FACULTY OF ELECTRONICS, TELECOMMUNICATIONS AND INFORMATICS
GDAŃSK UNIVERSITY OF TECHNOLOGY, POLAND

Experimental Analysis of Binary Search Models in Graphs

Michał Szyfelbein

Supervisor: prof. dr. hab. inż. Dariusz Dereniowski

September 20, 2025

Abstract

In this work, we conduct an experimental analysis of the generalized binary search problem in graphs. The analysis explores various query models including: edge, vertex, and general queries, across multiple classes of search spaces, such as: paths, trees, general graphs, and beyond. The study is structured into two main sections:

The first part focuses on the theoretical foundations of the problem. It introduces key definitions, fundamental concepts, and pseudocodes of the analyzed procedures, along with a formal analysis of their parameters. The significance of these results was evaluated based on two primary metrics: the computational complexity and theoretical bounds on the quality of the solutions obtained.

The second part provides experimental verification of the theoretical claims established in the previous chapters. It also presents a practical comparison of the algorithmic approaches developed for different problem variants. The proposed procedures were evaluated across diverse graph classes thus ensuring complete results. To guarantee thorough and unbiased coverage of the problem space, all of the test instances were generated using randomized techniques and multiple input sizes were tested.

Keywords and phrases Trees, Graph Searching, Binary Search, Decision Trees, Ranking Colorings, Graph Theory, Approximation Algorithm, Combinatorial Optimization, Experimental Analysis of Algorithms

Glossary

This document is incomplete. The external file associated with the glossary 'main' (which should be called `main.gls`) hasn't been created.

Check the contents of the file `main.glo`. If it's empty, that means you haven't indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can't be generated. If the file isn't empty, the document build process hasn't been completed.

If you don't want this glossary, add `nomain` to your package option list when you load `glossaries-extra.sty`. For example:

```
\usepackage[nomain]{glossaries-extra}
```

Try one of the following:

- Add `automake` to your package option list when you load `glossaries-extra.sty`. For example:

  ```
  \usepackage[automake]{glossaries-extra}
  ```

- Run the external (Lua) application:

  ```
  makeglossaries-lite.lua "main"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "main"
  ```

Then rerun LaTeX on this document.
This message will be removed once the problem has been fixed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The search problem

The Binary Search is a classical algorithm used to efficiently locate a hidden target element in a linearly ordered set. To do so, the searcher repeatedly picks the median element of such set, performs a comparison operation and in constant time learns if the target was found and if not, whether the target is above or below the median (For example see Figure 1.1).

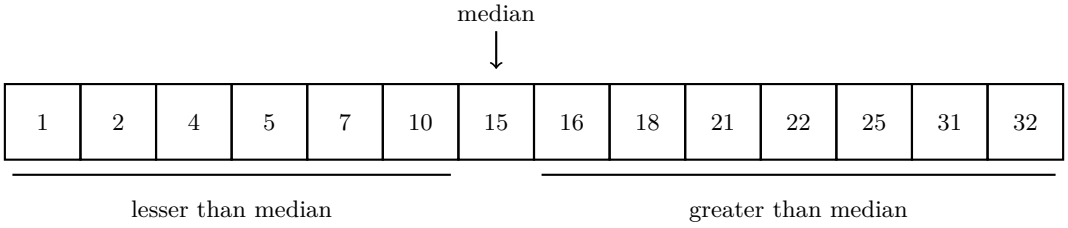

Figure 1.1: Example of a sorted array containing 14 elements. The subarrays with elements lesser than and greater then the median (15) are underlined. If the hidden element is for example 2, then the result of the comparison operation is "below" and the searcher can immediately discard all elements of value above 10. If the target were to be 15, then the comparison operation would yield "equal" meaning that the median element is in fact the target.

The study of searching was initiated by D. Knuth in his seminal book [Knu73] in which he discussed its various variants. However, the origins of the search problem reach the famous Rényi-Ulam game of twenty one questions in which a player is required to guess an unnamed object by asking yes-or-no questions[1]. Throughout the years, the searching and its variants have been

[1] Note that in the twenty-one questions game one answer to a question may be a lie.

continuously rediscovered under various definitions and names. This hints that the intuitions behind this problem resurface among multiple use cases and research domains. In fact, the search problem in its many variants is deeply connected with many other algorithmic notions including: parallelization of the Cholesky factorization, scheduling join operations in database queries, VLSI-layouts, learning theory, data clustering, graph cuts and parallel assembly of multi-part products. This work aims to serve as a survey of the results obtained for the problem and an experimental analysis of algorithms aimed at solving it.

The importance of searching is also due to its various practical applications. For example consider the following scenario: a complex procedure contains a hidden bug required to be fixed. The procedure is composed of multiple (often nested) blocks of code. In order to find this hidden bug the searcher can perform tests which allow him to check whether the given block of code contains the bug. After performing each such test they learn whether the bug is in or outside of the tested block. This process then continuous, until the bug is found. The problem is to find the best testing strategy for the tester in order to find the bug efficiently.

A different scenario may occur in the medical diagnostics. The so called *House M.D. Problem* is concerned with diagnosing a potentially lethal, hidden disease. In order to do so, House and his medical team perform series of tests on the patient. These tests (often avant-garde in their nature) may include blood tests, a family survey or even breaking into the patients house. After performing each such test the team learns some new information about the patient which allows them to iteratively narrow the size of the space of possible diagnoses. The outcome of such test may include for example: the sugar level in blood is low (or high), the patients mother died of a disease with similar symptoms, the patient consumes very large amount of tuna etc. As the condition of the patient is deteriorating rapidly, the team needs to determine the disease as quickly as possible.

1.2 Problem statement

Tree Search More formally, we model the search space as a tree T. The *Vertex Tree Search Problem* is as follows: Among vertices of T there is a hidden target vertex x which is required to be located[2]. During the search process, the searcher is allowed to perform queries, each about a chosen vertex $v \in V(T)$. In constant time the oracle responds whether the target is v and if not, it identifies which connected component of $T - v$ contains x. Upon learning this information the searcher then iteratively picks the next vertex to query until the target is found. The goal is to create the optimal strategy for the searcher. One may also define an analogous process in which the queries concern edges. After a query to an edge e the searcher learns which connected component of $T - e$ contains the target. We will call this problem the *Edge Tree Search Problem*. For a visual example for both query models see Fig 1.5.

[2]It should be pointed out that the target vertex might not always be the same across multiple searches.

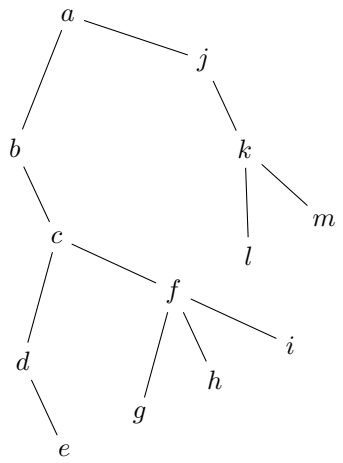

Figure 1.2

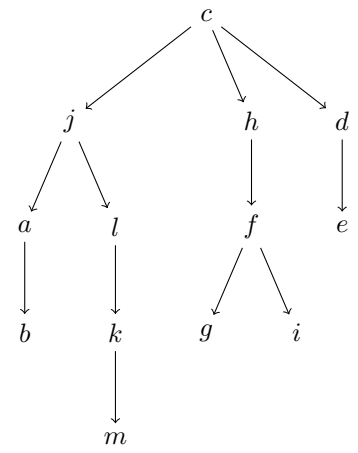

Figure 1.3

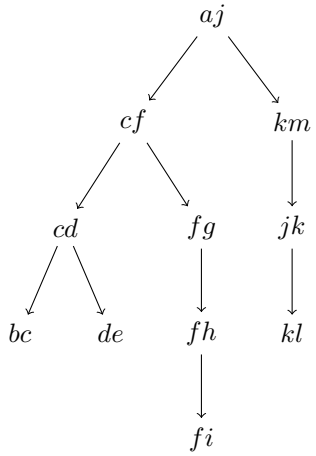

Figure 1.4

Figure 1.5: Sample input tree T (Figure 1.2) and two decision trees for T: one for the Vertex Tree Search Problem (Figure 1.3) and one for the Edge Tree Search Problem (Figure 1.4).

When the input tree is a path both problems become the clasical binary search in the linearly ordered set. We remark that for the vertex variant sometimes an alternative definition is provided. Upon query to v, if it is not the target, the response is an edge[3] incident to v which is the closest towards the target. This definition is equivalent as each component of $T - v$ has exactly one edge/vertex connecting it to v. A similar alternative/equivalent definition holds for the edge Tree Search Problem in which the response is the unique endpoint of e laying closer to the target. The distinction between the two ways of defining the problem becomes significant when attempting to generalize it to arbitrary graphs.

Graph Search In the following considerations we will be concerned with the generalization of the first definition of the Tree Search Problem. Given a graph G the *Vertex Graph Search Problem* is as follows: Among vertices of G there exists a hidden target vertex x which is required to be located. During the search process, the searcher is allowed to perform queries, each about a chosen vertex $v \in V(G)$. In constant time the oracle responds whether the target is v and if not, it identifies which connected component of $G - v$ contains x. Again, the goal is to create the optimal strategy for the searcher. As before, a similar definition can be provided for the edge query model, which provides us with the *Edge Graph Search Problem*. For a visual example of both query models see Figure 1.9.

[3]Or equivaletly a vertex.

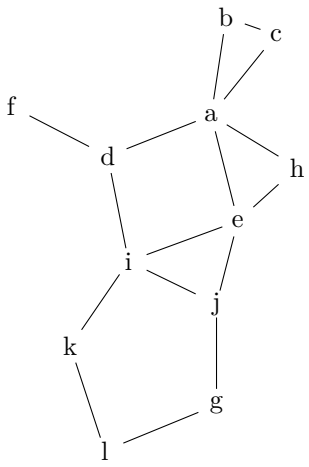

Figure 1.6

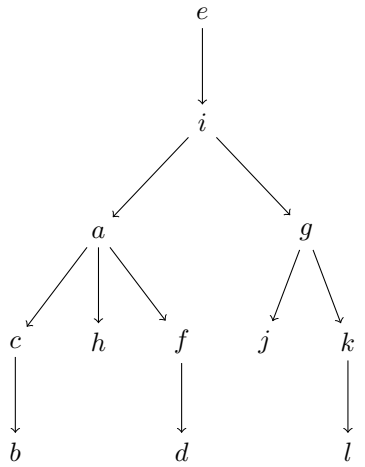

Figure 1.7

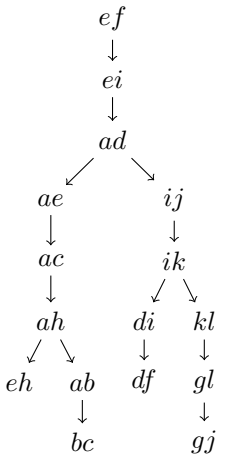

Figure 1.8

Figure 1.9: Sample input graph G (Figure 1.6) and two decision trees for G: one for the Vertex Tree Graph Problem (Figure 1.7) and one for the Edge Graph Search Problem (Figure 1.8).

Poset Search and the Binary Identification Problem A different generalization of the Edge Tree Search Problem is the *Poset Search Problem*. In this problem we are given a partially ordered set (poset) $\mathcal{P} = (X, \preceq)$, where X denotes the ground set of objects and $\preceq$ is a partial ordering of elements of X. In order for the problem to be properly defined we also require that $\mathcal{P}$ contains a unique maximum element r. Among X there is a unique hidden target element x which is required to be located. During the search process, the searcher is allowed to perform queries asking whether $x \preceq v$ for a chosen $v \in X$. Once again, the goal is to create the optimal strategy for the searcher. For a visual example see Figure 1.12.

The Poset Search Problem can be generalized even further by allowing each available query to be any partition of the search space. In the *Binary Identification Problem* we are given a pair $(\mathcal{H}, \mathcal{Q})$ where $\mathcal{H}$ is a set of hypotheses and $\mathcal{Q}$ is a set of queries. Each query $q = \{R_1, R_2, ...R_k\}$ is a partition of $\mathcal{H}$ (we require that $\bigcup_{R \in q} R = \mathcal{H}$ and for any $R_1, R_2 \in q$: $R_1 \cap R_2 = \emptyset$). After performing a chosen query the searcher obtains information which $R \in q$ contains the hidden target. This process continuous iteratively until the target if found. For a visual example see Figure 1.15.

In this broad sense this general model of searching can be interpreted using the information theory point of view. There exists a unique true hypotheses h among the set $H = \{h_1, ..., h_n\}$ which the learner is trying to obtain. In order to do so, they iteratively collect small pieces of information which allow them to rule out certain hypothesis narrowing the search space. They do so until there is only one such hypothesis left. Analogously, the goal is to design a learning strategy which enables an efficient obtaining of this hidden hypothesis.

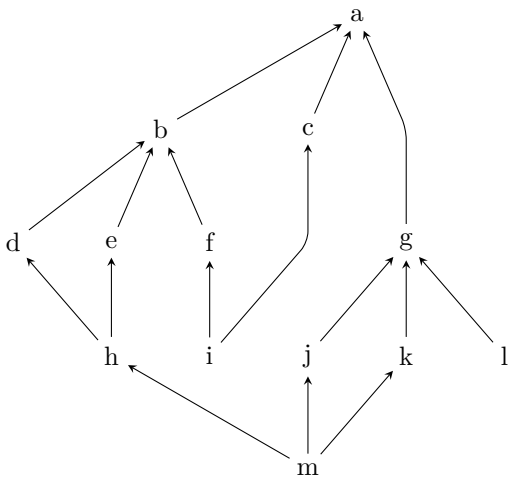

Figure 1.10

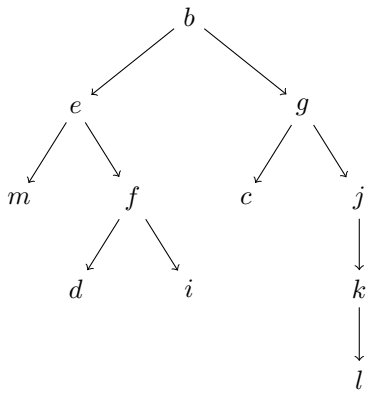

Figure 1.11

Figure 1.12: Sample poset $\mathcal{P}$ (Figure 1.10) and a decision tree for it (Figure 1.11).

	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9
a	1	2	1	1	1	1	1	1	1
b	1	1	1	1	2	2	1	1	1
c	1	3	1	1	3	3	1	2	1
d	1	3	2	1	3	1	1	2	1
e	2	1	2	2	1	2	1	2	1
f	2	2	2	1	1	1	1	2	1
g	2	1	2	1	1	2	2	2	1
h	2	1	2	3	2	3	1	2	1
i	2	1	2	1	2	3	1	2	2
j	2	1	2	1	2	2	2	2	2
k	2	1	2	4	2	2	2	2	2

Figure 1.13

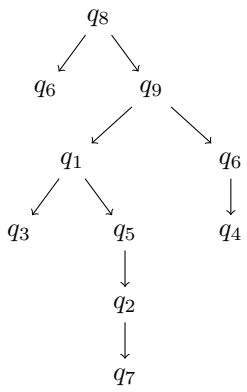

Figure 1.14

Figure 1.15: Sample Binary Identification Problem instance (Figure 1.13) and a decision tree for it (Figure 1.14).

Strategies, Decision Trees and their costs Notice, that while defining the searching the term strategy was never properly defined. The *search strategy* is an adaptive algorithm which (in polynomial time) provides the searcher with the next query to perform (given the previous responses). A natural way to visualize such strategy is to see it as a decision tree. A *decision tree* D is a rooted tree in which each vertex represents a query and each edge represents a possible response. The search is conducted by choosing as the next query the root of D. After receiving the response (If the search is not terminated) the searcher moves along the edge $e = (r, r_e)$ incident to r associated with the response. The process then recurses in D_{r_e} until the target is found. It should be noted however, that this is far from the only viable way of encoding the search strategy. The choice of the data structure used is a matter of taste and often leads to simpler design and analysis of the algorithms.

In order to sensibly talk about the quality of such strategy we need to also measure its cost. The cost of locating a vertex x using a strategy A is the amount of queries required to be performed to find x using $\mathcal{A}$. The two most intuitive ways to measure the overall cost of $\mathcal{A}$ are:

- The worst case search time which is maximum of costs of $\mathcal{A}$ over all vertices
- The average case which is the sum of costs of $\mathcal{A}$ over all vertices[4].

Even though similar, these two criterion often differ in their analysis and algorithms constructed for them usually exploit slightly different properties of the input. It is often the case that greedy heuristics perform much better when we measure the average case cost of the decision trees created by them. In contrast, in the worst case, often the best known solutions require some intricate dynamic programming procedure as an essential subroutine. Interestingly enough, it is not hard to show that given two decision trees: one with good performance in the average case and one with good performance in the worst case, a simple algorithm can be used to create new decision tree with fairly good performance in both metrics [SLC14].

Weights and Costs Above, we have made the assumption that performing each query costs us exactly the same. In real life applications it might not be a case. For example, determining the value of some complex comparison operation for two large objects may take a substantial amount of time. In such cases we associate with each query a cost function. To calculate the cost of finding x, instead of measuring the amount of queries, we measure the sum of their costs. The worst case and the average case criteria are then defined according to this new values.

Additionally, when dealing with the average case version of the problem, one may consider a scenario in which certain vertices are searched for more often then the others. In general, we can associate with each vertex a probability/frequency of it being searched for which we will call its

[4]The average of and the sum are equivalent up to a constant factor of n.

weight. In this case the average query time naturally becomes the weighted average according to this weight function [5].

1.3 The three field notation for the search problem

One may see that the multiplicity of variants for the problem have started to be somewhat problematic. Ideally, we would like to introduce some unified way of speaking about the problem to avoid ambiguity. This is problematic because historically, various variants of the problem were often explored independently. To alleviate this inconvenience we introduce the following three field notation resembling the notation commonly used in task scheduling problems. Similarly, our notation will consists of the three following fields: α, β and γ. The α field is the search space environment field resembling the machine environment. The β field is the query characteristics which resembles the job characteristics. The γ field is the objective function which we are trying to optimize. In order not to confuse these two notations in contrary to single line separator used in scheduling $(\alpha|\beta|\gamma)$, we will separate the three fields with doubled lines: $\alpha||\beta||\gamma$. The following table showcases example variants which may be considered:

α - search space	β - query characteristics	γ - objective value
P - paths	E - edge queries	C_{max} - maximum search time
T - trees	V - vertex queries	$\sum C_i$ - average search time
$POSET$ - POSETs	Q - any queries	$\sum U_i$ - throughtput
G - graphs	c - cost function on queries	F_{max} - maximum flow time
HT - hypertrees	w - weight function on vertices	$\sum F_i$ - average flow time
HG - hypergraphs	d - due dates	L_{max} - maximum lateness
S - any set of hypothesis	$\bar{d}$ - strict deadlines	$\sum L_i$ - average lateness
	r - release times	T_{max} - maximum tardiness
	$prec$ - precedences	$\sum T_i$ - average tardiness

Table 1.1: Sample values for the three field notation for the search problem.

The striking resemblance between these two notations suggests that we can view the search problem as a specific form of scheduling, in which the search strategy is the schedule and the queries are the jobs. From the perspective of the researcher however, the search problem is not nearly as explored as the scheduling problems and most of the variants which can be constructed using the table above are not even mentioned in the literature. It also seems, that the search problem is in a sense harder than the usual scheduling. For example, the best algorithm[6] known for the NP-hard variant $T||V, c||C_{max}$ achieves an $O\big(\sqrt{\log n}\big)$-approximation [Der+17]. A somewhat

[5] We assume that these cost and weight functions are known *a-priori*.

[6] This algorithm is obtained by a recursive usage of a QPTAS obtained via a non-trivial dynamic programming procedure. For details see: [insert ref here].

similar scheduling problem $P||C_{max}$ has a simple $\frac{4}{3}$-approximation algorithm based on sorting the jobs according to their costs [Gra69], admits a PTAS for an unbounded number of machines [Leu89] and if the number of machines is bounded an FPTAS can be obtained [Sah76].

1.4 Many names, one problem

As mentioned above, the search problem has been continuously rediscovered under various names and definitions. It is usually the case, that each of these definitions is equivalent when the search space is a tree. The following list consists of different formulations under which the problem have been studied in the context of graphs:

- Binary Search [OP06; Der+17; DMS19; EKS16; DW22; DW24; DŁU25; DGW24; DŁU21; DGP23],
- Tree Search Problem [Jac+10; Cic+14; Cic+16],
- Binary Identification Problem [Cic+12; KZ13],
- Ranking Colorings [Knu73; Der06; Der08; DK06; DN06; LY98],
- Ordered Colorings [KMS95],
- Elimination Trees [Pot88],
- Hub Labeling [Ang18],
- Tree-Depth [NO06; BDO23],
- Partition Trees [Høg+21; Høg24],
- Hierarchical Clustering [CC17],
- Search Trees on Trees [BK22; Ber+22],
- LIFO-Search [GHT12].

Various different problem definitions stem from the learning theory including:

- Decision Tree [LN04; LLM; GNR10; SLC14],
- Bayesian Active Learning [GKR10; Das04],
- Discrete Function evaluation [CLS14],
- Tree Split [KPB99],

- Query Selection [BBS12].

Each of these problems is equivalent when restricted to trees, however for arbitrary graphs/search spaces this might not be the case. For example, in the generalized binary search problem in graphs (by convention) the response to the query to v is the neighbor of v laying on the shortest path towards the target. In contrast, in the Hierarchical Clustering the response to such query is the connected component of $T - v$ containing the target.

Among ambiguities resulting form the multiplicity of the problem definitions is the interpretation of the notion of weight. When considering the worst case cost, usually the weight is interpreted as the cost of a query. On the other hand, while analyzing the average case cost, the weight is usually the probability that the vertex will be searched for. In such scenario, the cost of the query (if present) is usually called cost. In this work we follow the second convention and we usually denote them as $w(v)$ and $c(v)$ (or $c(e)$ in edge query model) accordingly.

1.5 The aim of the thesis

Hereby, we will be mostly concerned with the situation in which the input graph is tree. A motivation for this is twofold. Firstly, trees come up most often in the practical scenarios regarding the problem. Secondly, from the algorithmic perspective, the most interesting and structural results are obtained for trees. Beyond that, most of the algorithms with provable guarantees follow some simple greedy rule and the achieved approximations are far from the objective value. For example, the problem $T||V||C_{max}$ is solvable in linear time (the algorithm is non-trivial)[Sch89]. If we however allow arbitrary graphs ($G||V||C_{max}$) then the problem becomes NP-hard even in chordal graphs [DN06] and the best known approximation in general case is $O\left(\log^{\frac{3}{2}} n\right)$ which is trivially obtained via an almost blackbox use of the tree decomposition of the graph [Bod+98].

We conclude a series of experiments aimed at verifying whether the theoretical claims regarding the discussed algorithms are reflected in an experimental setup. In particular, we employ randomized techniques to generate various classes of inputs and test the performance of the implemented algorithms both in terms of running time and the quality of the solutions obtained.

1.6 Organization of the work

The second chapter serves as a more formal and detailed introduction necessary for further considerations. We formally restate all of the search models we are interested in and we recall the basic notions of graph theory required for the analysis.

The main part of the thesis is partitioned into two main chapters:

In the third chapter we focus ourselves on the formal analysis of the considered variants including the presentation of the most interesting algorithmic results for the problem. We showcase exact and approximation algorithms and few hardness results for the most complex variants of the search problem.

The fourth chapter is a description of the computer experiments conducted in order to verify the theoretical claims regarding the performance of the previously presented algorithms.

The fifth chapter serves as a summary of our considerations and points the further research directions regarding this field.

Chapter 2

Notions and Definitions

2.1 Graph theory

A *graph* is a pair $G = (V(G), E(G))$ where $V(G)$ is the set of *vertices* and $E(E)$ is the set of *edges* which are unordered pairs of vertices. We denote $n(G) = |V(G)|$ and $m(G) = |E(G)|$. For $u, v \in V(G)$ by uv we denote the edge which connects them. A *subgraph* of a graph G is another graph G' formed from a subset of the vertices and edges of G. For any $V' \subseteq V(G)$ by $G[V']$ we denote the *subgraph induced* by V' in G (i. e. for every $u, v \in V'$ if $uv \in E(G)$, then also $uv \in E(G')$). Additionally, by $G - V'$ we denote the set of connected components occurring after deleting all vertices in V' from G. The set of *neighbors* of $v \in V(G)$ will be denoted as $N_G(v) = \{u \in V(G) \,|uv \in E(G)\}$ and the set of neighbors of subgraph G' of G as $N_G(G') = \bigcup_{v \in V(G')} N_G(v) - V(G')$. By $\deg_G(v) = |N_G(v)|$ we will denote the *degree* of v in G. By $\Delta(G) = \max_{v \in V(G)}\{|\deg(v)|\}$ we denote the degree of G.

A *cycle* is a non-empty sequence of vertices in which for every two consecutive vertices u, v: $uv \in E(G)$ and only the first and last vertices are equal. A *tree* T is a connected graph that contains no cycle. A *forest* is a (not necessarily connected) graph that contains no cycle. A *path* P is a tree such that $\Delta(P) = 2$. Let $v \in V(T)$. The *outdegree* of v in T will be denoted as $\deg^+_T(v) = |\mathcal{C}_T(v)|$. By $P_T(u, v) = T\langle\{u, v\}\rangle - \{u, v\}$ we denote a path of vertices between u and v in T (excluding u and v). Analogously, for $V_1, V_2 \in V(T)$ we define $P_T(V_1, V_2) = T\langle V_1 \cup V_2\rangle - (V_1 \cup V_2)$. For any we denote the minimal connected subtree of T containing all vertices from V' by $T\langle V'\rangle$.

A partial ordering $\preceq$ is a two-argument relationship which is: reflective ($a \preceq a$), antisymmetric (if $a \preceq b$ and $b \preceq a$ then $a = b$) and transitive (if $a \preceq b$ and $b \preceq c$ then $a \preceq c$). A poset is a pair $\mathcal{P} = (X, \preceq)$ where X is the set of elements and $\preceq$ is a partial ordering of elements of in X. When clear from the context the set X itself is also sometimes called a poset.

2.2 Optimization algorithms

A *minimization problem* is one in which given an input I, the set of valid solutions S and a cost function $c : S \to \mathbb{R}^+$ we are required to find a solution $s^* \in S$ such that $c(s^*) = \min_{s \in S}\{c(s)\}$. Analogously, a *maximization problem* is one in which we are required to find a solution $s^* \in S$ such that $c(s^*) = \max_{s \in S}\{c(s)\}$. For both types of problems we define $\texttt{OPT}(I) = c(s^*)$. Given an instance (I, S, c) of a minimization problem such that $|I| = n$, an $\alpha(n)$-approximation algorithm is an algorithm which always outputs a solution s such that:

$$\frac{c(s)}{\texttt{OPT}(I)} \leq \alpha(n)$$

Analogously, for a maximization problem such an $\alpha(n)$-approximation algorithm is an algorithm which always outputs a solution s such that:

$$\frac{c(s)}{\texttt{OPT}(I)} \geq \alpha(n)$$

If $\alpha(n) = O(1)$ we say that the algorithm is a constant factor approximation algorithm for I. If $\alpha(n) = 1$ we say that the algorithm is an exact algorithm for I. For a minimization problem if for every $0 < \epsilon \leq 1$ the algorithm provides a $(1 + \epsilon)$-approximation (or a $(1 - \epsilon)$-approximation in case of a maximization problem) and:

- Runs in time $\text{poly}(n/\epsilon)$, then it is called a Fully-Polynomial Time Approximation Scheme (FPTAS).
- Runs in time $f(\epsilon) \cdot \text{poly}(n)$ for some computable function f, then it is called a Efficient-Polynomial Time Approximation Scheme (EPTAS).
- Runs in time $n^{O(1/\epsilon)}$, then it is called a Polynomial Time Approximation Scheme (PTAS).
- Runs in time $n^{\text{poly}(\log n/\epsilon)}$, then it is called a Quasi-Polynomial Time Approximation Scheme (QPTAS).

2.3 The Search Problem

Below we list the definitions regarding the search problem. We start with definitions of all viable search spaces and query models connected with them. Since the problem has a modular form and one can almost freely swap criteria and constraints, the number of separate variants is very large. Due to this we define all of the possible search spaces, query models and criteria separately and then we formulate the *General Search Problem* based on the chosen variant.

2.3.1 Search spaces with according query models

The *Tree Search Instance* consists of a pair $T = (V(T), E(T))$. Among $V(T)$ there is a unique hidden target element x which is required to be located. During the *Search Process* the searcher is allowed to iteratively perform a *query* which asks about chosen vertex (or alternatively an edge e). If the answer is affirmative, then v is the target, otherwise a connected component $H \in T - v$ is returned such that $x \in V(H)$ (for the edge version always $H \in T - e$ is returned). Based on this information the searcher narrows the subtree of T which might contain x until there is only one possible option left.

The *Graph Search Instance* consists of a pair $G = (V(G), E(G))$. Among $V(G)$ there is a unique hidden target element x which is required to be located. During the *Search Process* the searcher is allowed to iteratively perform a *query* which asks about chosen vertex (or alternatively an edge e). If the answer is affirmative, then v is the target, otherwise a connected component $H \in G - v$ is returned such that $x \in V(H)$ (for the edge version always $H \in G - e$ is returned). Based on this information the searcher narrows the subgraph of G which might contain x until there is only one possible option left.

Remark 2.3.1.1. *In the vertex query model we require that every vertex must be queried even when such vertex is the last among the candidate set. Note that it is sometimes assumed that in such case, this vertex does not need to be queried which may reduce the cost of the solution. Note that all of the algorithms showed in this work can be altered to take this assumption into account. For the sake of the brevity we do not include them but we encourage the reader to obtain them as an exercise.*

The *Poset Search Instance* consists of a pair $\mathcal{P} = (X, \preceq)$. Among X there is a unique hidden target element x which is required to be located. During the *Search Process* the searcher is allowed to iteratively perform a *query* which asks about chosen element $v \in X$ and as the answers receives information whether $x \preceq v$. Based on this information the searcher narrows the subset of X which might contain x until there is only one possible option left.

The *Binary Identification Problem Instance* consists of a pair $\mathcal{P} = (\mathcal{H}, \mathcal{Q})$. In the *Binary Identification Problem* we are given a pair $(\mathcal{H}, \mathcal{Q})$ where $\mathcal{H}$ is a set of hypotheses and $\mathcal{Q}$ is a set of queries. Each query $q = \{R_1, R_2, ...R_k\}$ is a partition of $\mathcal{H}$ (we require that $\bigcup_{R \in q} R = \mathcal{H}$ and for any $R_1, R_2 \in q$: $R_1 \cap R_2 = \emptyset$). Among $\mathcal{H}$ there is a unique hidden target hypothesis x which is required to be identified. During the *Search Process* the searcher is allowed to iteratively perform a chosen query q. As the response the searcher obtains information which $R \in q$ contains the hidden target. Based on this information the searcher narrows the subset of $\mathcal{H}$ which might contain x until there is only one possible option left.

2.3.2 Additional input parameters

As a part of the input we will also allow the cost function. Let $\mathcal{Q}$ be the space of possible queries. The cost of query $q \in \mathcal{Q}$ is then denoted as $c : \mathcal{Q} \to \mathbb{R}^+$. We will also allow the weight function. Let X be the space of the possible targets. The weight of query $x \in X$ is then denoted as $w : X \to \mathbb{R}^+$.

2.3.3 Decision trees, optimization criteria and the Generalized Search Problem

Let $I = (X(I), \mathcal{Q}(I))$ be an arbitrary search space and c, w be the cost and weight functions. A decision tree is pair $D = (V(D), E(D), m)$ where $V(D) = X(I)$ are vertices and $E(D)$ are edges of D.

It is required that each child of $q \in V(D)$ corresponds to a distinct response to the query at q, with respect to the subset of candidate solutions that remain after performing all queries along the path $r(D) \circ P(r(D), q)$.

Let $Q_D(X(I), x)$ denote the sequence of queries made to locate a target $x \in X$ using D. We define the cost of searching for x using D in (I, c) as:

$$\texttt{COST}_{max,D}(I, c, x) = \sum_{q \in Q_D(X(I),x)} c(q)$$

We define the worst case cost of a decision tree D in I with weight function w as:

$$\texttt{COST}_{avg,D}(I, w) = \max_{x \in X(I)} \{\texttt{COST}_D(I, x)\}$$

By a slight abuse of notation we will also sometimes use $Q_D(X(I), x)$ as the set consisting of queries in sequence $Q_D(X(I), x)$. This is done in order to not inflate the amount of symbols and will not become problematic during the analysis of the solutions. Whenever clear from the context, for the clarity of the analysis, we will occasionally drop any of the subscripts or arguments of the COST function. We are now ready to define the *Generalized Search Problem*:

Generalized Search Problem

Input: Search space, restricitions on the query model and the optimization criterion

Output: A viable decision tree for the input search space fulfilling all of the restrictions of the query model which optimizes the criterion.

Chapter 3

Theorethical Analysis

The following chapter is concerned with the presentation and theoretical analysis of the algorithms for the Search Problem. We partition the analysis into 5 main sections: Paths, Unitary costs in trees, Non-uniform costs in trees, Arbitrary graphs and Miscellaneous. The variants are grouped according to the similarity of structure, hardness and the techniques used to solve them. It should be noted that however this choice is arbitrary as sometimes distant versions of the problem remain connected and some techniques used to solve one version might be somewhat useful in the other.

3.1 Paths

In general, all of the variants of the problem dealing with paths are known to be solvable in polynomial time. This is due to the fact that the number of subpaths of a path is of size $\binom{n}{2} = O(n^2)$. Such property allows us to construct efficient dynamic programming solutions, which when naively implemented, usually run in time $O(n^3)$. The key part of the analysis is often to show how to optimize such solution in order to reduce the factor of $O(n)$ thus obtaining $O(n^2)$ running time. Let $V(P) = v_1, \ldots, v_n$. In the following considerations by $\texttt{OPT}_{sum}(i,j)$ we will denote the cost of optimal decision tree for a subpath $v_i, .., v_j$ according to the average case cost and similarly by $\texttt{OPT}_{max}(i,j)$ we will denote the cost of optimal decision tree for a subpath $v_i, .., v_j$ according to the worst case cost. Whenever clear from the context we will drop the subscript and simply write $\texttt{OPT}(i,j)$. For both variants we have that $\texttt{OPT} = \texttt{OPT}(1,n)$. By a slight abuse of notation we will also use $\texttt{OPT}(i,j)$ to denote the root query of the decision tree whose cost is equal to this value. In this section we will be only concerned with the edge query model, as each of the constructed solutions can be easily altered to solve the vertex query version of the problem. The following paragraph introduces us with the general recurrence relationships exploited in the dynamic programming.

A warm up: $O(n^3)$ algorithm for $P||E,c||\texttt{COST}_{max}$ and $P||E,c,w||\sum C_i$ First of all, following the definition of $\texttt{OPT}(i,j)$ we get that $\texttt{OPT}(i,i)=0$ for both worst and average case criteria. Fix some $i<j$. The two next recurrence relationships stem from a fact that there must exist some query among edges of $v_i,..v_j$ which is the root of the optimal decision tree. Define D_{max} to be this decision tree (for the worst case cost) and $q_{max}=r(D_{max})$ to be its root query. Let $P_1,P_2\in P-q_{max}$ and let $D_1,D_2\in D-q_{max}$ be the subtrees of D_{max} being decision trees for P_1 and P_2 accordingly. Let D'_{max} be the costiest of the two and P'_{max} be the according path. We have that: $\texttt{OPT}_{max}(i,j)=\texttt{COST}_{max}(D_{max})=c(q_{max})+\texttt{COST}_{max}(D'_{max})$. By the optimality of $\texttt{OPT}_{max}(i,j)$ we immediately obtain that D'_{max} is the optimal decision tree for P'_{max}. This results in the following recurrence relationship:

$$\texttt{OPT}_{max}(i,j)=\min_{i\le k<j}\{c(v_kv_{k+1})+\max\{\texttt{OPT}_{max}(i,k),\texttt{OPT}_{max}(k+1,j)\}\}$$

Similarly, fix again $i<j$ and define D_{sum} to be the optimal decision tree (for the average case cost) for $v_i,\ldots,v_j$ and $q_{sum}=r(D_{sum})$ to be its root query. Let $P_1,P_2\in P-q_{sum}$ and $D_1,D_2\in D-q_{sum}$ be the subtrees of D_{sum} begin decision trees for P_1 and P_2 accordingly. Let $C(D_{sum},x)$ denote the cost of finding $x\in\{v_i,\ldots,v_j\}$ using D_{sum}. Let $w(i,j)=\sum_{i\le k\le j}w(v_k)$. We have that:

$$\texttt{OPT}_{sum}(i,j)=\sum_{x\in v_i,\ldots,v_j}w(x)\cdot\texttt{COST}_{sum}(D_{sum},x)=w(i,j)\cdot c(q_{sum})+\texttt{COST}_{sum}(D_1)+\texttt{COST}_{sum}(D_2)$$

By the optimality of $\texttt{OPT}_{max}(i,j)$ we immediately obtain that D_1 is the optimal decision tree for P_1 and D_2 is the optimal decision tree for P_2. This results in the following recurrence relationship:

$$\texttt{OPT}_{sum}(i,j)=\min_{i\le k<j}\{w(i,j)\cdot c(v_kv_{k+1})+\texttt{OPT}_{sum}(i,k)+\texttt{OPT}_{sum}(k+1,j)\}$$

Both recurrences can be solved using dynamic programming in time $O(n^3)$. To see this, recall that there are at most $\binom{n}{2}$ values for all possible choices of values of i and j and each of them requires checking at most $n-1$ choices for the root of the optimal decision tree.

Despite being polynomial, $O(n^3)$ is substantial amount of calculation for the practical purposes we are interested in. In subsequent considerations we show how, using clever tricks, lower down the running time of these dynamic programming algorithms in some special cases.

3.1.1 Average case, non-uniform weights

Theorem 3.1.1.1. *There exists an $O(n^2)$ algorithm for $P||E,w|\sum C_i$*

Proof. The idea behind the speed-up described below is due to Knuth [Knu73] and [Yao80]. For every $i\le k<j$ let $\texttt{OPT}_k(i,j)=w(i,j)+\texttt{OPT}(i,k)+\texttt{OPT}(k,j)$ be the optimal cost of searching in

$v_i, \ldots, v_j$ assuming that the edge $v_k v_{k+1}$ is the root of the decision tree. We have that $\texttt{OPT}(i,j) \leq \texttt{OPT}_k(i,j)$. Additionally, define $K(i,j) = \max_{i \leq k < j}\{k | \texttt{OPT}_k(i,j) = \texttt{OPT}(i,j)\}$ to be the largest index such that setting $v_k v_{k+1}$ as the root of the decision tree yields the optimal solution.

Let $i \leq i' \leq j \leq j'$. Observe that the weight function fulfills the following inequalities:

1. Monotocity: $w(i',j) \leq w(i,j')$
2. The quadrangle inequality (QI): $w(i,j) + w(i',j') \leq w(i',j) + w(i,j')$

Using the above fact we will firstly show the following:

Lemma 3.1.1.2. *The OPT function satisfies the quadrangle inequality*

Proof. Let $i \leq i' \leq j \leq j'$. The proof is by induction on $l = j' - i$. Assume by induction that:

$$\texttt{OPT}(i,j) + \texttt{OPT}(i',j') \leq \texttt{OPT}(i',j) + \texttt{OPT}(i,j')$$

Whenever $i = i'$ or $j = j'$ the claim follows trivially and therefore is true for $j \leq 1$ so assume otherwise. There are two cases:

1. $i' = j$. Let $k = K(i,j')$. In this case the inequality reduces to $\texttt{OPT}(i,j) + \texttt{OPT}(j,j') \leq \texttt{OPT}(i,j')$. There are two subcases:
   (a) $k \leq j$. We have that:

$$\begin{aligned}
\texttt{OPT}(i,j) + \texttt{OPT}(j,j') &\leq \texttt{OPT}_k(i,j) + \texttt{OPT}(j,j') \\
&= w(i,j) + \texttt{OPT}(i,k) + \texttt{OPT}(k+1,j) + \texttt{OPT}(j,j') \\
&\leq w(i,j') + \texttt{OPT}(i,k) + \texttt{OPT}(k+1,j') \\
&= \texttt{OPT}_k(i,j') \\
&= \texttt{OPT}(i,j')
\end{aligned}$$

   Where the first inequality and the first equality are due to the definition of $\texttt{OPT}_k$, the second inequality is due to monotonicity of w and the induction hypothesis and the last two equalities are again due to the definition of $\texttt{OPT}_k$.
   (b) The case when $k \geq j$ is symmetrical.
2. $i' < j$. Let $y = K(i',j)$ and $z = K(i,j')$. There are again two symmetric cases:

(a) $z \leq y$. We have that:

$$\begin{aligned}
&\texttt{OPT}(i,j) + \texttt{OPT}(i',j') \\
&\leq \texttt{OPT}_z(i,j) + \texttt{OPT}_y(i',j') \\
&= w(i,j) + w(i',j') + \texttt{OPT}(i,z) + \texttt{OPT}_z(z+1,j) + \texttt{OPT}_y(i',y) + \texttt{OPT}_y(y+1,j') \\
&\leq w(i',j) + w(i,j') + \texttt{OPT}(i,z) + \texttt{OPT}_y(i',y) + \texttt{OPT}_z(y+1,j) + \texttt{OPT}_y(z+1,j') \\
&= \texttt{OPT}_y(i',j) + \texttt{OPT}_z(i,j') \\
&= \texttt{OPT}(i',j) + \texttt{OPT}(i,j')
\end{aligned}$$

Where the first inequality and the first equality are due to the definition of $\texttt{OPT}_k$, the second inequality is due to QI of w and the induction hypothesis at $z \leq y < j < j'$ and the last two equalities are again due to the definition of $\texttt{OPT}_k$.

(b) Also this time the other case when $k \geq j$ is symmetrical.

□

Having the above lemma we will now prove the following:

Lemma 3.1.1.3. $K(i,j-1) \leq K(i,j) \leq K(i+1,j)$

Proof. To prove the first inequality $K(i,j-1) \leq K(i,j)$ we will show that for $i \leq k \leq k' < j$ we have that: $\texttt{OPT}_{k'}(i,j-1) \leq \texttt{OPT}_k(i,j-1)$ implies that $\texttt{OPT}_{k'}(i,j) \leq \texttt{OPT}_k(i,j)$. This condition suffices as whenever $k' = K(i,j-1)$ this means that either $k' = k$ or $k \neq K(i,j)$ as choosing $v_{k'}v_{k'+1}$ as the root of the decision tree provides a solution which cannot be worse[1]. By using QI at $k \leq k' \leq j-1 < j$ we have:

$$\texttt{OPT}(k,j-1) + \texttt{OPT}(k',j) \leq \texttt{OPT}(k',j-1) + \texttt{OPT}(k,j)$$

By adding $w(i,j-1) + w(i,j) + \texttt{OPT}(i,k) + \texttt{OPT}(i,k')$ to both sides we obtain:

$$\texttt{OPT}_k(i,j-1) + \texttt{OPT}_{k'}(i,j-1) \leq \texttt{OPT}_{k'}(i,j-1) + \texttt{OPT}_k(i,j-1)$$

Which implies the claim. The second inequality $K(i,j) \leq K(i+1,j)$ follows similarly from the QI at $i < i+1 \leq k \leq k'$. □

The above lemma allows us to the amount of computation required. The idea is as follows: Before calculating $\texttt{OPT}(i,j)$ we firstly calculate the values of $\texttt{OPT}(i-1,j)$ and $\texttt{OPT}(i,j+1)$. In

[1]Note that by the definition we require $K(i,j)$ to be maximal.

doing so we also calculate the indices $K(i, j-1)$ and $K(i+1, j)$ required to narrow the space of possible choices for value of $K(i, j)$. We obtain the following recurrence relationship:

$$\texttt{OPT}_{sum}(i,j) = \min_{K(i,j-1) \leq k \leq K(i+1,j)} \{w(i,j) + \texttt{OPT}_{sum}(i,k) + \texttt{OPT}_{sum}(k+1,j)\}$$

It remains to show that the running time can be bounded by $O\big(n^2\big)$. The amount of computation steps required by the algorithm is equal to:

$$\begin{aligned}\sum_{i=1}^{n} \sum_{j=i+1}^{n} (K(i+1,j) - K(i,j-1)) &= \sum_{i=1}^{n} \sum_{j=i}^{n} (K(i+1,j+1) - K(i,j)) \\ &= \sum_{i=1}^{n} (K(i+1,n) - K(1,i)) = O\big(n^2\big)\end{aligned}$$

Where the second equality is due to the fact that all of the terms except $K(i+1, n)$ and $K(1, i)$ cancel, and the last equality is trivially due to the fact that $K(i+1, n) < n$. This proves the claim. □

3.1.2 Non-uniform costs, worst case

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

3.1.3 Non-uniform costs, Average case, uniform weights

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

3.2 Unitary costs, trees

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit

esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

3.2.1 Worst case

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

3.2.2 Average case, non-uniform weights

The problem of average case searching is also solvable in polynomial time assuming all of the weight are uniform. However the procedure is the same as for the weighted case and only the running time differ. Hence we combine these results in one section. Note that it is yet unknown whether the same holds for the weighted version of the problem and the fastest known algorithm runs in pseudopolynomial time. Using this one may also obtain a FPTAS using a standard rounding trick. Before that, however we show that a simple greedy heuristics achieves a 2-approximation for $T|V, w|\sum C_j$.

A warm up: greedy achieves 2-approximation for $T|V, w|\sum C_j$ The weight centroid is a vertex $c \in T$ such that for every $H \in T - c$ we have that $w(h) \leq \frac{w(T)}{2}$. The existence of the (unweighted) centroid has been known since 19th century [Jor69]. The proof of the existence of the weight centroid i straightforward and can be summarized as follows: pick any vertex $v \in T$ and if its not a weight centroid move to the neighbor v' of v such that the $H \in T - v$ such that $v' \in H$ has weight $w(H) > \frac{w(T)}{2}$. It is easily observable that the algorithm always succeeds and visits each vertex at most once. The greedy algorithm is as follows: pick the centroid c of T as the root of the decision tree for T and proceed recursively in $T - c$. The following analysis of greedy is due to [Ber+22].

Theorem 3.2.2.1. *Let D_c be the cost of the greedy solution. Then $\texttt{COST}_{D_c}(T) \leq 2\texttt{OPT}(T) - w(T)$.*

Proof. We start with the following lemma:

Lemma 3.2.2.2. *Let D be any decision tree for T and let c be the centroid of T. Then:*

$$\texttt{OPT}(T) \geq \frac{w(T)}{2} + \frac{w(c)}{2} + \sum_{H \in T-c} \texttt{OPT}(H)$$

Proof. Let $r = r(D)$. There are two cases:

1. $r = c$. In such case the cost of the solution is trivially lower bounded by:

$$\texttt{COST}_D(T) \geq w(T) + \sum_{H \in T-r} \texttt{OPT}(H) \geq \frac{w(T)}{2} + \frac{w(c)}{2} + \sum_{H \in T-c} \texttt{OPT}(H)$$

2. $r \neq c$. In such case denote by H_r the connected component of $T - c$ such that $r \in H_r$. We have that the contribution of each $v \in H_r$ is at least $\left|Q_{D|H_r}(v)\right|$ so the overall contribution of vertices in H_r is at least $\texttt{COST}_{D|H_r}(H_r)$. For every $H \in T - c$ such that $H \neq H_r$ and $v \in H$ we have that $\{r\} \cup Q_{D|H}(v) \subseteq Q_D(v)$ so we have that the contribution of vertices in H is at least $w(H) + \texttt{COST}_{D|H}(H)$. Additionally the contribution of c is at least $w(c)$ since query to r precedes the query to c. We have that:

$$\begin{aligned}
\texttt{COST}_D(T) &\geq 2w(c) + \texttt{COST}_{D|H_r}(H_r) + \sum_{H \in T-c, H \neq H_r} \left(w(H) + w(c) + \texttt{COST}_{D|H}(H)\right) \\
&\geq w(T) - w(H_r) + \sum_{H \in T-c} \texttt{OPT}_{D|H}(H) \\
&\geq \frac{w(T)}{2} + w(c) + \sum_{H \in T-c} \texttt{OPT}_{D|H}(H)
\end{aligned}$$

   where in the last inequality we used the fact that c is a centroid of T.

□

The proof is by induction on the size of T. When $n(T) = 1$ we have that $\texttt{COST}_{D_c}(T) = w(T) = 2\texttt{OPT}(T) - w(T)$. Assume therefore that $n(T) > 1$ and let c be the centroid of T. We have that:

$$\begin{aligned}
\texttt{COST}_{D_c}(T) &= w(T) + \sum_{H \in T-c} \texttt{COST}_{D_c|H}(H) \\
&\leq w(T) + \sum_{H \in T-c} \left(2 \cdot \texttt{OPT}(H) - w(H)\right) \\
&= w(c) + \sum_{H \in T-c} 2 \cdot \texttt{OPT}(H) \\
&\leq 2\texttt{OPT}(T) - w(T)
\end{aligned}$$

where the first inequality is by the induction hypothesis and the second inequality is by the Lemma 3.2.2.2. □

Theorem 3.2.2.3. *The greedy decision tree can be found in $O(n \log n)$ running time.*

Proof. We use the data structure called *top trees*. The top trees are used to maintain dynamic forests under insertion and deletion of edges. The following theorem is due to [Als+05]:

Theorem 3.2.2.4. *We can maintain a forest with positive vertex weights on n vertices under the following operations:*

1. *Add an edge between two given vertices u, v that are not in the same connected component.*
2. *Remove an existing edge.*
3. *Change the weight of a vertex.*
4. *Retrieve a pointer to the tree containing a given vertex.*
5. *Find the centroid of a given tree in the forest.*

Each operation requires $O(\log n)$ time. A forest without edges and with n arbitrarily weighted vertices can be initialized in $O(n)$ time.

We begin with building the top tree out of T. We begin with empty top tree and add each edge one by one. Then we find the centroid of T and remove each edge incident to it. Then we recurse on this new created tree (excluding the subtree consisting of c). Since the algorithm finds each vertex once and removes each edge once the total running time is of order $O(n \log n)$. □

PTAS for $T|V, w|\sum C_j$

FPTAS for $T|V, w|\sum C_j$ As it turns out one may employ a different dynamic programming technqieu to obtain an FPTAS for $T|V, w|\sum C_j$. To do so, we firstly design a pseudopolynomial time procedure which then combine with a standard rounding scheme. To do so, we begin with the following bound due to [Ber+22] (we managed to simplify the proof a bit):

Theorem 3.2.2.5. *Let D^* be the optimal decision tree for $T|V, w|\sum C_j$. Then we have:*

$$\texttt{COST}_{max,D^*}(T) \leq \left\lceil \log_{3/2} w(T) \right\rceil$$

Proof. For the sake of the argument we define the following operation. Let D be a decision tree for some tree T and $v \in V(T)$. We define D_v to be a decision tree such that $r(D) = v$. Additionally, for each $H \in T - v$ we hang $D_{|H}$ below v in D. This operation is called a *lifting* of a vertex. Let $x, v \in V(T)$ and $H_x \in T - v$ such that $x \in H_x$ if $x \neq v$. We have:

$$Q_{D^v}(x) = \begin{cases} \{v\} \text{ if } x = v \\ \{v\} \cap (Q_D(x) \cup V(H)) \text{ otherwise} \end{cases}$$

We will show that after each query the size of the candidate subset decreases by a factor of 2/3. To do so, assume contrary. Let D be a minimum height decision tree for which this is not the case. By doing so we can assume that $r = r(D)$ has a child c such that $w(D_y) > \frac{2w(T)}{3}$. Let H_r denote the set of vertices not in the same component of $T - r$ as c and H_c denote the set of vertices not in the same component of $T - c$ as r. We also define $H_{r,c} = V(T) - H_r - H_c$. By the assumption $w(H_c \cup H_{r,c}) > \frac{2w(T)}{3}$ and $w(H_r) < \frac{w(T)}{3}$. There are two cases:

1. $w(H_c) > \frac{w(T)}{3}$. In such case we augment D by lifting c. The query sequences of vertices in H_c decrease by one query, the query sequences of vertices in H_r increase by one and query sequences of vertices in $H_{r,c}$ remain unchanged. We have:

$$\texttt{COST}_{avg,D^v}(T) - \texttt{COST}_{avg,D}(T) = w(H_r) - w(H_c) < 0$$

   thus, a contradiction.

2. $w(H_c) \leq \frac{w(T)}{3}$. We have that $w(H_{r,c})$ and additionally $H_{r,c} \neq \emptyset$. Let $s \in P(r, c)$. In such case we augment D by lifting s. The query sequences of vertices in H_c remain unchanged, since these vertices gain t and lose r as ancestors. The query sequences of vertices in $H_{r,c}$ are decreased by at least one query, since each loses at least one ancestor from c, r. The query sequences of vertices in H_r increase by one, since each of these vertices gains t as ancestor. We have:

$$\texttt{COST}_{avg,D^v}(T) - \texttt{COST}_{avg,D}(T) = w(H_r) - w(H_{r,c}) < 0$$

   again, a contradiction.

As after each query to size of the candidate subset shrinks by the ratio of 2/3 the claim follows. □

The above bound on the cost allows us to assume that the height of the optimal decision tree is of order of $O(l \log w(T))$. We will exploit the fact to build a btoom-up dynamic programming algorithm which for every $v \in V(T)$ will enumerate all possible "shapes" of the query sequence toward v. As to place the query to v we only need an information about which spots in this query sequences are free and which are not, at most $2^{O(\log w(T))} = \text{poly}(W)$ options need to be checked.

3.3 Non-uniform costs, trees

In this section we will be only concerned with the vertex-query variant of the problem. This is due to the fact that the edge variant is easily reducible to the vertex variant of the problem and this reduction preserves the approximation ratio (note that this reduces the problem to the version in which the last query may be omitted but all of the algorithms can be easily altered to consider this assumption). This is done by subdividing each edge e with a new vertex v_e of cost $c(v_e) = c(e)$.

If we consider the average case criterion then we additionally set $w(v_e) = 0$. It is immediate that the optimal decision tree for the this vertex query model instance can be used to obtain a decision tree for the original instance. To do so, simply replace each query to vertex v_e with the query corresponding to e.

3.3.1 Worst case

The problem for non-uniform is NP-hard even when restricted to spiders of diameter 6 and binary trees. A simple greedy heuristics which always queries the middle vertex of the graph achieves a $O(\log n)$-approximation [Der06]. However one can obtain better results. We begin with the following simple lemma which will become useful in few arguments:

Lemma 3.3.1.1. *Let T' be a connected subtree of T. Then, $\texttt{OPT}(T') \leq \texttt{OPT}(T)$.*

The proof of this fact is trivial and will be left as an exercise for the reader.

A warm up: $O(\log n/\log\log n)$-approximation algorithm for $T||V,c||C_{max}$ This first algorithm is an adapted and simplified version of the algorithm due to [Cic+16] for the edge query model.

Theorem 3.3.1.2. *There exists a polynomial time, $O(\log n/\log\log n)$-approximation algorithm for the $T||V,c||C_{max}$ problem .*

Proof. To construct a decision tree we will use the following exact procedure:

Lemma 3.3.1.3. *There exists a $O(2^n n)$ algorithm for $T||V,c||C_{max}$*

Proof. The algorithm is a general version of the dynamic programming procedure for paths. We have that:

$$\texttt{OPT}_{max}(T) = \min_{v\in V(T)}\left\{c(v) + \max_{H\in T-v}\{\texttt{OPT}_{max}(H)\}\right\}$$

There are there are at most $O(2^n)$ different subtrees of T to be checked. Additionally, and for each $v \in V(T)$ there are at most $\deg_T(v)$ possible responses to check in the inner max function. Therefore for each subproblem there are at most $\sum_{v\in V(T)} \deg_T(v) = 2m = 2n-2$ comparison operations to be performed. As at each level of the recursion the algorithm considers all possible choices of the next queried vertex v it returns the optimal decision tree for T and the claim follows. □

Observation 3.3.1.4. *Let D be a partial decision tree for tree T. Let T' be a subtree of T. Let Q be a set of all queries to vertices from $N_T(V(T'))$ in D such that every for every $q \in Q$: q is queried before every vertex in T'. Then $D\langle Q\rangle$ is a path.*

Proof. Let $x \in N_T(V(T'))$. In such case, for every vertex $v \in V(T - V(T') - N_T(V(T')))$ the answer to a query to v is always towards the same $u \in N_T(v)$, so until a vertex from T' is queried, no query q can partition vertices from $N_T(V(T'))$ into disjoint subtrees of candidate vertices except when $q \in N_T(V(T'))$. After a query to q, the only different response is when $x = q$, in which case no further queries are needed, so queries in Q must belong to a path in D. □

Let $k = 2^{\lfloor \log \log n \rfloor + 2}$. The basic idea is as follows. The algorithm is recursive. Let $\mathcal{T}$ be the tree currently processed by the algorithm. If $n(T) \leq k$ then we use the exponential time algorithm to find the optimal solution in time $2^k k = \text{poly}(n)$.

If otherwise, to build a solution (see Algorithm **??**) we will firstly define a set $\mathcal{X} \subseteq V(\mathcal{T})$ which will be of size at most k. We build $\mathcal{X}$ iteratively. Starting with an empty set we pick the centroid x_1 of T which we add to $\mathcal{X}$. Then we take the forest $F = T - x$, find the largest $H \in F$, pick its centroid x_2 and append it to $\mathcal{X}$. We continue this in $F - H + (H - x_2)$ until $|\mathcal{X}| = k$.

Lemma 3.3.1.5. *For every $H \in \mathcal{T} - \mathcal{X}$ we have that $n(H) \leq n(\mathcal{T}) / \log(n)$.*

Proof. We prove by induction on t that deleting first 2^t centroids from T each connected components H_t has size at most $n(H_t) \leq n(\mathcal{T}) / 2^{t-1}$. For the case when $t = 0$ we have that after 1 iteration every H_1 has size at most $n(T) / 2 \leq 2(n)$ so the base of induction is complete.

Fix $t > 0$ and by assume by the induction hypothesis that after 2^{t-1} iterations all □

We also define set $\mathcal{Y} \subseteq V(\mathcal{T})$ which consists of vertices in $\mathcal{X}$ and all vertices in $v \in \mathcal{T}\langle X \rangle$ such that $\deg_{T\langle \mathcal{X} \rangle}(v) \geq 3$. Furthermore, we define set $\mathcal{Z} \subseteq V(\mathcal{T})$ as a set consisting of vertices in $\mathcal{Y}$ and for every $u, v \in \mathcal{Y}$ such that $\mathcal{P}_{\mathcal{T}}(u, v) \neq \emptyset$ and $\mathcal{P}_{\mathcal{T}}(u, v) \cap \mathcal{Y} = \emptyset$ we add to $\mathcal{Z}$ the vertex $\arg\min_{z \in \mathcal{P}_{\mathcal{T}}(u,v)} \{c(z)\}$ (for example see Figure 3.2). We then create an auxiliary tree $\mathcal{T}_{\mathcal{Z}} = (\mathcal{Z}, \{uv | \mathcal{P}_{\mathcal{T}}(u, v) \cap \mathcal{Z} = \emptyset\})$ (for example see Figure 3.3). The algorithm builds an optimal decision tree $D_{\mathcal{Z}}$ for $\mathcal{T}_{\mathcal{Z}}$ by applying the exponential time algorithm. Observe, that $D_{\mathcal{Z}}$ is a partial decision tree for $\mathcal{T}$, so we get that:

Observation 3.3.1.6. $\texttt{COST}_{D_{\mathcal{Z}}}(\mathcal{T}_{\mathcal{Z}}) = \texttt{COST}_{D_{\mathcal{Z}}}(\mathcal{T})$.

Then for each $H \in \mathcal{T} - \mathcal{Z}$ we recursively apply the same algorithm to obtain the decision tree D_H and we hang it in $D_{\mathcal{Z}}$ below the unique last query to vertex in $N_{\mathcal{T}'}(H)$ (By Observation 3.3.1.4).

Algorithm 1: Main recursive procedure (k is a global parameter)

Procedure `DecisionTree(`$\mathcal{T}$`):`

- **if** $n(\mathcal{T}) \leq k$ **then**
  - $D \leftarrow$`Exact(`$\mathcal{T}$`)`;
  - **return** D
- $\mathcal{X} \leftarrow \emptyset$;
- $F \leftarrow \{\mathcal{T}\}$;
- **for** $i \in 1, ..., k$ **do**
  - $H \leftarrow \arg\max_{H \in F}\{n(H)\}$;
  - Let x be the centroid of H.;
  - Add x to X.;
  - Add $H - x$ to F;
- $\mathcal{Z} \leftarrow \mathcal{Y} \leftarrow \mathcal{X} \cup \left\{v \in \mathcal{T}\langle\mathcal{X}\rangle | \deg_{\mathcal{T}\langle\mathcal{X}\rangle}(v) \geq 3\right\}$;
- `// Branching vertices in` $\mathcal{T}\langle X\rangle$.
- **foreach** $u, v \in \mathcal{Y}, \mathcal{P}_{\mathcal{T}}(u, v) \neq \emptyset, \mathcal{P}_{\mathcal{T}}(u, v) \cap \mathcal{Y} = \emptyset$ **do**
  - Add $\arg\min_{z \in \mathcal{P}_{\mathcal{T}}(u,v)}\{c(z)\}$ to $\mathcal{Z}$.;
  - `// Lightest vertex on path` $P_{\mathcal{T}}(u, v)$.
- $\mathcal{T}_{\mathcal{Z}} = (\mathcal{Z}, \{uv | \mathcal{P}_{\mathcal{T}}(u, v) \cap \mathcal{Z} = \emptyset\})$;
- $D \leftarrow D_{\mathcal{Z}} \leftarrow$`Exact(`$\mathcal{T}_{\mathcal{Z}}$`)`;
- **foreach** $H \in \mathcal{T} - \mathcal{Z}$ **do**
  - $D_H \leftarrow$`DecisionTree(`H`)`;
  - Hang D_H in D below the last query to a vertex $v \in N_{\mathcal{T}}(H)$;
- **return** D

Lemma 3.3.1.7. *Let $\mathcal{T}_{\mathcal{Z}}$ be the auxiliary tree. Then, $|V(\mathcal{T}_{\mathcal{Z}})| \leq 4k - 3$.*

Proof. We firstly show that $|\mathcal{Y}| \leq 2k - 1$. We use induction of the centroids in $\mathcal{X}$. For $1 \leq i \leq k$ let x_i denote the i-th centroid added to $\mathcal{X}$. We will construct a family of sets $\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_{|\mathcal{H}|}$ such that for any $1 \leq t \leq |\mathcal{X}|$: $|\mathcal{X}_t| = t$ and $\mathcal{X}_{|\mathcal{X}|} = \mathcal{X}$. For each $\mathcal{X}_t$ we will also construct a corresponding set $\mathcal{Y}_t$, ensuring $\mathcal{Y}_{|\mathcal{X}|} = \mathcal{Y}$. We will build the sets $\mathcal{Y}_t$ to ensure that $|\mathcal{Y}_t| \leq 2t - 1$.

Let $\mathcal{X}_1 = \{x_1\}$, $\mathcal{Y}_1 = \{x_1\}$. This establishes the base case. Assume by induction on $t \geq 1$ that $|\mathcal{Y}_t| \leq 2t - 1$ for some $t > 1$. Let $\mathcal{X}_{t+1} = \mathcal{X}_t \cup \{x_{t+1}\}$ and let $\mathcal{T}_t = \mathcal{T}\langle\mathcal{X}_t\rangle$ If $x_t \in V(\mathcal{T}_t)$ then $\mathcal{Y}_{t+1} = \mathcal{Y}_t \cup \{x_t\}$. If otherwise let $y_t \in V(\mathcal{T}_t)$ be the unique vertex such that $P(x_t, y_t) \cap V(\mathcal{T}_t) = \emptyset$. Then $\mathcal{Y}_{t+1} = \mathcal{Y}_t \cup \{x_t, y_t\}$. As by induction $|\mathcal{Y}_t| \leq 2t - 1$ and we add at most two vertices to it to obtain $\mathcal{Y}_{t+1}$ the induction step is complete.

As paths between vertices in $\mathcal{Y}$ form a tree, at most $2k - 2$ additional vertices are added to $\mathcal{Y}$ while constructing $\mathcal{Z}$ (at most one for each path) and the lemma follows. □

Lemma 3.3.1.8. *Let $\mathcal{T}_{\mathcal{Z}}$ be the auxiliary tree. Then, $\texttt{OPT}(\mathcal{T}_{\mathcal{Z}}) \leq \texttt{OPT}(\mathcal{T})$.*

Proof. Let D^* be the optimal strategy for $\mathcal{T}\langle\mathcal{Z}\rangle$. We build a new decision tree $D'_{\mathcal{Z}}$ for $\mathcal{T}_{\mathcal{Z}}$ by transforming D^*: Let $u, v \in \mathcal{Y}$ such that $\mathcal{P}_{\mathcal{T}}(u,v) \neq \emptyset$ and $\mathcal{P}_{\mathcal{T}}(u,v) \cap \mathcal{Y} = \emptyset$. Let $q \in V(D^*)$ such that $q \in \mathcal{P}_{\mathcal{T}}(u,v)$ is the first query among vertices of $\mathcal{P}_{\mathcal{T}}(u,v)$. We replace q in D^* by the query to the distinct vertex $v_{u,v} \in \mathcal{P}_{\mathcal{T}}(u,v) \cap \mathcal{Z}$ and delete all queries to vertices $\mathcal{P}_{\mathcal{T}}(u,v) - v_{u,v}$ from D^*. By construction, $D'_{\mathcal{Z}}$ is a valid decision tree for $\mathcal{T}_{\mathcal{Z}}$ and as for every $z \in \mathcal{P}_{\mathcal{T}}(u,v)$: $c(v_{u,v}) \leq c(z)$ such strategy has cost at most $\texttt{COST}_{D'_{\mathcal{Z}}}(\mathcal{T}_{\mathcal{Z}}) \leq \texttt{OPT}(\mathcal{T}\langle\mathcal{Z}\rangle)$. We get:

$$\texttt{OPT}(\mathcal{T}_{\mathcal{Z}}) \leq \texttt{COST}_{D'_{\mathcal{Z}}}(\mathcal{T}_{\mathcal{Z}}) \leq \texttt{OPT}(\mathcal{T}\langle\mathcal{Z}\rangle) \leq \texttt{OPT}(\mathcal{T})$$

where the first inequality is due to the optimality and the last inequality is due to the fact that $\mathcal{T}\langle\mathcal{Z}\rangle$ is a subtree of $\mathcal{T}$ (by Lemma 3.3.1.1). The lemma follows. □

Lemma 3.3.1.9. *Let D_T be the solution returned by the algorithm. Then the approximation factor of such solution is bounded by $\texttt{APP}_T(D_T) \leq \log n / \log\log n$.*

Proof. Let $\mathcal{T}$ be the tree processed at some level of the recursion and let $D_{\mathcal{T}}$ be the decision tree returned by the algorithm. The proof is by induction on the size of $\mathcal{T}$. We claim that $\texttt{APP}_{\mathcal{T}}(D_{\mathcal{T}}) \leq \max\{1, \log n(\mathcal{T}) / \log\log n\}$. If $n(\mathcal{T}) \leq k$ then $D_{\mathcal{T}}$ is the optimal decision tree for $\mathcal{T}$ which establishes the base case. Let $n(\mathcal{T}) > k$ and assume that claim holds for every $t < n(\mathcal{T})$. By construction, we have that:

$$\begin{aligned}
\texttt{APP}_{D_{\mathcal{T}}}(\mathcal{T}) &= \frac{\texttt{COST}_{D_{\mathcal{T}}}(\mathcal{T})}{\texttt{OPT}(\mathcal{T})} \\
&\leq \frac{\texttt{COST}_{D_{\mathcal{Z}}}(\mathcal{T}) + \max_{H \in \mathcal{T}-\mathcal{Z}}\{C_{D_H}(H)\}}{\texttt{OPT}(\mathcal{T})} \\
&\leq \frac{\texttt{COST}_{D_{\mathcal{Z}}}(\mathcal{T}_{\mathcal{Z}})}{\texttt{OPT}(\mathcal{T}_{\mathcal{Z}})} + \max_{H \in \mathcal{T}-\mathcal{Z}}\left\{\frac{C_{D_H}(H)}{\texttt{OPT}(H)}\right\} \\
&\leq 1 + \frac{\log \frac{n(\mathcal{T})}{\log n(\mathcal{T})}}{\log\log n} = \frac{\log n(\mathcal{T})}{\log\log n}
\end{aligned}$$

where the first inequality is by construction, the second is by usage of Observation 3.3.1.6, Lemma 3.3.1.8 and Lemma 3.3.1.1 and the last inequality is due to the Lemma 3.3.1.5 and the induction hypothesis. □

Using the fact that the call to the exponential time procedure requires $O\big(2^{4k-3}(4k-3)\big) = \text{poly}(n)$ time (Due to Lemma 3.3.1.7), all other computations require polynomial time, and each $v \in V(T)$ belongs to $\mathcal{Z}$ at most once during the execution we get that the overall running time is polynomial in n. □

In the above analysis we lose one factor of $\texttt{OPT}$ per each level of recursion of which there are at most $O(\log n/\log\log n)$. Notice however, that we can allow some more loss (i. e. $c\cdot\texttt{OPT}$) without affecting the asymptotical approximation factor. As it turns out it is possible to obtain a constant factor approximation for this problem in quasipolynomial time. This is the main idea behind the improvement of the approximation factor for this problem as in such case the size of the set $\mathcal{Z}$ may be greater and less recursion levels are needed which directly improves the approximation.

An $O\big(\sqrt{\log n}\big)$-approximation algorithm for $T||V,c||C_{max}$ We begin with the following proposition [Der+17] about the existence of QPTAS for $T||V,c||C_{max}$:

Proposition 3.3.1.10. *For any $0<\epsilon\leq 1$ there exists a $(1+\epsilon)$-approximation algorithm for the Tree Search Problem running in $2^{O\left(\frac{\log^2 n}{\epsilon^2}\right)}$ time.*

The algorithm and the proof of its correctness are very intricate and requires usage of an alternative notion of strategy. However, we rewrite it to use the language of the decision trees. Since the proof is involved for now we will use it as a black-box. The proof will be differed to a separate paragraph after the analysis below.

Theorem 3.3.1.11. *There exists a polynomial time, $O(\log n/\log\log n)$-approximation algorithm for the $T||V,c||C_{max}$ problem .*

Proof. We use the same procedure as in the $O(\log n/\log\log n)$-approximation algorithm, however we set $k=2^{\lfloor\sqrt{\log n}\rfloor+2}$ and we swap the exact procedure to the QPTAS with $\epsilon=1$. The analysis of the algorithm is largely the same except the evaluation of the cost of this solution.

Lemma 3.3.1.12. *Let D_T be the solution returned by the algorithm. Then the approximation factor of such solution is bounded by $\texttt{APP}_T(D_T)\leq 2\sqrt{\log n}$.*

Proof. Let $\mathcal{T}$ be the tree processed at some level of the recursion and let $D_{\mathcal{T}}$ be the decision tree returned by the algorithm. The proof is by induction on the size of $\mathcal{T}$. We claim that $\texttt{APP}_{\mathcal{T}}(D_{\mathcal{T}})\leq\max\big\{1,2\log n(\mathcal{T})/\sqrt{\log n}\big\}$. If $n(\mathcal{T})\leq k$ then $D_{\mathcal{T}}$ is the optimal decision tree for $\mathcal{T}$ which establishes the base case. Let $n(\mathcal{T})>k$ and assume that claim holds for every $t<n(\mathcal{T})$. By

construction, we have that:

$$\begin{aligned}
\texttt{APP}_{D_{\mathcal{T}}}(\mathcal{T}) &= \frac{\texttt{COST}_{D_{\mathcal{T}}}(\mathcal{T})}{\texttt{OPT}(\mathcal{T})} \\
&\leq \frac{\texttt{COST}_{D_{\mathcal{Z}}}(\mathcal{T}) + \max_{H \in \mathcal{T}-\mathcal{Z}}\{C_{D_H}(H)\}}{\texttt{OPT}(\mathcal{T})} \\
&\leq \frac{\texttt{COST}_{D_{\mathcal{Z}}}(\mathcal{T}_{\mathcal{Z}})}{\texttt{OPT}(\mathcal{T}_{\mathcal{Z}})} + \max_{H \in \mathcal{T}-\mathcal{Z}}\left\{\frac{C_{D_H}(H)}{\texttt{OPT}(H)}\right\} \\
&\leq 2 + \frac{2\log\frac{n(\mathcal{T})}{\sqrt{\log n}}}{\sqrt{\log n}} = \frac{2\log n(\mathcal{T})}{\sqrt{\log n}}
\end{aligned}$$

where the first inequality is by construction, the second is by usage of Observation 3.3.1.6, Lemma 3.3.1.8 and Lemma 3.3.1.1 and the last inequality is due to the Lemma 3.3.1.5 and the induction hypothesis. □

□

QPTAS for the $T||V,c||C_{max}$ problem

Observation 3.3.1.13. *Let T be a tree such that $|V(T)| > 1$ and $c : V \to \mathbb{R}^+$ be a normalized weight function. Then, $1 \leq \texttt{OPT}(T) \leq \lfloor \log n \rfloor + 1$.*

Constant factor approximation for monotonic query costs

$O(\log\log n)$-approximation algorithm parametrized by the k-up-modularity of the cost function Let $s \in \mathbb{R}^+$. We define a *heavy module* with respect to s as $H \subseteq V(T)$ such that: $T[H]$ is connected, for every $v \in H$: $c(v) \geq s$ and H is maximal - no vertex can be added to it without violating one of its properties. We then define the *heavy module set* of c in T as: $HM_{T,c}(s) = \{H \subseteq V(T) \,|H \text{ is a heavy module w.r.t. } s\}$ (abbreviated to $HM(s)$). Let $k(T,c) = \max_{s\in\mathbb{R}^+}\{|HM_{T,c}(s)|\}$. We say that a function c is *k-up-modular* in T, when $k \geq k(T,c)$. Whenever clear from the context we will use $k(T,c)$, $k(T)$, or k to denote the lowest value such that c is k-up modular in T.

The notion of k-up-modularity is a direct generalization of the concept of up-monotonicity of the cost function. It is easy to see that in fact 1-up-modularity is equivalent to up-monotonicity. Observe that if c is up-monotonic in T then for every $s \in \mathbb{R}^+$: $T[V(T) - \{v \in V(T) : c(v) < s\}]$ is connected and forms a single heavy module. Conversely, let $r = \arg\max_{v\in V(T)}\{c(v)\}$ and u be any other vertex. If c is 1-up-modular in T then there is no vertex v on path between r and u such that $c(v) < c(u)$. If otherwise then for any $s \in (v,u)$: v does not belong to any heavy module but u and r do. As v lays between them $|HM(s)| > 1$, a contradiction.

From now on we will again assume that all weights are normalized, so that the weight of the heaviest vertex is exactly 1. If not, the weights are scaled by dividing them by $\max_{v \in V(T)}\{w(v)\}$.

The main idea of the algorithm is to divide vertices into *cost levels* and process them in a recursive manner. At each level of the recursion the algorithm broadens the subset of vertices of T having queries assigned to them. This continues until every $v \in T$ has a query to it scheduled. We consider the following intervals which we call *cost levels*:

1. Firstly, an interval $\left(0, \frac{1}{\log n}\right]$.
2. Then, each next interval $\mathcal{I}' = (a', b']$ starts at the left endpoint of the previous interval $\mathcal{I} = (a, b]$, that is, $a' = b$ and ends with $b' = \min\{2b, 1\}$. This results in the following sequence of intervals:

$$\left(\frac{1}{\log n}, \frac{2}{\log n}\right], \left(\frac{2}{\log n}, \frac{4}{\log n}\right], \ldots, \left(\frac{2^{\lceil \log \log n \rceil - 1}}{\log n}, 1\right]$$

After obtaining a solution for previous interval recursively, the next consecutive weight level $\mathcal{I}$ is processed by extending the strategy to contain all queries to vertices v such that $c(v) \in \mathcal{I}$. Once the last interval $\left(\frac{2^{\lceil \log \log n \rceil - 1}}{\log n}, 1\right]$ is processed the algorithm returns a valid decision tree for T.

We are now ready to introduce the notions of heavy and light vertices (and queries to them). We say that a vertex v (or the query to it) is *heavy* with respect to the interval $\mathcal{I} = (a, b]$, when $c(v) > a$. If otherwise, i.e. $c(v) \leq a$ then, the vertex (and the query to it) is *light* with respect to $\mathcal{I}$. Whenever clear from the context, we will omit the term "with respect to" and just call the vertices and queries heavy and light.

At last, we introduce the contraction operation. Let $s > c(r(T))$ and let H be a heavy module with respect to s. Let v be a parent of $r(T[H])$ (observe that such a parent always exists as the tree is rooted at the lightest vertex and $s > c(r(T))$). A *contraction* of a heavy module H is an operation which consists of deleting all of the vertices in H from T and connecting every vertex u which was a child of some vertex in H to v (note, that this definition slightly differs from the standard contraction, for example see Figure 3.1). This leads to the following simple observations:

Observation 3.3.1.14. *Let T' be a tree created by contraction of a heavy module H in T. Then, $\texttt{OPT}(T') \leq \texttt{OPT}(T)$.*

Observation 3.3.1.15. *Let T' be a tree created by contraction of a heavy module H in T. Then, $k(T') \leq k(T)$.*

We are ready to present the main recursive procedure. To avoid ambiguity, let $\mathcal{T}$ be the tree processed at some level of the recursion. Alongside $\mathcal{T}$, the algorithm takes as an input the interval $(a, b]$ such that for every $v \in \mathcal{T}$: $c(v) \leq b$. The algorithm (see Algorithm 5) works in the following manner:

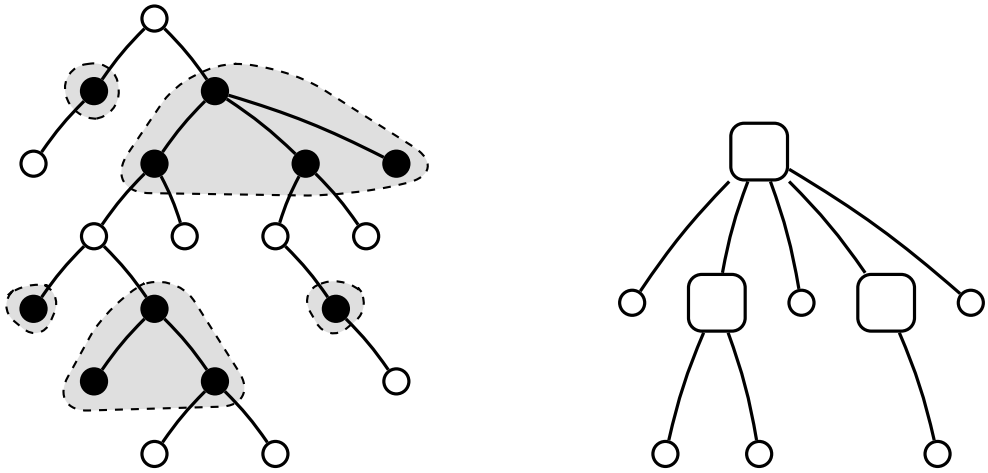

Figure 3.1: Example of contracting 5 heavy modules. Black vertices represent heavy vertices, white vertices represent light vertices and square vertices represent vertices which were a parent of at least one heavy module before contraction.

Algorithm 2: Main recursive procedure (n is a global parameter)

Procedure `CreateDecisionTree(`$\mathcal{T}$, $(a, b]$`):`

- **if** $b \leq \frac{1}{\log n}$ ***or*** *for every* $v \in \mathcal{T}$: $c(v) > a$ **then**
  - `// Every vertex in` $\mathcal{T}$ `is heavy`
  - **return** a decision tree D built by using the vertex ranking of $\mathcal{T}$;
- **else**
  - `// There are light vertices in` $\mathcal{T}$
  - Create $\mathcal{T}_C$ by contracting all heavy modules in $\mathcal{T}$;
  - $D_C \leftarrow$`CreateDecisionTree(`$\mathcal{T}_C$, $\left(\frac{a}{2}, a\right]$`);`
  - $D \leftarrow$`ExtendDecTree(`$\mathcal{T}$, D_C, $(a, b]$`)// Apply Proposition 3.3.1.16`
  - **return** D;

- The base case happens whenever for every $v \in \mathcal{T}$: $c(v) \leq \frac{1}{\log n}$ or for every $v \in \mathcal{T}$: $c(v) > a$, i.e. every vertex is heavy. In such situation a solution is build by disregarding the costs of vertices and building a decision tree by using the vertex ranking of $\mathcal{T}$.
- If otherwise, all heavy modules in $\mathcal{T}$ are contracted, forming a new tree $\mathcal{T}_C$ and a decision tree D_C for $\mathcal{T}_C$ is built in a recursive manner. By applying the following proposition, the decision tree D for $\mathcal{T}$ is build by extending D_C to contain queries to all vertices in $\mathcal{T}$:

Proposition 3.3.1.16. *There exists the algorithm* `ExtendDecTree` *which given a tree* $\mathcal{T}$*, a decision*

tree D_C for $\mathcal{T}_C$ and an interval $(a, b]$ returns decision tree D for $\mathcal{T}$ of cost at most: $\texttt{COST}_D(\mathcal{T}) \leq \texttt{COST}_{D_C}(\mathcal{T}_C) + \left(2 + \frac{b}{a}\right) \cdot \texttt{OPT}(\mathcal{T})$. The algorithm runs in time $2^{O(\log^2 k)} \cdot poly(n)$.

The algorithm and the proof of the Proposition 3.3.1.16 are differed to further paragraph.

Lemma 3.3.1.17. *Let D be a decision tree build for $\mathcal{T}$ in the base of the recursion in* CreateDecisionTree *by using the vertex ranking of $\mathcal{T}$. Then: $\texttt{COST}_D(\mathcal{T}, c) \leq 2\texttt{OPT}(T)$.*

Proof. There are two cases:

1. If $b \leq \frac{1}{\log n}$ we get that:

$$\texttt{COST}_D(\mathcal{T}, c) \leq \frac{\lfloor \log n \rfloor + 1}{\log n} \leq 2 \leq 2\texttt{OPT}(\mathcal{T}, c) \leq 2\texttt{OPT}(T, c)$$

   where the first inequality is due to the definition of the vertex ranking, the third inequality is due to Observation 3.3.1.13 and the last inequality is due to Observation 3.3.1.14.

2. If for every $v \in \mathcal{T}$: $c(v) > a$ then: For every $v \in \mathcal{T}$ let $c'(v) = a$ (note, that we can choose any cost here since we treat each query as unitary). As $2c'(v) \geq 2a = b \geq c(v)$ we get that $2\texttt{COST}_D(\mathcal{T}, c') \geq \texttt{COST}_D(\mathcal{T}, c)$. Additionally, using the fact that $c'(v) \leq c(v)$ we get that $\texttt{OPT}(\mathcal{T}, c') \leq \texttt{OPT}(\mathcal{T}, c)$. Therefore:

$$\texttt{COST}_D(\mathcal{T}, c) \leq 2\texttt{COST}_D(\mathcal{T}, c') = 2\texttt{OPT}(\mathcal{T}, c') \leq 2\texttt{OPT}(\mathcal{T}, c) \leq 2\texttt{OPT}(T, c)$$

   where the equality is due to the optimality of decision tree built using the vertex ranking and the last inequality is due to Observation 3.3.1.15. The lemma follows.

□

We are now ready to prove the main theorem:

Theorem 3.3.1.18. *There exists an $O(\log \log n)$-approximation algorithm for the Tree Search Problem running in $2^{O(\log^2 k(T))} \cdot poly(n)$ time.*

Proof. Let d be the depth of recursion call performed in the algorithm. We prove by induction that $\texttt{COST}_D(\mathcal{T}) \leq (4d + 2)\,\texttt{OPT}(T)$. When $d = 0$ (the base case) the induction hypothesis is true due to the Lemma 3.3.1.17. For $d > 0$ assume by induction that the cost of the decision tree build for D_C is at most $\texttt{COST}_{D_C}(\mathcal{T}_C) \leq (4(d-1) + 2)\,\texttt{OPT}(T)$. By using induction hypothesis and the fact that whenever `ExtendDecTree` is called $\frac{b}{a} \leq 2$ we get that:

$$\texttt{COST}_D(\mathcal{T}) \leq (4 \cdot (d-1) + 2)\,\texttt{OPT}(\mathcal{T}) + 4\texttt{OPT}(\mathcal{T}) \leq (4d + 2)\,\texttt{OPT}(T)$$

where the first inequality is due to the induction hypothesis and Proposition 3.3.1.16 and the second inequality is due to Observation 3.3.1.14. This proves the induction step.

Let D_T be a decision tree obtained by calling $\texttt{CreateDecisionTree}\left(T, \left(\frac{2^{\lceil \log\log n \rceil - 1}}{\log n}, 1\right]\right)$. As there are $\lceil \log\log(n) \rceil + 1$ intervals considered, the depth of recursion is bounded by $d \leq \lceil \log\log(n) \rceil \leq \log\log(n) + 1$. We obtain:

$$\texttt{COST}_{D_T}(T) \leq (4 \log\log n + 6)\, \texttt{OPT}(T) = O(\log\log n \cdot \texttt{OPT}(T))$$

As $d = \text{poly}(n)$ and due to Observation 3.3.1.15 for every $\mathcal{T}$ processed at some level of the recursion $k(\mathcal{T}) \leq k(T)$. Using the Proposition 3.3.1.16, we get that for every d the algorithm runs in $2^{O(\log^2 k(\mathcal{T}))} \cdot \text{poly}(n)$, so the overall running time is bounded by $d \cdot 2^{O(\log^2 k(T))} \cdot \text{poly}(n) = 2^{O(\log^2 k(T))} \cdot \text{poly}(n)$ as required.

□

Proof of Proposition 3.3.1.16: Extending the decision tree We prove Proposition 3.3.1.16 by showing the procedure `ExtendDecTree` which takes as an input: the tree $\mathcal{T}$, the partial decision tree D_C for the contracted tree $\mathcal{T}_C$ and the interval $(a, b]$ and extends D_C to contain all queries needed to find any target $x \in \mathcal{T}$. The basic idea of the algorithm is as follows:

1. Create an auxiliary tree $T_{\mathcal{Z}}$ such that each subtree $\mathcal{T}' \in \mathcal{T} - \mathcal{Z}$ contains at most one heavy module and create a new decision tree $D_{\mathcal{Z}}$ for $\mathcal{T}_{\mathcal{Z}}$.

2. For each $\mathcal{T}'$ build a decision tree D_H for the subtree $T\langle H \rangle$ induced by its heavy module H by using the vertex ranking of H. Then hang D_H below the last query to vertex from $N_{\mathcal{T}}(\mathcal{T}')$ in $D_{\mathcal{Z}}$.

3. For each $L \in \mathcal{T}' - H$ build a decision tree D_L by truncating queries to vertices outside of L from D_C. Then hang D_L below the last query to vertex from $N_{\mathcal{T}'}(L)$ in $D_{\mathcal{Z}}$.

We begin with the following observations:

Observation 3.3.1.19. *Let $\mathcal{H}$ be the set of heavy modules in $\mathcal{T}$. Then, $|\mathcal{H}| \leq k(\mathcal{T})$.*

To build a solution (see Algorithm 3) we will firstly define a set $\mathcal{X} \subseteq V(\mathcal{T})$. For every $H \in \mathcal{H}$ pick arbitrary $v \in H$ and add it to $\mathcal{X}$. We also define set $\mathcal{Y} \subseteq V(\mathcal{T})$ which consists of vertices in $\mathcal{X}$ and all vertices in $v \in \mathcal{T}\langle X \rangle$ such that $\deg_{T\langle \mathcal{X} \rangle}(v) \geq 3$.

Furthermore, we define set $\mathcal{Z} \subseteq V(\mathcal{T})$ as a set consisting of vertices in $\mathcal{Y}$ and for every $u, v \in \mathcal{Y}$ such that $\mathcal{P}_{\mathcal{T}}(u, v) \neq \emptyset$ and $\mathcal{P}_{\mathcal{T}}(u, v) \cap \mathcal{Y} = \emptyset$ we add to $\mathcal{Z}$ the vertex $\arg\min_{z \in \mathcal{P}_{\mathcal{T}}(u,v)}\{c(z)\}$ (for example see Figure 3.2). We then create an auxiliary tree $\mathcal{T}_{\mathcal{Z}} = (\mathcal{Z}, \{uv | \mathcal{P}_{\mathcal{T}}(u, v) \cap \mathcal{Z} = \emptyset\})$ (for example see Figure 3.3). The algorithm builds a decision tree $D_{\mathcal{Z}}$ for $\mathcal{T}_{\mathcal{Z}}$ by taking $\epsilon = 1$ and

applying the QPTAS from Theorem 3.3.1.10. Observe, that $D_{\mathcal{Z}}$ is a partial decision tree for $\mathcal{T}$, so we get that:

Observation 3.3.1.20. $\texttt{COST}_{D_{\mathcal{Z}}}(\mathcal{T}_{\mathcal{Z}}) = \texttt{COST}_{D_{\mathcal{Z}}}(\mathcal{T})$.

Let $D = D_{\mathcal{Z}}$. For each connected component $\mathcal{T}' \in \mathcal{T} - \mathcal{Z}$ we build a new decision tree in the following way: By construction of $\mathcal{Z}$, heavy vertices in $\mathcal{T}'$ form a singular heavy module $H \subseteq V(\mathcal{T}')$. We create a new decision tree D_H for $\langle H \rangle$ by using the vertex ranking of $\mathcal{T}\langle H \rangle$ and we hang D_H in D below the unique last query to a vertex in $N_{\mathcal{T}}(\mathcal{T}')$ (By Observation 3.3.1.4). As D_H is a partial decision tree for $\mathcal{T}'$, it follows that D is also a partial decision tree for $\mathcal{T}$. Then for each $L \in \mathcal{T}' - H$ we create a decision tree D_L by deleting all queries in D_C to vertices outside of $V(L)$ and hang D_L in D below the unique last query to vertex in $N_{\mathcal{T}'}(L)$ (By Observation 3.3.1.4). As D_L is a decision tree for L, we obtain a valid decision tree D for $\mathcal{T}$.

Lemma 3.3.1.21. *Let $\mathcal{T}_{\mathcal{Z}}$ be the auxiliary tree. Then, $|V(\mathcal{T}_{\mathcal{Z}})| \leq 4k - 3$.*

Proof. We firstly show that $|\mathcal{Y}| \leq 2k-1$. We use induction on elements of set $\mathcal{H}$. We will construct a family of sets $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_{|\mathcal{H}|}$ such that for any $1 \leq h \leq |\mathcal{H}|$: $|\mathcal{H}_h| = h$ and $\mathcal{H}_{|\mathcal{H}|} = \mathcal{H}$. For each $\mathcal{H}_h$ we will also construct a corresponding set $\mathcal{Y}_h$, ensuring $\mathcal{Y}_{|\mathcal{H}|} = \mathcal{Y}$.

Let $\mathcal{H}_1 = \emptyset$, $\mathcal{Y}_1 = \emptyset$. Pick any heavy module $H \subseteq V(\mathcal{T})$ and add it to $\mathcal{H}_1$. Additionally, add the unique vertex in $H \cap \mathcal{X}$ to $\mathcal{Y}$, so that $|\mathcal{Y}_1| = 1$. Assume by induction on $h \geq 1$ that $|\mathcal{Y}_h| \leq 2h - 1$. We say that heavy modules $H_1, H_2 \subseteq V(\mathcal{T})$ are *neighbors* when for every heavy module $H_3 \subseteq V(\mathcal{T})$ such that $H_3 \neq H_1, H_2$: $P_{\mathcal{T}}(H_1, H_2) \cap H_3 = \emptyset$. Let $H \subseteq V(\mathcal{T})$ such that $H \notin \mathcal{H}_h$ be a heavy module which is a neighbor of some heavy module in $\mathcal{H}_h$. Let $\mathcal{H}_{h+1} = \mathcal{H}_h \cup \{H\}$. Let z be the unique vertex in $H \cap \mathcal{X}$ and $\mathcal{Y}_{h+1} = \mathcal{Y}_h \cup \{z\}$. Let $\mathcal{T}_{h+1} = \mathcal{T}\langle\{v \in \mathcal{Y}_{h+1} | \mathcal{P}_{\mathcal{T}}(v, z) \cap \mathcal{Y}_{h+1} = \emptyset\}\rangle$. Notice, that $\mathcal{T}_{h+1}$ is a spider (tree with at most one vertex with degree above 3). Add to $\mathcal{Y}_{h+1}$ the unique vertex $v \in \mathcal{T}_{h+1}$ such that $\deg_{\mathcal{T}_{h+1}}(v) \geq 3$ if it exists. Clearly, $|\mathcal{Y}_{h+1}| \leq 2h + 1$, so the induction step is complete. By construction, eventually $\mathcal{H}_{|\mathcal{H}|} = \mathcal{H}$ and $\mathcal{Y}_{|\mathcal{H}|} = \mathcal{Y}$, so in consequence $|\mathcal{Y}| \leq 2|\mathcal{H}| - 1 \leq 2k - 1$.

As paths between vertices in $\mathcal{Y}$ form a tree, at most $2k - 2$ additional vertices are added to $\mathcal{Y}$ while constructing $\mathcal{Z}$ (at most one for each path) and the lemma follows. □

Algorithm 3: The extension procedure

Procedure `ExtendDecTree(`$\mathcal{T}$, D_C, $(a, b]$`):`

- $\mathcal{H} \leftarrow \{H \subseteq V \mid H \text{ is a heavy module}\}$;
- $\mathcal{X} \leftarrow \emptyset$;
- **for** $H \in \mathcal{H}$ **do**
  - Pick $v \in H$ and add v to $\mathcal{X}$;
- $\mathcal{Z} \leftarrow \mathcal{Y} \leftarrow \mathcal{X} \cup \left\{v \in \mathcal{T}\langle\mathcal{X}\rangle \mid \deg_{\mathcal{T}\langle\mathcal{X}\rangle}(v) \geq 3\right\}$;
- `// Branching vertices in` $\mathcal{T}\langle X\rangle$`.`
- **for** $u, v \in \mathcal{Y}$, $\mathcal{P}_{\mathcal{T}}(u, v) \neq \emptyset$, $\mathcal{P}_{\mathcal{T}}(u, v) \cap \mathcal{Y} = \emptyset$ **do**
  - Add $\arg\min_{z \in \mathcal{P}_{\mathcal{T}}(u,v)}\{c(z)\}$ to $\mathcal{Z}$;
  - `// Lightest vertex on path` $P_{\mathcal{T}}(u, v)$`.`
- $\mathcal{T}_{\mathcal{Z}} \leftarrow (\mathcal{Z}, \{uv \mid \mathcal{P}_{\mathcal{T}}(u, v) \cap \mathcal{Z} = \emptyset\})$;
- $D \leftarrow D_{\mathcal{Z}} \leftarrow$ `QPTAS(`$\mathcal{T}_{\mathcal{Z}}, \epsilon = 1$`)`;
- **for** $\mathcal{T}' \in \mathcal{T} - \mathcal{Z}$ **do**
  - Let H be the heavy module in $\mathcal{T}'$;
  - $D_H \leftarrow$ a decision tree built by using the vertex ranking of $\mathcal{T}\langle H\rangle$;
  - Hang D_H in D below the last query to a vertex $v \in N_{\mathcal{T}}(\mathcal{T}')$;
  - **for** $L \in \mathcal{T}' - H$ **do**
    - $D_L \leftarrow$ a decision tree built by deleting all queries in D_C outside of L;
    - Hang D_L in D below the last query to a vertex $v \in N_{\mathcal{T}'}(L)$;
- **return** D;

Lemma 3.3.1.22. *Let $\mathcal{T}_{\mathcal{Z}}$ be the auxiliary tree. Then, $\mathit{OPT}(\mathcal{T}_{\mathcal{Z}}) \leq \mathit{OPT}(\mathcal{T})$.*

Proof. Let D^* be the optimal strategy for $\mathcal{T}\langle\mathcal{Z}\rangle$. We build a new decision tree $D'_{\mathcal{Z}}$ for $\mathcal{T}_{\mathcal{Z}}$ by transforming D^*: Let $u, v \in \mathcal{Y}$ such that $\mathcal{P}_{\mathcal{T}}(u, v) \neq \emptyset$ and $\mathcal{P}_{\mathcal{T}}(u, v) \cap \mathcal{Y} = \emptyset$. Let $q \in V(D^*)$ such that $q \in \mathcal{P}_{\mathcal{T}}(u, v)$ is the first query among vertices of $\mathcal{P}_{\mathcal{T}}(u, v)$. We replace q in D^* by the query to the distinct vertex $v_{u,v} \in \mathcal{P}_{\mathcal{T}}(u, v) \cap \mathcal{Z}$ and delete all queries to vertices $\mathcal{P}_{\mathcal{T}}(u, v) - v_{u,v}$ from D^*. By construction, $D'_{\mathcal{Z}}$ is a valid decision tree for $\mathcal{T}_{\mathcal{Z}}$ and as for every $z \in \mathcal{P}_{\mathcal{T}}(u, v)$: $c(v_{u,v}) \leq c(z)$ such strategy has cost at most $\mathtt{COST}_{D'_{\mathcal{Z}}}(\mathcal{T}_{\mathcal{Z}}) \leq \mathtt{OPT}(\mathcal{T}\langle\mathcal{Z}\rangle)$. We get:

$$\mathtt{OPT}(\mathcal{T}_{\mathcal{Z}}) \leq \mathtt{COST}_{D'_{\mathcal{Z}}}(\mathcal{T}_{\mathcal{Z}}) \leq \mathtt{OPT}(\mathcal{T}\langle\mathcal{Z}\rangle) \leq \mathtt{OPT}(\mathcal{T})$$

where the first inequality is due to the optimality and the last inequality is due to the fact that $\mathcal{T}\langle\mathcal{Z}\rangle$ is a subtree of $\mathcal{T}$ (by Lemma 3.3.1.1). The lemma follows. □

Lemma 3.3.1.23. $\mathit{COST}_D(\mathcal{T}) \leq \mathit{COST}_{D_C}(\mathcal{T}_C) + \left(2 + \frac{b}{a}\right) \mathit{OPT}(\mathcal{T})$

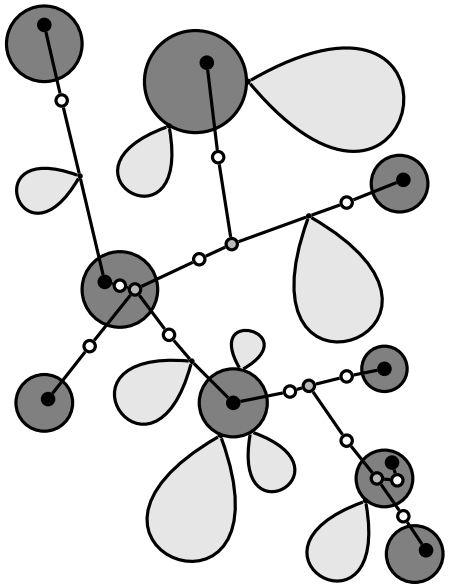

Figure 3.2: Example tree $\mathcal{T}$. Dark grey circles represent heavy modules. Light grey regions represent light subtrees. Black vertices represent $\mathcal{X}$. Gray and black vertices represent $\mathcal{Y}$. White, gray and black vertices represent $\mathcal{Z}$. Lines represent paths of vertices between vertices of $\mathcal{Z}$.

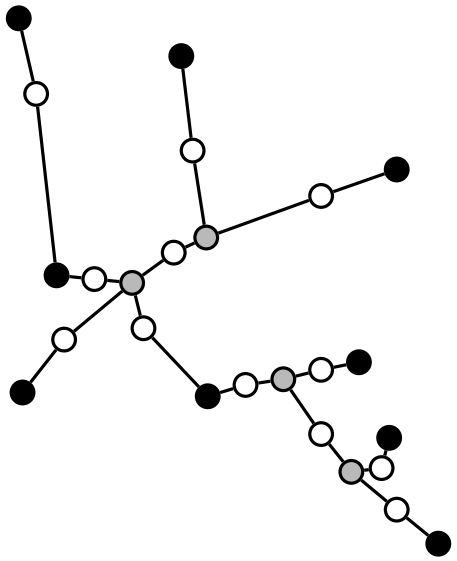

Figure 3.3: Auxiliary tree $\mathcal{T}_{\mathcal{Z}}$ built from vertices of set $\mathcal{Z}$. Lines represent edges between vertices of $\mathcal{T}_{\mathcal{Z}}$.

Proof. Let $H \subseteq V(\mathcal{T})$ be a heavy module. We show that: $\texttt{COST}_{D_H}(\mathcal{T}\langle H\rangle) \leq \frac{b}{a} \cdot \texttt{OPT}(\mathcal{T}, c)$. For every $v \in H$ let $c'(v) = a$ (note, that we can choose any cost here since we treat each query as unitary). As $\frac{bc'(v)}{a} \geq b \geq c(v)$ we get that $\frac{b}{a} \cdot \texttt{COST}_{D_H}(\mathcal{T}\langle H\rangle, c') \geq \texttt{COST}_{D_H}(\mathcal{T}\langle H\rangle, c)$. Additionally, using the fact that $c'(v) \leq c(v)$ we get that $\texttt{OPT}(\mathcal{T}\langle H\rangle, c') \leq \texttt{OPT}(\mathcal{T}\langle H\rangle, c)$. Therefore:

$$\texttt{COST}_{D_H}(\mathcal{T}\langle H\rangle, c) \leq \frac{b}{a} \cdot \texttt{COST}_{D_H}(\mathcal{T}\langle H\rangle, c') = \frac{b}{a} \cdot \texttt{OPT}(\mathcal{T}\langle H\rangle, c') \leq$$
$$\leq \frac{b}{a} \cdot \texttt{OPT}(\mathcal{T}\langle H\rangle, c) \leq \frac{b}{a} \cdot \texttt{OPT}(\mathcal{T}, c)$$

where the last inequality is due to the fact that $\mathcal{T}\langle H\rangle$ is a subtree of $\mathcal{T}$ (by Lemma **??**).

Let $Q_D(\mathcal{T}, x)$ be a sequence of queries performed in order to find target x. By construction of the Algorithm 3, $Q_D(\mathcal{T}, x)$ consists of at most three distinct subsequences of queries (some subsequences might be empty). Firstly, there is a sequence of queries belonging to $Q_{D_\mathcal{Z}}(\mathcal{T}_\mathcal{Z}, x)$. If $x \notin \mathcal{Z}$, then it is followed by a sequence of queries belonging to $Q_{D_H}(\mathcal{T}\langle H\rangle, x)$ such that $H \in \mathcal{T}'$ is a heavy module and $x \in \mathcal{T}'$. If $x \notin H$, at last, there is a sequence of queries belonging to $Q_{D_L}(L, x)$ for $L \in \mathcal{T}' - H$ such that $x \in L$. We have:

$$\begin{aligned}
\texttt{COST}_D(\mathcal{T}) &\leq \texttt{COST}_{D_\mathcal{Z}}(\mathcal{T}) + \max_{\mathcal{T}' \in \mathcal{T} - \mathcal{Z}} \left\{ \texttt{COST}_{D_H}(\mathcal{T}\langle H\rangle) + \max_{L \in \mathcal{T}' - H} \{\texttt{COST}_{D_L}(L)\} \right\} \\
&\leq \texttt{COST}_{D_\mathcal{Z}}(\mathcal{T}_\mathcal{Z}) + \max_{\mathcal{T}' \in \mathcal{T} - \mathcal{Z}} \left\{ \frac{b}{a} \cdot \texttt{OPT}(\mathcal{T}) + \texttt{COST}_{D_C}(\mathcal{T}_C) \right\} \\
&\leq 2\texttt{OPT}(\mathcal{T}_\mathcal{Z}) + \frac{b}{a} \cdot \texttt{OPT}(\mathcal{T}) + \texttt{COST}_{D_C}(\mathcal{T}_C) \leq \texttt{COST}_{D_C}(\mathcal{T}_C) + \left(2 + \frac{b}{a}\right) \cdot \texttt{OPT}(\mathcal{T})
\end{aligned}$$

where the first inequality is due to the construction of the returned decision tree, the second is due to Observation 3.3.1.20 and due to the fact that L is a subtree of $\mathcal{T}_C$ (Lemma 3.3.1.1), the third is due is due to Theorem 3.3.1.10 and the last is due to Lemma 3.3.1.22.

□

Using Lemma 3.3.1.21 we get that the QPTAS runs in time $2^{O\left(\log^2(4k(\mathcal{T}))\right)} = 2^{O\left(\log^2(k(\mathcal{T}))\right)}$ and all other computations can be performed in polynomial time which completes the proof of the Proposition 3.3.1.16.

3.3.2 Average case, non-uniform weights

In this section we show how to obtain a constant factor approximation for the $T||V, c, w||\sum C_i$ problem. Since the problem is NP-hard, one should not hope for a polynomial time exact solution (The proof of this fact is differed to the appendix). However, even constant factor approximation is quite surprising as in contrast, recall that the state-of-the-art algorithm for $T||V, c|C_{max}$ achieves only much weaker, $O\left(\sqrt{\log n}\right)$-approximation. The idea behind the solution is inspired by the work

of [CC17] and [BMN13]. In order to tightly lower bound the cost of the optimum we use the connection of the problem to the following weighted α-separator problem which is a generalization of the k-separator problem introduced in [BMN13]:

Weighted α-separator problem

Input: Graph $G = (V(G), E(G))$, the weight function $w : V \to \mathbb{N}^+$, the cost function $c : V \to \mathbb{N}^+$ and a real number α.
Output: A set $S \subseteq V(G)$ called *separator* such that for every $H \in G - S$: $w(S) \leq w(G)/\alpha$ and $c(S)$ is minimized.

Note that the authors of [BMN13] use an alternative definition for the unweighted version of the problem in which the size of each $H \in G - S$ is required to be $|H| \leq k$. As we are concerned with the more general variant of the problem it is easier for us to use our definition. Given the value α or k one can obtain the other by simply using the fact that the problems are equivalent when $k = w(G)/\alpha$. We will use these two notions whenever more comfortable. Note that for the special case when $\alpha = 0$ we have that $k = \infty$ and the optimal separator for G is $\emptyset$.

For the sake of our analysis we will be only concerned with the variant in which the underlying graph is a tree. When the weights are uniform, this problem is solvable in polynomial time [BMN13]. If otherwise, the problem is (weakly) NP-hard. The proof of this fact is differed to the appendix. Despite the hardness of the problem we show that it can be solved in a pseudopolynomial time by means of dynamic programming.

Theorem 3.3.2.1. *There exists a* $O\Big(n(\min\{c(T), w(T)/\alpha\})^2\Big)$ *time algorithm for the Weighted* α*-separator Problem.*

Proof. We use separate subprocedures depending on which of the values $c(T), w(T)$ is smaller. Both of them are inspired by the polynomial time algorithm for the unweighted version of the problem [BMN13]. We assume that that the input tree is rooted in an arbitrary vertex r. For any vertex $v \in V(T)$ let $\mathcal{C}_T(v) = \Big\{c_1, c_2, \ldots, c_{\deg_v^+}\Big\}$.

1. $w(T)/\alpha \leq c(T)$. Let C_v be the cost of the optimal separator of the subtree T_v. Define C_v^{in} as the cost of the optimal α-separator S under the condition that v belongs to S. We immediately have the following relationship:

$$C_v^{in} = c(v) + \sum_{c \in \mathcal{C}(v)} C_c$$

Assume that $v \notin S$. Let $H^v \in T - S$ such that $v \in H^v$. For each $1 \leq w \leq k$ let $C_v^{out}(w)$

denote the cost of the optimal α-separator S such that $v \notin S$ and $w(H^v) = w$. We have that:

$$C_v = \min\left\{C_v^{in}, \min_{1 \le w \le k} C_v^{out}(w)\right\}$$

For any vertex $v \in V(T)$ and any number $1 \le i \le \deg_v^+$ let $H_{v,i} \in T_{v,i} - S$ such that $v \in H_{v,i}$. Additionally, for any number $1 \le w \le k$ let $C_{v,i}^{out}(w)$ denote the weight of an optimal α-separator such that $v \notin S$ and $w(H_i^v) = w$. We have that:

$$C_v^{out}(w) = C_{v,\deg_v^+}^{out}(w)$$

$$C_{v,1}^{out}(w) = \begin{cases} \infty, & \text{if } w < w(v) \\ C_{c_1}^{in}, & \text{if } w = w(v) \\ C_{c_1}^{out}(w - w(v)), & \text{if } w > w(v) \end{cases}$$

$$C_{v,i}^{out}(w) = \min\left\{C_{v,i-1}^{out}(w) + C_{c_i}^{in}, \min_{1 \le j \le w-1}\left\{C_{v,i-1}^{out}(w-j) + C_{c_i}^{out}(j)\right\}\right\}$$

Where in the last equality the first term of the minimum function represents a situation in which $c_i \in S$. In such case all of the weight of $H_{v,i}$ belongs to H_{i-1}^v. The second term assumes contrary and checks all possible partitions of the weight between $T_{v,i-1}$ and T_{c_i}.

The relationships above suffice to calculate the value of C_r which is the cost of the optimal separator S. The computation is performed in a bottom-up, left to right manner starting with the leafs. To finish the description of the algorithm we observe then when this is the case (v is a leaf) we have that $C_v^{in} = c(v)$, $C_v^{out}(w) = 0$, $C_v = 0$ and all other values are equal to ∞.

As there are $O(nw(T)/\alpha)$ subproblems and each of them requires at most $w(T)/\alpha$ computational steps we obtain that the running time is $O\left(n(w(T)/\alpha)^2\right)$.

2. $w(T)/\alpha \ge c(T)$. For $v \in V(T)$ and $1 \le c \le c(T)$ let $W_v(c)$ be the optimal weight of H_v in the separator of the subtree T_v of cost at most c. Let $B(v,c)$ be a boolean value defined as:

$$B_v(c) = \begin{cases} 0, & \text{if } W_v(c) \le k \\ 1, & \text{if } W_v(c) > k \end{cases}$$

Define a boolean value $B_v^{in}(c)$ to be 0: if there exists a separator of the subtree T_v of cost at most c under the condition that v belongs to S and 1 if otherwise. For $1 \le i \le \deg_v^+$ we also define a boolean value $B_{v,i}^{in}(c)$ to be 0: if there exists a separator of the subtree $T_{v,i}$ of cost at most c under the condition that v belongs to S and 1 if otherwise. Of course

$B^{in}_v(c) = B^{in}_{v,\deg^+_v}(c)$. We have the following relationships:

$$B^{in}_{v,1}(c) = \begin{cases} 1, & \text{if } c < c(v) \\ B_{c_1}(c - c(v)), & \text{if } c \geq c(v) \end{cases}$$

$$B^{in}_{v,i}(c) = \min_{1 \leq j \leq c} \left\{ B^{in}_{v,i-1}(c-j) \vee B_{c_i}(j) \right\}$$

For each $1 \leq c \leq c(T)$ let $W^{out}_v(w)$ denote the optimal weight of H^v in the separator of the subtree T_v of cost at most c under assumption that $v \notin S$. We have that:

$$W_v(c) = \begin{cases} 0, & \text{if } \neg B^{in}_v(c) \\ W^{out}_v(c), & \text{otherwise} \end{cases}$$

For any vertex $v \in V(T)$ and for any number $1 \leq c \leq c(T)$ let $W^{out}_{v,i}(c)$ denote the optimal weight of H^v_i in the separator of the subtree $T_{v,i}$ of cost at most c under assumption that $v \notin S$. We have that:

$$W^{out}_v(c) = W^{out}_{v,\deg^+_v}(c)$$

If v has only one child we have that:

$$W^{out}_{v,1}(c) = \begin{cases} \infty, & \text{if } B^{in}_{c_1}(c) \\ w(v) + W_{c_1}(c), & \text{otherwise} \end{cases}$$

It is also easy to see that for $i > 1$ we have:

$$W^{out}_{v,i}(c) = \min_{1 \leq j \leq c} \left\{ W^{out}_{v,i-1}(c-j) + W_{c_i}(j) \right\}$$

To obtain the cost of the optimal separator we calculate the value $\min\{1 \leq c \leq c(T) \,|B_r(c) = 0\}$ which is the minimal value for which a valid separator of cost at most c exists. The computation is performed in a bottom-up, left to right manner starting with the leafs. To finish the description of the algorithm we observe then when this is the case (v is a leaf) we have that:

$$B^{in}_v(c) = \begin{cases} 0, & \text{if } c = c(v) \\ 1, & \text{otherwise} \end{cases}$$

And also:

$$W^{out}_v(c) = \begin{cases} w(v), & \text{if } c = 0 \\ \infty, & \text{otherwise} \end{cases}$$

As there are $O(nc(T))$ subproblems and each of them requires at most $c(T)$ computational steps we obtain that the running time is $O\Big(nc(T)^2\Big)$.

To obtain not only the value of the optimum but also a solution, in both cases, by using a standard tabling one is able to store the information about the vertices belonging to the optimal separator for all calculated subproblems. □

We would like to use the previously shown procedure the create a constant-factor approximation algorithm for $T||V, c, w||\sum C_i$. However, the running time of this algorithm may not be polynomial if both $c(T)$ and $w(T)$ are of order $\omega(\text{poly}(n))$. To alleviate this difficulty we will use a fairly standard rounding trick which will provide us with a bi-criteria FPTAS for the weighted α-separator.

Let $\texttt{OPT}_\alpha$ be the cost of the optimal solution for the chosen value of α. We will still require that the cost of the returned solution will be at most $\texttt{OPT}_\alpha$. However, we will relax the condition demanding that $H \in T - S$: $w(H) \leq \frac{w(T)}{\alpha}$ and we will only require that $w(H) \leq \frac{(1+\delta)w(T)}{\alpha}$ for some chosen value of δ.

Theorem 3.3.2.2. *Let S^* be an optimal weighted α-separator for (T, c, w, α). For any $\delta > 0$ there exists an algorithm which returns a separator S such that:*

1. *$w(H) \leq \frac{(1+\delta)\cdot w(T)}{\alpha}$ for every $H \in T - S$.*
2. *$c(S) \leq c(S^*)$.*

The algorithm runs in $O\big(n^3/\delta^2\big)$ time.

Proof. The procedure is as follows:

1. Let $K = \frac{\delta W}{n\alpha}$, for every $v \in V(T)$: $w'(v) = \left\lfloor \frac{w(v)}{K} \right\rfloor$, $w'(T) = \sum_{v \in V(T)} w'(v)$ and $\alpha' = \frac{w'(T)K\alpha}{w(T)}$.
2. Return the optimal separator S for (T, w', c, α') found using the $O\Big(n\big(\frac{W}{\alpha}\big)^2\Big)$ time algorithm.

Lemma 3.3.2.3. *Let S^* be the optimal separator for the original instance. $C(S) \leq C(S^*)$.*

Proof. We prove that S^* is a valid separator for the (T, w', c, α') instance. Recall that $k = W(T)/\alpha$. To simplify the analysis we introduce the auxiliary instance with values:

$$w''(v) = \left\lfloor \frac{w(v)}{K} \right\rfloor \cdot K, w''(T) = \sum_{v \in V(T)} w''(v)\,, k'' = k \text{ and consequently: } \alpha'' = \frac{w''(T)}{k''} = \frac{\alpha w''(T)}{w(T)}$$

Notice that in this new instance the absolute threshold for required size of each $H \in T - S^*$ has not changed ($k'' = k$) and for every $v \in V(T)$: $w''(v) \leq w(v)$. Therefore we have that $w''(H) \leq w(H) \leq k \leq \frac{w''(T)}{\alpha''}$. Now notice that the one prim instance has all of its weights scaled by a constant value of K and the relative factor of $\alpha' = \alpha''$. As multiplying by a constant does not

influence the validity of a solution S^* is a valid α'-separator for (T, w', c, α'). Additionally, as the costs remained unchanged and S is optimal for (T, w', c, α') the claim follows. □

Lemma 3.3.2.4. *For every $H \in T - S$ we have that $w(H) \leq \frac{(1+\delta)w(T)}{\alpha}$.*

Proof. By definition we have that $w'(v) \leq \frac{w(v)}{K} \leq w'(v) + 1$ and therefore also $Kw'(v) \leq w(v) \leq Kw'(v) + K$. We have that:

$$\sum_{v \in H} w(v) \leq K \cdot \sum_{v \in H} w'(v) + Kn \leq \frac{Kw'(T)}{\alpha'} + nK = \frac{w(T)}{\alpha} + \frac{\delta w(T)}{\alpha} = \frac{(1+\delta)\, w(T)}{\alpha}$$

where the second inequality is due to the fact that S is a α'-separator for $(T, w'(T), c, \alpha')$ instance. □

Combining the two above lemmas with the fact that $\frac{w'(T)}{\alpha'} = \frac{w(T)}{K\alpha} = \frac{n}{\delta}$ we have that the algorithm runs in time $O\big(n^3/\delta^2\big)$ as required. □

Constant-factor approximation for $T||V, w, c||\sum C_i$ As it turns out the $T||V, w, c||\sum C_i$ is deeply connected with the previously regarded Separator Problem. The idea behind the construction and the analysis of the algorithm is largely inspired by [CC17]. In order to tightly lower bound the value of the optimal solution, we split the cost of the optimal solution into levels, one for each $0 \leq k < w(T)$, such that each level consists of a $w(T)/k$-separator of T.

We start with some additional notation. By $L_{\texttt{OPT}}(k)$ we will denote the set of all maximal subtrees of weight at most k in OPT. We call such set the k-th *level* of OPT. Observe that as two candidate subtrees H_1, H_2 may only have a common part when $H_1 \subseteq H_2$ or $H_2 \subseteq H_1$, by the minimality we have that $H_1 \cap H_2 = \emptyset$. We denote $S_{\texttt{OPT}}(k) = V(T) - L_{\texttt{OPT}}(k)$. These are the vertices belonging to the separator at the level $L_{\texttt{OPT}}(k)$.

Lemma 3.3.2.5. $\mathit{OPT} = \sum_{k=0}^{w(T)-1} c(S_{\mathit{OPT}}(k))$

Proof. Consider any vertex v and let r be the weight of the subtree of candidates right before v is queried. Then the contribution of v to the LHS is $r \cdot c(v)$. On the other hand, for every $0 \leq k < r$ we have that $v \in S_{\texttt{OPT}}(k)$ so the contribution to the RHS is $r \cdot c(v)$ as well. □

Using the above lemma one easily obtains the following lower bound on the cost of the optimal solution:

$$2 \cdot \texttt{OPT} = 2 \cdot \sum_{k=0}^{w(T)-1} c(S_{\texttt{OPT}}(k)) \geq \sum_{k=0}^{w(T)} c(S_{\texttt{OPT}}(\lfloor k/2 \rfloor))$$

Using the above lower bound on the value of OPT we will prove the main result of this section, namely:

Theorem 3.3.2.6. *There exists a 4-approximation algorithm for $T||V,w,c||\sum C_i$ running in time $O\left(n^2(\min\{c(T),w(T)/\alpha\})^2\right)$ and a $(4+\epsilon)$-approximation algorithm running in time $O\left(n^4/\epsilon^2\right)$.*

Proof. We describe how the obtain the $(4+\epsilon)$-approximation since the pseudopolynomial time algorithm can be formulated and analyzed similarly (using the exact algorithm for the weighted separator problem with $\alpha = 2$). The procedure is as follows:

Algorithm 4: The $(4+\epsilon)$-approximation algorithm for $T||V,c,w||\sum C_i$.

Procedure `DecisionTree`(T,c,w,ϵ)**:**

- $S_T \leftarrow$ `SeparatorFPTAS` $\left(T,c,w,\alpha=2,\delta=\frac{\epsilon}{4+\epsilon}\right)$
- Build an arbitrary decision tree D_T on vertices of S_T.
- **foreach** $H \in T - S$ **do**
  - $D_H \leftarrow$`DecisionTree`(H,c,w,ϵ)
  - Hang D_H below the last query consistent with H in D_T.
- **return** D

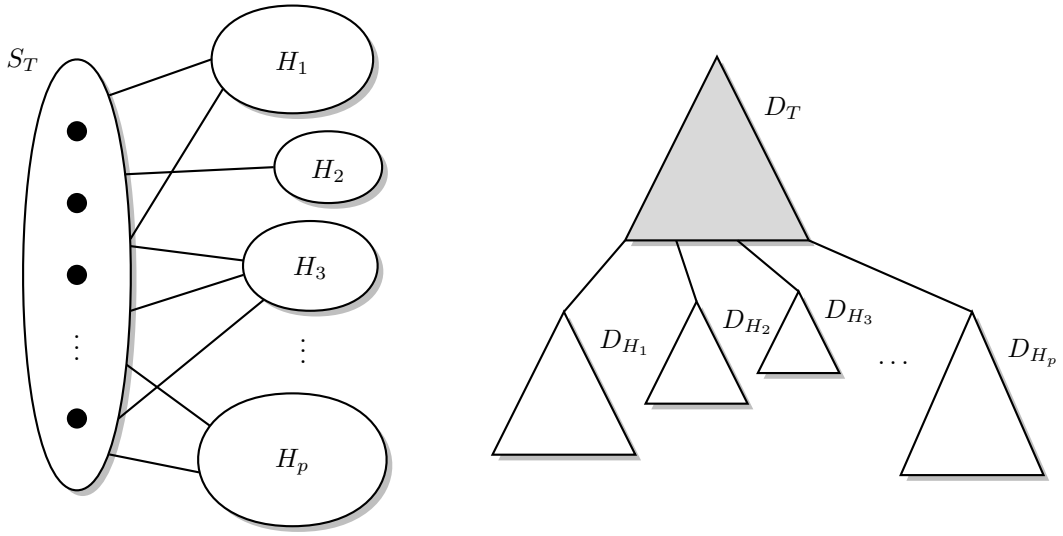

Figure 3.4: The separator S_T produced by the algorithm and the structure of the decision tree built using S_T.

Let $\mathcal{T}$ be a subtree for which the procedure was called. By the optimality of $S_{\mathcal{T}}$, we have that $c(S_{\mathcal{T}}) \leq c(S_{\texttt{OPT}}(\lfloor w(\mathcal{T})/2 \rfloor) \cap \mathcal{T})$. Let $0 < \beta < 1$ be some constant. We have the following bound:

$$\beta \cdot w(\mathcal{T}) \cdot c(S_{\texttt{OPT}}(\lfloor w(\mathcal{T})/2 \rfloor) \cap \mathcal{T}) \leq \sum_{k=(1-\beta)w(\mathcal{T})+1}^{w(\mathcal{T})} c(S_{\texttt{OPT}}(\lfloor k/2 \rfloor) \cap \mathcal{T})$$

since as k decreases, more vertices belong to the separator. Using $\beta = \frac{1-\delta}{2}$ we have that the

contribution of the decision tree $D_{\mathcal{T}}$ is therefore bounded by:

$$\begin{aligned} w(\mathcal{T}) \cdot c(S_{\mathcal{T}}) &\leq w(\mathcal{T}) \cdot c(S_{\texttt{OPT}}(\lfloor w(\mathcal{T})/2 \rfloor) \cap \mathcal{T}) \\ &\leq \frac{2}{1-\delta} \cdot \sum_{k=\frac{1+\delta}{2} \cdot w(\mathcal{T})+1}^{w(\mathcal{T})} c(S_{\texttt{OPT}}(\lfloor k/2 \rfloor) \cap \mathcal{T}) \end{aligned}$$

To bound the cost of the whole solution we will firstly show the following lemma which is necessary to proceed:

Lemma 3.3.2.7. $\sum_{\mathcal{T}} \sum_{k=w(\mathcal{T})/2+1}^{w(\mathcal{T})} c(S_{OPT}(\lfloor k/2 \rfloor) \cap \mathcal{T}) \leq \sum_{k=0}^{w(T)} c(S_{OPT}(\lfloor k/2 \rfloor))$

Proof. Fix a value of $\mathcal{T}$ and k. Their contribution to the LHS is $c(S_{\texttt{OPT}}(\lfloor k/2 \rfloor) \cap \mathcal{T})$. Consider which subtrees for which the algorithm is called contribute such term to the LHS. As $S_{\mathcal{T}}$ is a weighted $\frac{2}{1+\delta}$-separator we have that $\mathcal{T}$ is the minimal candidate subtree such that $w(\mathcal{T}) \geq k \geq \frac{(1+\delta)w(\mathcal{T})}{2} + 1 > w(H)$ for every $H \in \mathcal{T} - S_{\mathcal{T}}$. This means that if for every $H \in \mathcal{T} - S_{\mathcal{T}}$: $w(H) < t$ then $\mathcal{T}$ contributes such a term. Using the fact that for every query, the set of possible responses consists of disjoint subtrees, we conclude for all $H_1, H_2 \in \mathcal{T} - S_{\mathcal{T}}$ we have that $H_1 \cap H_2 = \emptyset$. Therefore also $(S_{\texttt{OPT}}(\lfloor k/2 \rfloor) \cap H_1) \cap (S_{\texttt{OPT}}(\lfloor k/2 \rfloor) \cap H_2 = \emptyset)$ and the claim follows by summing over all k. □

We are now ready to bound the cost of the solution. Let D_T be the decision tree returned by the procedure. Using the fact that by definition $\frac{4}{1-\delta} = 4 + \epsilon$, we have:

$$\begin{aligned} c(D_T) &\leq \sum_{\mathcal{T}} w(\mathcal{T}) \cdot c(S_{\mathcal{T}}) \\ &\leq \frac{2}{1-\delta} \cdot \sum_{\mathcal{T}} \sum_{k=\frac{1+\delta}{2} \cdot w(\mathcal{T})+1}^{w(\mathcal{T})} c(S_{\texttt{OPT}}(\lfloor k/2 \rfloor) \cap \mathcal{T}) \\ &\leq \frac{2}{1-\delta} \cdot \sum_{k=0}^{w(T)} c(S_{\texttt{OPT}}(\lfloor k/2 \rfloor)) \leq \frac{4}{1-\delta} \cdot \texttt{OPT} = (4+\epsilon) \cdot \texttt{OPT} \end{aligned}$$

As $1/\delta = \frac{4+\epsilon}{\epsilon} = 1 + 4/\epsilon$ and each $v \in V(T)$ belongs to the set $S_{\mathcal{T}}$ exactly once we have that the overall running time is at most $O\big(n^4/\epsilon^2\big)$ as required. □

3.4 Arbitrary graphs

3.4.1 Non-unitary costs, average case

Below we show how to generalize the approach we used for $T||V, c, w|| \sum C_i$. We follow a similar notation as before. By $L_{\texttt{OPT}}(k)$ we will denote the set of all maximal connected subgraphs of weight at most k in $\texttt{OPT}$. We will call such subgraphs clusters and we call such set the k-th *level* of $\texttt{OPT}$. Observe that as two clusters H_1, H_2 may only have a common part when $H_1 \subseteq H_2$ or $H_2 \subseteq H_1$, by the minimality we have that $H_1 \cap H_2 = \emptyset$. We denote $S_{\texttt{OPT}}(k) = V(G) - L_{\texttt{OPT}}(k)$. These are the vertices belonging to the separator at the level $L_{\texttt{OPT}}(k)$. However, in contrast to the analysis for $T||V, c, w|| \sum C_i$, we will use a connection to a different problem. The (generalized) min-ratio vertex cut is as follows:

Min-ratio vertex cut

Input: Graph $G = (V(G), E(G))$, the weight function $w : V \to \mathbb{N}^+$ and the cost function $c : V \to \mathbb{N}^+$.

Output: A partition (A, S, B) of $V(G)$ such that there is no $u \in A$ and $v \in B$ such that $uv \in E(G)$ which minimizes the ratio:

$$\alpha_{c,w}(A, S, B) = \frac{c(S)}{w(A \cup S) \cdot w(B \cup S)}$$

Let $\texttt{OPT}_{c,w}(G)$ denote the optimal value of such vertex cut. We invoke the following result by [FHL05] however we do not present their proof if this fact since it is quite involved. However we use the result below as a black box:

Theorem 3.4.1.1. *Given a graph $G = (V(G), E(G))$, the weight function $w : V \to \mathbb{N}^+$ and the cost function $c : V \to \mathbb{N}^+$, there exists a polynomial-time algorithm which computes a partition (A, S, B) for which:*

$$\alpha_{c,w}(A, S, B) = O\Big(\sqrt{\log n}\Big) \cdot \texttt{OPT}_{c,w}(G)$$

We combine the latter with the following result which combined yield an $O\big(\sqrt{\log n}\big)$-approximation algorithm for $G||V, c, w|| \sum C_i$:

Theorem 3.4.1.2. *Let f_n be the approximation ratio of any (polynomial time) algorithm for the min-ratio vertex cut. Then, given a graph $G = (V(G), E(G))$, the weight function $w : V \to \mathbb{N}^+$, the cost function $c : V \to \mathbb{N}^+$ there exists an $O(f_n)$-approximation algorithm for $G||V, c, w|| \sum C_i$ running in polynomial time.*

Algorithm 5: The f_n-approximation algorithm for $G||V, c, w|| \sum C_i$.

Proof.

Procedure `DecisionTree`(G, c, w)**:**
  $S_G \leftarrow$ `AlgorithmMinCut` (G, c, w)
  Build an arbitrary decision tree D_G on vertices of S_G.
  foreach $H \in G - S$ **do**
    $D_H \leftarrow$`DecisionTree`(H, c, w)
    Hang D_H below the last query consistent with H in D_G.
  return D

As before we have the following property (we omit the proof as it is the same for case of arbitrary graphs):

Lemma 3.4.1.3. $\mathit{OPT} = \sum_{k=0}^{w(G)-1} c(S_{\mathit{OPT}}(k))$

By which follows that:

$$2 \cdot \texttt{OPT} = 2 \cdot \sum_{k=0}^{w(G)-1} c(S_{\texttt{OPT}}(k)) \geq \sum_{k=0}^{w(G)} c(S_{\texttt{OPT}}(\lfloor k/2 \rfloor))$$

Let $\mathcal{G}$ be any cluster for which the procedure was called and let $r_{\mathcal{G}} = w(\mathcal{G})$. Let $0 < \beta < 1$ be some constant. We have the following upper bound:

$$\beta \cdot r_{\mathcal{G}} \cdot c(S_{\texttt{OPT}}(\lfloor r_{\mathcal{G}}/2 \rfloor) \cap \mathcal{G}) \leq \sum_{k=(1-\beta)r_{\mathcal{G}}+1}^{r_{\mathcal{G}}} c(S_{\texttt{OPT}}(\lfloor k/2 \rfloor) \cap \mathcal{G})$$

Denote by $H_1, \ldots, H_p$ the connected components of $\mathcal{G} - S_{\texttt{OPT}}(\lfloor r_{\mathcal{G}}/2 \rfloor \cap \mathcal{G})$. Let $\gamma_j = w(H_j)/r_{\mathcal{G}}$ and $s = w(S_{\texttt{OPT}}(\lfloor r_{\mathcal{G}}/2 \rfloor \cap \mathcal{G}))$. We have the following lemma:

Lemma 3.4.1.4. *Let $\lambda = 6 + 2\sqrt{5}$. We can partition $H_1, \ldots, H_p$ into two sets A and B, with $a = w(A)$ and $b = w(B)$ such that $(a + s)(b + s) \geq r_{\mathcal{G}}^2/\lambda$.*

Proof. There are two cases:

1. $s \geq r_{\mathcal{G}}/\sqrt{\lambda}$. In this case we take arbitrary partition A, B of $H_1, \ldots, H_p$. We have: $(a + s)(b + s) \geq s^2 = r_{\mathcal{G}}^2/\lambda$.

2. $s \leq r_{\mathcal{G}}/\sqrt{\lambda}$. Let $\gamma_A = \sum_{H_j \in A} \gamma_j$ and $\gamma_B = \sum_{H_j \in B} \gamma_j$. We have that $\gamma_A + \gamma_B \geq 1 - \frac{1}{\sqrt{\lambda}} = \frac{\sqrt{\lambda}-1}{\sqrt{\lambda}}$. Let A, B be the partition of $H_1, \ldots, H_p$ into two subsets both of size at least $\left(\frac{\sqrt{\lambda}-1}{\sqrt{\lambda}} - \frac{1}{2}\right) \cdot r_{\mathcal{G}}$ (this is always possible as $\frac{\sqrt{\lambda}-1}{\sqrt{\lambda}} > \frac{1}{2}$ and for each H_j: $\gamma_j \leq 1/2$). We have:

$$(a + s)(b + s) \geq ab \geq r_{\mathcal{G}} \cdot \left(\frac{\sqrt{\lambda} - 1}{\sqrt{\lambda}} - \frac{1}{2}\right) \cdot \frac{r_{\mathcal{G}}}{2} = \frac{r_{\mathcal{G}}^2}{\lambda}$$

(one can easily check that the equality is correct by substituting the value of λ).

□

We therefore have the following upper bound on the value of the optimal min-ratio cut:

$$\texttt{OPT}_{c,w}(\mathcal{G}) \leq \frac{\lambda \cdot c(S_{\texttt{OPT}}(\lfloor r_{\mathcal{G}}/2 \rfloor) \cap \mathcal{G})}{r_{\mathcal{G}}^2}$$

Let $(A_{\mathcal{G}}, S_{\mathcal{G}}, B_{\mathcal{G}})$ (with $a_{\mathcal{G}} = w(A_{\mathcal{G}})$, $s_{\mathcal{G}} = w(S_{\mathcal{G}})$, $b_{\mathcal{G}} = w(B_{\mathcal{G}})$ and $a_{\mathcal{G}} \geq b_{\mathcal{G}}$) be the partition returned by the f_n-approximate algorithm so:

$$\frac{c(S_{\mathcal{G}})}{(a_{\mathcal{G}} + s_{\mathcal{G}})(b_{\mathcal{G}} + s_{\mathcal{G}})} \leq f_n \cdot \frac{\lambda}{r_{\mathcal{G}}^2} \cdot c(S_{\texttt{OPT}}(\lfloor r_{\mathcal{G}}/2 \rfloor) \cap \mathcal{G})$$

Let $\beta = (b_{\mathcal{G}} + s_{\mathcal{G}})/r_{\mathcal{G}}$. As $r_{\mathcal{G}} = a_{\mathcal{G}} + s_{\mathcal{G}} + b_{\mathcal{G}}$ we have that $(1-\beta) r_{\mathcal{G}} + 1 = a(\mathcal{G}) + 1$ so we conclude that the contribution of the decision tree $D_{\mathcal{G}}$ is bounded by:

$$\begin{aligned} r_{\mathcal{G}} \cdot c(S_{\mathcal{G}}) &\leq \lambda \cdot f_n \cdot \frac{(a_{\mathcal{G}} + s_{\mathcal{G}})(b_{\mathcal{G}} + s_{\mathcal{G}})}{a_{\mathcal{G}} + s_{\mathcal{G}} + b_{\mathcal{G}}} \cdot c(S_{\texttt{OPT}}(\lfloor r_{\mathcal{G}}/2 \rfloor) \cap \mathcal{G}) \\ &\leq \lambda \cdot f_n \cdot (b_{\mathcal{G}} + s_{\mathcal{G}}) \cdot c(S_{\texttt{OPT}}(\lfloor r_{\mathcal{G}}/2 \rfloor) \cap \mathcal{G}) \\ &\leq \lambda \cdot f_n \cdot \sum_{k=a_{\mathcal{G}}+1}^{r_{\mathcal{G}}} c(S_{\texttt{OPT}}(\lfloor k/2 \rfloor) \cap \mathcal{G}) \end{aligned}$$

As before, to bound the cost of the whole solution we will firstly show the following lemma. The argument is mostly the same, however there are few differences and we include it for completeness:

Lemma 3.4.1.5. $\sum_{\mathcal{G}} \sum_{k=a_{\mathcal{G}}+1}^{r_{\mathcal{G}}} c(S_{OPT}(\lfloor k/2 \rfloor) \cap \mathcal{G}) \leq \sum_{k=0}^{w(G)} c(S_{OPT}(\lfloor k/2 \rfloor))$

Proof. Fix a value of $\mathcal{G}$ and k. Their contribution to the LHS is $c(S_{\texttt{OPT}}(\lfloor k/2 \rfloor) \cap \mathcal{G})$. Consider which clusters for which the algorithm is called contribute such term to the LHS. By definition of $S_{\mathcal{G}}$ we have that $\mathcal{G}$ is the minimal cluster such that $w(\mathcal{G}) \geq k \geq a_{\mathcal{G}} + 1 > w(H)$ for every $H \in \mathcal{G} - S_{\mathcal{G}}$. This means that if for every $H \in \mathcal{G} - S_{\mathcal{G}}$: $w(H) < k$ then $\mathcal{G}$ contributes such a term. Using the fact that for every query, the set of possible responses consists of disjoint subgraphs, we conclude for all $H_1, H_2 \in \mathcal{G} - S_{\mathcal{G}}$ we have that $H_1 \cap H_2 = \emptyset$. Therefore also $(S_{\texttt{OPT}}(\lfloor k/2 \rfloor) \cap H_1) \cap (S_{\texttt{OPT}}(\lfloor k/2 \rfloor) \cap H_2 = \emptyset)$ and the claim follows by summing over all k. □

We are now ready to bound the cost of the solution. Let D_G be the decision tree returned by

the procedure.

$$
\begin{aligned}
c(D_G) &\leq \sum_{\mathcal{G}} r_{\mathcal{G}} \cdot c(S_{\mathcal{G}}) \\
&\leq \lambda \cdot f_n \cdot \sum_{\mathcal{G}} \sum_{k=a_{\mathcal{G}}+1}^{r_{\mathcal{G}}} c(S_{\texttt{OPT}}(\lfloor k/2 \rfloor) \cap \mathcal{G}) \\
&\leq \lambda \cdot f_n \cdot \sum_{k=0}^{w(G)} c(S_{\texttt{OPT}}(\lfloor k/2 \rfloor)) \leq 2 \cdot \lambda \cdot f_n \cdot \texttt{OPT}(G) = \left(12 + 4\sqrt{5}\right) \cdot f_n \cdot \texttt{OPT}(G)
\end{aligned}
$$

□

Chapter 4

Experimental Results

Chapter 5

Conclusions

Bibliography

[Als+05] Stephen Alstrup et al. "Maintaining information in fully dynamic trees with top trees". In: *ACM Trans. Algorithms* 1.2 (Oct. 2005), pp. 243–264. ISSN: 1549-6325. DOI: 10.1145/1103963.1103966. URL: https://doi.org/10.1145/1103963.1103966.

[Ang18] Haris Angelidakis. "Shortest path queries, graph partitioning and covering problems in worst and beyond worst case settings". In: *ArXiv* abs/1807.09389 (2018). URL: https://api.semanticscholar.org/CorpusID:51718679.

[BBS12] Gowtham Bellala, Suresh K. Bhavnani, and Clayton Scott. "Group-Based Active Query Selection for Rapid Diagnosis in Time-Critical Situations". In: *IEEE Transactions on Information Theory* 58.1 (2012), pp. 459–478. DOI: 10.1109/TIT.2011.2169296.

[BDO23] Piotr Borowiecki, Dariusz Dereniowski, and Dorota Osula. "The complexity of bicriteria tree-depth". In: *Theoretical Computer Science* 947 (2023), p. 113682. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2022.12.032. URL: https://www.sciencedirect.com/science/article/pii/S0304397522007666.

[Ber+22] Benjamin Berendsohn et al. *Fast approximation of search trees on trees with centroid trees*. Sept. 2022. DOI: 10.48550/arXiv.2209.08024.

[BK22] Benjamin Berendsohn and László Kozma. "Splay trees on trees". In: Jan. 2022, pp. 1875–1900. ISBN: 978-1-61197-707-3. DOI: 10.1137/1.9781611977073.75.

[BMN13] Walid Ben-Ameur, Mohamed S. A. Mohamed, and José Neto. "The k-separator problem". In: *COCOON '13 : 19th International Computing & Combinatorics Conference*. Vol. 7936. Hangzhou, China: Springer, June 2013, pp. 337–348. DOI: 10.1007/978-3-642-38768-5_31. URL: https://hal.science/hal-00843860.

[Bod+98] Hans L. Bodlaender et al. "Rankings of Graphs". In: *SIAM Journal on Discrete Mathematics* 11.1 (1998), pp. 168–181. DOI: 10.1137/S0895480195282550. eprint: https://doi.org/10.1137/S0895480195282550. URL: https://doi.org/10.1137/S0895480195282550.

[CC17] Moses Charikar and Vaggos Chatziafratis. "Approximate hierarchical clustering via sparsest cut and spreading metrics". In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '17. Barcelona, Spain: Society for Industrial and Applied Mathematics, 2017, pp. 841–854.

[Cic+12] Ferdinando Cicalese et al. "The binary identification problem for weighted trees". In: *Theoretical Computer Science* 459 (2012), pp. 100–112. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2012.06.023.

[Cic+14] Ferdinando Cicalese et al. "Improved Approximation Algorithms for the Average-Case Tree Searching Problem". In: *Algorithmica* 68 (Apr. 2014). DOI: 10.1007/s00453-012-9715-6.

[Cic+16] Ferdinando Cicalese et al. "On the tree search problem with non-uniform costs". In: *Theoretical Computer Science* 647 (2016), pp. 22–32. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2016.07.019.

[CLS14] Ferdinando Cicalese, Eduardo Laber, and Aline Medeiros Saettler. "Diagnosis determination: decision trees optimizing simultaneously worst and expected testing cost". In: *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*. ICML'14. Beijing, China: JMLR.org, 2014, pp. I–414–I–422.

[Das04] Sanjoy Dasgupta. "Analysis of a greedy active learning strategy". In: *Advances in Neural Information Processing Systems*. Ed. by L. Saul, Y. Weiss, and L. Bottou. Vol. 17. MIT Press, 2004. URL: https://proceedings.neurips.cc/paper_files/paper/2004/file/c61fbef63df5ff317aecdc3670094472-Paper.pdf.

[Der+17] Dariusz Dereniowski et al. "Approximation Strategies for Generalized Binary Search in Weighted Trees". In: *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*. Ed. by Ioannis Chatzigiannakis et al. Vol. 80. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, 84:1–84:14. ISBN: 978-3-95977-041-5. DOI: 10.4230/LIPIcs.ICALP.2017.84.

[Der06] Dariusz Dereniowski. "Edge ranking of weighted trees". In: *Discrete Applied Mathematics* 154.8 (2006), pp. 1198–1209. ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2005.11.005.

[Der08] Dariusz Dereniowski. "Edge ranking and searching in partial orders". In: *Discrete Applied Mathematics* 156.13 (2008). Fifth International Conference on Graphs and Optimization, pp. 2493–2500. ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2008.03.007.

[DGP23] Dariusz Dereniowski, Przemysław Gordinowicz, and Paweł Prałat. “Edge and Pair Queries—Random Graphs and Complexity”. In: *The Electronic Journal of Combinatorics* 30.2 (2023). DOI: 10.37236/11159. URL: https://www.combinatorics.org/ojs/index.php/eljc/article/view/v30i2p34.

[DGW24] Dariusz Dereniowski, Przemysław Gordinowicz, and Karolina Wr'obel. “On multidimensional generalization of binary search”. In: *ArXiv* abs/2404.13193 (2024). URL: https://api.semanticscholar.org/CorpusID:269293685.

[DK06] Dariusz Dereniowski and Marek Kubale. “Efficient Parallel Query Processing by Graph Ranking”. In: *Fundam. Inform.* 69 (Feb. 2006), pp. 273–285. DOI: 10.3233/FUN-2006-69302.

[DŁU21] Dariusz Dereniowski, Aleksander Łukasiewicz, and Przemysław Uznański. “An Efficient Noisy Binary Search in Graphs via Median Approximation”. In: *Combinatorial Algorithms*. Ed. by Paola Flocchini and Lucia Moura. Cham: Springer International Publishing, 2021, pp. 265–281. ISBN: 978-3-030-79987-8.

[DŁU25] Dariusz Dereniowski, Aleksander Łukasiewicz, and Przemysław Uznański. “Noisy (Binary) Searching: Simple, Fast and Correct”. In: *42nd International Symposium on Theoretical Aspects of Computer Science (STACS 2025)*. Ed. by Olaf Beyersdorff et al. Vol. 327. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 29:1–29:18. ISBN: 978-3-95977-365-2. DOI: 10.4230/LIPIcs.STACS.2025.29. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.STACS.2025.29.

[DMS19] Argyrios Deligkas, George B. Mertzios, and Paul G. Spirakis. “Binary Search in Graphs Revisited”. In: *Algorithmica* 81.5 (May 2019), pp. 1757–1780. ISSN: 1432-0541. DOI: 10.1007/s00453-018-0501-y.

[DN06] Dariusz Dereniowski and Adam Nadolski. “Vertex rankings of chordal graphs and weighted trees”. In: *Information Processing Letters* 98.3 (2006), pp. 96–100. ISSN: 0020-0190. DOI: https://doi.org/10.1016/j.ipl.2005.12.006.

[DW22] Dariusz Dereniowski and Izajasz Wrosz. “Constant-Factor Approximation Algorithm for Binary Search in Trees with Monotonic Query Times”. In: *47th International Symposium on Mathematical Foundations of Computer Science (MFCS 2022)*. Ed. by Stefan Szeider, Robert Ganian, and Alexandra Silva. Vol. 241. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 42:1–42:15. ISBN: 978-3-95977-256-3. DOI: 10.4230/LIPIcs.MFCS.2022.42.

[DW24] Dariusz Dereniowski and Izajasz Wrosz. *Searching in trees with monotonic query times*. 2024. arXiv: 2401.13747 [cs.DS]. URL: https://arxiv.org/abs/2401.13747.

[EKS16] Ehsan Emamjomeh-Zadeh, David Kempe, and Vikrant Singhal. "Deterministic and probabilistic binary search in graphs". In: June 2016, pp. 519–532. DOI: 10.1145/2897518.2897656.

[FHL05] Uriel Feige, MohammadTaghi Hajiaghayi, and James R. Lee. "Improved approximation algorithms for minimum-weight vertex separators". In: *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*. STOC '05. Baltimore, MD, USA: Association for Computing Machinery, 2005, pp. 563–572. ISBN: 1581139608. DOI: 10.1145/1060590.1060674. URL: https://doi.org/10.1145/1060590.1060674.

[GHT12] 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". In: *Discrete Applied Mathematics* 160.15 (2012), pp. 2089–2097. ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2012.03.015. URL: https://www.sciencedirect.com/science/article/pii/S0166218X12001199.

[GKR10] Daniel Golovin, Andreas Krause, and Debajyoti Ray. "Near-optimal Bayesian active learning with noisy observations". In: *Proceedings of the 24th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'10. Vancouver, British Columbia, Canada: Curran Associates Inc., 2010, pp. 766–774.

[GNR10] Anupam Gupta, Viswanath Nagarajan, and R. Ravi. "Approximation Algorithms for Optimal Decision Trees and Adaptive TSP Problems". In: *Automata, Languages and Programming*. Ed. by Samson Abramsky et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 690–701. ISBN: 978-3-642-14165-2.

[Gra69] R. L. Graham. "Bounds on Multiprocessing Timing Anomalies". In: *SIAM Journal on Applied Mathematics* 17.2 (1969), pp. 416–429. DOI: 10.1137/0117039. eprint: https://doi.org/10.1137/0117039. URL: https://doi.org/10.1137/0117039.

[Høg+21] Svein Høgemo et al. "On Dasgupta's Hierarchical Clustering Objective and Its Relation to Other Graph Parameters". In: *Fundamentals of Computation Theory*. Ed. by Evripidis Bampis and Aris Pagourtzis. Cham: Springer International Publishing, 2021, pp. 287–300. ISBN: 978-3-030-86593-1.

[Høg24] Svein Høgemo. "Tight Approximation Bounds on a Simple Algorithm for Minimum Average Search Time in Trees". In: *ArXiv* abs/2402.05560 (2024). URL: https://api.semanticscholar.org/CorpusID:267547530.

[Jac+10] Tobias Jacobs et al. “On the Complexity of Searching in Trees: Average-Case Minimization”. In: *Automata, Languages and Programming*. Ed. by Samson Abramsky et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 527–539. ISBN: 978-3-642-14165-2.

[Jor69] Camille Jordan. “Sur les assemblages de lignes.” fre. In: *Journal für die reine und angewandte Mathematik* 70 (1869), pp. 185–190. URL: http://eudml.org/doc/148084.

[KMS95] Meir Katchalski, William McCuaig, and Suzanne Seager. “Ordered colourings”. In: *Discrete Mathematics* 142.1 (1995), pp. 141–154. ISSN: 0012-365X. DOI: https://doi.org/10.1016/0012-365X(93)E0216-Q. URL: https://www.sciencedirect.com/science/article/pii/0012365X93E0216Q.

[Knu73] Donald Knuth. *The Art Of Computer Programming, vol. 3: Sorting And Searching*. Addison-Wesley, 1973, pp. 391–392.

[KPB99] S. Rao Kosaraju, Teresa M. Przytycka, and Ryan Borgstrom. “On an Optimal Split Tree Problem”. In: *Algorithms and Data Structures*. Ed. by Frank Dehne et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 157–168. ISBN: 978-3-540-48447-9.

[KZ13] Amin Karbasi and Morteza Zadimoghaddam. “Constrained Binary Identification Problems”. In: *30th International Symposium on Theoretical Aspects of Computer Science (STACS)*. Vol. 20. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2013, pp. 550–561. DOI: 10.4230/LIPIcs.STACS.2013.550. URL: https://drops.dagstuhl.de/opus/volltexte/2013/3942.

[Leu89] Joseph Y-T. Leung. “Bin packing with restricted piece sizes”. In: *Information Processing Letters* 31.3 (1989), pp. 145–149. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(89)90223-8. URL: https://www.sciencedirect.com/science/article/pii/0020019089902238.

[LLM] Ray Li, Percy Liang, and Stephen Mussmann. “A Tight Analysis of Greedy Yields Subexponential Time Approximation for Uniform Decision Tree”. In: *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 102–121. DOI: 10.1137/1.9781611975994.7. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611975994.7. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611975994.7.

[LN04] Eduardo S. Laber and Loana Tito Nogueira. “On the hardness of the minimum height decision tree problem”. In: *Discrete Applied Mathematics* 144.1 (2004). Discrete Mathematics and Data Mining, pp. 209–212. ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2004.06.002. URL: https://www.sciencedirect.com/science/article/pii/S0166218X04001787.

[LY98] Tak Wah Lam and Fung Ling Yue. "Edge ranking of graphs is hard". In: *Discrete Applied Mathematics* 85.1 (1998), pp. 71–86. ISSN: 0166-218X. DOI: https://doi.org/10.1016/S0166-218X(98)00029-8.

[NO06] Jaroslav Nešetřil and Patrice Ossona de Mendez. "Tree-depth, subgraph coloring and homomorphism bounds". In: *European Journal of Combinatorics* 27.6 (2006), pp. 1022–1041. ISSN: 0195-6698. DOI: https://doi.org/10.1016/j.ejc.2005.01.010. URL: https://www.sciencedirect.com/science/article/pii/S0195669805000570.

[OP06] Krzysztof Onak and Pawel Parys. "Generalization of Binary Search: Searching in Trees and Forest-Like Partial Orders". In: *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*. 2006, pp. 379–388. DOI: 10.1109/FOCS.2006.32.

[Pot88] Alex Pothen. *The Complexity of Optimal Elimination Trees*. Technical Report CS-88-16. Penn State University CS-88-16. Pennsylvania State University, Department of Computer Science, Apr. 1988.

[Sah76] Sartaj K. Sahni. "Algorithms for Scheduling Independent Tasks". In: *J. ACM* 23.1 (Jan. 1976), pp. 116–127. ISSN: 0004-5411. DOI: 10.1145/321921.321934. URL: https://doi.org/10.1145/321921.321934.

[Sch89] Alejandro A. Schäffer. "Optimal node ranking of trees in linear time". In: *Information Processing Letters* 33.2 (1989), pp. 91–96. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(89)90161-0.

[SLC14] Aline Saettler, Eduardo Laber, and Ferdinando Cicalese. "Trading off Worst and Expected Cost in Decision Tree Problems and a Value Dependent Model". In: (June 2014).

[Yao80] F. Frances Yao. "Efficient dynamic programming using quadrangle inequalities". In: *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*. STOC '80. Los Angeles, California, USA: Association for Computing Machinery, 1980, pp. 429–435. ISBN: 0897910176. DOI: 10.1145/800141.804691. URL: https://doi.org/10.1145/800141.804691.

Appendix A

Hardness proofs

Theorem A.0.0.1. *The decision version of $T||V, c, w||\sum C_j$ is NP-complete in the class of trees with diameter at most 8 and in the class of trees with degree at most 16.*

Proof. Of course the problem is in NP since given a decision tree D one can in polynomial time check whether all of the requirements are fulfilled.

To show the hardness we use the fact that $T||E, w||\sum C_j$ is NP-complete in the class of trees with diameter at most 4 and in the class of trees with degree at most 16 [Jac+10]. Let (T, w, K) be such instance. We create a new instance (T', c, w, K') for $T||V, c, w||\sum C_j$ in the following way. For every $v \in V(T)$ we set $c(v) = K + 1$. We subdivide each edge $e \in E(T)$ by adding new vertex v_e with $w(v_e) = 0$ and $c(v_e) = 1$. We set $K' = K + w(T) \cdot (K + 1)$. Assume that we have a decision tree D of cost at most K for the original instance. To obtain a decision tree D' for the second instance we simply replace each query in D by a query to a vertex which subdivided the queried edge. Additionally, below each leaf of D we hang appropriate queries to the left vertices (as D contains a query to every edge of T each vertex is separated, so for every $v \in V(T)$ one such additional query is added. This results in a decision tree D' of cost at most $K' = K + w(T)\,(K + 1)$ as required.

Observe that in the new instance for every vertex $v \in T$ we have that the cost of searching for v is at least $K + 1$ since $c(v) = K + 1$. Therefore for these vertices, at least $w(T) \cdot (K + 1)$ cost is required assuming that in the query sequence of each such vertex there is only one such costly query, namely query to v itself. Notice that if it was not the case the cost would exceed K' so we conclude that every such v is queried only when the subtree of candidates consists of v as the only vertex. Assume that we have a tree D for the second instance of cost at most K'. We show how to obtain a decision tree D' for the original instance. We delete all of the queries to the vertices $v \in V(T)$. We also replace each query to the vertex in a subdivided edge with a query to this

edge. As each query to $v \in V(T)$ was a query to a last vertex in the set of candidates we obtain that D' is a valid decision tree for the original instance. Additionally, the cost of D' is at most $K' - w(T) \cdot (K+1) = K$ as required.

Note that by subdividing each edge the diameter of the tree doubles and the degree remains unchanged so the claim follows. □

Theorem A.0.0.2. *The decision version of the Weighted α-separator Problem is (weakly) NP-complete even when restricted to stars.*

Decision weighted α-separator problem

Input: *Graph $G = (V(G), E(G))$, the weight function $w : V \to \mathbb{N}^+$, the cost function $c : V \to \mathbb{N}^+$, a real number α and an integer number K.*
Output: *Whether there exists a set $S \subseteq V(G)$ such that for every $H \in G - S$: $w(H) \leq w(G)/\alpha$ and $c(S) \leq K$.*

Proof. Of course the problem is in NP since given a set $S \subseteq V(T)$ one can in polynomial time check whether all of the requirements are fulfilled.

To show the hardness we use the following Partition problem which is known to be weakly NP-complete.

Partition problem

Input: A set of integers $A = \{a_1, ..., a_n\}$.
Output: Whether there exists a subset $A' \subseteq A$ such that $\sum_{a \in A'} a = \frac{1}{2} \sum_{a \in A} a$.

Let $A = \{a_1, \ldots, a_n\}$ be an arbitrary Partition instance. To convert it to a corresponding α-separator instance we set $K = \frac{1}{2} \sum_{a \in A} a$, $\alpha = 2$ and create the tree T with following vertices: Create a vertex r such that $w(r) = 0$ and $c(r) = K + 1$. Then for each $a \in A$ create a vertex v_a such that $w(v_a) = a$ and $w(c_a) = a$ and attach it r by an edge rv_a. First of all, observe that $r \notin S$ since it is required that $c(S) \leq K$. Notice that as $\sum_{v \in V(T)} w(w) = K$ if some $v \in S$ then the $c(S)$ increases by a_v and if otherwise for the unique $H \in T - S$: $w(H)$ increases by a_v. Therefore a valid solution for the Partition problem exists iff it is possible to partition the vertices of T into two distinct subsets of which the one not containing r is S such that $c(S) = w(V(T - S))$. The claim follows. □