (V, E, w)$ in a query and learns in which of the two components of $T - e$ the target node is located. Indeed, this edge variant can be reduced to our problem as follows: first assign a 'large' weight to each node of $T$ (for example, one plus the sum of the weights of all edges in the graph) and then subdivide each edge $e$ of $T$ giving to the new node the weight of the original edge, $w(e)$. It is apparent that an optimal search strategy for the new node-weighted tree should never query the nodes with large weights, thus immediately providing a search strategy for the edge variant of $T$.

We also point out that the considered problem, as well as the edge variant, being quite fundamental, were historically introduced several times under different names: minimum height elimination trees [27], ordered colourings [15], node and edge rankings [13], tree-depth [25] or LIFO-search [11]. Table 1 summarizes the complexity status of the node-query model and places our result in the general context.

## 1.1 State-of-the-Art

In this work we focus on the worst case search time for a given input graph and we only remark that other optimization criteria has been also considered [3, 16, 17, 30]. For other closely related models and corresponding results see e.g. [1, 12, 19, 21, 29].

**The node-query model.** An optimal search strategy can be computed in linear-time for an unweighted tree [26, 28]. The number of queries performed in the worst case may vary from being constant (for a star one query is enough) to being at most $\log_2 n$ for any tree [26] (by

always querying a node that halves the search space). Several following results have been obtained in [10]. First, it turns out that $\log_2 n$ queries are always sufficient for general simple graphs and this implies a $O(m^{\log_2 n} n^2 \log n)$-time optimal algorithm for arbitrary unweighted graphs. The algorithm which performs $\log_2 n$ queries also serves as a $O(\log n)$-approximation algorithm, also for the weighted version of the problem. On the other hand, it is shown in the same work that an optimal algorithm (for unweighted case) with a running time of $O(n^{o(\log n)})$ would be in contradiction with the Exponential-Time-Hypothesis. When weighted graphs are considered, the problem becomes PSPACE-complete.

**The edge-query model.** In the case of unweighted trees, an optimal search strategy can be computed in linear time [20, 24]. (See [7] for a correspondence between edge rankings and the searching problem.) The computational complexity of the problem on weighted trees attracted a lot of attention. On the negative side, it has been proved that it is strongly NP-hard to compute an optimal search strategy [6] for bounded diameter trees, which has been improved by showing hardness for several specific topologies: trees of diameter at most 6, trees of degree at most 3 [4] and spiders [5] (trees having at most one node of degree greater than two). On the other hand, polynomial-time algorithms exist for weighted trees of diameter at most 5 and weighted paths [4]. We note that for weighted paths there exists a linear-time but approximate solution given in [16]. For approximate polynomial-time solutions, a simple $O(\log n)$-approximation has been given in [6] and a $O(\log n/\log\log\log n)$-approximate solution is given in [4]. Then, the best known approximation ratio has been further improved to $O(\log n/\log\log n)$ in [5].

Some bounds on the number of queries for unweighted trees have been developed. Observe that an optimal search strategy needs to perform at least $\log_2 n$ queries in the worst case. However, there exist trees of maximum degree $\Delta$ that require $\Delta \log_{\Delta+1} n$ queries [2]. On the other hand, $\Theta(\Delta \log n)$ queries are always sufficient for each tree [2], which has been improved to $(\Delta+1)\log_\Delta n$ [18], $\Delta \log_\Delta n$ [8] and $1 + \frac{\Delta-1}{\log_2(\Delta+1)-1}\log_2 n$ [10].

## 1.2 Organization of the Paper

The aim of Section 2 is to give the necessary notation and a formal statement of the problem (Sections 2.1 and 2.2) and to provide two different but equivalent problem formulations that will be more convenient for our analysis. As opposed to the classical problem formulation in which a strategy is seen as a *decision tree*, Section 2.3 restates the problem in such a way that with each vertex $v$ of the input tree we associate a sequence of vertices that need to be iteratively queried when $v$ is the root of the current subtree that contains the target node. In Section 2.4 we extend this approach by associating with each vertex a sequence of not only vertices to be queried but also time points of the queries.

The latter problem formulation is suitable for a dynamic programming algorithm provided in Section 3.1. In this section we introduce an auxiliary, slightly modified measure of the cost of a search strategy. First we provide a quasi-polynomial time dynamic programming scheme that provides an arbitrarily good approximation of the output search strategy with respect to this modified cost (the analysis is deferred to Section 4), and then we prove that the new measure is sufficiently close to the original one (the analysis is deferred to Section 5). These two facts provide the quasi-polynomial time scheme for the tree search problem, achieving a $(1+\varepsilon)$-approximation with a computation time of $n^{O(\log n/\varepsilon^2)}$, for any $0 < \varepsilon < 1$.

In Section 3.2 we observe how to use the above algorithm to derive a polynomial-time $O(\sqrt{\log n})$-approximation algorithm for the tree search problem. This is done by a divide and conquer approach: a sufficiently small subtree $T^*$ of the input tree $T$ is first computed

so that the quasi-polynomial time algorithm runs in polynomial (in the size of $T$) time for $T^*$. This decomposes the problem: having a search strategy for $T^*$, the search strategies for $T - T^*$ are computed recursively.

# 2 Preliminaries

## 2.1 Notation and Query Model

We now recall the problem of searching of an unknown target node $x$ by performing queries on the vertices of a given node-weighted rooted tree $T = (V, E, w)$ with weight function $w\colon V \to \mathbb{R}_+$. Each *query* selects one vertex $v$ of $T$ and after $w(v)$ time units receives an answer: either the query returns *true*, meaning that $x = v$, or it returns a neighbor $u$ of $v$ which lies closer to the target $x$ than $v$. Since we assume that the queried graph $T$ is a tree, such a neighbor $u$ is unique and is equivalently described as the unique neighbor of $v$ belonging to the same connected component of $T \setminus \{v\}$ as $x$.

All trees we consider are rooted. Given a tree $T$, the root is denoted by $r(T)$. For a node $v \in V$, we denote by $T_v$ the subtree of $T$ rooted at $v$. For any subset $V' \subseteq V$ (respectively, $E' \subseteq E$) we denote by $T[V']$ (resp., $T[E']$) the minimal subtree of $T$ containing all nodes from $V'$ (resp., all edges from $E'$). For $v \in V$, $N(v)$ is the set of neighbors of $v$ in $T$.

For $U \subseteq V$ and a target node $x \notin U$, there exists a unique maximal subtree of $T \setminus U$ that contains $x$; we will denote this subtree by $T\langle U, x\rangle$.

We denote $|V| = n$. We will assume w.l.o.g. that the maximum weight of a vertex is normalized to 1. (This normalization is immediately obtained by a proportional scaling of all units of cost.) We will also assume w.l.o.g. that the weight function satisfies the following *star condition*: for all $v \in V$, $w(v) \leq \sum_{u \in N(v)} w(u)$. Observe that if this condition is not fulfilled, i.e., for some vertex $v$ will have $w(v) > \sum_{u \in N(v)} w(u)$, then vertex $v$ will never be queried by any optimal strategy in $v$, since a query to $v$ can then be replaced by a sequence of queries to all neighbors of $v$, obtaining not less information at strictly smaller cost. In general, given an instance which does not satisfy the star condition, we enforce it by performing all necessary weight replacements $w(v) \leftarrow \min\{w(v), \sum_{u \in N(v)} w(u)\}$, for $v \in V$. Replacements terminate definitely since no vertex will be replaced more than once.

For $a, \omega \in \mathbb{R}_{\geq 0}$, we denote the rounding of $a$ down (up) to the nearest multiple of $\omega$ as $\lfloor a \rfloor_\omega = \omega \lfloor a/\omega \rfloor$ and $\lceil a \rceil_\omega = \omega \lceil a/\omega \rceil$, respectively.

## 2.2 Definition of a Search Strategy

*A search strategy* $\mathcal{A}$ for a rooted tree $T = (V, E, w)$ is an adaptive algorithm which defines successive queries to the tree, based on responses to previous queries, with the objective of locating the target vertex in a finite number of steps. Note that search strategies can be seen as decision trees in which each node represents a subset of vertices of $T$ that contains $x$, with leaves representing singletons consisting of $x$.

Let $\mathtt{Q}_\mathcal{A}(T, x)$ be the time-ordering (sequence) of queries performed by strategy $\mathcal{A}$ on tree $T$ to find a target vertex $x$, with $\mathtt{Q}_{\mathcal{A},i}(T, x)$ denoting the $i$-th queried vertex in this time ordering, $1 \leq i \leq |\mathtt{Q}_\mathcal{A}(T, x)|$.

We denote by $\mathtt{COST}_\mathcal{A}(T, x) = \sum_{i=1}^{|\mathtt{Q}_\mathcal{A}(T,x)|} w(\mathtt{Q}_{\mathcal{A},i}(T, x))$ the sum of weights of all vertices queried by $\mathcal{A}$ with $x$ being the target node, i.e., the time after which $\mathcal{A}$ finishes. Let $\mathtt{COST}_\mathcal{A}(T) = \max_{x \in V} \mathtt{COST}_\mathcal{A}(T, x)$ be the *cost of* $\mathcal{A}$. We define the *cost of* $T$ to be $\mathtt{OPT}(T) = \min\{\mathtt{COST}_\mathcal{A}(T) \mid \mathcal{A} \text{ is a search strategy for } T\}$. We say that a search strategy is *optimal* for $T$ if its cost equals $\mathtt{OPT}(T)$.

**Algorithm 2.1** Search strategy $\mathcal{A}_S$ for a query sequence assignment $S$.

```
1: v ← r(T)    // stores current root
2: U ← ∅
3: while |T⟨U, x⟩| > 1 do
4:     for u ∈ S(v) do
5:         if u ∈ T⟨U, x⟩ then    // u is the first vertex in S(v) that belongs to T⟨U, x⟩
6:             QueryVertex(u)
7:             U ← U ∪ {u}
8:             if v ≠ r(T⟨U, x⟩) then    // query reply is 'down'
9:                 v ← r(T⟨U, x⟩)
10:                break    // for loop
```

As a consequence of normalization and the star condition, we have the following bound.

▶ **Observation 2.1.** *For any tree $T$, we have* $1 \leq \mathtt{OPT}(T) \leq \lceil \log_2 n \rceil$.

All omitted proofs are provided in the full version.

We also introduce the following notation. If the first $|U|$ queried vertices by a search strategy $\mathcal{A}$ are exactly the vertices in $U$, $U = \{\mathtt{Q}_{\mathcal{A},i}(T, x) : 1 \leq i \leq |U|\}$, then we say that $\mathcal{A}$ *reaches* $T\langle U, x\rangle$ *through* $U$, and $w(U)$ is the *cost of reaching* $T\langle U, x\rangle$ *by* $\mathcal{A}$. We also say that we receive an 'up' reply to a query to a vertex $v$ if the root of the tree remaining to be searched remains unchanged by the query, i.e., $r(T\langle U, x\rangle) = r(T\langle U \cup \{v\}, x\rangle)$, and we call the reply a 'down' reply when the root of the remaining tree changes, i.e., $r(T\langle U, x\rangle) \neq r(T\langle U \cup \{v\}, x\rangle)$.

## 2.3 Query Sequences and Stable Strategies

By a slight abuse of notation, we will call a search strategy *polynomial-time* if it can be implemented using a dynamic (adaptive) algorithm which computes the next queried vertex in polynomial time.

We give most of our attention herein to search strategies in trees which admit a natural (non-adaptive, polynomial-space) representation called a *query sequence assignment*. Formally, for a rooted tree $T$, the *query sequence assignment* $S$ is a function $S : V \to V^*$, which assigns to each vertex $v \in V$ an ordered sequence of vertices $S(v)$, known as the *query sequence* of $v$. The query sequence assignment directly induces a strategy $\mathcal{A}_S$, presented as Algorithm 2.1. Intuitively, the strategy processes successive queries from the sequence $S(v)$, where $v$ is the root vertex of the current search tree, $v = r(T\langle U, x\rangle)$, where $U$ is the set of queries performed so far. This processing is performed in such a way that the strategy iteratively takes the first vertex in $S(v)$ that belongs to $T\langle U, x\rangle$ and queries it. As soon as the root of the search tree changes, the procedure starts processing queries from the sequence of the new root, which belong to the remaining search tree. The procedure terminates as soon as $T\langle U, x\rangle$ has been reduced to a single vertex, which is necessarily the target $x$.

In what follows, in order to show that our approximation strategies are polynomial-time, we will confine ourselves to presenting a polynomial-time algorithm which outputs an appropriate sequence assignment.

A sequence assignment is called *stable* if the replacement of line 9 in Algorithm 2.1 by any assignment of the form $v \leftarrow v''$, where $v''$ is an arbitrary vertex which is promised to lie on the path from $r(T\langle U, x\rangle)$ to the target $x$, always results in a strategy which performs a (not necessarily strict) subsequence of the sequence of queries performed by the original strategy $\mathcal{A}_S$. Sequence assignments computed on trees with a bottom-up approach usually have the stability property; we provide a proof of stability for one of our main routines in Section 4.

Without loss of generality, we will also assume that if $v \in S(v)$, then $v$ is the last element of $S(v)$. Indeed, when considering a subtree rooted at $v$, after a query to $v$, if $v$ was not the target, then the root of the considered subtree will change to one of the children of $v$, hence any subsequent elements of $S(v)$ may be removed without changing the strategy.

## 2.4 Strategies Based on Consistent Schedules

Intuitively, we may represent search strategies by a schedule consisting of some number of jobs, with each job being associated to querying a node in the tree (cf. e.g. [14, 22, 23]). Each job has a fixed processing time, which is set to the weight of a node. Formally, in this work we will refer to the schedule $\hat{S}$ only in the very precise context of search strategies $\mathcal{A}_S$ based on some query sequence assignment $S$. The *schedule assignment* $\hat{S}$ is the following extension of the sequence assignment $S$, which additionally encodes the starting time of any search query job. If the query sequence $S$ of a node $v$ is of the form $S(v) = (v_1, \ldots, v_k)$, $k = |S(v)|$, then the corresponding schedule for $v$ will be given as $\hat{S}(v) = ((v_1, t_1), \ldots, (v_k, t_k))$, with $t_i \in \mathbb{R}_{\geq 0}$ denoting the starting time of the query for $v_i$. We will call $\hat{S}(v)$ the *schedule of node* $v$. We will call a schedule assignment $\hat{S}$ *consistent* with respect to search in a given tree $T$ if the following conditions are fulfilled:

- **(i)** No two jobs in the schedule of a node overlap: for all $v \in V$, for two distinct jobs $(u_1, t_1), (u_2, t_2) \in \hat{S}(v)$, we have $|[t_1, t_1 + w(u_1)] \cap [t_2, t_2 + w(u_2)]| = 0$.
- **(ii)** If $v$ is the parent of $v'$ in $T$ and $(u, t) \in \hat{S}(v')$, then we either also have $(u, t) \in \hat{S}(v)$, or the job $(v, t_v) \in \hat{S}(v)$ completes before the start of job $(u, t)$: $t_v + w(v) \leq t$.

It follows directly from the definition that a consistent schedule assignment (and the underlying query sequence assignment) is uniquely determined by the collection of jobs $\{(v, t_v) : (v, t_v) \in \hat{S}(u), u \in V\}$. Note that not every vertex has to contain a query to itself in its schedule; we will occasionally write $t_v = \perp$ to denote that such a job is missing.

By extension of notation for sequence assignments, we will denote a strategy following a consistent schedule assignment $\hat{S}$ (i.e., executing the query jobs of schedule $\hat{S}$ at the prescribed times) as $\mathcal{A}_{\hat{S}}$. We will then have: $\texttt{COST}_{\mathcal{A}_{\hat{S}}}(T) = |\hat{S}|$, where $|\hat{S}|$ is the *duration* of schedule assignment $\hat{S}$, given as: $|\hat{S}| = \max_{v \in V} |\hat{S}(v)|$, with: $|\hat{S}(v)| = \max_{(u,t) \in \hat{S}(v)} (t + w(u))$.

We remark that there always exists an optimal search strategy which is based on a consistent schedule. By a well-known characterization (cf. e.g. [6]), tree $T$ satisfies $\texttt{OPT}(T) = \tau \in \mathbb{R}$ if and only if there exists an assignment $I : V \to \mathcal{I}_\tau$ of intervals of time to nodes before deadline $\tau$, $\mathcal{I}_\tau = \{[a, b] : 0 \leq a < b \leq \tau\}$, such that $|I(v)| = w(v)$ and if $|I(u) \cap I(v)| > 0$ for any pair of nodes $u, v \in V$, then the $u - v$ path in $T$ contains a separating vertex $z$ such that $\max I(z) \leq \min(I(u) \cup I(v))$. The corresponding schedule assignment of duration $\tau$ is obtained by adding, for each node $u \in V$, the job $(u, \min I(u))$ to the schedule of all nodes on the path from $u$ towards the root, until a node $v$ such that $\max I(v) \leq \min I(u)$ is encountered on this path. The consistency and correctness of the obtained schedule is immediate to verify.

▶ **Observation 2.2.** *For any tree $T$, there exists a query sequence assignment $S$ and a corresponding consistent schedule $\hat{S}$ on $T$ such that $|\hat{S}| = \texttt{OPT}(T)$.*

# 3 The Results

## 3.1 $(1 + \varepsilon)$-Approximation in $n^{O(\log n/\varepsilon^2)}$ Time

We first present an approximation scheme for the weighted tree search problem with $n^{O(\log n)}$ running time. The main difficulty consists in obtaining a constant approximation ratio for

the problem with this running time; we at once present this approximation scheme with tuned parameters, so as to achieve $(1+\varepsilon)$-approximation in $n^{O(\log n/\varepsilon^2)}$ time.

Our construction consists of two main building blocks. First, we design an algorithm based on a bottom-up (dynamic programming) approach, which considers exhaustively feasible sequence assignments and query schedules over a carefully restricted state space of size $n^{O(\log n)}$ for each node. The output of the algorithm provides us both with a lower bound on $\texttt{OPT}(T)$, and with a sequence assignment-based strategy $\mathcal{A}_S$ for solving the tree search problem. The performance of this strategy $\mathcal{A}_S$ is closely linked to the performance of $\texttt{OPT}(T)$, however, there is one type of query, namely a query on a vertex of small weight leading to a 'down' response, due to whose repeated occurrence the eventual cost difference between $\texttt{COST}_{\mathcal{A}_S}(T)$ and $\texttt{OPT}(T)$ may eventually become arbitrarily large. To alleviate this difficulty, we introduce an alternative measure of cost which compensates for the appearance of the disadvantageous type of query.

We start by introducing some additional notation. Let $\omega \in \mathbb{R}_+$, be an arbitrarily fixed value of weight and let $c \in \mathbb{N}$. The choice of constant $c \in \mathbb{N}$ will correspond to an approximation ratio of $(1+\varepsilon)$ of the designed scheme for $\varepsilon = 168/c$.

We say that a query to a vertex $v$ is a *light down query* in some strategy if $w(v) < c\omega$ and $x \in V(T_v)$, i.e., it is also a 'down' query, where $x$ is the target vertex.

For any strategy $\mathcal{A}$, we denote by $\texttt{COST}^{(\omega,c)}_{\mathcal{A}}(T,x)$ its modified cost of finding target $x$, defined as follows. Let $d_x$ be the number of light down queries when searching for $x$: $d_x = \left|\{i : w(\mathtt{Q}_{\mathcal{A},i}(T,x)) < c\omega \text{ and } x \in V(T_{\mathtt{Q}_{\mathcal{A},i}(T,x)})\}\right|$. Then, the modified cost $\texttt{COST}^{(\omega,c)}_{\mathcal{A}}(T,x)$ is:

$$\texttt{COST}^{(\omega,c)}_{\mathcal{A}}(T,x) = \texttt{COST}_{\mathcal{A}}(T,x) - (2c+1)\omega d_x. \tag{1}$$

and by a natural extension of notation: $\texttt{COST}^{(\omega,c)}_{\mathcal{A}}(T) = \max_{x\in V} \texttt{COST}^{(\omega,c)}_{\mathcal{A}}(T,x)$.

The technical result which we will obtain in Section 4 may now be stated as follows.

▶ **Proposition 3.1.** *For any $c \in \mathbb{N}$, $L \in \mathbb{N}$, there exists an algorithm running in time $(cn)^{O(L)}$, which for any tree $T$ constructs a stable sequence assignment $S$ and computes a value of $\omega$ such that $\omega \le \frac{1}{L}\texttt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T)$ and: $\texttt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T) \le \left(1+\frac{12}{c}\right)\texttt{OPT}(T)$.*

In order to convert the obtained strategy $\mathcal{A}_S$ with a small value of $\texttt{COST}^{(\omega,c)}$ into a strategy with small $\texttt{COST}$, we describe in Section 5 an appropriate strategy conversion mechanism. The approach we adopt is applicable to any strategy based on a stable sequence assignment and consists in concatenating, for each vertex $v \in V$, a prefix to the query sequence $S(v)$ in the form of a separately computed sequence $R(v)$, which does not depend on $S(v)$. The considered query sequences are thus of the form $R(v) \circ S(v)$, where the symbol "$\circ$" represents sequence concatenation. Intuitively, the sequences $R$, taken over the whole tree, reflect the structure of a specific solution to the unweighted tree search problem on a contraction of tree $T$, in which each edge connecting a node to a child with weight at least $c\omega$ is contracted. We recall that the optimal number of queries to reach a target in an unweighted tree is $O(\log n)$, and the goal of this conversion is to reduce the number of light down queries in the combined strategy to at most $O(\log n)$.

▶ **Proposition 3.2.** *For any fixed $\omega > 0$ there exists a polynomial-time algorithm which for a tree $T$ computes a sequence assignment $R : V \to V^*$, such that, for any strategy $\mathcal{A}_S$ based on a stable sequence assignment $S$, the sequence assignment $S^+$, given by $S^+(v) = R(v) \circ S(v)$ for each $v \in V$, has the following property:*

$$\texttt{COST}_{\mathcal{A}_{S^+}}(T) \le \texttt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T) + 4(2c+1)\omega \log_2 n.$$

The proof of Proposition 3.2 is provided in Section 5.

We are now ready to put together the two bounds. Combining the claims of Proposition 3.1 for $L = \lceil c^2 \log_2 n \rceil$ (with $\omega \leq \frac{1}{L}\mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T) \leq \frac{\mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T)}{c^2 \log_2 n}$) and Proposition 3.2, we obtain:

$$\begin{aligned}\mathtt{COST}_{\mathcal{A}_{S^+}}(T) &\leq \mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T) + 4(2c+1)\omega \log_2 n \leq \mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T) + 12c\omega \log_2 n \;\leq \\ &\leq \mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T) + 12c \log_2 n \frac{\mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T)}{c^2 \log_2 n} \leq \left(1 + \frac{12}{c}\right) \mathtt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T) \leq \\ &\leq \left(1 + \frac{12}{c}\right)^2 \mathtt{OPT}(T) \leq \left(1 + \frac{168}{c}\right) \mathtt{OPT}(T).\end{aligned}$$

After putting $\varepsilon = \frac{168}{c}$ and noting that in stating our result we can safely assume $c = O(\mathrm{poly}(n))$ (beyond this, the tree search problem can be trivially solved optimally in $O(n^n)$ time using exhaustive search), we obtain the main theorem of the section.

▶ **Theorem 3.3.** *There exists an algorithm running in $n^{O(\frac{\log n}{\varepsilon^2})}$ time, providing a $(1+\varepsilon)$-approximation solution to the weighted tree search problem for any $0 < \varepsilon < 1$.*

## 3.2 Extension: A Poly-Time $O(\sqrt{\log n})$-Approximation Algorithm

We now present the second main result of this work. By recursively applying the previously designed QPTAS (Theorem 3.3) with $\varepsilon = 1$, we obtain a polynomial-time $O(\sqrt{\log n})$-approximation algorithm for finding search strategy for an arbitrary weighted tree. We start by informally sketching the algorithm – we follow here the general outline of the idea from [5]. The algorithm is recursive and starts by finding a minimal subtree $T^*$ of an input tree whose removal disconnects $T$ into subtrees, each of size bounded by $n/2^{\sqrt{\log n}}$. The tree $T^*$ will be processed by our QPTAS algorithm described in Section 3.1. This results either in locating the target node, if it belongs to $T^*$, or identifying the component of $T - T^*$ containing the target, in which case the search continues recursively in the component. Subtrees considered at each level of recursion are disjoint, thus factors of approximation add up over recursion levels. However, for the final algorithm to have polynomial running time, the tree $T^*$ needs to be of size $2^{O(\sqrt{\log n})}$. This is obtained by contracting paths in $T^*$ (each vertex of the path has at most two neighbors in $T^*$) into single nodes having appropriately chosen weights. Since $T^*$ has $2^{O(\sqrt{\log n})}$ leaves, this narrows down the size of $T^*$ to the required level and we argue that an optimal search strategy for the 'contracted' $T^*$ provides a search strategy for the original $T^*$ that is within a constant factor from the cost of $T^*$.

A formal exposition and analysis of the obtained algorithm is provided in the full version.

▶ **Theorem 3.4.** *There is a $O(\sqrt{\log n})$-approximation polynomial time algorithm for the weighted tree search problem.*

# 4 Quasi-Polynomial Computation of Strategies with Small $\mathtt{COST}^{(\omega,c)}$

## 4.1 Preprocessing: Time Alignment in Schedules

We adopt here a method similar but arguably more refined than rounding techniques in scheduling problems of combinatorial optimization, showing that we could discretize the starting and finishing time of jobs, as well as weights of vertices, in a way to restrict the size of state space for each node to $n^{O(\log n)}$, without introducing much error.

Fix $c \in \mathbb{N}$ and $\omega = \frac{a}{cn}$ for some $a \in \mathbb{N}$. (In subsequent considerations, we will have $c = \Theta(1/\varepsilon)$, $a = O(\frac{n}{\log n})$ and $\omega = \Omega(\varepsilon/\log n)$.) Given a tree $T = (V, E, w)$, let $T' = (V, E, w')$ be a tree with the same topology as $T$ but with weights rounded up as follows:

$$w'(v) = \begin{cases} \lceil w(v) \rceil_{\omega}, & \text{if } w(v) > c\omega, \\ \lceil w(v) \rceil_{\frac{1}{cn}}, & \text{otherwise.} \end{cases} \tag{2}$$

We will informally refer to vertices with $w(v) > c\omega$ (equivalently $w'(v) > c\omega$) as *heavy vertices* and vertices with $w(v) \leq c\omega$ (equivalently $w'(v) \leq c\omega$) as *light vertices*. (Note that $w(v) \leq c\omega$ if and only if $w'(v) \leq c\omega$.)

When designing schedules, we consider time divided into *boxes* of duration $\omega$, with the $i$-th box equal to $[i\omega, (i+1)\omega]$. Each box is divided into $a$ identical *slots* of length $\frac{1}{cn}$.

In the tree $T'$, the duration of a query to a heavy vertex is an integer number of boxes, and the duration of a query to a light vertex is an integer number of slots. We next show that, without affecting significantly the approximation ratio of the strategy, we can align each query to a heavy vertex in the schedule so that it occupies an interval of full adjacent boxes, and each query to a light vertex in the schedule so that it occupies an interval of full adjacent slots (possibly contained in more than one box).

We start by showing the relationship between the costs of optimal solutions for trees $T$ and $T'$.

▶ **Lemma 4.1.** $\texttt{OPT}(T) \leq \texttt{OPT}(T') \leq (1 + \frac{2}{c})\texttt{OPT}(T)$.

▶ **Lemma 4.2.** *There exists a consistent schedule assignment $\hat{S}$ for tree $T'$ such that $\texttt{COST}_{\mathcal{A}_{\hat{S}}}(T') \leq (1 + \frac{3}{c})\texttt{OPT}(T')$ and for all $v \in V$ we have that*

- *if $w'(v) > c\omega$, ($v$ is heavy), then the starting time $t$ of any job $(v, t)$ in the schedule $\hat{S}(u)$ of any $u \in V$ is an integer multiple of $\omega$ (aligned to a box),*
- *if $w'(v) \leq c\omega$, ($v$ is light), then the starting time $t$ of any query $(v, t)$ in the schedule $\hat{S}(u)$ of any $u \in V$ is an integer multiple of $\frac{1}{cn}$ (aligned to a slot).*

A schedule on tree $T'$ satisfying the conditions of Lemma 4.2, and the resulting search strategy, are called *aligned*. Subsequently, we will design an aligned strategy on tree $T'$, and compare the quality of the obtained solution to the best aligned strategy for $T'$.

The intuition between the separate treatment of heavy vertices (aligned to boxes) and light vertices (aligned to slots) in aligned schedules is the following. Whereas the time ordering of boxes is essential in the design of the correct strategy, in our dynamic programming approach we will not be concerned about the order of slots within a single box (i.e., the order of queries to light vertices placed in a single box). This allows us to reduce the state space of a node. Whereas the ordering of slots in the box will eventually have to be repaired to provide a correct strategy, this will not affect the quality of the overall solution too much (except for the issue of light down queries pointed out earlier, which are handled separately in Section 5).

## 4.2 Dynamic Programming Routine for Fixed Box Size

Let the values of parameter $c$ and box size $\omega$ be fixed as before. Additionally, let $L \in \mathbb{N}$ be a parameter representing the time limit for the duration of the considered vertex schedules when measured in boxes, i.e., the longest schedule considered by the procedure will be of length $L\omega$ (we will eventually choose an appropriate value of $L = O(\log n)$ as required when showing Theorem 3.3).

In order to lower-bound the duration of the consistent aligned schedule assignment with minimum cost, we perform an exhaustive bottom-up evaluation of aligned schedules which

satisfy constraints on the occupancy of slots. However, instead of considering individual slots of a schedule which may be empty or full, for reasons of efficiency we consider the *load* $s_v[p]$ of each box, $0 \leq p < L$, in the same schedule, defined informally as the proportion of the duration of the occupied slots within the box to the duration $\omega$ of the box. In the full version, we formally show the following claim.

▶ **Lemma 4.3.** *Assume that the data structure $(s_v, t_v)_{v \in V}$ corresponds to a consistent schedule. Let $v \in V$ be an arbitrarily chosen node with set of children $\{v_1, \ldots, v_l\}$. Then the set of queried nodes forms an edge cover of the tree:*

$$\text{If } t_v = \perp, \text{ then } t_{v_j} \neq \perp, \text{ for all } 1 \leq j \leq l. \tag{3}$$

*Moreover, let completion time $t^v_{end}$ of the query to $v$ given as:*

$$t^v_{end} = \begin{cases} t_v + w'(v), & \text{if } t_v \neq \perp, \\ +\infty, & \text{if } t_v = \perp. \end{cases}$$

*Let $a_p$ be the contribution to the load of the $p$-th time box of the query job for vertex $v$, i.e.*

$$a_p = \begin{cases} \frac{1}{\omega} |[t_v, t^v_{end}] \cap [p\omega, (p+1)\omega]| & \text{if } t_v \neq \perp, \\ 0 & \text{if } t_v = \perp. \end{cases}$$

*Then, for any box $[p\omega, (p+1)\omega]$, $0 \leq p < L$, we have the following bounds on the amount of load which can be packed into the box:*

$$\left.\begin{array}{ll} s_v[p] = a_p + \sum_{j=1}^{l} s_{v_j}[p] \in [0,1], & \text{when } t^v_{end} \geq (p+1)\omega, \\ s_v[p] \geq a_p, & \text{when } p\omega < t^v_{end} < (p+1)\omega, \\ s_v[p] = 0, & \text{when } t^v_{end} \leq p\omega. \end{array}\right\} \tag{4}$$

*Moreover, for any box $[p\omega, (p+1)\omega]$, $0 \leq p < L$, we have that the total load of a query to $v$ and queries propagated from any of the subtrees cannot exceed 1:*

$$\text{For all } 1 \leq j \leq l, \text{ the following bound holds: } s_{v_j}[p] + a_p \leq 1. \tag{5}$$

We now show that the shortest schedule assignments satisfying the set of constraints (3), (4), and (5) can be found in $n^{O(\log n)}$ time. This is achieved by using the procedure BuildStrategy, presented in Algorithm 4.1, which returns for a node $v$ a non-empty set of schedules $\hat{\mathcal{S}}[v]$, such that each $s_v \in \hat{\mathcal{S}}[v]$ can be extended into the sought assignment of schedules in its subtree, $(s_u, t_u)_{u \in V(T_v)}$. In the statement of Algorithm 4.1, we recall that, given a tree $T = (V, E, w)$, tree $T' = (V, E, w')$ is the tree with weights rounded up to the nearest multiple of the length of a slot (see Equation (2)).

The subsequent steps taken in procedure BuildStrategy can be informally sketched as follows. The input tree $T'$ is processed in a bottom-up manner and hence, for an input vertex $v$, the recursive calls for its children $v_1, \ldots, v_l$ are first made, providing schedule assignments for the children (see lines 3–4). Then, the rest of the pseudocode is responsible for using these schedule assignments to obtain all valid schedule assignments for $v$. Lines 10–14 merge the schedules of the children in such a way that a set $\hat{S}^*_i$, $i \in \{1, \ldots, l\}$, contains all schedule assignments computed on the basis of the schedules for the children $v_1, \ldots, v_i$. Thus, the set $\hat{S}^*_l$ is the final product of this part of the procedure and is used in the remaining part. Note

**Algorithm 4.1** Dynamic programming routine BUILDSTRATEGY for a tree $T'$. $L, c \in \mathbb{N}$ are global parameters. Subroutines MERGESCHEDULES and INSERTVERTEX are provided in the full version.

```
1: procedure BuildStrategy(vertex v, box size ω ∈ ℝ)
2:     l ← number of children of v in T'    // Denote by v_1, ..., v_l the children of v.
3:     for i = 1..l do
4:         Ŝ[v_i] ← BuildStrategy(v_i, ω);
5:     s ← 0^L
6:     s.max_child_load ← 0^L
7:     s.must_contain_v ← false
8:     Ŝ_0 ← {s}    // Ŝ_0 contains the schedule with no queries.
9:     // Inductively, Ŝ_i^* is based on merging schedules at v_1, ..., v_i.
10:    for i = 1..l do
11:        Ŝ_i^* ← ∅
12:        for each schedule s ∈ Ŝ_{i-1}^* do
13:            for each schedule s_add ∈ Ŝ[v_i] do
14:                Ŝ_i^* ← Ŝ_i^* ∪ MergeSchedules(s, s_add, ω);
15:    Ŝ[v] ← ∅
16:    for each s ∈ Ŝ_l^* do
17:        if w'(v) > cω then    // v is heavy
18:            for p = 0..L − 1 do    //attempt to insert (into s) query to v starting from time-box p
19:                Ŝ[v] ← Ŝ[v] ∪ InsertVertex(s, v, ω, p · ω)
20:        else    //v is light
21:            for real t = 0..L · ω step 1/(cn) do
22:                //attempt to insert (into s) query to v at a slot from time t
23:                Ŝ[v] ← Ŝ[v] ∪ InsertVertex(s, v, ω, t)
24:        if s.must_contain_v = false then
25:            Ŝ[v] ← Ŝ[v] ∪ InsertVertex(s, v, ω, ⊥)
26:    return Ŝ[v]
```

that a schedule assignment in $\hat{S}_l^*$ may not be valid since a query to $v$ is not accommodated in it – the rest of the pseudocode is responsible for taking each schedule $s \in \hat{S}_l^*$ and inserting a query to $v$ into $s$. More precisely, the subroutine INSERTVERTEX is used to place the query to $v$ at all possible time points (depending whether $v$ is heavy or light). We note that the subroutine MERGESCHEDULES, for each schedule $s$ it produces, sets a Boolean 'flag' $s.must\_contain\_v$ that whenever equals $false$, indicates that querying $v$ is not necessary in $s$ to obtain a valid schedule for $v$ (this happens if $s$ queries all children of $v$). A detailed analysis of procedure BUILDSTRATEGY can be found in the full version.

▶ **Lemma 4.4.** *For fixed constants $L, c \in \mathbb{N}$, calling procedure* BUILDSTRATEGY$(r(T), \omega)$*, where $r(T)$ is the root of the tree, determines if there exists a tuple $(s_v, t_v)_{v \in V}$ which satisfies constraints* (3)*,* (4)*, and* (5)*, or returns an empty set otherwise.*

It follows directly from Lemma 4.4 that, for any value $\omega^*$, tree $T$ may only admit an aligned schedule assignment of duration at most $\omega^* L$ if a call to procedure BUILDSTRATEGY $(r(T), \omega^*)$ returns a non-empty set. Taking into account Lemmas 4.1 and 4.2, we directly obtain the following lower bound on the length of the shortest aligned schedule in tree $T'$.

▶ **Lemma 4.5.** *If* BUILDSTRATEGY$(r(T), \omega^*) = \emptyset$*, then:*

$$\omega^* L < \left(1 + \frac{3}{c}\right) \texttt{OPT}(T') \leq \left(1 + \frac{3}{c}\right)\left(1 + \frac{2}{c}\right) \texttt{OPT}(T) \leq \left(1 + \frac{11}{c}\right) \texttt{OPT}(T).$$

▶ **Lemma 4.6.** *The running time of procedure* BuildStrategy$(r(T), \omega)$ *is at most* $O((cn)^{\gamma L})$*, for some absolute constant* $\gamma = O(1)$*, for any* $\omega \leq n$*.*

To complete the proof of Proposition 3.1, we can now provide a strategy which achieves a small value of $\texttt{COST}^{(\omega,c)}$. This relies on procedure BuildStrategy$(r(T), \omega)$ as an essential subroutine, first determining the minimum value of $\omega = \frac{i}{cn}$, $i \in \mathbb{N}$, for which BuildStrategy produces a schedule. Details of the approach are provided in the full version.

## 5 Reducing the Number of Down-Queries

We start with defining a function $\ell\colon V \to \{1, \ldots, \lceil \log_2 n \rceil\}$ which in the following will be called a *labeling of* $T$ and the value $\ell(v)$ is called the *label of* $v$. We say that a subset of nodes $H \subseteq V$ is an *extended heavy part* in $T$ if $H = \{v\} \cup H'$, where all nodes in $H'$ are heavy, no node in $H'$ has a heavy neighbor in $T$ that does not belong to $H'$ and $v$ is the parent of some node in $H'$. Let $H_1, \ldots, H_l$ be all extended heavy parts in $T$. Obtain a tree $T_C = (V_C, E_C)$ by contracting, in $T$, the subgraph $H_i$ into a node denoted by $h_i$ for each $i \in \{1, \ldots, l\}$. In the tree $T_C$, we want to find its labeling $\ell'\colon V_C \to \{1, \ldots, \lceil \log_2 |V_C| \rceil\}$ that satisfies the following condition: for each two nodes $u$ and $v$ in $V_C$ with $\ell'(u) = \ell'(v)$, the path between $u$ and $v$ has a node $z$ satisfying $\ell'(z) < \ell'(u)$. One can obtain such a labeling by a following procedure that takes a subtree $T'_C$ of $T_C$ and an integer $i$ as an input. Find a central node $v$ in $T'_C$, set $\ell'(v) = i$ and call the procedure for each subtree $T''_C$ of $T'_C - v$ with input $T''_C$ and $i + 1$. The procedure is initially called for input $T$ and $i = 1$. We also remark that, alternatively, such a labeling can be obtained via vertex rankings [13, 28].

Once the labeling $\ell'$ of $T_C$ is constructed, we extend it to a labeling $\ell$ of $T$ in such a way that for each node $v$ of $T$ we set $\ell(v) = \ell'(v)$ if $v \notin H_1 \cup \cdots \cup H_l$ and $\ell(v) = \ell'(h_i)$ if $v \in H_i$, $i \in \{1, \ldots, l\}$.

Having the labeling $\ell$ of $T$, we are ready to define a query sequence $R(v)$ for each node $v \in V$. $R(v)$ contains all nodes $u$ from $T_v$ such that $\ell(u) < \ell(v)$ and each internal node $z$ of the path connecting $v$ and $u$ in $T$ satisfies $\ell(z) > \ell(u)$. Additionally, the nodes in $R(v)$ are ordered by increasing values of their labels.

By $x$ we refer to the target node in $T$. Fix $S$ to be a stable sequence assignment in the remaining part of this section and by $R$ we refer to the sequence assignment constructed above. Then, we fix $S^+$ to be $S^+(v) = R(v) \circ S(v)$ for each $v \in V$. A query made by $\mathcal{A}_{S^+}$ to a node that belongs to $R(v)$ for some $v \in V$ is called an *$R$-query*; otherwise it is an *$S$-query*. In the full version we show that, in $\mathcal{A}_{S^+}$, the total number of $R$-queries does not exceed $2\log_2 n$. Moreover, since $S$ is stable, for each target node $x$, the $S$-queries performed by $\mathcal{A}_{S^+}$ are a subsequence of the queries performed by $\mathcal{A}_S$. Therefore, the potentially additional queries made by $\mathcal{A}_{S^+}$ with respect to $\mathcal{A}_S$ are $R$-queries. We then formally show that each $R$-query is made on a light node and that any $R$-query increases the value of $\texttt{COST}^{(\omega,c)}$ of $\mathcal{A}_{S^+}$ with respect to the value of $\texttt{COST}^{(\omega,c)}$ of $\mathcal{A}_S$ by at most $(2c+1)\omega$. Hence we have: $\texttt{COST}^{(\omega,c)}_{\mathcal{A}_{S^+}}(T) \leq \texttt{COST}^{(\omega,c)}_{\mathcal{A}_S}(T) + 2(2c+1)\omega \log_2 n$.

Moreover, we show in the full version that the total number of queries in strategy $\mathcal{A}_{S^+}$ to light nodes receiving 'down' replies can be likewise bounded by $2\log_2 n$. Since each such query introduces a rounding difference of at most $(2c+1)\omega$ when comparing cost functions $\texttt{COST}$ and $\texttt{COST}^{(\omega,c)}$, we thus obtain: $\texttt{COST}_{\mathcal{A}_{S^+}}(T) \leq \texttt{COST}^{(\omega,c)}_{\mathcal{A}_{S^+}}(T) + 2(2c+1)\omega \log_2 n$.

Combining the above observations gives the claim of the Proposition.

## References


1 Esther M. Arkin, Henk Meijer, Joseph S. B. Mitchell, David Rappaport, and Steven Skiena. Decision trees for geometric models. *Int. J. Comput. Geometry Appl.*, 8(3):343–364, 1998. doi:10.1142/S0218195998000175.

2 Yosi Ben-Asher and Eitan Farchi. The cost of searching in general trees versus complete binary trees. Technical report, Technical report, 1997.

3 Ferdinando Cicalese, Tobias Jacobs, Eduardo Sany Laber, and Marco Molinaro. On the complexity of searching in trees and partially ordered structures. *Theor. Comput. Sci.*, 412(50):6879–6896, 2011. doi:10.1016/j.tcs.2011.08.042.

4 Ferdinando Cicalese, Tobias Jacobs, Eduardo Sany Laber, and Caio Dias Valentim. The binary identification problem for weighted trees. *Theor. Comput. Sci.*, 459:100–112, 2012. doi:10.1016/j.tcs.2012.06.023.

5 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.

6 Dariusz Dereniowski. Edge ranking of weighted trees. *Discrete Applied Mathematics*, 154(8):1198–1209, 2006. doi:10.1016/j.dam.2005.11.005.

7 Dariusz Dereniowski. Edge ranking and searching in partial orders. *Discrete Applied Mathematics*, 156(13):2493–2500, 2008. doi:10.1016/j.dam.2008.03.007.

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

9 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.

10 Ehsan Emamjomeh-Zadeh, David Kempe, and Vikrant Singhal. Deterministic and probabilistic binary search in graphs. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 519–532, 2016. doi:10.1145/2897518.2897656.

11 Archontia C. Giannopoulou, Paul Hunter, and Dimitrios M. Thilikos. Lifo-search: A min-max theorem and a searching game for cycle-rank and tree-depth. *Discrete Applied Mathematics*, 160(15):2089–2097, 2012. doi:10.1016/j.dam.2012.03.015.

12 Brent Heeringa, Marius Catalin Iordan, and Louis Theran. Searching in dynamic tree-like partial orders. In *Algorithms and Data Structures – 12th International Symposium, WADS 2011, New York, NY, USA, August 15-17, 2011. Proceedings*, pages 512–523, 2011. doi:10.1007/978-3-642-22300-6_43.

13 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.

14 Ananth V. Iyer, H. Donald Ratliff, and Gopalakrishnan Vijayan. Parallel assembly of modular products – an analysis. Technical report, Technical Report 88-86, Georgia Institute of Technology, 1988.

15 Meir Katchalski, William McCuaig, and Suzanne M. Seager. Ordered colourings. *Discrete Mathematics*, 142(1-3):141–154, 1995. doi:10.1016/0012-365X(93)E0216-Q.

16 Eduardo Sany Laber, Ruy Luiz Milidiú, and Artur Alves Pessoa. On binary searching with nonuniform costs. *SIAM J. Comput.*, 31(4):1022–1047, 2002. doi:10.1137/S0097539700381991.

17 Eduardo Sany Laber and Marco Molinaro. An approximation algorithm for binary searching in trees. *Algorithmica*, 59(4):601–620, 2011. doi:10.1007/s00453-009-9325-0.

18 Eduardo Sany Laber and Loana Tito Nogueira. Fast searching in trees. *Electronic Notes in Discrete Mathematics*, 7:90–93, 2001. doi:10.1016/S1571-0653(04)00232-X.

**19** Eduardo Sany Laber and Loana Tito Nogueira. On the hardness of the minimum height decision tree problem. *Discrete Applied Mathematics*, 144(1-2):209–212, 2004. doi:10.1016/j.dam.2004.06.002.

**20** Tak Wah Lam and Fung Ling Yue. Optimal edge ranking of trees in linear time. *Algorithmica*, 30(1):12–33, 2001. doi:10.1007/s004530010076.

**21** Nathan Linial and Michael E. Saks. Searching ordered structures. *J. Algorithms*, 6(1):86–103, 1985. doi:10.1016/0196-6774(85)90020-3.

**22** Joseph W. H. Liu. Computational models and task scheduling for parallel sparse cholesky factorization. *Parallel Computing*, 3(4):327–342, 1986. doi:10.1016/0167-8191(86)90014-1.

**23** Joseph W. H. Liu. The role of elimination trees in sparse factorization. *SIAM. J. Matrix Anal. & Appl.*, 11(1):134–172, 1990.

**24** Shay Mozes, Krzysztof Onak, and Oren Weimann. Finding an optimal tree searching strategy in linear time. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*, pages 1096–1105, 2008.

**25** Jaroslav Nesetril 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 *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), 21-24 October 2006, Berkeley, California, USA, Proceedings*, pages 379–388, 2006. doi:10.1109/FOCS.2006.32.

**27** Alex Pothen. The complexity of optimal elimination trees. Technical report, Technical Report CS-88-13, Pennsylvannia State University, 1988.

**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** George Steiner. Searching in 2-dimensional partial orders. *J. Algorithms*, 8(1):95–105, 1987. doi:10.1016/0196-6774(87)90029-0.

**30** Jayme Luiz Szwarcfiter, Gonzalo Navarro, Ricardo A. Baeza-Yates, Joísa de S. Oliveira, Walter Cunto, and Nivio Ziviani. Optimal binary search trees with costs depending on the access paths. *Theor. Comput. Sci.*, 290(3):1799–1814, 2003. doi:10.1016/S0304-3975(02)00084-1.