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 28, 2025

# Abstract

In this work, we conduct an experimental analysis of the generalized binary search problem in graphs. The analysis explores various classes of graphs, including: paths, trees and general graphs. 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).

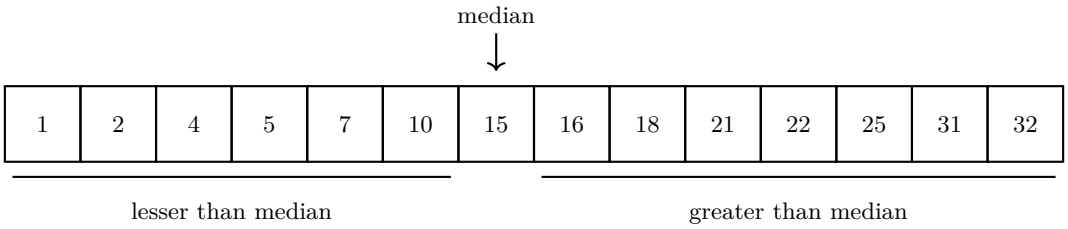


Figure 1.1: Example of a sorted array containing 14 elements. The subarrays with elements lesser than and greater then the median (15) are underlined. If the hidden element is for example 2, then the result of the comparison operation is "below" and the searcher can immediately discard all elements of 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 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

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

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.

## 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.

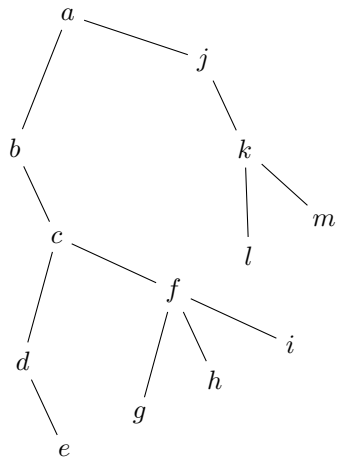


Figure 1.2

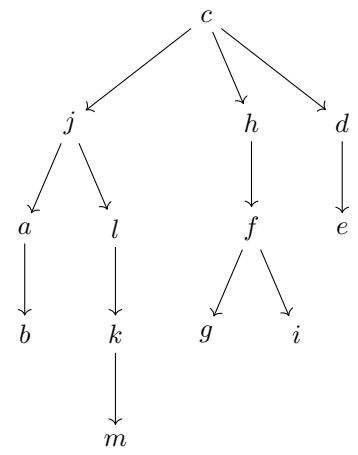


Figure 1.3

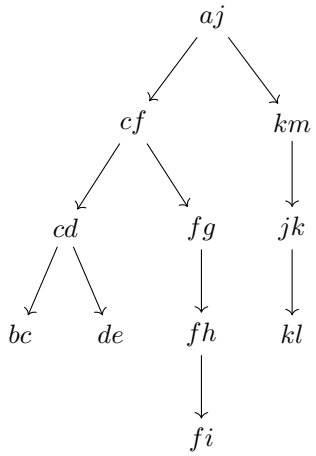


Figure 1.4

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).

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

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.

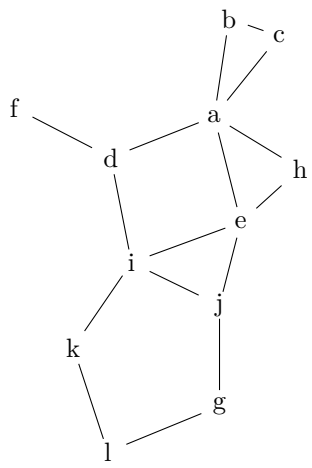


Figure 1.6

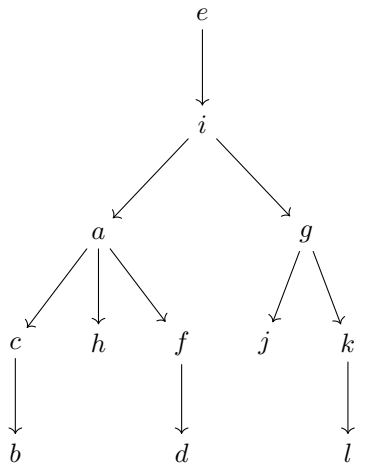


Figure 1.7

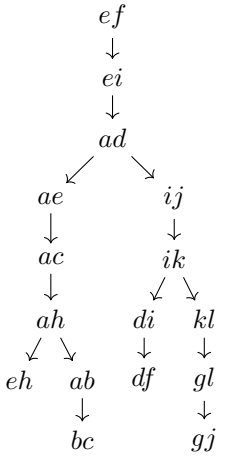


Figure 1.8

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).

**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*

[3]Or equivaletly a vertex.

*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 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: $\alpha, \beta$ and $\gamma$. The $\alpha$ field is the search space

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

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

environment field resembling the machine environment. The $\beta$ field is the query characteristics which resembles the job characteristics. The $\gamma$ 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:

| $\alpha$ - search space | $\beta$ - query characteristics | $\gamma$ - 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 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],

[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].

- 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 will also be mostly concerned with the vertex query variant of the problem, since it is usually, the more general variant.

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 Motivation and applications

## 1.7 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 problems

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 graph search problem

Below we list the definitions regarding the search problem. 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 present a general Graph Search Problem, which we will later specify

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 2.3.0.1.** *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.*

### 2.3.1 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 (either vertex or edge queries). The *cost* of query $q \in \mathcal{Q}$ is then denoted as $c : \mathcal{Q} \to \mathbb{R}^+$. We will also allow each vertex to have a *weight function* $c : V(G) \to \mathbb{R}^+$ on vertices.

### 2.3.2 Decision trees, optimization criteria and the Graph Search Problem

Let $G$ be a graph. A decision tree is a rooted tree $D = (V(D), E(D))$, where $V(D) = V(G)$ are the vertices of $D$ and $E(D)$ are the 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 subtree of candidate solutions that remain after performing all previous queries.

Let $Q_D(G, x)$ denote the sequence of queries made to locate a target $x \in V(G)$ using $D$, i. e., the sequence of vertices belonging to the unique path in $D$ starting at $r(D)$ and ending at $x$. We define the worst case cost of a decision tree $D$ in $(G, c)$

$$\texttt{COST}_{max,G}(D, c) = \max_{x \in V(G)} \left\{ \sum_{q \in Q_D(V(G),x)} c(q) \right\}$$

We define the average case cost of a decision tree $D$ in $(G, c, w)$ with as:

$$\texttt{COST}_{avg,D}(I, w) = \sum_{v \in V(G)} \sum_{q \in Q_D(V(G),x)} c(q)$$

By a slight abuse of notation we will also sometimes use $Q_D(V(G), x)$ as the set consisting of queries in sequence $Q_D(V(G), x)$. 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 *Graph Search Problem*:

**Generalized Search Problem**

**Input:** Graph $G$, the query model and the optimization criterion
**Output:** A viable decision tree for $G$ according to 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.

### 3.1.1 A warm up: $O(n^3)$ algorithm for $P||E,c||C_{max}$ 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:

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

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:

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

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.

### 3.1.2 An $O(n^2)$ algorithm for $P||V,w||\sum C_i$.

**Theorem 3.1.2.1.** *There exists an $O(n^2)$ algorithm for $P||V,w||\sum C_i$*

*Proof.* The idea behind the speed-up described below is due to Knuth [Knu73] and [Yao80]. For every $i \leq k < j$ 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 3.1.2.2.** *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 3.1.2.3.** $K(i,j-1) \leq K(i,j) \leq K(i+1,j)$

*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 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 $O(n^2)$. The amount of computation

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

steps required by the algorithm is equal to:

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

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. □

### 3.1.3 An $O(n^2)$ algorithm for $P||V,c||C_{max}$.

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 Trees, Worst Case, Uniform Costs

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.

## 3.3 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 $T|V,w|\sum C_j$.

### 3.3.1 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 3.3.1.1.** *Let $D_c$ be the greedy decision tree. Then $\texttt{COST}_{D_c}(T) \leq 2\texttt{OPT}(T) - w(T)$.*

*Proof.* We start with the following lemma:

**Lemma 3.3.1.2.** *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:

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

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:

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

where the first inequality is by the induction hypothesis and the second inequality is by the Lemma 3.3.1.2. □

**Theorem 3.3.1.3.** *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 3.3.1.4.** *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 $O(n \log n)$. □

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

**Theorem 3.3.2.1.** *Fix $\epsilon > 1$. There exists an $(1+\epsilon)$-approximation algorithm for $T||V, w|| \sum C_i$ running in $O\big(n^{(2/\epsilon+3)} \log n/\epsilon\big)$ time.*

*Proof.* To design our PTAS we will make use of the following lemma combined with a non trivial dynamic programming procedure due to [Cic+14; Ang18; Ber24]. Note that, this is not the only way to obtain PTAS, see [BK22].

**Lemma 3.3.2.2.** *Fix $\epsilon > 1$. For every tree $T$, there exists a decision tree $D$, such that:*

1. $\texttt{COST}_{avg,D}(T, w) \leq (1+\epsilon) \cdot \texttt{OPT}_{avg}(T, w)$,
2. $\texttt{COST}_{max,D}(T, w) \leq \big(1 + \frac{1}{\epsilon}\big) \cdot (\lfloor \log n \rfloor + 1)$

*Proof.* Let $D^*$ be any optimal startegy for $T$. If $\texttt{COST}_{max,D}(T, w) \leq \lfloor \log n \rfloor + 1$, then the claim follows. Assume contrary. In such case let $T'$ be any non empty subtree of $T$ occuring as the candidate subtree after first $\lfloor \log n \rfloor + 1/\epsilon$ queries of some branch of the strategy. We build $D$ by altering $D^*$ from now on. At each next level of the decision tree a centroid of a current candidate subtree is scheduled to be queried. In such case each vertex belonging to $T'$ gains additional query time equal to at most $\log \lfloor \log n(T') \rfloor + 1 \leq \lfloor \log n \rfloor + 1$ and the depth of $D$ is bounded by $\texttt{COST}_{max,D}(T, w) \leq \big(1 + \frac{1}{\epsilon}\big) \cdot (\lfloor \log n \rfloor + 1)$. Additionally, the cost of $D$ is at most:

$$\begin{aligned}\texttt{COST}_{avg,D}(T, w) &\leq \sum_{v \in V(T)} w(v) \left(\epsilon \cdot |Q_{D^*}(T, v)| + |Q_{D^*}(T, v)|\right) \\ &\leq (1+\epsilon) \cdot \texttt{COST}_{avg,D^*}(T, w) = (1+\epsilon) \cdot \texttt{OPT}_{avg}(T, w)\end{aligned}$$

□

We assume that the input tree $T$ is rooted at an arbitrary vertex. If the response to a query contains $r(T)$ we say that such response is an *up* response and we say that it is an *down* response otherwise. Let $D$ be a decision tree for $T$. We say that a child of $q \in V(D)$ is a *left* child if it is associated with an up response to query at $q$. We say that, it is a *right* response otherwise. Note that any query in $D$ may have at most one left child.

To devise our dynamic program, we will need to use the following generalization of decision trees. An *extended decision tree* $D = V(V(D), E(D))$ for the tree $T$ is defined analogously as ordinary decision tree, however we allow $V(D) = V \cup U \cup B$, where $V \subseteq V(T)$, $U$ is a set of nodes in $V(D)$ labeled as *unassigned* and $B$ is a set of nodes in $V(D)$ label as *blocked*. We also require if $q(D) \in U \cup B$, then $q$ has no right children. The cost of such decision tree is defined the same as the cost of an ordinary decision tree. Note that, any decision tree is also an extended decision tree, and we can easily transform any extended decision tree to obtain an ordinary decision tree. To do so, simply delete every query $q \in U \cup B$. If $q \neq r(D)$ and $q$ has a left child, then: If $q$ was a left child of $p(q)$, hang the left child of $q$ as a left child of $p(q)$. Else if $q$ was a right child of $p(q)$, hang the left child of $q$ as a right child of $p(q)$.

We will also define a timeline $P$ to be an extended decision tree consisting of sequence of queries $\langle p_1, \ldots, p_k \rangle$, such that every query of $P$ is either blocked or unassigned. We will build our decision trees around timelines. Let $D$ be any extedned decision tree. Define the *left path* $P_D = \langle q_1, \ldots, q_h \rangle$ of $D$ as the sequence of queries in $D$, obtained by traversing $D$ starting from $r(D)$, and stepping to the left child until there is none. We will say that $D$ with a left path $P_D = \langle p_1, \ldots, p_k \rangle$ is *compatible* with a timeline $P = \langle q_1, \ldots, q_h \rangle$, such that $k \leq k$ if for every integer $1 \leq l \leq h$, if $q_k \in B$, then $p_k \in B$.

We will now introduce the subproblems which our dynamic programming solves. A problem $\texttt{OPT}(T_{v,i}, P)$ consist of finding an optimal extended decision tree for the tree $T_{v,i}$, which is compatible with $P$. Additionally, a global parameter $h$ is given which bounds the maximum height of the solution found by the algorithm and in consequence, the length of $P$. The algorithm will compute the solutions to subproblems in an bottom-up and left to right manner. If at any point there is no way to create an extended decision tree with given parameters we simply declare such instance *unfeasible*. The choice of the constant $h$ will ensure existence of at least one such solution. We will now show how to compute $\texttt{OPT}(T_{v,i}, P)$ efficiently. The Algorithm 1 consists of 3 cases:

1. $T_{v,0}$, in this case we greedily pick the smallest index $1 \leq k \leq |P|$, such that $p_k$ is unassigned. If there is no such index, we declare the subproblem unfeasible. In other case, the solution obtained by taking timeline $P$ and setting $p_k = v$. The cost of such solution is $w(v) \cdot k$.

2. $T_{v,1}$, let $u$ be the unique child of $v$ in $T_{v,1}$. We assume that we have already solved all the subproblems of $T_u$. We iterate through all possible choices of $1 \leq k \leq h$, such that $p_k$ is unassigned. If there are no such choices, we declare the subproblem unfeasible. If otherwise, for each such $k$, we create an auxiliary timeline $P'_k = \langle p'_1, \ldots, p'_h \rangle$, such that $p'_l = p_l$ for $l < k$, $p'_k$ is blocked and $p'_l$ is unassigned for $l > k$. We consider an optimal extended decision tree $D'_k$ for an instance $\mathcal{P}(T_u, P'_k)$. In order to create a new decision tree $D_k$, for each choice of $k$, we proceed as follows: Let $q'_k$ be the $k$-th vertex of the left path of $D'_k$. We set $q'_k = v$ Then, we take the left child of $q'_k$ in $D'$ and we rehang it as the right child of $q'_k$. The cost of each such extedned decision tree is $\texttt{OPT}(T_u, P'_k) + w(v) \cdot k$. We then return an optimal extended decision tree $D$, which minimizes the cost.

3. $T_{v,i}$ for $i > 1$, we assume that we have already solved all the subproblems of $T_{v,i-1}$ and $T_{c_i}$. Let $I$ be a set of indices of unassigned nodes of $P$, i.e. $I = \{l | p_l \text{ is unassigned}\}$. Consider any bipartition $(I_1, I_2)$ of $I$. We create a timeline $P_1 = \langle p_{1,1}, \ldots, p_{1,h} \rangle$ from timeline $P$, by blocking all of the nodes in $P$ whose indices do not belong to $I_1$. We now consider an extended decision tree $D_1$ for $\mathcal{P}(T_{v,i-1}, P_1)$, with a left path $\langle q_{1,1}, \ldots, q_{1,d_1} \rangle$. Let $k$ be the index of query to $v$, such that $q_{1,k} = v$. We construct $P_2 = \langle p_{2,1}, \ldots, p_{2,h} \rangle$ as follows: for any $1 \leq l \leq h$, we set $p_{2,l}$ to be unassigned if $l \in I_k^2$ or $l < k$ and we set $p_{2,l}$ to be blocked otherwise. Let $D_2$ be an optimal extended decision tree for $\mathcal{P}(T_{c_i}, P_2)$ and let $\langle q_{2,1}, \ldots, q_{2,d_2} \rangle$ be its left path. We proceed as follows. Firstly, we rehang the left child of $q_{2,k}$ in $D_2$ as its unique right child (by construction $q_{2,k}$ is blocked in $D_2$). Then, we build $D$ by aligning $D_1$ and $D_2$ by their left paths: For $1 \leq l \leq k$, if $p_l$ is blocked then $q_l$ is blocked. Else, if $p_{1,l}$ is unassigned, then $q_l = q_{1,l}$. Else, if $p_{2,l}$ is unassigned, then $q_l = q_{2,l}$. For $k < l$ we set $q_l = q_{1,l}$. Since the unassigned nodes above the $k$-th node of left paths of $D_1$ and $D_2$ have no conflicts and $D_2$ has no vertices in its left path beyond $q_{2,k}$, by construction we obtain a valid extended decision tree $D$. The cost of such solution is $\texttt{OPT}(T_{v,i-1}, P_1) + \texttt{OPT}(T_{c_i}, P_2)$. We then return an optimal extended decision tree $D$, which minimizes the cost. For a visual example, see 3.8.

**Algorithm 1:** The dynamic programming procedure finding $\texttt{OPT}(T_{v,i}, P)$.

**Procedure** $\texttt{DPTimelines}(T_{v,i}, w, P, h)$**:**

- **if** $i = 0$ **then**
  - **for** $1 \le k \le h$ **do**
    - **if** $p_k = \textit{unassigned}$ **then**
      - $p_k \leftarrow v$.
      - **return** $P$.
  - **return** $\emptyset$.
- $\mathcal{D} \leftarrow \emptyset$.
- **if** $i = 1$ **then**
  - **for** $1 \le k \le h$ **do**
    - **if** $p_k = \textit{unassigned}$ **then**
      - $P'_k \leftarrow P$.
      - $p'_k \leftarrow \textit{blocked}$.
      - **for** $k < l \le h$ **do**
        - $p'_k \leftarrow \textit{unassigned}$.
      - $D'_k \leftarrow \texttt{DPTimelines}(T_{c_1}, w, P'_k, h)$.
      - $q'_k \leftarrow v$.
      - Rehang the left child of $q'_k$ as its right child.
      - **for** $k < l \le h$ **do**
        - $q'_l \leftarrow p_l$.
      - $\mathcal{D} \leftarrow \mathcal{D} \cup \{D'_k\}$.
- **else**
  - $I \leftarrow \{l | p_l \text{ is unassigned}\}$.
  - **foreach** bipartition$(I_1, I_2)$of $I$ **do**
    - $P_1 \leftarrow P$.
    - **for** $1 \le k \le h$ **do**
      - **if** $k \notin I$ **then**
        - $p_{1,k} \leftarrow \textit{blocked}$
    - $D_1 \leftarrow \texttt{DPTimelines}(T_{v,i-1}, w, P_1, h)$.
    - $k \leftarrow l | q_{1,l} = v$.
    - $P_2 \leftarrow P$.
    - **for** $1 \le l \le h$ **do**
      - **if** $k \in I$ *or* $l > k$ **then**
        - $p_{1,k} \leftarrow \textit{unassigned}$.
      - **else**
        - $p_{1,k} \leftarrow \textit{blocked}$.
    - $D_2 \leftarrow \texttt{DPTimelines}(T_{c_i}, w, P_2, h)$.
    - Rehang left child of $q_{2,k}$ as its right child.
    - $D \leftarrow D_1$ and $D_2$ with their left paths aligned.
    - $\mathcal{D} \leftarrow \mathcal{D} \cup \{D\}$.
- **return** $\arg\min_{D \in \mathcal{D}} \{\texttt{COST}_D(T_{v,i})\}$.

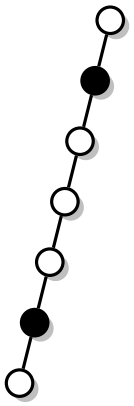

Figure 3.1

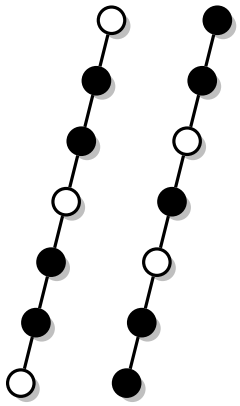

Figure 3.2

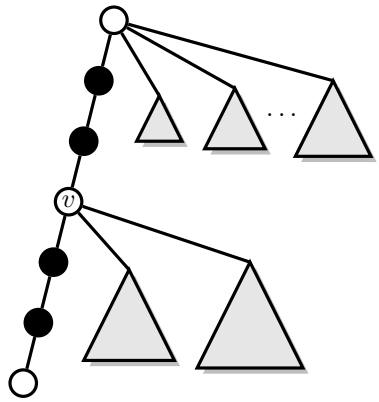


Figure 3.3

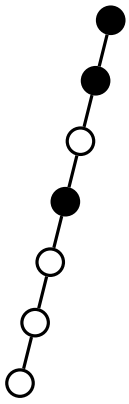

Figure 3.4

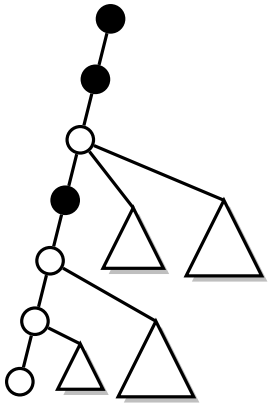

Figure 3.5

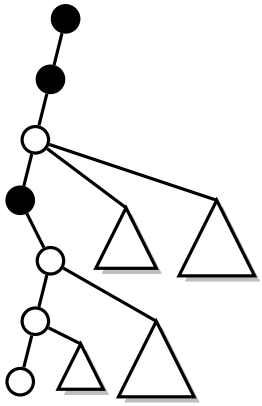

Figure 3.6

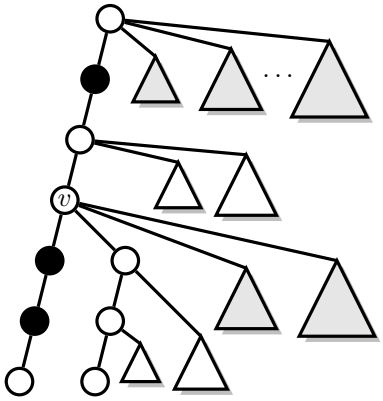


Figure 3.7

Figure 3.8: Basic steps of the case when $i > 1$. The black nodes are blocked and white are unassigned. Fig. 3.1: example timeline $P$. Fig. 3.2: timelines $P_1$ and $P_2$ induced by bipartition $(I_1, I_2)$ of $I$. Fig 3.3: decision tree $D_1$ compatible with timeline $P_1$. Fig 3.4: timeline $P_2$ after unblocking nodes with indices below $k$. Fig 3.5: a decision tree $D_2$ compatible with $P_2$. Fig 3.6: $D_2$ with left child of $q_k$ rehanged to right. Fig 3.7: $D_1$ and $D_2$ aligned by their left paths.

Let $P = \langle p_1, \ldots, p_h \rangle$, such that for every integer $1 \leq k \leq h$, $p_k \in U$. Let $h = \left(1 + \frac{1}{\epsilon}\right) \cdot (\lfloor \log n \rfloor + 1)$. Since, any extended decision tree of depth at most $h$ is compatible with $P$ by Lemma 3.3.2.2 we have that for $D$ calculated for $\texttt{OPT}(T, P)$, $\texttt{COST}_D(T, w) \leq (1 + \epsilon) \cdot \texttt{OPT}(T, w)$.

There are at most $O(n)$ subtrees $T_{v,i}$, $2^h$ different timelines and each subproblem requires $O\big(h + 2^h \cdot h\big) = O\big(2^h \cdot h\big)$ amount of computation, since there are at most $2^h$ bipartitions of unassigned vertices of any timeline and aligning two decision trees requires $O(h)$ time. Therefore, the running time of the procedure is bounded by $O\big(n \cdot 2^{2h} \cdot h\big) = O\big(n^{(2/\epsilon+3)} \cdot \log(n/\epsilon)\big)$ as required.

□

### 3.3.3 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 3.3.3.1.** *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. □

**Theorem 3.3.3.2.** *Fix $0 < \epsilon \leq n$. There exists a $(1+\epsilon)$-approximation algorithm for $T||V,w||\sum C_i$ running in $O\Big(n \cdot (n/\epsilon)^{2\cdot\log_{3/2}(2)} \cdot \log(n/\epsilon)\Big)$ time.*

*Proof.* To obtain the FPTAS we combine this bound with a standard rounding trick and the Algorithm 1. The algorithm 2 is as follows: Fix $\epsilon > 0$ and let $K = \frac{\epsilon \cdot w(T)}{n^2}$. For every $v \in V(T)$ we define $w'(v) = \left\lceil \frac{w(v)}{K} \right\rceil$. We set $h = \left\lceil \log_{3/2} w'(T) \right\rceil$ and initialize $P\langle p_1, \ldots, p_h \rangle$, such that for every $1 \leq k \leq h$, $p_h =$ *unassigned*. We then call $\texttt{DPTimelines}(T, w', P, h)$ and return the resulting decision tree $D'$.

**Algorithm 2:** The FPTAS for $T||V,w,||\sum C_i$

**Procedure** $\texttt{FPTAS}(T, w, \epsilon)$**:**
- $K \leftarrow \frac{\epsilon \cdot w(T)}{n^2}$.
- **foreach** $v \in V(T)$ **do**
  - $w'(v) \leftarrow \left\lceil \frac{w(v)}{K} \right\rceil$.
- $h \leftarrow \left\lceil \log_{3/2} w'(T) \right\rceil$.
- $P \leftarrow \langle p_1, \ldots, p_h \rangle$, such that for every $1 \leq k \leq h$, $p_h \leftarrow$ *unassigned*.
- $D' \leftarrow \texttt{DPTimelines}(T, w', P, h)$.
- **return** $D'$.

**Lemma 3.3.3.3.**

$$\texttt{COST}_{D'}(T, w) \leq (1+\epsilon) \cdot \texttt{OPT}(T, w)$$

*Proof.* By definition, for every $v \in V(T)$, we have $w'(v) \leq \frac{w(v)}{K} + 1$ and therefore $K \cdot w'(v) \leq w(v) + K$. Let $D^*$ be the optimal solotion for the $(T, w)$ instance. We have:

$$\begin{aligned}
\texttt{COST}_{D'}(T, w) &\leq K \cdot \texttt{COST}_{D'}(T, w') \leq K \cdot \texttt{COST}_{D^*}(T, w') \\
&\leq \texttt{COST}_{D^*}(T, w) + K \cdot \sum_{v \in V(T)} |Q_{D^*}(T, v)| \\
&\leq \texttt{COST}_{D^*}(T, w) + K \cdot n^2 = \texttt{COST}_{D^*}(T, w) + \epsilon \cdot w(T) \\
&\leq \texttt{COST}_{D^*}(T, w) + \epsilon \cdot \texttt{COST}_{D^*}(T, w) = (1+\epsilon) \cdot \texttt{OPT}_{D^*}(T, w)
\end{aligned}$$

where the first inequality is by definition of $w'$, the second inequality is by the optimality of $D'$ in $(T, w')$, the fourth inequality is using the fact that $\sum_{v \in V(T)} |Q_{D^*}(T, v)|$ is trivially upper bounded by $n^2$, the first equality is by definition of $K$, the last inequality is using the fact that

$\texttt{COST}_{D^*}(T, w)$ is trivially lower bounded by $w(T)$ and the last equality is by the optimality of $D^*$ in $(T, w)$. The claim follows. □

We have that $w'(T) = \sum_{v \in V(T)} \frac{w(v)}{K} \leq n^2/\epsilon + n = O\big(n^2/\epsilon\big)$. Hence, the running time of the procedure is bounded by $O\big(n \cdot 2^{2h} \cdot h\big) = O\Big(n \cdot w'(T)^{\log_{3/2}(2)} \cdot \log w'(T)\Big) = O\Big(n \cdot (n/\epsilon)^{2 \cdot \log_{3/2}(2)} \cdot \log(n/\epsilon)\Big)$ and the claim follows. □

## 3.4 Trees, worst case, non-uniform costs

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

### Vertex ranking

The *vertex ranking* of $T$ is a labeling of vertices $l : V \to \{1, 2, \ldots, \lceil \log n \rceil + 1\}$, which satisfies the following condition: for each pair of vertices $u, v \in V(T)$, whenever $l(u) = l(v)$, there exists $z \in \mathcal{P}_T(u, v)$ for which $l(z) > l(v)$. Such a labeling always exists and can be computed in linear time by means of dynamic programming [Sch89; OP06; MOW08]. For a visual example, see Figure 3.12

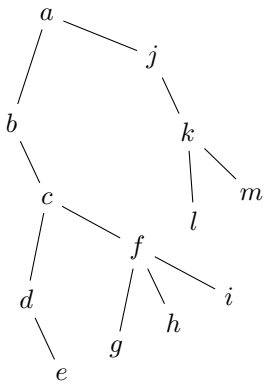


Figure 3.9

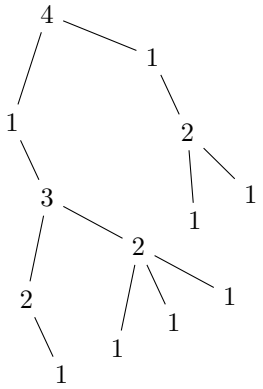


Figure 3.10

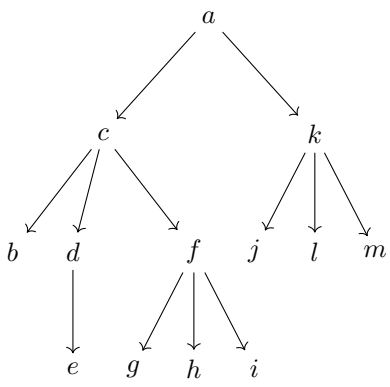


Figure 3.11

Figure 3.12: Sample input tree $T$ (Figure 3.9), vertex ranking labeling $l$ of $T$ (Figure 3.10) and a decision tree $D$ for $T$ built using $l$ (Figure 3.11).

Having a vertex ranking of $T$, one can easily obtain a decision tree for $T$ using the following procedure:

1. Let $z \in V(T)$ be the unique vertex, such that for every $v \in V(T)$, $l(z) \geq l(v)$.
2. Schedule a query to $z$ as the root of the decision tree $D$ for $T$.
3. For each $T' \in T - z$, build a decision tree $D_{T'}$ recursively and hang it below the query to $z$ in $D$.

When the input tree has uniform costs and the ranking uses the minimal number of labels, the decision tree built in this way is optimal and never uses more than $\lfloor \log n \rfloor + 1$ queries [OP06]. Let `RankingBasedDT` be the name of the latter procedure. We have the following corollary:

**Corollary 3.4.0.2.** *There exists an $O(n)$ time procedure* `RankingBasedDT` *that finds the optimal decision tree for the Tree Search Problem when all costs are uniform. Moreover, the depth of such a decision tree, i.e., the worst-case number of queries, is at most $\lfloor \log n \rfloor + 1$.*

### 3.4.1 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 3.4.1.1.** *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 3.4.1.2.** *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 at most $O(2^n)$ different subtrees of $T$ to be checked. Additionally, 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 necessarily returns the optimal decision tree for $T$ and the claim follows. □

We will denote the above procedure. Let $k = 2^{\lfloor \log \log n \rfloor + 2}$. The basic idea of the Algorithm 3 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 call $\texttt{Exact}(\mathcal{T}, c)$ to find the optimal solution in time $2^k k = \text{poly}(n)$.

If otherwise, to build a solution 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 3.4.1.3.** *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 3.18). 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 3.19). The algorithm builds an optimal decision tree $D_{\mathcal{Z}}$ for $\mathcal{T}_{\mathcal{Z}}$ by applying the `Exact` procedure for $(\mathcal{T}_{\mathcal{Z}}, c)$. Observe, that $D_{\mathcal{Z}}$ is a partial decision tree for $\mathcal{T}$, so we get that:

**Observation 3.4.1.4.** $\texttt{COST}_{D_{\mathcal{Z}}}(\mathcal{T}_{\mathcal{Z}}) = \texttt{COST}_{D_{\mathcal{Z}}}(\mathcal{T})$.

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 3.4.4.5).

**Algorithm 3:** Main recursive procedure ($k$ is a global parameter)

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

**Lemma 3.4.1.5.** *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 the elements of set $\mathcal{X}$. 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 every integer $1 \leq t \leq |\mathcal{X}|$: $|\mathcal{X}_t| = t$ and $\mathcal{X}_{|\mathcal{X}|} = \mathcal{X}$. For each $\mathcal{X}_t$, we will also construct a corresponding set $\mathcal{Y}_t$, eventually ensuring that $\mathcal{Y}_{|\mathcal{X}|} = \mathcal{Y}$. 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.

By construction, $\mathcal{X}_{|\mathcal{X}|} = \mathcal{X}$ and $\mathcal{Y}_{|\mathcal{H}|} = \mathcal{Y}$, so $|\mathcal{Y}| \leq 2 \cdot |\mathcal{H}| - 1 \leq 2k - 1$. As paths between vertices in $\mathcal{Y}$ form a tree when contracted, at most $2k - 2$ additional vertices are added while constructing $\mathcal{Z}$ (at most one per path). The lemma follows. □

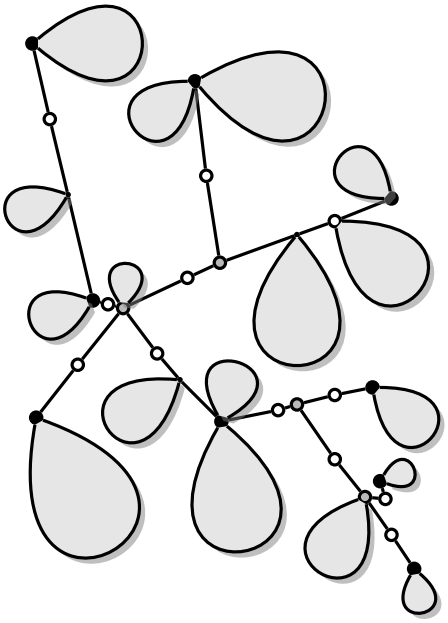

Figure 3.13: Example tree $\mathcal{T}$. Light grey regions represent light subtrees. Black vertices represent $\mathcal{X}$. Gray and black vertices represent $\mathcal{Y}$. White, gray and black vertices represent $\mathcal{Z}$. Lines represent paths of vertices between vertices of $\mathcal{Z}$.

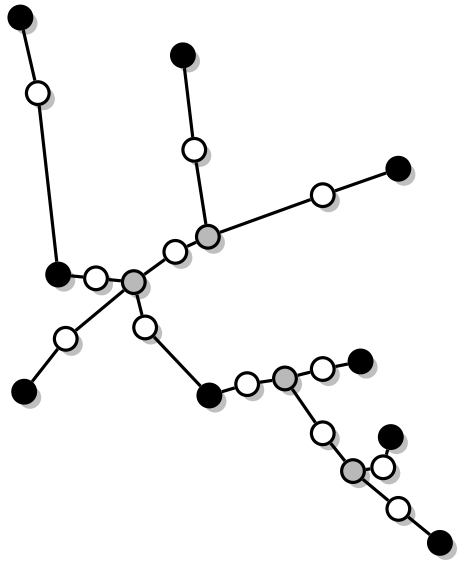

Figure 3.14: Auxiliary tree $\mathcal{T}_{\mathcal{Z}}$ built from vertices of set $\mathcal{Z}$.

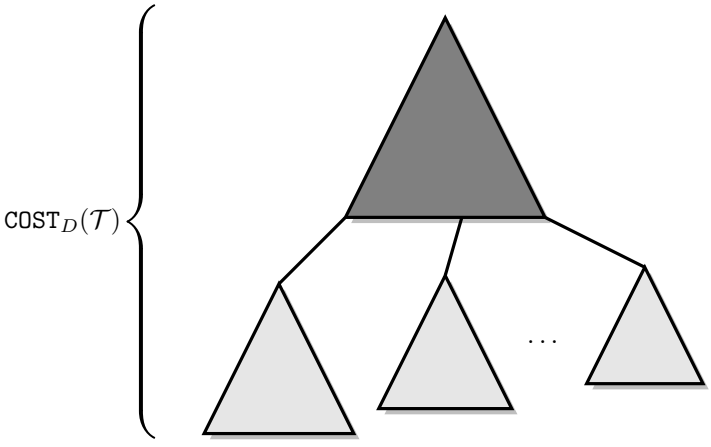


Figure 3.15: The structure of the decision tree $D$, built by the Algorithm 3. The dark gray subtree represents the decision tree $D_{\mathcal{Z}}$, obtained by calling the `Exact` procedure for $\mathcal{T}_{\mathcal{Z}}$ and $c$. Light gray subtrees represent decision trees $D_L$, built for each $H \in \mathcal{T} - \mathcal{Z}$, by recursively calling DecisionTree with $H$ and $c$.

**Lemma 3.4.1.6.** *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 3.4.0.1). The lemma follows. □

**Lemma 3.4.1.7.** *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:

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

where the first inequality is by construction, the second is by usage of Observation 3.4.1.4, Lemma 3.4.1.6 and Lemma 3.4.0.1 and the last inequality is due to the Lemma 3.4.1.3 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 3.4.1.5), 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.

### 3.4.2 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 3.4.2.1.** *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 3.4.2.2.** *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 while evaluating the cost of the resulting decision tree.

**Lemma 3.4.2.3.** *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:

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

where the first inequality is by construction, the second is by usage of Observation 3.4.1.4, Lemma 3.4.1.6 and Lemma 3.4.0.1 and the last inequality is due to the Lemma 3.4.1.3 and the induction hypothesis. □

□

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

The following algorithh is a simplified version of the QPTAS provided in [Der+17]. The core idea of the algorithm is the same, however, our solution uses the language of decision trees instead of the language of sequence assignments, which makes the algorithm more intuitive.

For the rest of the analysis, without loss of the generality, we will assume that $T$ is rooted in a vertex $v$ minimizing $c(v)$. We will also assume that all costs are normalized, so that $\max_{v \in V(T)}\{c(v)\} = 1$. If not, the costs are scaled by dividing them by $\max_{v \in V(T)}\{c(v)\}$. Note that this operation does not affect the optimality of a strategy or the quality of an approximation.

**Observation 3.4.3.1.** *Let $T$ be a tree such that $|V(T)| > 1$ and $c : V \to \mathbb{R}^+$ be a normalized weight function. Then, $1 \leq \mathit{OPT}(T) \leq \lfloor \log n \rfloor + 1$.*

*Proof.* The first inequality is due to the fact that there exists $v \in V(T)$, such that $c(v) = 1$ and for any decision tree $D$ we have $v \in Q_D(T, v)$. The second inequality is due to the fact that we can always locate the target using $\lfloor \log n \rfloor + 1$ queries [OP06]. □

**Rounding**

We will use the following rounding schem which will allow us to discretise the space of possible solutions to process it efficiently. Let $p \in \mathbb{N}$, and $c = a/pn$ for some $a \in \mathbb{N}$. Define:

$$c'(v) = \begin{cases} \lceil c(c) \rceil_c , & \text{if } c(v) > pc, \text{ in which case the vertex will be called } \textit{heavy}, \\ \lceil c(c) \rceil_{\frac{1}{pn}} , & \text{otherwise, in which case the vertex will be called } \textit{light}. \end{cases}$$

**Lemma 3.4.3.2.**

$$\mathit{OPT}(T, c') \leq \left(1 + \frac{2}{p}\right) \cdot \mathit{OPT}(T, c)$$

*Proof.* Let $D^*$ be an optimal strategy for $(T, c)$. By definition, we have that for every vertex $v \in V(T)$, $c'(v) \leq \left(1 + \frac{1}{p}\right) \cdot c(v) + \frac{1}{p}n$ and therefore:

$$\begin{aligned} \mathtt{OPT}(T, c') \leq \mathtt{COST}_{D^*}(T, c') &= \max_{v \in V(T)} \left\{ \sum_{q \in Q_{D^*}(T,v)} c'(q) \right\} \\ &\leq \max_{v \in V(T)} \left\{ \sum_{q \in Q_{D^*}(T,v)} \left( \left(1 + \frac{1}{p}\right) \cdot c(v) + \frac{1}{pn} \right) \right\} \\ &\leq \frac{1}{p} + \left(1 + \frac{1}{p}\right) \cdot \max_{v \in V(T)} \left\{ \sum_{q \in Q_{D^*}(T,v)} c(v) \right\} \leq \left(1 + \frac{2}{p}\right) \mathtt{OPT}(T, c) \end{aligned}$$

where in the third inequality we used the fact that for every $v \in V(T)$, $|Q_{D^*}(T, v)| \leq n$ and in the last inequality we used Observation 3.4.3.1. □

While calculating the decision tree, we will divide the time into boxes of duration $c$, which will be further subdivided into $a$ identical slots of length $\frac{1}{pn}$. Let $t_q$ denote the start of some query in a decision tree $D$. Note that the numbers $t_v$ provide a complete information about any decision tree, and are an equiavalent representation of any strategy. We will assume that for any heavy vertex

$v \in V(T)$, $t_v$ is an integer multiple of $c$ and for any light vertex $v \in V(T)$, $t_v$ is an integer multiple of $\frac{1}{pn}$. [2] We have the following lemma:

**Lemma 3.4.3.3.** *There exists a decision tree $D$ for $(T, c')$, such that $\texttt{COST}_D(T, c') \leq \left(1 + \frac{3}{p}\right) \cdot \texttt{OPT}(T, c')$ and for every vertex $v \in V(T)$ we have:*

1. *if $c(v) > pc$, then $t_v/c \in \mathbb{N}$ (every heavy query is aligned to a multiple of $c$),*
2. *if $c(v) \leq pc$, then $t_v pn \in \mathbb{N}$ (every light query is aligned to a multiple of $\frac{1}{pn}$).*

*Proof.* Let $D^*$ by any optimal decision tree for $(T, c')$. For any $v \in V(T)$, let $t^*_v$ be the start of query to $v$ in $D^*$ and let $t'_v = \left(1 + \frac{2}{p}\right) t^*_v$, thus construction a new decision tree $D'$. Since in this new decision tree $D'$, the ordering of vertices is exactly the same as in $D^*$, for any two consecutive queries $v, u$ in $D'$ we have:

$$t'_u - t'_v = \left(1 + \frac{2}{p}\right) \cdot (t_u - t_v) \geq \left(1 + \frac{2}{p}\right) \cdot c(v)$$

We now construct $D$ as follows: If $v \in V(T)$ is heavy, we assign $t_v = \lceil t'_v \rceil_c$ and $t_v = t'_v$ otherwise. For any two consecutive queries $v, u$ in $D$, such that $v$ is heavy we have:

$$t_u - \lceil t'_v \rceil_c > t_u - t_v - c \geq \left(1 + \frac{2}{p}\right) \cdot c(v) - c > w(v) + c > c'(v)$$

So we conclude that no two queries overlap. To obtain the second part of the claim, we round up the starting time of each query to a light vertex in $D$ to an integer multiple of $\frac{1}{pn}$. We have:

$$\begin{aligned}
\texttt{COST}_D(T, c') &\leq \max_{v \in V(T)} \left\{ \sum_{q \in Q_{D^*}(T,v)} \left( \left(1 + \frac{2}{p}\right) \cdot c'(v) + \frac{1}{pn} \right) \right\} \\
&\leq \frac{1}{p} + \left(1 + \frac{2}{p}\right) \cdot \max_{v \in V(T)} \left\{ \sum_{q \in Q_{D^*}(T,v)} c'(v) \right\} \leq \left(1 + \frac{3}{p}\right) \texttt{OPT}(T, c')
\end{aligned}$$

where in the third inequality we used the fact that for every $v \in V(T)$, $|Q_D(T, v)| \leq n$ and in the last inequality we used Observation 3.4.3.1. □

We will call a decision tree fulfilling above conditions *aligned*. In subsequent considerations, we will focus ourselves of finding such decision trees, whose properties will allow us to devise an efficient dynamic programming procedure finding an optimal, aligned decision tree.

### Heavy module contraction, up and down responses

Since, our decision tree is rooted, we can reasonably talk about up and down responses to a query. An *up* response to a query to $v$ in $T$ occurs when the connected component $\in T - v$, which

[2] Note that by doing so, we allow decision trees to contain idle time intervals, in which no queries are scheduled. However, if this occurs, after obtaining such decision tree, we simply delete the idle times, which results in a valid decision tree

is the reply happens to contain $r(T)$. If this is not the case, then such response is called a *down* response. As it will turns out, a repeating occurance of light queries with down responses will become problematic for our algorithm. To account for this issue we will use the following notions:

We will define a new measure of cost for aligned deision trees called the *aligned cost*. Let $D$ be any aligned strategy for $(t, c')$. For any query $q \in V(D)$ and a vertex $v \in V(T)$, let the contribution $\kappa_{T,c}(q, v)$ be defined as:

$$\kappa_{T,c}(q,v) = \begin{cases} 0, & \text{if } c(v) \leq pc \text{ and the response to query } q \text{ in } T, \text{ towards } v \text{ is down,} \\ c(q)\,, & \text{otherwise.} \end{cases}$$

Then, the *aligned cost* of $D$ is defined as:

$$\texttt{COST}'_D(T, c') = \max_{v \in V(T)} \left\{ \sum_{q \in Q_D(T,v)} \kappa_{T,c'}(q, v) \right\}.$$

Let $\texttt{OPT}'(T, c')$ denote the optimal aligned cost among all aligned decision trees for $(T, c')$. The above cost, serves as a way to "ignore" all od the costs of light queries with down responses. Since of course, the amount of such queries may be of order $O(n)$, the difference between $\texttt{COST}'_D(T, c')$ and $\texttt{COST}_D(T, c')$ may grow almost arbitrarily large. However, we will make sure that this does not happen to often, which will give us the desired bound on the cost of the solution.

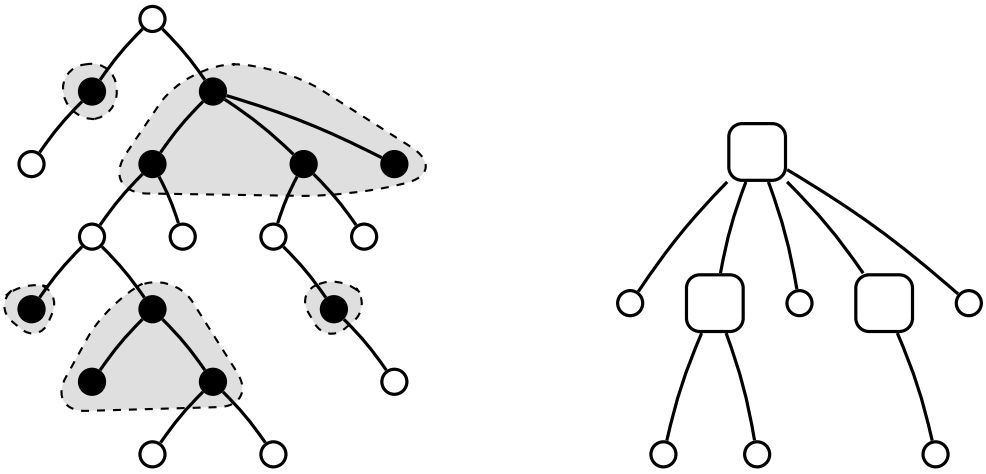

Figure 3.16: Example of contracting 5 heavy modules. Black vertices represent heavy vertices, white vertices represent light vertices and square vertices represent vertices which were a parent of at least one heavy module before contraction.

We define a *heavy module* as $H \subseteq V(T)$ such that: $T[H]$ is connected, every $v \in H$ is heavy, i. e., $c(v) \geq pc$ and $H$ is maximal - no vertex can be added to it without violating one of its properties. 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 the parent of $r(T\langle H \rangle)$ if it exists. For example see Figure 3.16.

**The main procedure**

We will use the following proposition about the existence of the `BuildDT` procedure:

**Proposition 3.4.3.4.** *Let $T$ be a tree, $D_C$ a decision tree for $T$ with all heavy groups contracted, $p \in \mathbb{N}$ be a constant, $c' \in \mathbb{R}_{>0}$ be the box size and $d \in \mathbb{N}$ be maximal depth. There exists a* `BuildDT` *procedure which:*

1. *Either determines that $cd \leq \texttt{OPT}'(T, c')$ and returns an aligned decision tree $D$, such that:*

$$\texttt{COST}_D(T, c') = \texttt{OPT}'(T, c') + \texttt{COST}_{D_C}(T, c')\,.$$

2. *Or determines that $cd > \texttt{OPT}'(T, c')$ and such decision tree does not exist.*
3. *The procedure runs in $(pn)^d$ time.*

The proof will be provided in the next section. We will now prove the Proposition 3.4.2.1.

**Algorithm 4:** The QPTAS for $T||V, c, w||C_{max}$.

**Procedure** `QPTAS`$(T, c, \epsilon)$**:**
- $p \leftarrow \lceil 24/\epsilon \rceil$, $c \leftarrow 0$, $D \leftarrow \emptyset$.
- $d \leftarrow p^2 \cdot (\lfloor \log(n) \rfloor + 1)$.
- **repeat**
  - $c \leftarrow c + \frac{1}{pn}$.
  - **foreach** $v \in V(T)$ **do**
    - **if** $c(v) > pc$ **then**
      - $c'(v) \leftarrow \lceil c(v) \rceil_c$.
    - **else**
      - $c'(v) \leftarrow \lceil c(v) \rceil_{\frac{1}{pn}}$.
  - $T_C \leftarrow T$ with all heavy modules contracted.
  - $D_C \leftarrow$ `RankingBasedDT`$(T_C)$.
  - $D \leftarrow$ `BuildDT`$(T, D_C, p, c, d)$.
- **until** $D \neq \emptyset$;
- **return** $D$.

The Algorithm 4 starts by picking $p = \left\lceil \frac{24}{\epsilon} \right\rceil$ and assigning $c = 0$ and $d = p^2 \cdot (\lfloor \log n \rfloor + 1)$. At each iteration of the repeat loop, the algorithm picks $c$ to be the next integer multiple of $\frac{1}{pn}$, performs the rounding operation introduced above, creates a decision tree $D_C$ for tree $T$ with all heavy modules contracted and tries to create decision tree $D$, by applying Proposition 3.4.3.4. Once the procedure returns a non-empty solution, the `QPTAS` procedure terminates and returns the constructed decision tree $D$. Let $c'$ be the value of $c$ for which $D$ was found. Since we know that

$c' \leq \frac{\texttt{OPT}'(T,c')}{d} = \frac{\texttt{OPT}'(T,c')}{p^2 \cdot (\lfloor \log n \rfloor + 1)}$, we have that:

$$\begin{aligned}
\texttt{COST}_D(T,c') &\leq \texttt{OPT}'(T,c') + pc' \cdot (\lfloor \log n \rfloor + 1) \leq \texttt{OPT}'(T,c') + p \cdot \left(p^2 \cdot (\lfloor \log n \rfloor + 1)\right) \cdot \frac{\texttt{OPT}'(T,c')}{\lfloor \log n \rfloor + 1} \\
&\leq \left(1 + \frac{1}{p}\right) \cdot \texttt{OPT}'(T,c') \leq \left(1 + \frac{1}{p}\right) \cdot \left(1 + \frac{2}{p}\right) \cdot \left(1 + \frac{3}{p}\right) \cdot \texttt{OPT}(T,c) \\
&\leq \left(1 + \frac{24}{p}\right) \cdot \texttt{OPT}(T,c) = \left(1 + \frac{24}{\lceil \frac{24}{\epsilon} \rceil}\right) \cdot \texttt{OPT}(T,c) \leq (1+\epsilon) \cdot \texttt{OPT}(T,c)
\end{aligned}$$

where the second inequality is by applying Corollary 3.4.0.2 and the fourth inequality is by Lemma 3.4.3.2 and Lemma 3.4.3.3.

We can assume that $c = \text{poly}(n)$, since beyond that the problem can be solved to optimality in $O(2^n n)$ time. Therefore the running time of the procedure is bounded by:

$$n^{O(d)} = n^{O(p^2 \log n)} = n^{O(\log n/\epsilon^2)}.$$

**Dynamic programming procedure for fixed box size**

### 3.4.4 An $O(\log \log n)$-approximation algorithm parametrized by the $k$-up-modularity of the cost function

**$k$-up-modularity**

The main algorithmic difficulty in dealing with the problem arises when the values of the cost function vary drastically. We would like to measure this "irregularity" in a quantifiable way. To do so, we introduce the notion of $k$-up-modularity.

Let $t \in \mathbb{R}_{\geq 0}$. We define a *heavy module* with respect to $t$ as $H \subseteq V(T)$ such that, $T[H]$ is connected, for every $v \in H$, $c(v) > t$, and $H$ is maximal - no vertex can be added to it without violating one of its properties. We then define the *heavy module set* with respect to $t$ in $(T,c)$ as:

$$\mathcal{H}_{T,c}(t) = \{H \subseteq V(T) \mid H \text{ is a heavy module w.r.t. } t\},$$

Let $k(T,c,t) = |\mathcal{H}_{T,c}(t)|$ be the size of the heavy module set, and finally let $k(T,c) = \max_{s \in \mathbb{R}_{\geq 0}}\{k(T,c,t)\}$. 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 $T$. To illustrate the notion of $k$-up-modularity, see Figure 3.17.

The concept of $k$-up-modularity is a direct generalization of the notion of up-monotonicity of the cost function introduced in [DW22] (as monotonicity) and in [DW24] (as up-monotonicity). Let $z = \arg\max_{v \in V(T)}\{c(v)\}$. A function $c$ is *up-monotonic* in $T$ if for every $v, u \in V(T)$, whenever $v$ lies on the path between $z$ and $u$, we have $c(v) \geq c(u)$.

It is easy to see that 1-up-modularity is equivalent to up-monotonicity. Observe that if $c$ is up-monotonic in $T$, then for every $t \in \mathbb{R}_{\geq 0}$, $T[V(T) - \{v \in V(T) \mid c(v) \leq t\}]$ 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 the path between $r$ and $u$ such that $c(v) < c(u)$. Otherwise, for any $t \in (c(v), c(u))$, $v$ does not belong to any heavy module, but $u$ and $r$ do. Since $v$ lies between them, $|\mathcal{H}_{T,c}(t)| > 1$, a contradiction.

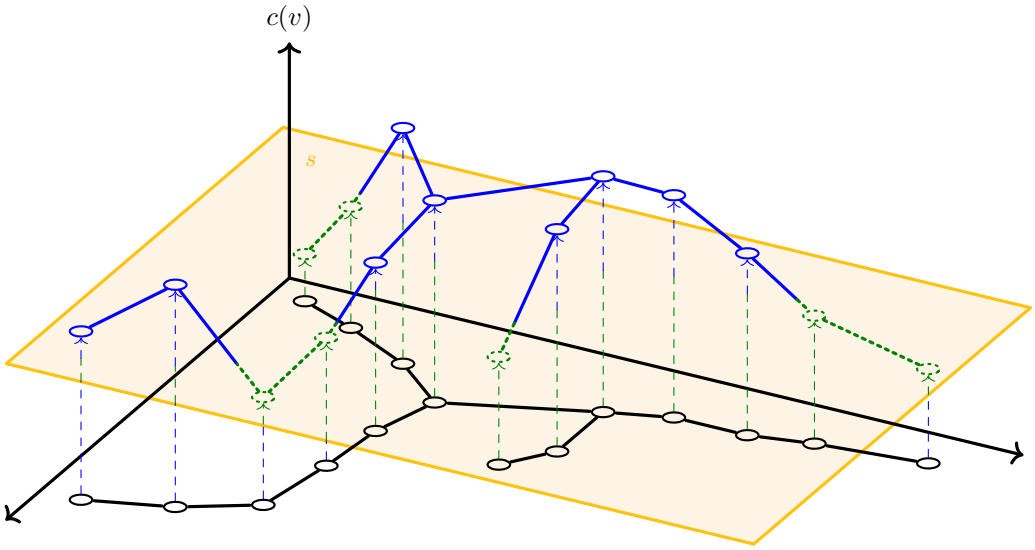


Figure 3.17: A visual depiction of a tree $T$ with a 3-up-modular cost function $c$. Each vertex of a tree is mapped onto some value of $c$. The yellow plane represents some threshold value $t \in \mathbb{R}_{\geq 0}$ (in this particular example $k(T, c, t) = 2$). The (two) blue subtrees represent members of $\mathcal{H}_{T,c}(t)$.

### The parametrized $O(\log \log n)$-approx. solution

#### Cost levels

The main idea of the algorithm is to partition vertices into intervals called *cost levels* and process them in a top-down manner. At each level of the recursion, the algorithm schedules all necessary queries to vertices belonging to the given cost level. The rest of the decision tree is then built recursively. We consider the following intervals[3]:

1. Firstly, an interval $(0, 1/\log n]$.
2. Then, each subsequent 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, which partitions the interval $(0, 1]$:

$$(0, 1/\log n], (1/\log n, 2/\log n], (2/\log n, 4/\log n], \ldots, \left(2^{\lceil \log \log n \rceil - 1}/\log n, 1\right].$$

We will ensure that when we call our procedure with parameters $\left(T, c, \left(2^{\lceil \log \log n \rceil - 1}/\log n, 1\right]\right)$, the returned decision tree will be 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 a query to it) is *heavy* with respect to the interval $\mathcal{I} = (a, b]$ when $c(v) > a$. Otherwise, i.e., if $c(v) \leq a$, the vertex (and the query to it) is *light* with respect to $\mathcal{I}$.

[3]We present the intervals in the ascending order in which a complete solution for each of them is obtained. However, since the procedure is recursive, the order in which the recursive calls are made is reverse.

Note that each heavy vertex belongs to some heavy module. Whenever clear from the context, we will omit the phrase "with respect to" and simply call the vertices and queries heavy and light.

**The main recursive procedure**

We are ready to present the main recursive procedure. To avoid ambiguity, let $\mathcal{T}$ be the subtree of $T$ processed at some level of the recursion. Alongside $\mathcal{T}$ and a cost function $c$, the algorithm takes as input an interval $(a, b]$, such that for every $v \in V(\mathcal{T})$, $c(v) \leq b$ and $2a \geq b$. The basic steps of the Algorithm 5 are as follows:

1. If every vertex is heavy, return a decision tree built by calling the `RankingBasedDT` procedure for $\mathcal{T}$.
2. Otherwise, find a set $\mathcal{Z}$, such that each connected component of $\mathcal{T}' \in \mathcal{T} - \mathcal{Z}$ contains at most one heavy module.
3. Create an auxiliary tree $T_{\mathcal{Z}}$ using the vertices of $\mathcal{Z}$ and create a new decision tree $D_{\mathcal{Z}}$ for $T_{\mathcal{Z}}$, using the QPTAS from [Der+17].
4. For each $\mathcal{T}' \in \mathcal{T} - \mathcal{Z}$, build a decision tree $D_H$, by calling the `RankingBasedDT` procedure for $\mathcal{T}'\langle H \rangle$. Then, hang $D_H$ below the last query to $v \in N_{\mathcal{T}}(\mathcal{T}')$ in $D_{\mathcal{Z}}$.
5. For each $L \in \mathcal{T}' - H$, build a decision tree recursively. Then, hang $D_L$ below the last query to a vertex $v \in N_{\mathcal{T}'}(L)$ in $D_{\mathcal{Z}}$.
6. Return the resulting decision tree $D$.

Before providing a detailed description and analysis of the above procedure, we first present some basic properties necessary for the subsequent considerations. In particular, we will make use of the following well-known lemma [CLS16]:

**Lemma 3.4.4.1.** *Let $T'$ be a subtree of $T$. Then, $\texttt{OPT}(T') \leq \texttt{OPT}(T)$.*

For the rest of the analysis, fix $\mathcal{H} = \mathcal{H}_{\mathcal{T},c}(a)$ to be the set of heavy modules in $\mathcal{T}$. We have the following observations, which will be useful in the description and analysis of the algorithm:

**Observation 3.4.4.2.** *Let $\mathcal{H}$ be the set of heavy modules in $T$. Then, $|\mathcal{H}| \leq k(T)$.*

*Proof.* Since $\mathcal{H} = \mathcal{H}_{\mathcal{T},c}(a)$, we have $|\mathcal{H}| = k(\mathcal{T}, c, a) \leq \max_{t \in \mathbb{R}_{\geq 0}} k(T, c, t) = k(T, c)$. □

**Observation 3.4.4.3.** *Let $T'$ be a subtree of $T$. Then, $k(T') \leq k(T)$.*

*Proof.* Fix any $t \in \mathbb{R}_{\geq 0}$ and let $H \in \mathcal{H}_{T,c}(t)$. We show that each such $H$ contributes at most 1 to $k(T', c, t)$. If $H \cap V(T') = \emptyset$, then $H$ contributes 0. Otherwise, $H \cap V(T')$ forms a connected subtree of $T'$, and thus contributes at most 1. The lemma follows. □

**Observation 3.4.4.4.** *Let $T'$ be a subtree of a tree $T$ and let $D'$ be a decision tree for $T'$. Then, $D'$ is a partial decision tree for $T$.*

**Observation 3.4.4.5.** *Let $T'$ be a subtree of a tree $T$ and let $D$ be a partial decision tree for $T$ having no queries to vertices of $T'$, but containing at least one query to the vertices of $N_T(V(T'))$. Let $Q$ denote the set of all such queries to vertices of $N_T(V(T'))$ in $D$. Then, $D\langle Q \rangle$ forms a path in $D$.*

**Algorithm 5:** The main recursive procedure

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

- **if** $b \leq 1/\log n$ **or** for every $v \in V(\mathcal{T}), c(v) > a$ ; `// Every` $v \in \mathcal{T}$ `is heavy`
  **then**
  - **return** `RankingBasedDT`$(\mathcal{T})$ ; `// Apply Corollary 3.4.0.2`
- **else**
  - $\mathcal{X} \leftarrow \emptyset$.
  - **foreach** $H \in \mathcal{H}_{\mathcal{T},c}(a)$ **do**
    - Pick arbitrary $v \in H$.
    - $\mathcal{X} \leftarrow \mathcal{X} \cup \{v\}$.
  - $\mathcal{Z} \leftarrow \mathcal{Y} \leftarrow \mathcal{X} \cup \left\{v \in V(\mathcal{T}\langle\mathcal{X}\rangle) \mid \deg_{\mathcal{T}\langle\mathcal{X}\rangle}(v) \geq 3\right\}$.
  - **foreach** $u, v \in \mathcal{Y}$ with $\mathcal{P}_{\mathcal{T}}(u, v) \neq \emptyset$ and $\mathcal{P}_{\mathcal{T}}(u, v) \cap \mathcal{Y} = \emptyset$ **do**
    - $\mathcal{Z} \leftarrow \mathcal{Z} \cup \left\{\arg\min_{z \in \mathcal{P}_{\mathcal{T}}(u,v)}\{c(z)\}\right\}$. ; `// Lightest vertex on path`
  - $\mathcal{T}_{\mathcal{Z}} \leftarrow (\mathcal{Z}, \{uv \mid \mathcal{P}_{\mathcal{T}}(u, v) \cap \mathcal{Z} = \emptyset\})$ ; `// Build auxiliary tree`
  - $D \leftarrow D_{\mathcal{Z}} \leftarrow$ `QPTAS`$(\mathcal{T}_{\mathcal{Z}}, c, \epsilon = 1)$ ; `// Apply Theorem 3.4.2.1`
  - **foreach** $\mathcal{T}' \in \mathcal{T} - \mathcal{Z}$ **do**
    - $H \leftarrow$ the unique heavy module in $\mathcal{T}'$.
    - $D_H \leftarrow$ `RankingBasedDT`$(\mathcal{T}'\langle H\rangle)$ ; `// Apply Corollary 3.4.0.2`
    - Hang $D_H$ in $D$ below the last query to $v \in N_{\mathcal{T}}(\mathcal{T}')$ ; `// By Obs. 3.4.4.5`
    - **foreach** $L \in \mathcal{T}' - H$ **do**
      - $D_L \leftarrow$ `CreateDecisionTree`$(L, c, (a/2, a])$.
      - Hang $D_L$ in $D$ below the last query to $v \in N_{\mathcal{T}'}(L)$ ; `// By Obs. 3.4.4.5`
  - **return** $D$.

*Proof.* Let $q$ be any query in $D$. There are two cases:

1. $q \in V(T - V(T') - N_T(V(T')))$. Then, for every $x \in N_T(V(T'))$ being the target, $x$ belongs to the same connected component of $T - q$. Thus, no matter which vertex is the target, the answer is always the same. Therefore, $q$ has at most one child $u$ in $D$, such that $V(D_u) \cap Q \neq \emptyset$.

2. $q \in N_T(V(T'))$. After a query to $q$, the situation is as in the first case, except when $x = q$. Then, the response is $x$ itself, so no further queries are needed, and again $q$ has at most one child $u$ in $D$, such that $V(D_u) \cap Q \neq \emptyset$.

Since each $q \in Q$ has at most one child $u$ in $D$, with $D_u \cup Q \neq \emptyset$, $D\langle Q \rangle$ forms a path and the claim follows. □

**Base of the recursion**

We begin the description of our algorithm with the recursion base, which occurs whenever $b \leq 1/\log n$ or for every $v \in V(\mathcal{T})$, $c(v) > a$, i.e., every vertex is heavy. In such a situation, a solution is built by disregarding the costs of vertices and constructing a decision tree using the vertex ranking of $\mathcal{T}$.

**Lemma 3.4.4.6.** *Let $D$ be a decision tree built, by calling* `RankingBasedDT` $(\mathcal{T})$ *in line 5 of the* `CreateDecisionTree` *procedure. Then,*

$$\mathit{COST}_D(\mathcal{T}) \leq 2 \cdot \mathit{OPT}(T)\,.$$

*Proof.* There are two cases:

1. If $b \leq \frac{1}{\log n}$, then:

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

   where the first inequality is due to Corollary 3.4.0.2, the fourth inequality follows from Observation 3.4.3.1, and the last inequality is due to Observation 3.4.4.1.

2. If for every $v \in V(\mathcal{T})$, we have $c(v) > a$, then, define $c'(v) = a$ for all $v \in V(\mathcal{T})$ (note that any value could be chosen here, since we treat each query as unitary). As $2c'(v) = 2a \geq b \geq c(v)$, we obtain $2 \cdot \texttt{COST}_D(\mathcal{T}, c') \geq \texttt{COST}_D(\mathcal{T}, c)$. Additionally, using the fact that $c'(v) \leq c(v)$, we have $\texttt{OPT}(\mathcal{T}, c') \leq \texttt{OPT}(\mathcal{T}, c)$. Therefore:

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

   where the equality is due to Corollary 3.4.0.2 and the last inequality is due to Observation 3.4.4.1. The lemma follows.

□

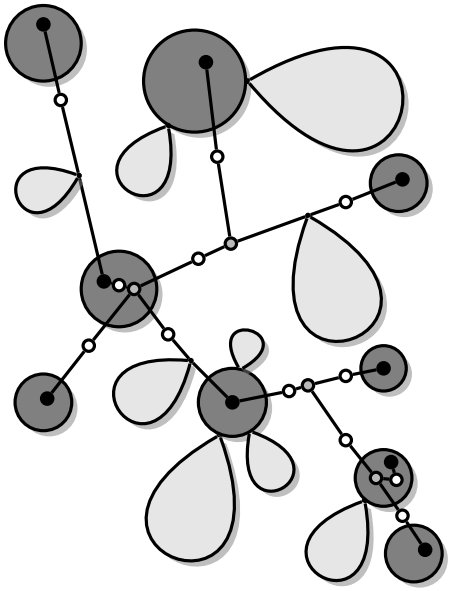

Figure 3.18: Example tree $\mathcal{T}$. Dark grey circles represent heavy modules. Light grey regions represent light subtrees. Black vertices represent $\mathcal{X}$. Gray and black vertices represent $\mathcal{Y}$. White, gray and black vertices represent $\mathcal{Z}$. Lines represent paths of vertices between vertices of $\mathcal{Z}$.

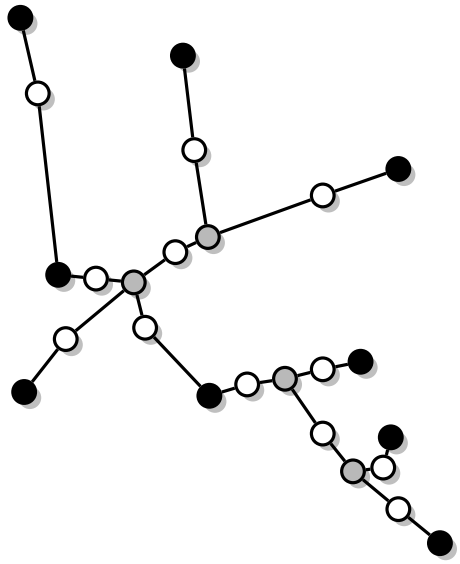

Figure 3.19: Auxiliary tree $\mathcal{T}_{\mathcal{Z}}$ built from vertices of set $\mathcal{Z}$.

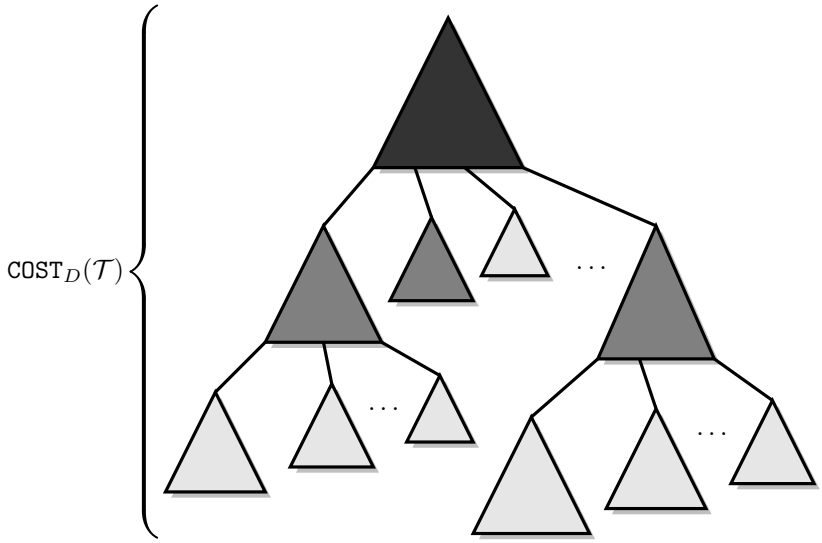


Figure 3.20: The structure of the decision tree $D_{\mathcal{T}}$ built by the Algorithm 5. The dark gray subtree represents the decision tree $D_{\mathcal{Z}}$, obtained by calling the QPTAS for $\mathcal{T}_{\mathcal{Z}}$, $c$ and $\epsilon = 1$. Gray subtrees represent decision trees $D_H$, each built for a unique heavy module $H \subseteq V(\mathcal{T}')$ of every $\mathcal{T}' \in \mathcal{T} - \mathcal{Z}$, by calling the RankingBasedDT procedure for $\mathcal{T}'\langle H \rangle$. Light gray subtrees represent decision trees $D_L$, built for each $L \in \mathcal{T}' - H$, by recursively calling CreateDecisionTree with $L$, $c$ and $(a/2, a]$.

**Construction of the Auxiliary Tree**

To obtain the solution for the non-base case of our algorithm, we first construct the so-called *auxiliary tree*. To do so, we begin by defining a set $\mathcal{X} \subseteq V(\mathcal{T})$. For every heavy module $H \in \mathcal{H}$, we pick an arbitrary $v \in H$ and add it to $\mathcal{X}$. We also define a set $\mathcal{Y} = \mathcal{X} \cup \left\{ v \in V(\mathcal{T}\langle\mathcal{X}\rangle) \,|\, \deg_{\mathcal{T}\langle\mathcal{X}\rangle}(v) \geq 3 \right\}$, by extending $\mathcal{X}$ to contain all vertices with degree at least 3 in $T\langle\mathcal{X}\rangle$. Furthermore, we define a set $\mathcal{Z} \subseteq V(\mathcal{T})$ consisting of the 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 lightest vertex between them, i. e., $v_{u,v} = \arg\min_{z \in \mathcal{P}_{\mathcal{T}}(u,v)}\{c(z)\}$. To see an example of construction of the sets $\mathcal{X}, \mathcal{Y}, \mathcal{Z}$, see Figure 3.18.

We then create the auxiliary tree $\mathcal{T}_{\mathcal{Z}} = (\mathcal{Z}, \{uv | \mathcal{P}_{\mathcal{T}}(u, v) \cap \mathcal{Z} = \emptyset\})$ (for an example, see Figure 3.19). Our algorithm starts by building a decision tree $D_{\mathcal{Z}}$ for $\mathcal{T}_{\mathcal{Z}}$, by taking $\epsilon = 1$ and applying the QPTAS from Theorem 3.4.2.1. Observe that, since $D_{\mathcal{Z}}$ is a partial decision tree for $\mathcal{T}$ and corresponding vertices in $\mathcal{T}$ and $\mathcal{T}_{\mathcal{Z}}$ have the same costs, we have that:

**Observation 3.4.4.7.** $\texttt{COST}_{D_{\mathcal{Z}}}(\mathcal{T}_{\mathcal{Z}}) = \texttt{COST}_{D_{\mathcal{Z}}}(\mathcal{T})$.

Let $D = D_{\mathcal{Z}}$. For each connected component $\mathcal{T}' \in \mathcal{T} - \mathcal{Z}$, we build a new decision tree as follows: By the construction of $\mathcal{Z}$, all heavy vertices in $V(\mathcal{T}')$ form a single heavy module $H \subseteq V(\mathcal{T}')$. We create a new decision tree $D_H$ for $\mathcal{T}'\langle H\rangle$, by calling the `RankingBasedDT` procedure with argument $\mathcal{T}'\langle H\rangle$ and we hang $D_H$ in $D$ below the unique last query to a vertex in $N_{\mathcal{T}}(\mathcal{T}')$ (which is possible due to Observation 3.4.4.5). As, by Observation 3.4.4.4, $D_H$ is a partial decision tree for $\mathcal{T}'$, it follows that $D$ is also a partial decision tree for $\mathcal{T}$.

Now notice that for each $L \in \mathcal{T}' - H$, there is no $v \in V(L)$, such that $c(v) > a$. This allows us to create a decision tree $D_L$ recursively, by calling the `CreateDecisionTree` procedure with arguments $L$, $c$ and $(a/2, a]$. Next, we hang $D_L$ in $D$ below the unique last query to a vertex in $N_{\mathcal{T}'}(L)$ (again, using Observation 3.4.4.5). Since after all such operations, every vertex $v \in V(\mathcal{T})$ also belongs to $D$, we obtain a valid decision tree $D$ for $\mathcal{T}$. To see example structure of such solution, see Figure 3.20.

**Analysis of the algorithm**

**Lemma 3.4.4.8.** *Let $\mathcal{T}_{\mathcal{Z}}$ be the auxiliary tree. Then, $|V(\mathcal{T}_{\mathcal{Z}})| \leq 4k - 3$.*

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

Let $\mathcal{H}_1 = \emptyset$, $\mathcal{Y}_1 = \emptyset$. Pick any heavy module $H \subseteq V(\mathcal{T})$ and add it to $\mathcal{H}_1$. Add the unique vertex $v$, such that $v \in H \cap \mathcal{X}$ to $\mathcal{Y}_1$, so that $|\mathcal{Y}_1| = 1$. Assume by induction that for some $h \geq 1$, $|\mathcal{Y}_h| \leq 2h - 1$. Two heavy modules $H_1, H_2 \subseteq V(\mathcal{T})$ will be called *neighbors* if for every $H_3 \subseteq V(\mathcal{T})$ with $H_3 \neq H_1, H_2$, we have $\mathcal{P}_{\mathcal{T}}(H_1, H_2) \cap H_3 = \emptyset$. Pick $H \in \mathcal{H}$, such that $H \notin \mathcal{H}_h$ to be a heavy module that is a neighbor of some member of $\mathcal{H}_h$. We define $\mathcal{H}_{h+1} = \mathcal{H}_h \cup \{H\}$. Let $z$ be the unique vertex, such that $v \in H \cap \mathcal{X}$, and let $\mathcal{Y}_{h+1} = \mathcal{Y}_h \cup \{z\}$. Define $\mathcal{T}_{h+1} = \mathcal{T}\langle\{v \in \mathcal{Y}_{h+1} | \mathcal{P}_{\mathcal{T}}(v, z) \cap \mathcal{Y}_{h+1} = \emptyset\}\rangle$. Note that $\mathcal{T}_{h+1}$ is a spider (a tree with at most one vertex of degree above 2). Add to $\mathcal{Y}_{h+1}$ the unique vertex $v \in V(\mathcal{T}_{h+1})$, such that $\deg_{\mathcal{T}_{h+1}}(v) \geq 3$, if it exists. Clearly, $|\mathcal{Y}_{h+1}| \leq 2h + 1$, completing the induction.

By construction, $\mathcal{H}_{|\mathcal{H}|} = \mathcal{H}$ and $\mathcal{Y}_{|\mathcal{H}|} = \mathcal{Y}$, so $|\mathcal{Y}| \leq 2 \cdot |\mathcal{H}| - 1 \leq 2k - 1$ where the last inequality is by Observation 3.4.4.2. As paths between vertices in $\mathcal{Y}$ form a tree when contracted, at most

$2k-2$ additional vertices are added while constructing $\mathcal{Z}$ (at most one per path). The lemma follows. □

**Lemma 3.4.4.9.** *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 decision tree for $\mathcal{T}\langle\mathcal{Z}\rangle$. We build a new decision tree $D'_{\mathcal{Z}}$ for $\mathcal{T}_{\mathcal{Z}}$ by transforming $D^*$ as follows:

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 $q \in V(D^*)$ be the first query to a vertex among $\mathcal{P}_{\mathcal{T}}(u,v)$. Recall that we picked $v_{u,v} = \arg\min_{z \in \mathcal{P}_{\mathcal{T}}(u,v)}\{c(z)\}$, so $c(v_{u,v}) \leq c(q)$. We replace $q$ in $D^*$ with the query to $v_{u,v}$ and delete all queries to vertices in $\mathcal{P}_{\mathcal{T}}(u,v) - v_{u,v}$. By construction, $D'_{\mathcal{Z}}$ is a valid decision tree for $\mathcal{T}_{\mathcal{Z}}$, and by choosing $v_{u,v}$ to minimize $c$, we did not increase the cost, so we have that:

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

Therefore, we have:

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

where the first inequality is due to the definition of optimality and the last inequality follows by Lemma 3.4.4.1. □

**Lemma 3.4.4.10.** *Let $H$ be the unique heavy module of $\mathcal{T}' \in \mathcal{T} - \mathcal{Z}$. Then, the decision tree $D_H$ is of cost at most:*

$$\mathit{COST}_{D_H}(\mathcal{T}'\langle H\rangle) \leq 2 \cdot \mathit{OPT}(\mathcal{T})\,.$$

*Proof.* For every $v \in H$ let $c'(v) = a$. We have $2c'(v) \geq bc'(v)/a = b \geq c(v)$ so we get that $2 \cdot \mathtt{COST}_{D_H}(\mathcal{T}'\langle H\rangle, c') \geq \mathtt{COST}_{D_H}(\mathcal{T}'\langle H\rangle, c)$. Additionally, using the fact that $c'(v) \leq c(v)$ we have that $\mathtt{OPT}(\mathcal{T}'\langle H\rangle, c') \leq \mathtt{OPT}(\mathcal{T}'\langle H\rangle, c)$. Hence:

$$\begin{aligned}\mathtt{COST}_{D_H}(\mathcal{T}'\langle H\rangle, c) &\leq 2 \cdot \mathtt{COST}_{D_H}(\mathcal{T}'\langle H\rangle, c') = 2 \cdot \mathtt{OPT}(\mathcal{T}'\langle H\rangle, c') \\ &\leq 2 \cdot \mathtt{OPT}(\mathcal{T}'\langle H\rangle, c) \leq 2 \cdot \mathtt{OPT}(\mathcal{T}, c)\end{aligned}$$

where the equality is by the Corollary 3.4.0.2 and the last inequality is due to the fact that $\mathcal{T}'\langle H\rangle$ is a subtree of $\mathcal{T}'$, which is a subtree of $\mathcal{T}$ (Lemma 3.4.4.1). □

**The main result**

Let $d$ be the remaining depth of the recursion call performed in Line 5 of the algorithm, i.e., the number of recursive steps from the current call to the base case (for the base case, this value is equal to $d = 0$). We show that at each level of the recursion we pay $O(\mathtt{OPT}(T))$, so the approximation ratio of the algorithm is bounded by $O(d)$:

**Lemma 3.4.4.11.** $\mathit{COST}_D(\mathcal{T}) \leq (4d+2) \cdot \mathit{OPT}(T)$.

*Proof.* Let $Q_D(\mathcal{T}, x)$ be the sequence of queries performed in order to find $x \in V(\mathcal{T})$. By construction of the Algorithm 5, $Q_D(\mathcal{T}, x)$ consists of at most three distinct subsequences of queries (see Figure 3.20):

1. Firstly, there is a sequence of queries belonging to $Q_{D_{\mathcal{Z}}}(\mathcal{T}_{\mathcal{Z}}, x)$.
2. If $x \notin \mathcal{Z}$, then, there is a sequence of queries belonging to $Q_{D_H}(\mathcal{T}'\langle H\rangle, x)$ for a unique heavy group $H \subseteq V(\mathcal{T}')$ of $\mathcal{T}' \in \mathcal{T} - \mathcal{Z}$, such that $x \in \mathcal{T}'$.
3. At last, if $x \notin H$, there is a sequence of queries belonging to $Q_{D_L}(L, x)$ for $L \in \mathcal{T}' - H$, such that $x \in V(L)$.

Note that it sometimes may happen that some of the above sequences are empty.

We prove by induction that $\texttt{COST}_D(\mathcal{T}) \leq (4d+2) \cdot \texttt{OPT}(T)$. When $d = 0$ (the base case), the induction hypothesis is true, due to the Lemma 3.4.4.6. For $d > 0$, assume by induction that the cost of the decision tree built for each $L$, is at most $\texttt{COST}_{D_L}(L) \leq (4(d-1)+2) \cdot \texttt{OPT}(T)$. We have:

$$\begin{aligned}
\texttt{COST}_D(\mathcal{T}) &\leq \texttt{COST}_{D_{\mathcal{Z}}}(\mathcal{T}) + \max_{\mathcal{T}' \in \mathcal{T} - \mathcal{Z}} \left\{ \texttt{COST}_{D_H}(\mathcal{T}'\langle H\rangle) + \max_{L \in \mathcal{T}' - H} \{\texttt{COST}_{D_L}(L)\} \right\} \\
&\leq \texttt{COST}_{D_{\mathcal{Z}}}(\mathcal{T}_{\mathcal{Z}}) + \max_{\mathcal{T}' \in \mathcal{T} - \mathcal{Z}} \{2 \cdot \texttt{OPT}(\mathcal{T}) + (4(d-1)+2) \cdot \texttt{OPT}(T)\} \\
&\leq 2 \cdot \texttt{OPT}(\mathcal{T}_{\mathcal{Z}}) + 2 \cdot \texttt{OPT}(T) + (4(d-1)+2) \cdot \texttt{OPT}(T) \\
&\leq 2 \cdot \texttt{OPT}(\mathcal{T}) + 4d \cdot \texttt{OPT}(T) = (4d+2) \cdot \texttt{OPT}(T)
\end{aligned}$$

where the first inequality is due to the construction of the decision tree returned by the Algorithm 5, the second inequality is by Observation 3.4.4.7, Observation 3.4.4.10 and by the induction hypothesis, the third inequality is due to Theorem 3.4.2.1 and using the fact that $\mathcal{T}$ is a subtree of $T$ (Lemma 3.4.4.1) and the last inequality is due to Lemma 3.4.4.9.

□

We are now ready to prove our main theorem:

**Theorem 3.4.4.12.** *There exists an $O(\log\log n)$-approximation algorithm for the Tree Search Problem running in $k^{O(\log k)} \cdot poly(n)$ time.*

*Proof.* Let $D = \texttt{CreateDecisionTree}\left(T, \left(2^{\lceil \log\log n \rceil - 1}/\log n, 1\right]\right)$. Since there are at most $\lceil \log\log n \rceil + 1$ intervals processed, the depth of the recursion is bounded by $d \leq \lceil \log\log n \rceil \leq \log\log n + 1$. Hence, using Lemma 3.4.4.11 we get that:

$$\texttt{COST}_D(T) \leq (4 \cdot \log\log n + 6) \cdot \texttt{OPT}(T) = O(\log\log n \cdot \texttt{OPT}(T)) .$$

By Observation 3.4.4.3, for every subtree $\mathcal{T}$ of $T$, processed at some level of the recursion, we have $k(\mathcal{T}) \leq k(T)$. Using Lemma 3.4.4.8, at each such level the call to the QPTAS from Theorem 3.4.2.1 (line 5 of the CreateDecisionTree) runs in time bounded by:

$$k(\mathcal{T})^{O(\log(4 \cdot k(\mathcal{T})))} = k(T)^{O(\log k(T))} .$$

Since $d = O(\text{poly}(n))$ and all other computation can be performed in polynomial time, the overall running time is bounded by $k^{O(\log k)} \cdot \text{poly}(n)$, as required. □

## 3.5 Non-uniform weights and costs, average case

In this section we will be concerned with the following variant of the problem:

**$G||V$**

**Input:** Graph $G$, a query cost function $c : V \to \mathbb{N}$ and a weight function $w : V \to \mathbb{N}$.
**Output:** A decision tree $D$, minimizing the weighted average search cost:

$$c_G(D) = \sum_{x \in V(G)} w(x) \cdot \sum_{q \in Q(D,x)} c(q)\,.$$

We introduce the following reinterpretation of the latter cost function, for each node $v \in D$, let $G_{D,v}$ be the subgraph of $G$ in which $v$ is queried when using $D$. Then, the contribution of $v$ to the total cost is $w(G_{D,v}) \cdot c(v)$, and therefore we obtain the following simple lemma:

**Lemma 3.5.0.1.**

$$c_G(D) = \sum_{v \in V(G)} w(G_{D,v}) \cdot c(v)\,.$$

### Cuts and separators

To obtain a tight lower bound on the cost of our solution, we establish a connection between the $G||V, c, w||\sum C_i$ and the following vertex separator problems. We define the *Weighted $\alpha$-Separator Problem* as follows:

**Weighted $\alpha$-Separator Problem**

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

We also define the Min-Ratio Vertex Cut Problem as follows:

**Min-Ratio Vertex Cut Problem**

**Input:** Graph $G = (V(G), E(G))$, the cost function $c : V \to \mathbb{N}$ and the weight function $w : V \to \mathbb{N}$.
**Output:** A partition $(A, S, B)$ of $V(G)$ called *vertex-cut*, such that there are no $u \in A$ and $v \in B$ for which $uv \in E(G)$, minimizing the ratio:

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

**Levels of OPT and basic bounds**

We begin with additional notation. For any graph $G$ and decision tree $D$, denote by $\mathcal{R}_D(G) = \{V(G_{D,v}) : v \in V(G)\}$ the family of all candidate subsets of $D$ in $G$.

Let $D^*$ be an arbitrary decision tree for the $G||V, c, w||\sum C_i$ such that $\texttt{COST}_{D^*}(G) = \texttt{OPT}(G)$. We denote by $\mathcal{L}_k^*$ the subfamily of $\mathcal{R}_{D^*}(G)$ consisting of all maximal elements $H$ of $\mathcal{R}_{D^*}(G)$ with $w(H) \leq k$, that is, if some superset $H'$ of $H$ belongs to $\mathcal{R}_{D^*}(G)$, then $w(H') > k$. We call such a set the $k$-th *level* of $\texttt{OPT}(G)$. Let $S_k^* = V(G) - \mathcal{L}_k^*$. These are the vertices belonging to the separator at level $\mathcal{L}_k^*$.

Notice that $S_k^*$ forms a Weighted $w(G)/k$-separator of $G$. Furthermore, for any $H_1, H_2 \in \mathcal{R}_D(G)$, we have $H_1 \cup H_2 \neq \emptyset$ if and only if $H_1 \subseteq H_2$ or $H_2 \subseteq H_1$, so $\mathcal{R}_D(G)$ is laminar. Therefore, for any $k_1 \neq k_2$, we have $\mathcal{L}_{k_1}^* \cap \mathcal{L}_{k_2}^* = \emptyset$.

**Lemma 3.5.0.2.**

$$\texttt{OPT}(G) = \sum_{k=0}^{w(G)-1} c(S_k^*)\,.$$

*Proof.* Consider any vertex $v$. For every $0 \leq k < w(G_{D^*,v})$, $v \notin \bigcup_{H \in \mathcal{L}_k^*} H$, so $v \in S_k^*$ and the contribution of $v$ to the cost is $w(G_{D^*,v}) \cdot c(v)$:

$$\sum_{k=0}^{w(G)-1} c(S_k^*) = \sum_{v \in V(G)} \sum_{k=0}^{w(G_{D^*,v})-1} c(v) = \texttt{OPT}(G)$$

where the second equality is by Lemma 3.5.0.1. □

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

**Lemma 3.5.0.3.**

$$2 \cdot \texttt{OPT}(G) = 2 \cdot \sum_{k=0}^{w(T)-1} c(S_k^*) \geq \sum_{k=0}^{w(T)} c\Big(S^*_{\lfloor k/2 \rfloor}\Big)\,.$$

We also have the following upper bound:

**Lemma 3.5.0.4.** *Let $\mathcal{G}$ be any subgraph of $G$ and $0 \leq \beta \leq 1$. Then:*

$$\beta \cdot w(\mathcal{G}) \cdot c\Big(S^*_{\lfloor w(\mathcal{G})/2 \rfloor} \cap \mathcal{G}\Big) \leq \sum_{k=(1-\beta)w(\mathcal{G})+1}^{w(\mathcal{G})} c\Big(S^*_{\lfloor k/2 \rfloor} \cap \mathcal{G}\Big)\,.$$

*Proof.* The inequality is due to the fact that as $k$ decreases, more vertices belong to the separator. □

### 3.5.1 A $(4+\epsilon)$-approximation for $T||V, c, w||\sum C_j$

In this section, we present a $(4+\epsilon)$-approximation algorithm for the case where the input graph is a tree. To achieve this, we establish a connection between searching in trees and the Weighted $\alpha$-Separator Problem. This connection provides a lower-bounding scheme for our recursive algorithm,

which at each level of recursion, constructs a decision tree using the $\alpha$-separator obtained by the following procedure:

**Theorem 3.5.1.1.** *Let $S$ be an optimal weighted $\alpha$-separator for $(T, c, w, \alpha)$. For any $\delta > 0$ there exists an algorithm , which returns a separator $S'$, such that:*

1. $c(S') \leq c(S)$.
2. $w(H) \leq \frac{(1+\delta)\cdot w(T)}{\alpha}$ *for every* $H \in T - S'$.
3. *The algorithm runs in* $O\left(n^3/\delta^2\right)$ *time.*

*Proof.* We devise a dynamic programming procedure similar to the one in [BMN13] and combine it with a rounding trick to obtain a bi-criteria FPTAS. Note that the authors considered only the case in which all weights are uniform. However, we generalize their algorithm to arbitrary integer weights and introduce an additional case that was previously lacking[4].

**Theorem 3.5.1.2.** *Let $T$ be a tree. There exists an optimal algorithm for the Weighted $\alpha$-Separator Problem running in* $O\left(n \cdot \left(w(T)/\alpha\right)^2\right)$ *time.*

*Proof.* Assume that the input tree is rooted at an arbitrary vertex $r(T)$. Let $k = \lfloor w(T)/\alpha \rfloor$. We want to find a separator $S$ such that for every $H \in T - S$, $w(H) \leq k$. Let $C_v$ denote the cost of the optimal separator $S_v$ in $T_v$ with this property. Define $C_v^{in}$ as the cost of the optimal separator for $T_v$, under the condition that $v \in S_v$. We immediately have:

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

Assume that $v \notin S_v$. Let $H_v \in T_v - S_v$ be the component containing $v$. For every integer $0 \leq w \leq k$, let $C_v^{out}(w)$ be the cost of the optimal separator for $T_v$, such that $v \notin S_v$ and $w(H_v) = w$. Then:

$$C_v = \min\left\{C_v^{in}, \min_{0 \leq w \leq k} C_v^{out}(w)\right\}.$$

For any vertex $v \in V(T)$ and any integer $1 \leq i \leq \deg^+_{T,v}$, let $S_{v,i}$ be the optimal separator for $T_{v,i}$ and $H_{v,i} \in T_{v,i} - S_{v,i}$ be the component containing $v$. For any integer $0 \leq w \leq k$, let $C_{v,i}^{out}(w)$ be the cost of an optimal separator for $T_{v,i}$, such that $v \notin S_{v,i}$ and $w(H_{v,i}) = w$. Then

$$C_v^{out}(w) = C^{out}_{v,\deg^+_{T,v}}(w).$$

For $i = 1$ we have:

$$C_{v,1}^{out}(w) = \begin{cases} \infty, & \text{if } w < w(v), \\ \min\left\{C_{c_1}^{in}, C_{c_1}^{out}(0)\right\}, & \text{if } w = w(v), \\ C_{c_1}^{out}(w - w(v)), & \text{if } w > w(v). \end{cases}$$

[4]Probably due to an oversight.

For $i > 1$:

$$C^{out}_{v,i}(w) = \min\left\{ C^{out}_{v,i-1}(w) + C^{in}_{c_i}, \min_{0 \le j \le w} \left\{ C^{out}_{v,i-1}(w-j) + C^{out}_{c_i}(j) \right\} \right\}.$$

In the above, the first term of the outer minimum corresponds to the case $c_i \in S_{v,i}$, so $H_{v,i} = H_{v,i-1}$. The second term considers the alternative, checking all possible partitions of the weight between $H_{v,i-1}$ and $H_{c_i}$.

These relationships suffice to compute $C_{r(T)}$, the cost of the optimal separator $S$ for $T$. Computation is performed in a bottom-up, left-to-right manner, starting from the leaves. For a leaf $v$, we have $C^{in}_v = c(v)$ and:

$$C^{out}_v(w) = \begin{cases} 0, & \text{if } w = w(v) \le k, \\ \infty, & \text{otherwise.} \end{cases}$$

Since each of the $C^{in}_v$ subproblems requires $O\left(\deg^+_{T,v}\right)$ computational steps we get that they require $O(n)$ running time. As there are $O(n \cdot k) = O(n \cdot w(T)/\alpha)$ remaining subproblems and each requires at most $O(k) = O(w(T)/\alpha)$ computational steps, the running time is $O\left(n \cdot (w(T)/\alpha)^2\right)$. □

Note that, the running time of the above procedure depends on $w(T)$ which may not be polynomial. To alleviate this difficulty, we slightly relax the condition on the size of components in $T - S$ using a controlled parameter $\delta$. Based on this relaxation, we show how to construct a bicriteria FPTAS for the problem. Let $\delta > 0$ be any fixed constant and let  be the dynamic programming procedure from Theorem 3.5.1.2. The algorithm is as follows:

**Algorithm 6:** The bicriteria FPTAS for the Weighted $\alpha$-separator Problem

**Procedure** $(T, c, w, \alpha, \delta)$**:**
- $K \leftarrow \frac{\delta \cdot w(T)}{n \cdot \alpha}$.
- **foreach** $v \in V(T)$ **do**
  - $w'(v) \leftarrow \left\lfloor \frac{w(v)}{K} \right\rfloor$.
- $\alpha' \leftarrow \frac{\alpha \cdot K \cdot w'(T)}{w(T)}$.
- $S' \leftarrow (T, c, w', \alpha')$.
- **return** $S'$.

**Lemma 3.5.1.3.** *Let $S$ be the optimal separator for the $(T, c, w, \alpha)$ instance. We have that $c(S') \le c(S)$.*

*Proof.* We prove that $S$ is a valid separator for the $(T, c, w', \alpha')$ instance, so that $c(S') \le c(S)$. To simplify the analysis, we will define the auxiliary instance: For every $v \in V(T)$, let $w''(v) = K \cdot \left\lfloor \frac{w(v)}{K} \right\rfloor$. Additionally, let $\alpha'' = \frac{\alpha \cdot w''(T)}{w(T)}$.

In this new instance, for $v \in V(T)$ we have $w''(v) \le w(v)$, so for every $H \in T - S$,

$$w''(H) \le w(H) \le w(T)/\alpha = w''(T)/\alpha''$$

where the second inequality is by the definition of the $\alpha$-separator and the equality is by the definition of $\alpha''$.

We conclude that $S$ is an $\alpha''$-separator for the auxiliary instance $(T, c, w'', \alpha'')$. Now notice that the $(T, c, w', \alpha')$ instance has all of its weights scaled by a constant value of $K$, relatively to $(T, c, w'', \alpha'')$ and $\alpha' = \alpha''$. As multiplying weights by a constant does not influence the validity of a solution, $S$ is an $\alpha'$-separator for $(T, w', c, \alpha')$ and the claim follows. □

**Lemma 3.5.1.4.** *For every $H \in T - S'$, we have that $w(H) \leq \frac{(1+\delta)\cdot w(T)}{\alpha}$.*

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

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

where the second inequality is due to the fact that $S'$ is a $\alpha'$-separator for $(T, c, w', \alpha')$ instance and the first equality is by the definition of $\alpha'$ and $K$. □

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

**How to search in trees**

Below, we show how to use the  procedure to construct a solution for $T||V, c, w||\sum C_i$. At each level of the recursion, the algorithm greedily finds an (almost) optimal weighted $\alpha$-separator of $T$, denoted $S_T$, and then builds an arbitrary decision tree $D_T$ using the vertices in $S_T$ (which can be done in $O\big(n^2\big)$ time).

Next, for each $H \in T - S_T$, the procedure is called recursively, and each resulting decision tree $D_H$ is attached below the appropriate query in $D_T$. The resulting decision tree is then returned by the procedure.

**Theorem 3.5.1.5.** *For any $\epsilon > 0$, there exists $(4+\epsilon)$-approximation algorithm for $T||V, c, w||\sum C_i$ running in time $O\big(n^4/\epsilon^2\big)$.*

*Proof.* The procedure is as follows:

**Algorithm 7:** The $(4+\epsilon)$-approximation algorithm for $T||V, c, w||\sum C_i$

**Procedure** `DecisionTree`$(T, c, w, \epsilon)$**:**
- $S_T \leftarrow$ `SeparatorFPTAS`$\Big(T, c, w, \alpha = 2, \delta = \frac{\epsilon}{4+\epsilon}\Big)$.
- $D_T \leftarrow$ arbitrary partial decision tree for $T$, built from vertices of $S_T$.
- **foreach** $H \in T - S_T$ **do**
  - $D_H \leftarrow$ `DecisionTree`$(H, c, w, \epsilon)$.
  - Hang $D_H$ in $D_T$ below the last query to $v \in N_T(H)$.
- **return** $D_T$.

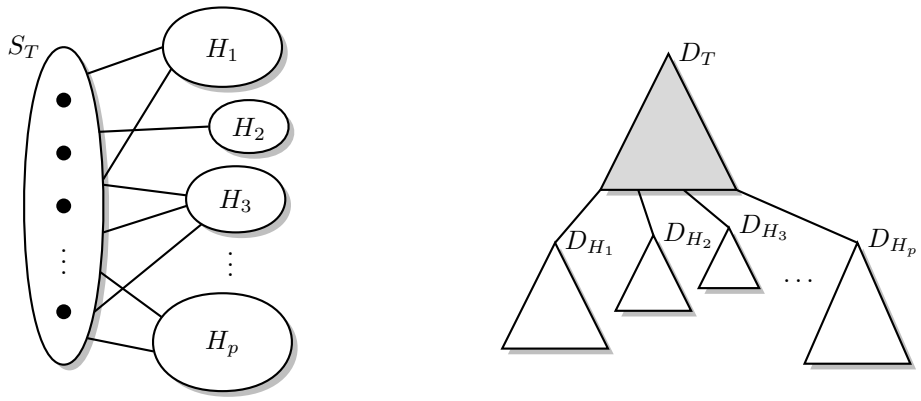


Figure 3.21: The separator $S_T$ produced by the algorithm and the structure of the decision tree built using $S_T$.

Let $\mathcal{T}$ be a subtree of $T$ for which the procedure was called and let $S^*_{\mathcal{T}} = S^*_{\lfloor w(\mathcal{T})/2 \rfloor} \cap \mathcal{T}$. By Theorem 3.5.1.1, we have that $c(S_{\mathcal{T}}) \leq c(S^*_{\mathcal{T}})$. Using $\beta = \frac{1-\delta}{2}$ and applying Lemma 3.5.0.4 we have that the contribution of the decision tree $D_{\mathcal{T}}$ is bounded by:

$$w(\mathcal{T}) \cdot c(S_{\mathcal{T}}) \leq w(\mathcal{T}) \cdot c(S^*_{\mathcal{T}}) \leq \frac{2}{1-\delta} \cdot \sum_{k=\frac{1+\delta}{2} \cdot w(\mathcal{T})+1}^{w(\mathcal{T})} c\Big(S^*_{\lfloor k/2 \rfloor} \cap \mathcal{T}\Big).$$

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

**Lemma 3.5.1.6.**

$$\sum_{\mathcal{T}} \sum_{k=\frac{1+\delta}{2} \cdot w(\mathcal{T})+1}^{w(\mathcal{T})} c\Big(S^*_{\lfloor k/2 \rfloor} \cap \mathcal{T}\Big) \leq \sum_{k=0}^{w(T)} c\Big(S^*_{\lfloor k/2 \rfloor}\Big).$$

*Proof.* Fix a value of $\mathcal{T}$ and $k$. Their contribution to the cost is $c\Big(S^*_{\lfloor k/2 \rfloor} \cap \mathcal{T}\Big)$. Consider which candidate subtrees contribute such a term. As $S_{\mathcal{T}}$ is a weighted $\frac{2}{1+\delta}$-separator, we have that $\mathcal{T}$ is the minimal candidate subtree, such that $w(\mathcal{T}) \geq k \geq \frac{(1+\delta) \cdot w(\mathcal{T})}{2} + 1 > w(H)$, for every $H \in \mathcal{T} - S_{\mathcal{T}}$. This means that if for every $H \in \mathcal{T} - S_{\mathcal{T}}$, $w(H) < k$, then $\mathcal{T}$ contributes such a term. Since for all $H_1, H_2 \in \mathcal{T} - S_{\mathcal{T}}$ we have that $H_1 \cap H_2 = \emptyset$, $\Big(S^*_{\lfloor k/2 \rfloor} \cap H_1\Big) \cup \Big(S^*_{\lfloor k/2 \rfloor} \cap H_2\Big) = \emptyset$, the claim follows by summing over all values of $k$. □

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

procedure. Using the fact that by definition $\frac{4}{1-\delta} = 4+\epsilon$, we have:

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

where the third inequality is due to Lemma 3.5.1.6 and the last inequality is by Lemma 3.5.0.3.

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. □

### 3.5.2 An $O\big(\sqrt{\log n}\big)$-approximation for $G||V,c,w||\sum C_j$

To construct decision trees for general graphs, we exploit a connection to a different problem, namely the Min-Ratio Vertex Cut Problem. Let $\alpha_{c,w}(G)$ denote the optimal value of such a vertex cut. We invoke the following result given in [FHL05]:

**Theorem 3.5.2.1.** *Given a graph $G = (V(G), E(G))$, the cost function $c : V \to \mathbb{N}$ and the weight function $w : V \to \mathbb{N}$, there exists a polynomial-time algorithm, which computes a partition $(A, S, B)$, such that:*

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

Combining the latter procedure, with the following result, yields an $O\big(\sqrt{\log n}\big)$-approximation algorithm for the $G||V,c,w||\sum C_i$:

**Theorem 3.5.2.2.** *Let $f_n$ be the approximation ratio of any polynomial time algorithm for the Min-Ratio Vertex Cut Problem. Then, there exists an $O(f_n)$-approximation algorithm for the GSP, running in polynomial time.*

*Proof.* Let be the procedure which achieves the $f_n$-approximation ratio for the Min-Ratio Vertex Cut Problem. The algorithm is as follows:

**Algorithm 8:** The $f_n$-approximation algorithm for $G||V,c,w||\sum C_i$.

**Procedure** `DecisionTree`$(G,c,w)$**:**
- $A_G, S_G, B_G \leftarrow (G,c,w)$.
- $D_G \leftarrow$ arbitrary partial decision tree for $G$, built from vertices of $S_G$.
- **foreach** $H \in G - S_G$ **do**
  - $D_H \leftarrow$ `DecisionTree`$(H,c,w)$.
  - Hang $D_H$ in $D_G$ below the last query to $v \in N_G(H)$.
- **return** $D_G$.

Let $\mathcal{G}$ be any subgraph of $G$, for which the procedure was called and let $S^*_{\mathcal{G}} = S^*_{\lfloor w(\mathcal{G})/2 \rfloor} \cap \mathcal{G}$.

**Lemma 3.5.2.3.** *Let $\mathcal{H} = \mathcal{G} - S^*_{\mathcal{G}}$ and let $\lambda = 6+2\sqrt{5}$ be the unique, positive solution of the equation $\frac{1}{4} - \frac{1}{2\sqrt{\lambda}} = \frac{1}{\lambda}$. Then, we can partition $\mathcal{H}$ into two sets, $\mathcal{A}$ and $\mathcal{B}$ such that for $A = \bigcup_{H\in\mathcal{A}} V(H)$*

*and* $B = \bigcup_{H \in \mathcal{B}} V(H)$*, we have:*

$$w\big(A \cup S^*_{\mathcal{G}}\big) \cdot w\big(B \cup S^*_{\mathcal{G}}\big) \geq w(\mathcal{G})^2 / \lambda.$$

*Proof.* There are two cases:

1. $w\big(S^*_{\mathcal{G}}\big) \geq w(\mathcal{G}) / \sqrt{\lambda}$. In this case we take arbitrary partition $\mathcal{A}, \mathcal{B}$ of $\mathcal{H}$. We have:

$$w\big(A \cup S^*_{\mathcal{G}}\big) \cdot w\big(B \cup S^*_{\mathcal{G}}\big) \geq w\big(S^*_{\mathcal{G}}\big)^2 \geq w(\mathcal{G})^2 / \lambda.$$

2. $w\big(S^*_{\mathcal{G}}\big) \leq w(\mathcal{G}) / \sqrt{\lambda}$. For any choice of the partition $\mathcal{A}, \mathcal{B}$ of $\mathcal{H}$, we have $\frac{w(A \cup B)}{w(\mathcal{G})} \geq 1 - \frac{1}{\sqrt{\lambda}}$. We pick $\mathcal{A}, \mathcal{B}$ to be a partition of $\mathcal{H}$, such that $w(A) \geq w(B) \geq \left(\frac{1}{2} - \frac{1}{\sqrt{\lambda}}\right) \cdot w(\mathcal{G})$ (this is always possible as $\frac{1}{2} - \frac{1}{\sqrt{\lambda}} > 0$ and for each $H \in \mathcal{H}$, $w(H) \leq w(\mathcal{G})/2$). We have:

$$\begin{aligned} w\big(A \cup S^*_{\mathcal{G}}\big) \cdot w\big(B \cup S^*_{\mathcal{G}}\big) &\geq w(A) \cdot w(B) \\ &\geq \left(\left(1 - 1/\sqrt{\lambda}\right) \cdot w(\mathcal{G}) - w(B)\right) \cdot w(B) \\ &\geq w(\mathcal{G})^2/2 \cdot \left(1/2 - 1/\sqrt{\lambda}\right) = w(\mathcal{G})^2/\lambda \end{aligned}$$

   where the third inequality is by using the fact that the concave function $f(w(B)) = w(B) \cdot \left(\left(1 - \frac{1}{\sqrt{\lambda}}\right) \cdot w(\mathcal{G}) - w(B)\right)$ reaches its minimum in the interval $\left[\left(\frac{1}{2} - \frac{1}{\sqrt{\lambda}}\right) \cdot w(\mathcal{G}), w(\mathcal{G})/2\right]$ when $w(B) = \left(\frac{1}{2} - \frac{1}{\sqrt{\lambda}}\right) \cdot w(\mathcal{G})$.

□

Using the fact, that the partition $\big(A, S^*_{\mathcal{G}}, B\big)$ in the above lemma is a vertex cut of $\mathcal{G}$, we have the following upper bound on the optimal value of the min-ratio-vertex cut of $\mathcal{G}$, $\alpha_{c,w}(\mathcal{G})$:

$$\alpha_{c,w}(\mathcal{G}) \leq \alpha_{c,w}\big(A, S^*_{\mathcal{G}}, B\big) = \frac{c\big(S^*_{\mathcal{G}}\big)}{w\big(A \cup S^*_{\mathcal{G}}\big) \cdot w\big(B \cup S^*_{\mathcal{G}}\big)} \leq \frac{\lambda \cdot c\big(S^*_{\mathcal{G}}\big)}{w(\mathcal{G})^2}.$$

Let $(A_{\mathcal{G}}, S_{\mathcal{G}}, B_{\mathcal{G}})$, be the partition returned by $(\mathcal{G}, c, w)$. Without the loss of generality assume that $w(A_{\mathcal{G}}) \geq w(B_{\mathcal{G}})$. Using Theorem 3.5.2.1, we get that:

$$\alpha_{c,w}(A_{\mathcal{G}}, S_{\mathcal{G}}, B_{\mathcal{G}}) = \frac{c(S_{\mathcal{G}})}{w(A_{\mathcal{G}} \cup S_{\mathcal{G}}) \cdot w(B_{\mathcal{G}} \cup S_{\mathcal{G}})} \leq f_n \cdot \frac{\lambda \cdot c\big(S^*_{\mathcal{G}}\big)}{w(\mathcal{G})^2}.$$

Let $\beta = w(B_{\mathcal{G}} \cup S_{\mathcal{G}})/w(\mathcal{G})$. We have that $(1 - \beta) \cdot w(\mathcal{G}) = w(A_{\mathcal{G}})$, so we conclude that the contribution of the decision tree $D_{\mathcal{G}}$ is bounded by:

$$\begin{aligned} w(\mathcal{G}) \cdot c(S_{\mathcal{G}}) &\leq \lambda \cdot f_n \cdot \frac{w(A_{\mathcal{G}} \cup S_{\mathcal{G}}) \cdot w(B_{\mathcal{G}} \cup S_{\mathcal{G}})}{w(\mathcal{G})} \cdot c\big(S^*_{\mathcal{G}}\big) \\ &\leq \lambda \cdot f_n \cdot w(B_{\mathcal{G}} \cup S_{\mathcal{G}}) \cdot c\big(S^*_{\mathcal{G}}\big) \leq \lambda \cdot f_n \cdot \sum_{k=w(A_{\mathcal{G}})+1}^{w(\mathcal{G})} c\Big(S^*_{\lfloor k/2 \rfloor} \cap \mathcal{G}\Big) \end{aligned}$$

where the last inequality is by Lemma 3.5.0.4.

As before, to bound the cost of the whole solution we will firstly show the following lemma. The argument is mostly the same as for the Lemma 3.5.1.6, however, there are few differences and we include it for completeness:

**Lemma 3.5.2.4.**

$$\sum_{\mathcal{G}} \sum_{k=w(A_{\mathcal{G}})+1}^{w(\mathcal{G})} c\Big(S^*_{\lfloor k/2 \rfloor} \cap \mathcal{G}\Big) \leq \sum_{k=0}^{w(G)} c\Big(S^*_{\lfloor k/2 \rfloor}\Big)\,.$$

*Proof.* Fix a value of $\mathcal{G}$ and $k$. Their contribution to the cost is $c\Big(S^*_{\lfloor k/2 \rfloor} \cap \mathcal{G}\Big)$. Consider which candidate subgraphs contribute such a term. By definition of $S_{\mathcal{G}}$, we have that $\mathcal{G}$ is the minimal subgraph, such that $w(\mathcal{G}) \geq k \geq w(A_{\mathcal{G}}) + 1 > w(H)$, 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. Since for all $H_1, H_2 \in \mathcal{G} - S_{\mathcal{G}}$, $H_1 \cap H_2 = \emptyset$, we have that $\Big(S^*_{\lfloor k/2 \rfloor} \cap H_1\Big) \cup \Big(S^*_{\lfloor k/2 \rfloor} \cap H_2\Big) = \emptyset$, the claim follows by summing over all values of $k$. □

We are now ready to bound the cost of the solution. Let $D$ be the decision tree returned by the procedure. We have:

$$\begin{aligned}
\texttt{COST}_D(G) &\leq \sum_{\mathcal{G}} w(\mathcal{G}) \cdot c(S_{\mathcal{G}}) \\
&\leq \lambda \cdot f_n \cdot \sum_{\mathcal{G}} \sum_{k=w(A_{\mathcal{G}})+1}^{w(\mathcal{G})} c\Big(S^*_{\lfloor k/2 \rfloor} \cap \mathcal{G}\Big) \leq \lambda \cdot f_n \cdot \sum_{k=0}^{w(G)} c\Big(S^*_{\lfloor k/2 \rfloor}\Big) \\
&\leq 2 \cdot \lambda \cdot f_n \cdot \texttt{OPT}(G) = \Big(12 + 4\sqrt{5}\Big) \cdot f_n \cdot \texttt{OPT}(G)
\end{aligned}$$

where the third inequality is due to Lemma 3.5.2.4 and the last inequality is by Lemma 3.5.0.3. □

# 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.

[Ber24] Benjamin Aram Berendsohn. "Search trees on graphs". Dissertation. 2024. URL: http://dx.doi.org/10.17169/refubium-45704.

[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.

[CLS16] Ferdinando Cicalese, Eduardo Laber, and Aline Saettler. “Decision Trees for Function Evaluation: Simultaneous Optimization of Worst and Expected Cost”. In: *Algorithmica* 79 (Oct. 2016). DOI: 10.1007/s00453-016-0225-9.

[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.

[MOW08] Shay Mozes, Krzysztof Onak, and Oren Weimann. "Finding an optimal tree searching strategy in linear time". In: *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*. Ed. by Shang-Hua Teng. SIAM, 2008, pp. 1096–1105. URL: http://dl.acm.org/citation.cfm?id=1347082.1347202.

[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 even when restricted to trees with* $\Delta(T) \leq 16$ *and to trees with* $\text{diam}(T) \leq 8$.

*Proof.* The problem is in NP since, given a decision tree $D$, one can verify in polynomial time whether all the requirements are satisfied.

To show hardness, we use a black-box reduction from the edge-query, uniform-cost, and non-uniform-weight variant, which is NP-complete even when restricted to trees with $\Delta(T) \leq 16$ and to trees with $\text{diam}(T) \leq 4$ [Jac+10]. Let $(T, w, K)$ be such an instance. We construct a new instance $(T', c, w, K')$ for the TSP as follows: for every $v \in V(T)$, we set $c(v) = K + 1$. We subdivide each edge $e \in E(T)$ by adding a 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 new instance, we replace each query in $D$ with a query to the vertex that subdivides the corresponding edge. Additionally, below each leaf of $D$, we attach the appropriate queries to the original 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 $K + w(T) \cdot (K + 1) = K'$, as required.

Observe that in the new instance, for every vertex $v \in T$, 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. This implies that each such vertex has exactly one such query in its query sequence, namely the query to $v$ itself. Otherwise, the cost would exceed $K'$, and we conclude that every such $v$ is queried only when the candidate subset consists solely of $v$.

Conversely, assume there exists a decision tree $D'$ of cost at most $K'$ for the new instance. We show how to obtain a decision tree $D$ for the original instance. We replace each query to a vertex $v_e$ with a query to edge $e$, and delete all queries to vertices $v \in V(T)$. Since each $v \in V(T)$ was the last query in $Q_{D'}(T', v)$, performed when the candidate set consisted only of $v$, the resulting $D$ 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.

Finally, note that subdividing each edge doubles the diameter of the tree while leaving the degree unchanged, so the claim follows.

□

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

***Decision weighted $\alpha$-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 $\alpha$ 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 $\alpha$-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. □