Algorithms for optimal search trees on trees

Bachelor-Thesis

Johannes Voderholzer

submitted on: December 5, 2024

supervisor: Prof. Dr. László Kozma

first reviewer: Prof. Dr. László Kozma

second reviewer: Prof. Dr. Wolfgang Mulzer

Abstract

Finding optimal binary search trees in terms of minimal search cost for a sequence of key queries is a well studied problem in computer science. An important theorem over the combinatorial structure of the problem found by Knuth in 1971 can be used to speed up a dynamic program for the problem to $\mathcal{O}(n^2)$ time, where n is the number of keys. Recently, Berendsohn and Kozma gave a 2-approximation algorithm for a generalization of optimal binary search trees to search trees on trees, which runs in $\mathcal{O}(n^3)$ time[2]. We show, that Knuth theorem also holds in this setting and use it to improve the running time of the algorithm to $\mathcal{O}(\min(D^2 \cdot L^2, n^3))$, where D denotes the diameter and L the number of leaves of the search space tree. We give a full implementation of both algorithms and verify the correctness and running time on randomly generated trees of different types. Further, we show that a bound given by Mehlhorn for binary search trees can be generalized to search trees on trees.

Acknowledgements

First of all, i would like to thank my supervisor, László Kozma, whose inspiring lectures got me interested in theoretical computer science. In addition, I am grateful to him for introducing me to the topic of "search trees on trees" and for giving me insightful feedback while replying to my queries in a very short time-frame. It was always fascinating to talk with him about the thesis and other topics. His extensive knowledge of algorithms and theoretical computer science has often enabled him to help me with questions that I have been struggling with for a long time.

I would also like to thank Wolfgang Mulzer, who is the second examiner in this thesis and with whom I had the pleasure of working as a tutor last semester. It was a really fun experience to work with him and I would also like to thank him for his interesting lectures that got me excited about the subject of theoretical computer science.

Lastly, I also want to thank my family for the extensive support throughout this thesis and for keeping me motivated during challenging times.

Contents

1 Introduction

In the context of a totally ordered set K composed of keys, binary search trees represent an organized tree structure used for the purpose of locating specific keys within the set K. Each node x of the search tree stores a key (denoted by $key(x)$), so that the binary-search-tree property is satisfied: For every node y in the left subtree of x the inequality $key(y) \leq key(x)$ holds. Similarly, for every node z in the right subtree of x the inequality $key(z) \geq key(x)$ holds.

Now, instead of considering K as a totally ordered set, we can also view it as a path graph P, where each node from P represents a key from K, so that keys are sorted from smallest to largest along this path according to the total order. Let r be the root of a search tree T_{search} for K. Assume we remove r from the path graph, then P falls apart into at most two connected components. One only with keys smaller than the key of r (call it C_1) and one with keys larger than the key of r (call it C_2). So in T_{search} the left subtree of the root contains only keys from C_1 and the right subtree contains only keys from C_2. This holds recursively, so for example removing the root of the left subtree from C_1 results in a split of C_1 itself into at most two connected components, that again correspond to the left and right subtree of the removed node. For example, consider a path consisting of keys $1, 2, 3, 4, 5, 6, 7$. We can build a valid binary search tree in the following way: First pick key 4 as the root. This splits the path into subpaths with keys $\{1, 2, 3\}$ and $\{5, 6, 7\}$. In the first subpath we pick node 2 as the root, so that 1 is the left and 3 the right child of 2. In the second, we pick node 5 as the root, which leaves nodes with keys 6 and 7. From this component we pick 7 as the root with 6 as the left child. The resulting search tree then looks like in Figure 1.

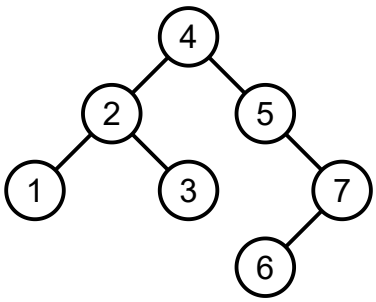
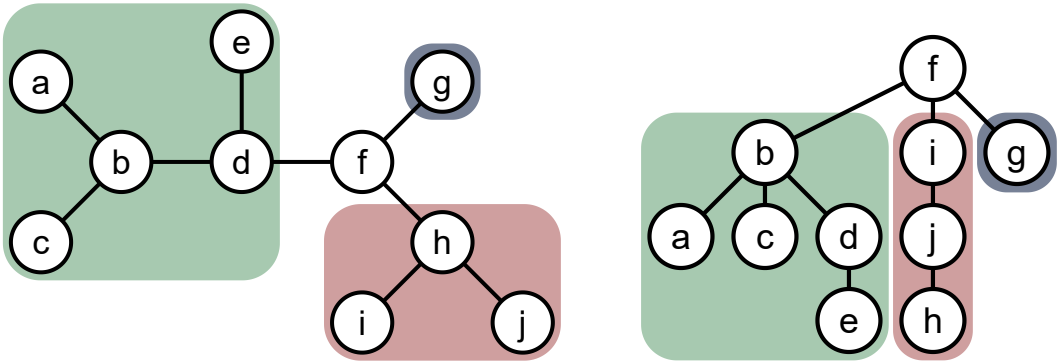


Figure 1: A valid binary search tree

This perspective on binary search trees directly leads to some interesting generalizations. So far, we only considered the search space as a path graph. However, we could also pick arbitrary undirected connected graphs as the search space instead. This generalization is called search trees on graphs or in a different context elimination trees [3]. Analogous to binary search trees, we define a search tree T on a graph S (search space) recursively. First, the root r of T must be in S. Then the subtrees of r are recursively build on the connected components of $S \backslash \{r\}$, where $S \backslash \{r\}$ denotes the graph S after we remove r from it.

In this work, we will restrict ourselves to the case, where the search space itself has the form of a general unrooted tree. We will refer to a search tree built on top of such a search space as an STT (search tree on tree). For example, Figure 2a shows such a search space S with keys $a, b, ..., j$, and Figure 2b a valid search tree on S. In this example, f is chosen as the root. When the node f is removed from search space S, it results in three connected components. The first component contains keys $\{a, b, c, d, e\}$ (marked green), the

second component contains keys $\{h, i, j\}$ (marked red), and the third component contains only the key $\{g\}$ (marked blue). These components build the subtrees of f, as marked in Figure 2b. In the green component, we pick b as the root node. When we remove b from the component, it splits into sub-components with keys $\{a\}, \{c\}$ and $\{d, e\}$. This forces a and c to be child nodes of b. However, we can pick either d or e as the root for the last sub-component. In our example, d is chosen as the root. In the red component, i is picked as the root, which leaves a connected sub-component with keys h, j. From this sub-component, j is chosen as the root.



(a) search space S (b) search tree on S

Figure 2: a valid search tree on a tree-shaped search space S

In case we want to search for a specific key x in such an STT, we can use the following algorithm to find x: Start in the root. Then a so called "oracle call" is performed, which compares the key r of the root node with x. If $x == r$, we can stop the search. Otherwise, the oracle tells us the connected component that contains x, after we remove r from S. We can then continue our search in the subtree, that corresponds to the identified connected component. The oracle calls are assumed to have constant cost. For example, in the binary search tree case, the oracle call corresponds to a simple "<" or ">" comparison of the keys. This tells us, if we have to move into the left or the right subtree.

1.1 Thesis results

The problem of calculating an optimal binary search trees in terms of total search time for a sequence of accesses is a well studied problem in computer science[10]. In this thesis we will discuss, how certain findings can be generalized to the STT setting. In Section 3.2 we show a generalization of a bound given by Mehlhorn in [8]. Further, we explore the concept of k-cut trees introduced by Berendsohn and Kozma[2] and prove a generalization of a theorem from Knuth to 2-cut STT's in Section 5.1. We use this to improve the original optimal 2-cut STT algorithm by integrating the theorem in Section 5.2. We show, that the performance of the algorithm can be improved even more with efficient result caching and give an elementary analysis for the running time in Section 5.2.3. Experimental results in Section 6.3 suggest, that this bound can be improved. Further we give a small counter example in Section 5.3, which shows that Knuth's theorem does not to k-cut trees or optimal STT's in the same way it does to 2-cut STT's. However, it is still an open question if other generalizations are possible.

In the following sections, we provide comprehensive definitions and present our results in greater detail.

2 Optimal search trees on trees

2.1 Basic Definitions

In this section, we are formalizing what we already talked about in Section 1 and mostly follow the definitions given in [2]. We use standart terminology for graphs and trees, so $V(G)$ denotes the set of vertices of a graph G.

Definition 1. A *search space* S is an unrooted tree, where $V(S)$ is a set of *keys*. Further $S \setminus \{v\}$ denotes the induced subgraph after removing a vertex $v \in V(S)$ from S.

Definition 2. With $Neig_S(v)$ we denote the set of neighbouring vertices of a node v in S. The *neighbour-subtree* for a neighbour w of a node v in S denotes the component of $S \setminus \{v\}$ containing w.

Definition 3. For a rooted tree T and a vertex $v \in T$, we denote with T_v the subtree of T, that is rooted at vertex v. The *child-subtrees* of v are all the subtrees T_x, where x is a child of v in T. Further, $d_T(x)$ denotes the *depth* of x in T, starting with depth one for the root.

Definition 4. The root of an undirected tree is given by $R(T)$.

Definition 5. A *STT (search tree on tree)* T for a search space S is a rooted tree with $R(T) \in V(S)$, so that the child-subtrees of $R(T)$ are STT's on the connected components of the forest $S \setminus \{R(T)\}$.

2.2 Problem Definition

In this work we consider the problem of finding an optimal STT for a search space S. We are given a search sequence $X = (x_1, ..., x_m) \in V(S)^m$ of length m. Then $f : V(S) \to \mathbb{N}$ calculates the search frequency of a node, i.e. how often a key appears in X. We want to find an STT with minimum **total search cost** according to this sequence. The total search cost of a given STT T on S is defined by:

$$Cost(T) = \sum_{x \in S} f(x) \cdot d_T(x)$$

2.3 Exponential algorithm

To my knowledge, there is no exact algorithm known that can solve this problem in polynomial time with respect to number of nodes. In this section, we describe a simple exact exponential algorithm first, and later discuss polynomial time approximations. The following idea is fundamental for the algorithm. Let T be an optimal STT for a search space S and let $T_{v_1}, ..., T_{v_k}$ be the child-subtrees of $R(T)$ in T. Then the following holds:

Lemma 6. $Cost(T) = m + \sum_{i=1}^{k} Cost(T_{v_i})$.

Proof. We can rewrite the cost of T by splitting up the sum formula and the use the fact, that $d_{T_{v_i}}(x) + 1 = d_T(x)$ for every node x and child-subtree T_{v_i}:

$$\begin{aligned} Cost(T) &= \sum_{x \in V(S)} f(x) \cdot d_T(x) \\ &= f(r) + \sum_{i=1}^{k} \sum_{x \in V(T_{v_i})} f(x) \cdot (d_{T_{v_i}}(x) + 1) \\ &= \left(\sum_{x \in V(S)} f(x) \right) + \sum_{i=1}^{k} Cost(T_{v_i}) \\ &= m + \sum_{i=1}^{k} Cost(T_{v_i}) \end{aligned} \tag{1}$$

□

This also means that every child-subtree T_{v_i} must be an optimal STT for the subtree induced by $V(T_{v_i})$. Assume, that one of the child-subtrees is not optimal. Then we could replace this child-subtree with the optimal child-subtree and therefore lower the total cost given in Eq. (1). This would mean, that T is not optimal, which is a contradiction.

This directly yields a simple exponential time algorithm: Pick a vertex v from S, then recursively build optimal STT's on the connected component of $S \setminus \{v\}$ and sum up the total cost according to Eq. (1). Do this for every vertex in S and return the minimum cost. See Algorithm 1 for more detail.

Algorithm 1 simple exponential time algorithm for optimal STT

Input: Search Space S with frequencies $f : V(S) \to \mathbb{N}$
Output: cost of an optimal STT

```
1: procedure OptSTT(S)
2:     minCost ← inf
3:     for all v ∈ V(S) do
4:         cost ← 0
5:         for all C ∈ Connected Components of S \{v} do
6:             cost ← cost + OptSTT(C)
7:         minCost ← min(minCost, cost)
8:     return minCost + Σ_{x∈V(S)} f(x)
```

This algorithm has exponential running time even in the best case. We can estimate a lower bound for the running time $T(n)$ by constructing a recurrence relation. For this, we only consider the cases, where v is a leaf node. Then $S \setminus \{v\}$ contains a single component, allowing us to simplify the formula. This results in the following equation:

$$\begin{aligned} T(n) &\geq \sum_{v \in V(S)} \sum_{C \in \substack{\text{components} \\ \text{of } S \setminus \{v\}}} T(|V(C)|) \\ &\geq \sum_{\substack{v \in V(S) \\ v \text{ is leaf}}} T(n-1) \geq 2 \cdot T(n-1) \end{aligned}$$

In the last step we used, that every tree with at least two nodes has at least two leaves. For the base case we use $T(1) = 1$. Expanding this recurrence relation yields $T(n) \geq 2^{n-1}$, so $T(n) \in \Omega(2^n)$.

However, we can use dynamic programming to decrease the running time of the above algorithm. Instead of recalculating everything recursively, we can also cache already calculated results. For this, we can create a dictionary storing solutions for individual subtrees. We can then store the calculated cost in the end of the procedure and return already calculated results in the beginning right before Line 2 by reading of the result from the dictionary, if it already exists. This does not improve the worst case running time however. For this, consider a simple star tree (Figure 3) as the search space.

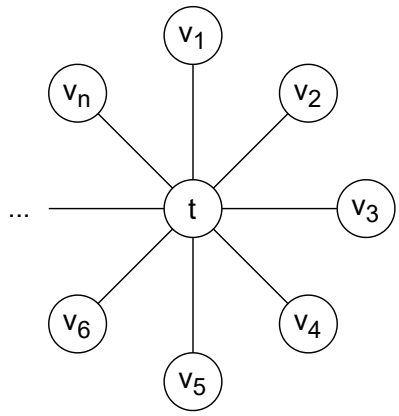


Figure 3: Star Tree

There are $\mathcal{O}(2^n)$ subtrees possible, since it is possible to pick any subset from vertices $\{v_1, ..., v_n\}$ and connect them with t. Since the algorithm checks each of these subsets, the running time remains exponential, and we are even dealing with exponential space complexity. For a simple path tree however, we would only need $\mathcal{O}(n^3)$ time complexity using this algorithm, since in this case there are only $\mathcal{O}(n^2)$ possible subpaths. Note, that the approach described above is a generalized version of the standard dynamic programming for optimal binary search trees.

2.4 Knuth's trick for optimal binary search trees

The optimal binary search tree problem (search space is given as a path) is well studied. There are some important findings, that might translate to the general case, where the search space is a tree. The first important observation by Knuth[6] is, that we can improve the dynamic program by a factor of n, so that the running time is $\mathcal{O}(n^2)$ instead of $\mathcal{O}(n^3)$. Assume the path-graph consists of nodes $v_1, v_2, ..., v_n$ in this order, and let $T_{i,j}$ denote an optimal binary search tree for nodes $v_i, ..., v_j$ with $i \leq j$. For the next statement, we also say that a vertex on the path is smaller than another vertex, if the index is smaller, i.e. $v_i \leq v_j$, if and only if $i \leq j$.

Theorem 7. *Adding the last vertex v_n to the right side of the path doesn't force the root of an optimal binary search tree to move to the left. Formally, there exists an optimal binary search tree $T_{1,n}$, so that $R(T_{1,n-1}) \leq R(T_{1,n})$, when $n \geq 2$.*

We now revisit the proof for this theorem given by Knuth[6]. However note, that we are looking at a special case of the problem presented in the original paper, where also the

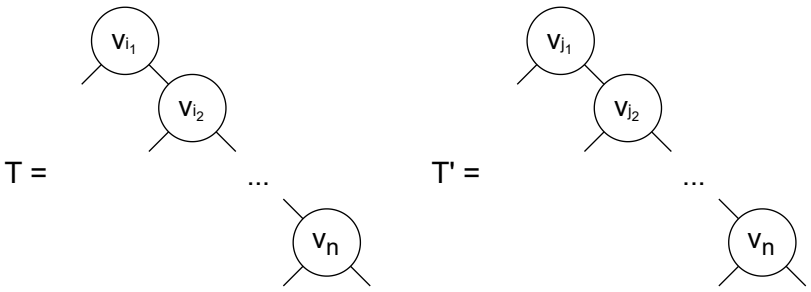
search frequencies for keys not in the search space are considered. Hence the proof may vary a bit from the original one.

Proof. The proof uses induction over n. When $n = 2$, the root can not move to the left, because there are no additional vertices.

For the induction step, assume the theorem holds for $n-1$ vertices. Let $f(v_n)$ be the frequency of node v_n. The proof then proceeds in the following two steps:

Step 1: Assume $f(v_n) = 0$
Then we can use the same root for $T_{1,n}$ as in $T_{1,n-1}$. Assume there was a better tree for $T_{1,n}$. Then deleting v_n from it would yield a better result for $T_{1,n-1}$. We can always delete this node, since it is either a leaf node or has exactly one child, in which case we can do a rotations around v_n until it has no child and delete it afterwards.

Step 2: Let α be a "threshold" value, so that so that there exists an optimal tree T for $f(v_n) = \alpha - \epsilon$ with $R(T) \geq R(T_{1,n-1})$, but not for $f(v_n) = \alpha + \epsilon$, for all sufficiently small $\epsilon > 0$.
Then let T' be an optimal binary search tree for nodes $v_1, ..., v_n$ and $f(v_n) = \alpha + \epsilon$ with $R(T') < R(T_{1,n-1})$. Now consider trees T and T':



One crucial observation is, that v_n must be positioned higher in T' than in T, so $d_T(v_n) > d_{T'}(v_n)$. Intuitively, this makes sense, since the increase of the frequency of v_n forces T to change its structure to T', and moving the node down doesn't help. This can be proven by the following argument. If we set $f(v_n) = \alpha - \epsilon$ in both trees, then $Cost(T) \leq Cost(T')$, since T was optimal for $\alpha - \epsilon$. If we set $f(v_n) = \alpha + \epsilon$ in both trees, then $Cost(T) > Cost(T')$, Since T is not optimal for $f(v_n) = \alpha + \epsilon$. But the only thing that changed in both of the cost calculations is the frequency of v_n. In order for the sign to flip from "$\leq$" to "$>$", $d_T(v_n) \cdot 2\epsilon > d_{T'}(v_n) \cdot 2\epsilon$ must hold, since the cost increased by $d_T(v_n) \cdot 2\epsilon$ for T and $d_{T'}(v_n) \cdot 2\epsilon$ for T'. Then $d_T(v_n) > d_{T'}(v_n)$ directly follows.

By our assumptions, $i_1 > j_1$. Now consider vertices v_{i_2} and v_{j_2}. These are the roots for the paths consisting of nodes $\{v_{i_1+1}, ..., v_n\}$ and $\{v_{j_1+1}, ..., v_n\}$ respectively. Then by induction hypothesis and symmetry of the theorem we can conclude that $i_2 \geq j_2$ (append vertices back to the path $\{v_{i_1+1}, ..., v_n\}$ on the left, until it becomes $\{v_{j_1+1}, ..., v_n\}$). If $i_2 > j_2$, we can use the same argument and conclude $i_3 \geq j_3$ and so on. However since $d_T(v_n) > d_{T'}(v_n)$, it holds that $j_{d_{T'}(v_n)} = n > i_{d_{T'}(v_n)}$. Thus our "chain" has to break somewhere, meaning $i_k = j_k$ holds at some point $1 < k < d'_T(v_n)$. But then we could replace the right subtree of the node v_{i_k} in T with the right subtree of v_{j_k} in T'. Then the total cost of the resulting tree T^* is the same as of T' for $f(v_n) = \alpha + \epsilon$, since now $d_T(v_n) = d'_T(v_n)$. If it was smaller, then T' would not be optimal for $f(v_n) = \alpha + \epsilon$. If it was larger than $Cost(T')$, then pasting the original right subtree of T into T' and T^* would change the cost of both trees by the same amount since they get pasted at the same depth,

giving $Cost(T) = Cost(T^*) > Cost(T')$, also for $f(v_n) = \alpha - \epsilon$. This is a contradiction, since we assumed T to be an optimal tree. Thus we can take T^* instead of T', meaning we do not have to move the root. By combining *step 1* and *2*, we can then achieve any frequency for $f(v_n)$ and therefore the theorem holds. □

This observation from Knuth can be used to improve the algorithm. The idea here is, that we do not have to check every node as potential root. For example, if we want to calculate the optimal binary search tree for the path $v_1, ..., v_n$, we could first calculate roots r_1 and r_2 for $v_1, ..., v_{n-1}$ and $v_2, ..., v_n$ recursively. Then the root r for the total path must be between r_1 and r_2, e.g. $r_1 \leq r \leq r_2$. This changes the running time of the algorithm from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$. For this, consider the total amount of root checks for all subpaths of fixed length d, where $R_{i,j}$ denotes the root calculated by the dynamic program for the subpath with nodes $v_i, ..., v_j$. The resulting sum formula then gives a telescoping sum:

$$\begin{aligned}\sum_{i=1}^{n-d+1} R_{i+1,i+d-1} - R_{i,i+d-2} &= \sum_{i=1}^{n-d+1} R_{i+1,i+d-1} - \sum_{i=0}^{n-d} R_{i+1,i+d-1} \\ &= R_{n-d+2,n} - R_{1,d} \leq n\end{aligned}$$

The total cost is then upper bounded by $\sum_{d=1}^{n} n \in \mathcal{O}(n^2)$.

2.5 Fast approximation for optimal binary search trees

There is a simple approximation for optimal binary search trees described in [8] again for a more generalized version, where also the search frequencies for keys are considered, that are not in the search space. In the generalized version, we get keys $v_1, ..., v_n$ with normalized frequency distribution $\beta_1, ..., \beta_n$ and also $\alpha_0, \alpha_1, ..., \alpha_n$, where α_i is the probability of searching for a key between v_i and v_{i+1} (α_0 and α_n have obvious interpretations). These probabilities sum up to 1, so $\sum_{i=1}^{n} \beta_i + \sum_{i=0}^{n} \alpha_i = 1$. The algorithm always selects the root so that the total weight of the left and right subtrees is balanced as much as possible, e.g. we pick element v_i as the root, so that $|(\alpha_0 + \beta_0 + ... + \alpha_{i-1}) - (\alpha_i + \beta_{i+1} + ... + \beta_n + \alpha_n)|$ is minimized. A straightforward implementation of this algorithms runs in $\mathcal{O}(n \log n)$ time, however $\mathcal{O}(n)$ time is also possible[5]. In 1975, Mehlhorn showed that this algorithm gives almost optimal binary search trees[8]. For this, he first showed, that a search tree created by this algorithm has total cost of at most

$$2 + \frac{H}{1 - \log(\sqrt{5} - 1)},$$

where $H = \sum \alpha_i \log(\frac{1}{\alpha_i}) + \sum \beta_j \log(\frac{1}{\beta_j})$ denotes the Entropy of the distribution. If C_{opt} denotes the optimal achievable cost, he also shows that the following inequality holds:

$$C_{opt} \geq \frac{H}{\log 3} \tag{2}$$

This means, if C_{bal} is the cost of the binary search tree constructed by the above algorithm, then

$$0.63 \cdot H \leq C_{opt} \leq C_{bal} \leq 2 + 1.44 \cdot H.$$

2.6 Other algorithms

There is no algorithm known yet, that calculates optimal binary search trees in subquadratic time [1]. However under certain assumptions on the search frequency distribution over the keys, better algorithms are possible. For example, if the frequency for every key is at least ϵn for some constant $\epsilon > 0$, we can get achieve running time $\mathcal{O}(n^{1.6})$ [7]. The same authors also suggest an algorithm that exhibits a trade-off between time and accuracy, where it is possible to achieve a constant error in $\mathcal{O}(n^{1.6})$ time.

3 Generalizations to STT's

So far, our focus has primarily been on the current state of the art for binary search trees. In the next sections, we will see how the previously discussed findings can generalize to the setting, where the search space is a tree. Also, we only have an exponential time algorithm for the general case, and to the best of my knowledge, there is no exact polynomial time algorithm known yet. So we will also briefly discuss a 2-approximation algorithm for finding STT's, that uses ideas from the algorithm used in Section 2.5 .

3.1 Centroid Trees

The idea the approximation algorithm described in Section 2.5 can be translated to STT's. For this, a so called centroid tree over S is defined. A centroid vertex is a vertex $v \in S$, so that the sum of frequencies in every connected component of $S \setminus \{v\}$ is less than half the total sum of frequencies. A centroid tree can then be build by picking a centroid vertex v and recursively building centroid trees on the connected component of S after the removal of v. Recently, it was shown by Berendsohn, Golinsky, Kaplan and Kozma [1], that such a centroid tree can be constructed in $\mathcal{O}(n \log h) \in \mathcal{O}(n \log h)$ time, where h is the height of the resulting centroid tree. In the typical case, where the height of the resulting centroid tree is $\mathcal{O}(\log n)$, this gives $\mathcal{O}(n \log \log n)$ running time. Further they showed, that the cost of a centroid tree is at most twice as large as the optimal cost. This guarantee is the best possible.

3.2 lower bound

The proof for the lower bound described in Eq. (2) can be generalized to search trees on trees. If C_{opt} denotes the optimal achievable cost for an STT on S, then the following holds:

$$C_{opt} \geq \frac{H \cdot |X|}{\log(\delta(S)+1)}, \tag{3}$$

where $\delta(S)$ is the maximum degree of S. We define $p(v) = \frac{f(v)}{|X|}$, where $|X|$ is the length of the search sequence (see Section 2.2), so $\sum_{v \in V(S)} p(v) = 1$. The entropy H of the distribution modeled by the frequencies of the search keys is then given by $\sum_{v \in V(S)} p(v) \cdot \log(\frac{1}{p(v)})$. We now give a proof for the lower bound:

Proof. Let T be any STT on S. Let $ch(v)$ denote the number of child nodes of v in T and $d(v)$ denote the depth of v in T. We start by defining $\beta(v) = (\delta(S)+1-ch(v)) \cdot (\delta(S)+1)^{-d(v)}$ and

$$L = \sum_{v \in V(S)} \beta(v),$$

which is equal to 1. This can be proven by induction over the number of nodes in T. If $|V(T)| = 1$, the formula gives $(\delta(S)+1)(\delta(S)+1)^{-1} = 1$, since there is only one node and it has no children. For the induction step, assume we append a new node w to some node v in T. Before adding w we have $\beta(v) = (\delta(S)+1-ch(v))\cdot(\delta(S)+1)^{-d(v)}$. Adding a new node to v will increase $ch(v)$ by one and therefore decrease $\beta(v)$ by $(\delta(S)+1)^{-d(v)}$. However, w has $d(v)+1$ depth and no children, so $\beta(w)$ will be $(\delta(S)+1)\cdot(\delta(S)+1)^{-(d(v)+1)} = (\delta(S)+1)^{-d(v)}$. So the sum will remain the same, which was 1 by induction hypothesis.

Since $L = 1$, we can see β as a distribution over the structure of T. We can then conclude, that $L = \sum_{v\in V(S)} p(v)\cdot\log(\frac{1}{p(v)}) \leq \sum_{v\in V(S)} p(v)\cdot\log(\frac{1}{\beta(v)})$, which is a well-known inequality (Gibbs' inequality). Then

$$
\begin{aligned}
H &= \sum_{v\in V(S)} p(v)\cdot\log(\frac{1}{p(v)}) \\
&\leq \sum_{v\in V(S)} p(v)\cdot\log(\frac{1}{\beta(v)}) \\
&= \sum_{v\in V(S)} p(v)\cdot\left(\log((\delta(S)+1)^{d(v)}) + \log\left(\frac{1}{\delta(S)+1-ch(v)}\right)\right) \\
&= \sum_{v\in V(S)} p(v)\cdot\left(d(v)\log(\delta(S)+1) + \log\left(\frac{1}{\delta(S)+1-ch(v)}\right)\right) \qquad (4) \\
&\leq \log(\delta(S)+1)\sum_{v\in V(S)} p(v)\cdot d(v) \\
&= \frac{1}{|X|}Cost(T)\cdot\log(\delta(S)+1)
\end{aligned}
$$

In Eq. (4), we can drop the expression $\log\frac{1}{\delta(S)+1-ch(v)}$, since $\delta(S)$ is the maximum degree in S and therefore no vertex in T can have more than $\delta(S)$ children. Then the denominator is at least one, so the log expression will become negative. We can then rearrange the formula, so that we get:

$$Cost(T) \geq \frac{H\cdot|X|}{\log(\delta(S)+1)}$$

Since we didn't specify T in the beginning, this also holds for the optimal tree. □

Note, that the formula from Eq. (3) is the same as from Eq. (2), if the search space is a path. Then the maximum degree of a vertex in S is two, which gives the lower bound $\frac{H}{\log 3}$, if we see f as a probability distribution instead of a frequency count.

4 k-cut STT's

4.1 Definition

So far we only discussed centroid trees and how they can achieve a good approximation ratio in reasonable time (see Section 3.1). However, recently Berendsohn and Kozma [2] introduced a concept called k-cut tree, that allows $(1+\frac{1}{t})$-approximations of the optimal STT in $\mathcal{O}(n^{2t+1})$ time for all integers $t \geq 1$.

In their work, they defined k-cut trees in the following way:

Definition 8. The *cut* of a nonempty set $A \subseteq V(S)$ in S, denoted by $cut_S(A)$, is a set of directed edges (u, v), so that $u \in A$ and $v \in S \setminus A$. In other words, the cut is the set of edges, that are connected to only one endpoint after we remove the subgraph induced by A from S.

Definition 9. A *k-cut* tree on S is a STT T, so that for every subtree T_v (Definition 3) of T the number of edges in $cut_S(V(T_v))$ is at most k for all nodes $v \in V(S)$. In other words, the number of edges we need to "cut" to isolate T_v in S, is at most k. Note, that the subgraph induced by $V(T_v)$ in S defines a connected subtree because of the definition given for STT's.

One important observation is the following:

Observation 10. *The number of possible connected subtrees S' in S with $|cut_S(V(S'))| = k$ is in $\mathcal{O}(n^k)$.*

This is because there are $\mathcal{O}(n)$ edges in S, and we can enumerate all subtrees with k cuts by choosing k edges. This gives roughly $\binom{n}{k} \in \mathcal{O}(n^k)$ such subtrees.

In [2] it was shown, that an optimal k-cut STT approximates an optimal STT by a factor of $1 + \frac{2}{k}$. Optimal 2-cut STT's therefore give a 2-approximation. The idea for the proof is to show, that an arbitrary STT can be transformed into a k-cut STT, so that the depth of every node increases by a factor of no more than $1 + \frac{2}{k}$. This concludes the proof, since then every optimal binary search tree can also be transformed into an k-cut STT with a cost increase of a factor at most $1 + \frac{2}{k}$. They also give an $\mathcal{O}(n^{k+1})$ algorithm for finding an optimal k-cut STT using dynamic programming. Later, we will improve this algorithm for optimal 2-cut STT's.

4.2 When do k-cut trees give optimal results?

Sometimes it is not necessary to set k really high, since for example an optimal 2-cut STT will always give optimal results, if the search space is a path. This is because a subpath of S can at most define two cuts. The following lemma generalizes this idea to arbitrary trees:

Lemma 11. *If L is the number of leaves in S, then the cost of an optimal L-cut tree is the same as of an optimal STT.*

Proof. Assume there is a connected subtree S' in S, so that $|cut_S(V(S'))| = L + 1$. Since S is a tree, the resulting graph after removing S' from S is a forest consisting of exactly $L + 1$ trees. Since every tree must have at least two leaves, the total number of leaves is at least $2(L + 1)$. Since $L + 1$ of these leaves came from the cuts, S must have had at least $L + 1$ leaves, which is a contradiction.

The optimal STT T_{opt} therefore cannot have any subtree T_v for some node v, so that $|cut_S(V(T_v))| = L + 1$. Therefore T_{opt} is an L-cut tree. □

Using this lemma, we can sharpen the running time for an exact algorithm by calculating L and then constructing an optimal L-cut STT. The running time for this algorithm would then be $\mathcal{O}(n^{L+1})$.

5 Optimizing the algorithm for 2-cuts

In this section we improve the algorithm for calculating optimal 2-cut trees as described in [2]. We start with some definitions and observations and then show a generalization of Theorem 7. We give detailed descriptions for every algorithm and later analyse the running time.

5.1 Knuth's trick generalization for optimal 2-cut STT

Definition 12. Let $Sub_S(C)$ be the connected subtree on the search space S for a set of directed edges C, so that $cut_S(V(Sub_S(C))) = C$.

Definition 13. $P(x, y)$ denotes the set of vertices on the path from vertex x to vertex y in S, including x and y. For two cuts $c_1 = (v_1, w_1)$ and $c_2 = (v_2, w_2)$, we say that $P(c_1, c_2) = P(v_1, v_2)$.

Definition 14. $Dist_S(x, y)$ denotes the distance of two nodes x and y in S given by $|P(x, y)| - 1$ and $Neig_S(v)$ the set of neighbouring nodes in S of arbitrary $v \in S$.

Definition 15. A k-cut STT T on some subtree S^* of S with $|cut_S(V(S^*))| \leq k$ is not *valid*, if $|cut_S(V(T_v))| > k$ does hold for some $v \in S^*$. Otherwise we call it *valid*.

Definition 16. $OptTree_2(c_1, c_2)$ denotes an optimal and valid 2-cut STT of $Sub_S(\{c_1, c_2\})$ for two cuts c_1 and c_2. Further let $R(c_1, c_2)$ be the root of $OptTree_2(c_1, c_2)$, so $R(c_1, c_2) = R(OptTree_2(c_1, c_2))$.

Let $OptTree_2(c_1, c_2)$ be an optimal and valid 2-cut STT of $Sub_S(\{c_1, c_2\})$ with $c_1 = (v_1, w_1)$ and $c_2 = (v_2, w_2)$. Then the following holds:

Lemma 17. *The root of the optimal 2-cut STT for $Sub_S(\{c_1, c_2\})$ has to be on the unique path from c_1 to c_2 in S. Formally, $R(c_1, c_2) \in P(c_1, c_2)$. Further, any vertex of $P(v_1, v_2)$ is a valid root for a 2-cut STT.*

Proof. We mainly follow the proof idea given in [6]. Assume, some $v_k \notin P(c_1, c_2)$ is the root of $OptTree_2(c_1, c_2)$. Now, since v_k is not on the path, one child-subtree T_x of v_k must contain the whole path $P(c_1, c_2)$. Then the two cuts c_1 and c_2 are cuts for $V(T_x)$. A third cut is needed to exclude v_k from the component defined by $V(T_x)$ in S. Therefore we have at least a 3-cut STT, which is a contradiction. It remains to show that every node on the path is a valid root, i.e. the STT can still be a valid 2-cut tree. For that, let v_r be any vertex on the path $P(v_1, v_2)$. Consider now any child c of v_r in the STT and let T_c be the corresponding child-subtree. If the child is not in $P(c_1, c_2)$, then there is only one cut needed to isolate $V(T_c)$ in S. This is because v_1 and v_2 cannot be in T_c. Therefore the cut (c, v_r) remains the only cut. If c is still on the path, then either v_1 or v_2 is not in T_c. Otherwise v_r itself would not be on the path. Then we only need two cuts to isolate $V(T_c)$ in S. So every child-subtree of v_k satisfies the 2-cut definition, which means that v_k is a valid root. □

We now strengthen this lemma further. For this, we generalize Knuth theorem in the following way:

Theorem 18. *Extending the cut c_2 one step further away from c_1 never forces the root of a optimal 2-cut STT to move towards c_1. Formally, there always exist valid roots, so that $Dist_S(R(c_1, c_2), v_1) \leq Dist_S(R(c_1, c_2'), v_1)$, where $c_2' = (w_2, x)$ so that x was not previously in $Sub_S(\{c_1, c_2\})$.*

Note, that the same also holds for the symmetric case, i.e. extending c_1 away from c_2. For simplicity, we also denote with $A' = Sub_S(\{c_1, c_2'\})$ the new subtree after shifting the cut. Further, V_k with $w_2 \in V_k$ denotes the set of added vertices by shifting the cut. Formally, $V_k = V(Sub_S(\{c_1, c_2'\})) \setminus V(Sub_S(\{c_1, c_2\}))$. Note, that V_k can be very large as w_2 might have a lot of side-branches. The following lemma is fundamental for the proof:

Lemma 19. *V_k must be the nodes of a subtree T^k with w_2 as the root in any valid 2-cut STT on $Sub_S(\{c_1, c_2'\})$.*

Proof. Let k be any of these nodes from V_k. Assume w_2 is in some child-subtree of k in the STT. But then, similarly to Lemma 17, the 2-cut property is violated at T_k. So w_2 cannot be the child of any $k \in V_k$. This concludes the proof, since again by the definition of STT's the child-subtrees of any node correspond to the connected components after deleting the node in the search space. This means, any of the child-subtrees of w_2, that are part of V_k, remain together in the STT. □

Note, that T^k is also complete, meaning every child-subtree of w_2 in T^k contains exactly the same nodes as the corresponding child-subtree in T. Otherwise the STT property would be violated. This allows us to view w_2 together with the its children nodes from V_k as a "single big node". We use this now to prove Theorem 18.

Proof. We proceed by induction on the distance between the two cuts.

Base Case:
If $Dist_S(v_1, v_2) = 0$, and then extend c_2 further away from c_1, the root cannot move closer to c_1, because there was previously only one node in A.

Induction step:
Step 1: Assume, that for every new discovered vertex $k \in V_k$, $f(k) = 0$
In this case, we can use the same root as in the tree $T = OptTree_2(c_1, c_2)$. Assume, there was a 2-cut STT $T' = OptTree_2(c_1, c_2')$ with total cost less than the total cost of T. Because of Lemma 19, V_k are the nodes of a connected subtree in T' rooted at w_2. We can delete this subtree now in a way, that leaves a better 2-cut STT for the cuts c_1 and c_2 than T. If the child trees of w_2 only contain nodes from V_k, then we can delete the subtree without problems. There can however be a child-subtree of w_2 with a node $x \in P(c_1, c_2)$ as the root. There can only be one such subtree, because w_2 is part of a cut. After deleting V_k, we can append this subtree to the original parent of w_2 . Since the depth of x didn't increase, the total cost of the tree can only be smaller afterwards. The now obtained tree is a valid 2-cut STT for $Sub_S(\{c_1, c_2\})$, since T_x was a valid 2-cut STT before and V_k was removed, so the components are defined on $Sub_S(\{c_1, c_2\})$. This tree has lower cost than T, which means T was not optimal. This is a contradiction, so there cannot be a tree with total cost less than the cost of T. This means, we can append T^k at any valid vertex in T and get an optimal STT for the cuts c_1 and c_2', keeping the original root.

Step 2: Let $k \in V_k$ be any new discovered vertex. Fix the frequency for every other vertex and let α be the smallest threshold value, so that the optimal 2-cut STT is T, if $f(k) = \alpha - \epsilon$ but changes to $T' \neq T$, when $f(k) = \alpha + \epsilon$, for some very small value $\epsilon > 0$. Assume further, that the root of T' is closer to c_1 than T, i.e. $Dist_S(R(T'), v_1) < Dist_S(R(T), v_1)$. Here, an important observation is given by the following lemma.

Lemma 20. *$d'(w_2) < d(w_2)$, where $d'(w_2)$ is the depth of w_2 in T' and $d(w_2)$ the depth of w_2 in T'.*

Proof. Assume $d'(w_2) \geq d(w_2)$. We now transform T into a new tree called T^* in the following way: Replace the subtree T^k (Lemma 19) with the similar subtree $T^{k'}$ from T', using $f^*(k) = \alpha + \epsilon$ as the search frequency for k in T^*. By replacing, we mean deleting all the child-subtrees from w_2 in T that are part of V_k and replacing them with the similar child-subtrees of w_2 from T'. Because $V(T^k) = V(T^{k'}) = V_k$, the subtree rooted at w_2 contains the same nodes as before. All the child-subtrees of w_2 were valid in T', then also in T^*. So T^* is still a valid 2-cut STT. Similarly, we transform T' into T'^* by replacing $T^{k'}$ in T' with the similar subtree T^k from T, using the search frequency $f'^*(k) = \alpha - \epsilon$ for

k in T'^*. If $Cost(T^*) \leq Cost(T')$, we could use T^* instead of T', so $Cost(T^*) > Cost(T')$ must hold. Also, $Cost(T) \leq Cost(T'^*)$, otherwise T would not be optimal, as we could use T'^* instead of T. Let y be the assumed depth difference of the node w_2. Also for the following we denote with $d^*(x)$ and $d'^*(x)$ the depth for a node x in T^* and T'^*. Then we can formulate these observations in the following way.

$$
\begin{aligned}
& Cost(T^*) > Cost(T') \\
\Leftrightarrow \sum_{x \in V_k} f'(x) \cdot d^*(x) + \sum_{x \in A' \setminus V_k} f(x) \cdot d(x) &> \sum_{x \in V_k} f'(x) \cdot (d^*(x) + y) + \sum_{x \in A' \setminus V_k} f'(x) \cdot d'(x) \\
\Leftrightarrow \sum_{x \in A' \setminus V_k} f(x) \cdot d(x) &> y \cdot \sum_{x \in V_k} f'(x) + \sum_{x \in A' \setminus V_k} f'(x) \cdot d'(x) \qquad (5)
\end{aligned}
$$

And for the second inequality:

$$
\begin{aligned}
& Cost(T) \leq Cost(T'^*) \\
\Leftrightarrow \sum_{x \in V_k} f(x) \cdot d(x) + \sum_{x \in A' \setminus V_k} f(x) \cdot d(x) &\leq \sum_{x \in V_k} f(x) \cdot (d(x) + y) + \sum_{x \in A' \setminus V_k} f'(x) \cdot d'(x) \\
\Leftrightarrow \sum_{x \in A' \setminus V_k} f(x) \cdot d(x) &\leq y \cdot \sum_{x \in V_k} f(x) + \sum_{x \in A' \setminus V_k} f'(x) \cdot d'(x) \qquad (6)
\end{aligned}
$$

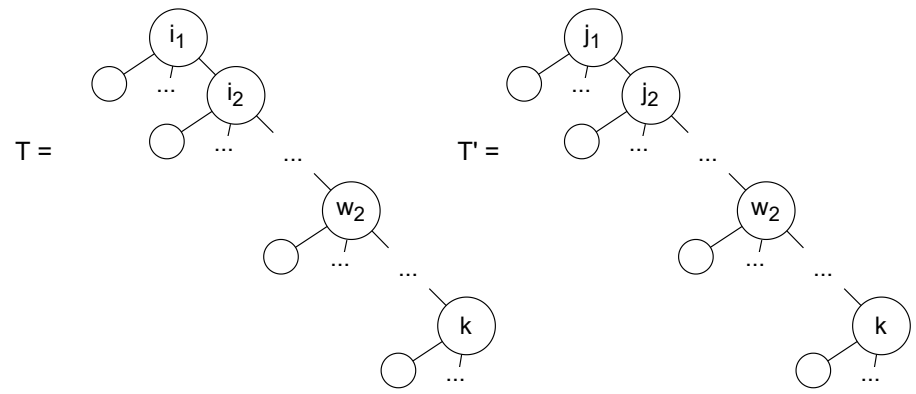
Then adding inequalities Eq. (5) and Eq. (6) together results in:

$$
\begin{aligned}
y \cdot \sum_{x \in V_k} f'(x) &< y \cdot \sum_{x \in V_k} f(x) \\
\Leftrightarrow (\epsilon + \alpha) + \sum_{x \in V_k \setminus \{k\}} f(x) &< (\epsilon - \alpha) + \sum_{x \in V_k \setminus \{k\}} f(x) \\
\Leftrightarrow (\alpha + \epsilon) &< (\alpha - \epsilon)
\end{aligned}
$$

This is a contradiction, since $\epsilon > 0$. So $d'(w_2) < d(w_2)$.

□

We now continue with the proof for Theorem 18. Consider the path from the root to k in both T and T':



Since we assumed, that the root of T' is closer to c_1, $Dist_S(j_1, v_1) < Dist_S(i_1, v_1)$ must hold. Consider the roots of the right subtrees of i_1 and j_1. These are given by $i_2 = R((i'_1, i_1), c'_2)$ and $j_2 = R((j'_1, j_1), c'_2)$. Here i'_1 is the next node along the path from i_1 to w_2 in S and similarly j_1 is the next node along the path from j_1 to w_2. Since we moved the cut by exactly one step in each tree, $Dist_S(j'_1, v_1) < Dist_S(i'_1, v_1)$ still holds. Since also the distance between the cuts decreased, we can use the induction hypothesis and by the symmetry of the theorem, we can conclude that there exist valid roots, so that $Dist_S(R((j'_1, j_1), c'_2), v_1) \leq Dist_S(R((i'_1, i_1), c'_2), v_1)$. Then $Dist_S(j_2, v_1) \leq Dist_S(i_2, v_1)$. Note, that here we might use the induction hypothesis multiple times. Assume now that $Dist_S(j_2, v_1) < Dist_S(i_2, v_1)$. Then by the same argument $Dist_S(j_3, v_1) \leq Dist_S(i_3, v_1)$. If now $Dist_S(j_3, v_1) < Dist_S(i_3, v_1)$, then $Dist_S(j_4, v_1) \leq Dist_S(i_4, v_1)$ and so on. However, since we have proven in Lemma 20, that $d'(w_2) < d(w_2)$ and because $j_{d'(w_2)} = w_2$, we have $Dist_S(j_{d'(w_2)}, v_1) > Dist_S(i_{d'(w_2)}, v_1)$. So at some point $1 < x < d'(w_2)$ the chain has to break, meaning that $Dist_S(j_x, v_1) = Dist_S(i_x, v_1)$.

Now, we can replace the child-subtree of i_x in T that contains w_2 with the similar subtree in T'. Since the depth of j_x and i_x are the same, T must have the same cost as T'. Assume not. If now $Cost(T) < Cost(T')$ (using $f(k) = f'(k) = \alpha + \epsilon$), then T' would not be optimal. If $Cost(T) > Cost(T')$ (again using $f(k) = f'(k) = \alpha + \epsilon$), we could replace the subtree with the original subtree of T in both T and T'. Now T is again the original tree. But because the depth of j_x and i_x are the same, the cost would change by the same amount in both trees and then $Cost(T) > Cost(T')$ (now using $f(k) = f'(k) = \alpha - \epsilon$) still holds. So T would not be optimal. This means, the transformation yields a tree, that is as good as T', but has the same root as T. Therefore, the root doesn't get forced towards c_1.

Since we have shown, that increasing the frequency of a single $k \in V_k$ while keeping everything else fixed doesn't force the root towards c_1, we can show this for any distribution of frequencies: Start with zero for every node, and increase the frequency of the nodes in V_k one by one towards the desired distribution. At each step, we do not need to shift the root towards c_1, so we also do not need to do it in total. □

5.2 Improved dynamic program

We now describe the dynamic programming algorithm for finding an optimal STT. The algorithm is similar to the original optimal 2-cut STT algorithm described in [2], except that we use Theorem 18 to save some computation time. For this, we first define an algorithm called $OptFromCuts(S, cuts)$, that returns the cost of a optimal 2-cut STT together with its root on the search space S for the subtree defined by the cuts $Sub_S(cuts)$ (see Definition 12). The optimal 2-cut STT on S can then be obtained by simply executing this algorithm with an empty list for the cuts.

The idea for the algorithm is as follows. We first check, if the cuts define a subtree of size one. If so, we have only one way of picking the root. Otherwise we check if we have exactly two cuts. If so, we shift the first cut one step towards the second cut along the path between them and calculate the optimal root r_1 recursively. We then do the same for the second cut, i.e. calculating the optimal root r_2 for the original two cuts, except that the second cut moved one step towards the first cut. Then using Theorem 18, there exists a optimal 2-cut STT T for the subtree defined by the original cuts, that uses a root from $P(r_1, r_2)$. Note, that we only shift the cuts, if the distance between them was originally larger than one. If not, we can just pick the middle node as the only possible root. Following Lemma 6, we can then calculate the total cost for T by recursively calculating the cost for each neighbour-subtree of the root. Finally, we just pick the root with the smallest total cost. If there is only one cut, we just try out every root in the subtree defined

by the cut. This doesn't have a negative effect on the running time, since there are only $\mathcal{O}(n)$ such cuts. For a detailed top-down description, see Algorithm 2

Algorithm 2 optimal 2-cut STT using "Knuth's trick"

Input: Search Space S with frequencies $f : V(S) \to \mathbb{N}$, set of cuts
Output: (c, r), cost c of a optimal 2-cut STT of the subtree defined by the cuts together with its root r.

1: **procedure** OptFromCuts(S, cuts)
2: Let $A = V(Sub_S(cuts))$ ▷ Subtree defined by the cuts(see Definition 12)
3: **if** $A = \{r\}$ **then** ▷ Base Case
4: **return** $(f(r), r)$
5: **if** cuts $= \{c_1, c_2\}$ **then** ▷ check if we have exactly two cuts
6: $c_1' \leftarrow c_1.ShiftTowards(c_2)$ ▷ move c_1 one step towards c_2 (if possible)
7: $c_2' \leftarrow c_2.ShiftTowards(c_1)$ ▷ move c_2 one step towards c_1 (if possible)
8: **if** $c_1 = c_1'$ or $c_2 = c_2'$ **then**
9: $allowedRoots \leftarrow P(c_1, c_2)$ ▷ path from c_1 to c_2
10: **else**
11: $(cost_1, r_1) \leftarrow$ OptFromCuts$(S, \{c_1', c_2\})$
12: $(cost_2, r_2) \leftarrow$ OptFromCuts$(S, \{c_1, c_2'\})$
13: $allowedRoots \leftarrow P(r_1, r_2)$ ▷ path from r_1 to r_2
14: **else**
15: $allowedRoots \leftarrow V(A)$
16: **for** $r \in allowedRoots$ **do**
17: $b_1, ..., b_t \leftarrow$ valid neighbours of r
18: **for** $i = 1, ..., t$ **do**
19: subtreeCuts $\leftarrow (b_i, r) \cup \{(v, w) \in$ cuts $\mid v$ is in the neighbour-subtree of $b_i\}$
20: $(cost_i, r_i) \leftarrow$ OptFromCuts$(S,$ subtreeCuts$)$
21: Let $C_r = \sum_{x \in A} f(x) + \sum_{i \in [t]} cost_i$
22: **return** (C_r, r) for r that minimizes C_r

5.2.1 Precalculation

For simplicity, we defined $A = V(Sub_S(cuts))$ in the pseudocode. However, we do not actually need to calculate this for every input. With some precalculation we can achieve constant time for most operations including calculating A. For any $a, b \in S$, we precalculate $N(a, b)$, which denotes the first node, that is on the path from a to b. To do this, we do a DFS for each node x and set $N(x, y)$ to the parent of y in the DFS-tree for every node y. Since we traverse for every node, this will take $\mathcal{O}(n^2)$ time in total. We also define $N(x, x) = x$. This information now allows us to do the following operations faster:

1. To calculate the path between two nodes, we can start with the first node and repeatedly follow the next node on the path, until we reach the second node. This takes $\mathcal{O}(n)$ time in total.

2. Shifting the cut takes only constant time. For two cuts $c_1 = (v_1, w_1)$ and $c_2 = (v_2, w_2)$ we can shift c_1 towards c_2 by setting $c_1 = (N(v_1, v_2), v_1)$.

3. Calculating the cuts for a neighbour-subtree of a root (Line 19) can also be done in constant time. To check if a node is in the subtree $Sub_S(\{c\})$ defined by exactly one cut $c = (v, w)$, we can use the following observation:

Observation 21. $N(w,x) = v \implies x \in Sub_S(\{c\})$

This is because there is always one unique path in a tree between two nodes. If the next node on the path from w to x is v, x must be somewhere in the subtree $Sub_S(\{c\})$. Otherwise there would be a circle in the tree.

To calculate Line 3 in constant time, we can just check if the cuts point outwards from the same node v, and the degree of v is 2.

Some additional precalculation also allows us to calculate the sum $\sum_{x\in A} f(x)$ from Line 21 in constant time (given $A = V(Sub_S(cuts))$). For this, we precalculate $F_{total} = \sum_{x\in A} f(x)$ and also $F_c = F_{total} - \sum_{x\in V(Sub_S(\{c\}))} f(x)$ for every possible cut c. Then, we can calculate the sum in the following way:

$$\sum_{x\in S} f(x) = F_{total} - \sum_{c\in cuts} F_c$$

Since there are only $\mathcal{O}(n)$ single cuts in the tree, we can precalculate again with a simple DFS for every cut in $\mathcal{O}(n^2)$ total time.

5.2.2 Running time

First, we can store the already calculated results, so that we do not have to recalculate them again. For this we could create a dictionary that stores the results indexed by the given cuts and add a simple check in the beginning of the procedure to read out already calculated results. Since the number of cuts in the list is never larger than two, this can be done for example by hashing. This means, the total running time cannot be larger than $\mathcal{O}(n^3)$, since there are only $\mathcal{O}(n^2)$ 2-cut subtrees on S. Since there is only a linear amount of work to do for each of these subtrees (checking maximum n possible roots), we have $\mathcal{O}(n^3)$ running time in total. Checking a root means running the inner loop in Line 18, i.e. calculating the cost for the root. Note, that we iterate through each child of each root we check, however the average degree in a tree is less than two. So, if we would check every node in the subtree defined by the cuts, we would need a constant amount of work on average.

However, Theorem 18 does not always help to improve the algorithm. For a simple star tree (same as in Figure 3), the worst case running time is still $\mathcal{O}(n^3)$, even if we only need to check a constant amount of roots for every set of cuts.

There are $\binom{n}{2} \in \mathcal{O}(n^2)$ possible ways to choose two cuts in this tree. For any of these pairs we have to consider the node t as the root, even with the improved algorithm. However, to calculate the total cost for this root, we have to consider the cost of every node v_i that is not part of the two cuts. Thus, we have to consider $n-2 \in \mathcal{O}(n)$ children for t. This is $\mathcal{O}(n^3)$ time on total.

Luckily, there is a simple fix for this problem. Instead of calculating the cost by iterating over all neighbours of the root, we could first calculate the total cost of all neighbour-subtrees of t and then subtract the cost of the missing branches. In our example, suppose we have (v_3, t) and (v_5, t) as a pair, we could subtract $f(v_3)$ and $f(v_5)$ from the total cost $\sum_{i\in[n]} f(v_i)$ and get the cost for all subtrees except the ones from the cuts. In this example, the calculation is trivial, however in the general case, it might happen that the root lies on the path between the cuts and is not directly part of the cut. In this case, assume we have a root r and the cut (v_1, w_1). We can then first subtract the cost of the optimal

2-cut STT for $Sub_S(\{(N(r, v_1), r)\})$, and then add back the cost of the optimal 2-cut STT for $Sub_S(\{(N(r, v_1), v_1), (v_1, w_1)\})$. Here, we remove the cost for the neighbour-subtree containing the cut and then add back the cost of the optimal 2-cut STT of the subtree between the root and the cut. This also works for two cuts by repeating the same process for both. The main idea now is, that we only need to calculate the total cost once and can then reuse it for future calculations, reducing the running time to a constant.

However, calculating the total cost is not easy, since we need to know the cost for each neighbour-subtree first. One might come up with the idea to calculate these costs recursively. This doesn't work however, since the neighbour-subtrees containing the cut can be arbitrarily large. Thus it might happen, that we have a recursive call on a subproblem larger than the original. To avoid this problem, instead of storing the total cost, we store the total cost we get when checking the root the first time, but only of the neighbour-subtrees, that do not contain one of the cuts. Formally, let C be the set of input cuts from the first recursive call, where r is considered as a root. Now, let N_C be the set of neighbours of r, that lead to a cut, so $N_C = \{x \in Neig_S(r) | \exists (v, w) \in C : N(r, w) = x\}$. We then store N_C together with the summed optimal cost of the neighbour-subtrees of r, that do not contain a node of N_C, inside a dictionary. For every subsequent call, we can check the cost of r in constant time, similar to the method described earlier by subtracting the branches we do not want to include and re-adding the optimal cost for the subtrees between the root and the cuts. Note, that we might need a few additional recursive calls, as we did not calculate the total cost but the total cost except up to two neighbour-subtrees. But this is not a problem, as we only have a constant amount of these calls. For more detail, consider the following pseudo-code:

Algorithm 3 Improving the cost-check for the allowed roots

```
1:  procedure OptFromCuts(S, cuts)
2:      RootOpts ← Dict()                          ▷ Empty dictionary storing costs for every node
3:      ...                                        ▷ See original algorithm (Algorithm 2)
4:      for r ∈ allowedRoots do
5:          N_C ← {x ∈ Neig_S(r) | ∃(v, w) ∈ cuts : N(r, w) = x}
6:          if RootOpts does not contain r then    ▷ first time looking at r
7:              b_1, ..., b_t = Neig_S(r) \ N_C    ▷ valid neighbours of r excluding N_C
8:              for i = 1, ..., t do
9:                  (cost_i, root_i) ← OptFromCuts(S, {(b_i, r)})
10:             RootOpts[r] ← (Σ_{i∈[t]} cost_i, N_C)   ▷ storing the result
11:         (CostPre, MissingBranchNeighs) ← RootOpts[r]   ▷ reading the result
12:         C_r ← CostPre + Σ_{x∈A} f(x)
13:         for each x ∈ MissingBranchNeighs do   ▷ Adding missing branches if needed
14:             if x not in N_C then
15:                 C_r += OptFromCuts(S, {(x, r)})
16:         for each (v, w) in cuts do
17:             cutNeigh ← N(r, w)
18:             if cutNeigh not in MissingBranchNeighs then
19:                 (cost_1, r) ← OptFromCuts(S, {(cutNeigh, r)})
20:                 C_r −= cost_1
21:             if cutNeigh ≠ w then
22:                 (cost_2, r_2) ← OptFromCuts(S, {(cutNeigh, r), (v, w)})
23:                 C_r += cost_2
24:     return (C_r, r) for r that minimizes C_r
```

We now argue, that checking a root takes constant amortized time. The first time we check a node x as the root takes $d_1 \cdot deg(x)$ time. Every subsequent check takes d_2 time, where d_1 and d_2 are some constants. Let t_x denote the number of times x is checked as a root. Then the total time for checking x is $d_1 \cdot deg(x) + d_2 \cdot (t_x - 1) \leq d \cdot (t_x + deg(x))$, where $d = \max(d_1, d_2)$. To get the total amortized cost, we sum over all nodes:

$$\begin{aligned}
\text{total cost} &\leq \sum_{x \in S} d \cdot (t_x + deg(x)) \\
&= d \cdot (\sum_{x \in S} t_x + \sum_{x \in S} deg(x)) \\
&= d \cdot (\sum_{x \in S} t_x + 2(n-1)) \qquad (7) \\
&\leq d \cdot \sum_{x \in S} t_x + 2 \\
&\leq 3d \cdot \sum_{x \in S} t_x \in \mathcal{O}(\sum_{x \in S} t_x) \qquad (8)
\end{aligned}$$

Eq. (7) follows, because S is a tree and the total sum of degrees in a tree is $2(n-1)$. We can conclude the last inequality (Eq. (8)), since every t_x must be at least one (every root gets checked when the set of cuts is empty). Since $\sum_{x \in S} t_x$ is the amount of root checks, we can conclude, that checking a root takes constant amortized cost.

However, the running time of the algorithm is still $\mathcal{O}(n^3)$ in the worst case, since we have not shown, that we only need $\mathcal{O}(n^2)$ root checks in total. And indeed, there exists such a counter example:

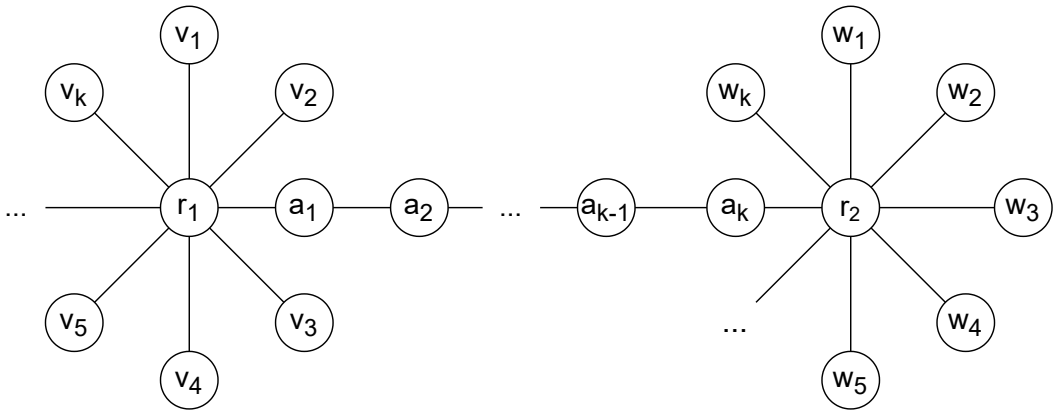


Figure 4: Counter example

Assume, that $f(r_1) = f(r_2) = 1$ and $f(x) = 0$ for every other node x. Consider all the pairs of cuts of the form, (r_1, v_i) and (r_2, w_j). There are k^2 ways of doing so. Since we have $n = 3k + 2$ number of nodes in this tree, in total we get $\mathcal{O}(n^2)$ such pairs. For each of these pairs, we then shift the left cut towards r_2 in our optimized algorithm, i.e. it becomes (a_1, r_1), no matter what cut we chose for the left star. Since r_2 is now the only remaining node with frequency 1, r_2 has to be the root of the subtree defined by the shifted left cut and the unmoved right cut. Similarly, if we shift back the left cut to the original position and instead move the right cut towards r_1 so that it becomes (a_k, r_2), the optimal root has

to be r_1. So even with Theorem 18, we have to consider roots $r_1, a_1, ..., a_k, r_2$. These are $k+2 \in \mathcal{O}(n)$ in total, which results in a running time of $\mathcal{O}(n^3)$.

5.2.3 improved analysis

As we have seen, the algorithm does not have $\mathcal{O}(n^2)$ running time in general. However, the original optimal binary search tree problem can be solved in $\mathcal{O}(n^2)$ time as shown by Knuth [6]. Since our algorithm in some sense generalizes this trick to tree search spaces, it can also run in $\mathcal{O}(n^2)$ time, if the input tree is a path. So there are cases, where the running time is better than $\mathcal{O}(n^3)$. We begin our improved analysis with the following lemma, which is analogous to the binary search tree case, but generalized for every path between two leaves.

Lemma 22. *Let l_1, l_2 be two leaves in S. Then the total amount of root-checks we need for pairs of cuts on the path between l_1 and l_2 is $\mathcal{O}(Dist_S(l_1, l_2)^2)$. This means, the amount of root checks needed for two cuts on the path is constant on average.*

Proof. Consider the nodes on the path between l_1 and l_2 given by $a_1, a_2, ..., a_k$ with $a_1 = l_1$ and $a_k = l_2$. Lets fix the distance between the cuts to d with $d > 0$. Remember, that given Theorem 18, for two cuts (a_{i+1}, a_i) and (a_j, a_{j+1}), we only have to check every node on the path from $R((a_{i+2}, a_{i+1}), (a_j, a_{j+1}))$ to $R((a_{i+1}, a_i), (a_{j-1}, a_j))$. If we sum over all cuts with distance d, we get a telescoping sum, which results in the following total cost:

$$\begin{aligned}
&\sum_{i=1}^{k-d-2} R((a_{i+2}, a_{i+1}), (a_{i+d+1}, a_{i+d+2})) - R((a_{i+1}, a_i), (a_{i+d}, a_{i+d+1})) \\
&= \sum_{i=2}^{k-d-1} R((a_{i+1}, a_i), (a_{i+d}, a_{i+d+1})) - \sum_{i=1}^{k-d-2} R((a_{i+1}, a_i), (a_{i+d}, a_{i+d+1})) \\
&= R((a_{k-d}, a_{k-d-1}), (a_{k-1}, a_k)) - R((a_2, a_1), (a_{d+1}, a_{d+2})) \qquad (9)
\end{aligned}$$

Since both of the roots from the last equation(Eq. (9)) are on the path from l_1 to l_2, their difference can only be of size k. If we now sum the total amount of root-checks needed classified by the distance between the cuts, we get:

$$\text{root checks} \leq \sum_{d=0}^{k-2} k \in \mathcal{O}(k^2)$$

Since there are also only $\mathcal{O}(k^2)$ number of ways to choose two valid cuts on the path, the number of roots to be considered for two given cuts is constant on average. □

Theorem 23. *The running time of the algorithm is in $\mathcal{O}(L^2 \cdot D^2)$, where L is the number of leaves in S and D denotes the diameter of S.*

Proof. We sum over all paths between leaves and use Lemma 22 to get:

$$\begin{aligned}
\text{root checks} &\leq \sum_{\substack{l_1, l_2 \in Leaves(S) \\ l_1 \neq l_2}} Dist(l_1, l_2)^2 \\
&\leq \sum_{\substack{l_1, l_2 \in Leaves(S) \\ l_1 \neq l_2}} D^2 \\
&= \binom{L}{2} \cdot D^2 \in \mathcal{O}(L^2 \cdot D^2)
\end{aligned}$$

However we didn't count the cases, where only one or zero cuts are given as input to the algorithm. There is a linear number of single cuts we can choose from S. For these we need $\mathcal{O}(n)$ time for the root check, which increases the total running time to $\mathcal{O}(L^2D^2 + n^2)$. It is possible to drop the n^2 term. This is because $D \cdot L \geq c \cdot n$, for some constant c. To show this, we can use the fact that the diameter is greater equal the average path-length between two leaves. If the leaves are given by $l_1, l_2, ..., l_L$, we can then conclude:

$$\begin{aligned} D &\geq \frac{1}{L^2} \sum_{i=1}^{L} \sum_{j=1}^{L} Dist_S(l_i, l_j) + 1 \\ &\geq \frac{1}{L^2} \sum_{i=1}^{L} n = \frac{1}{2L} n \end{aligned}$$

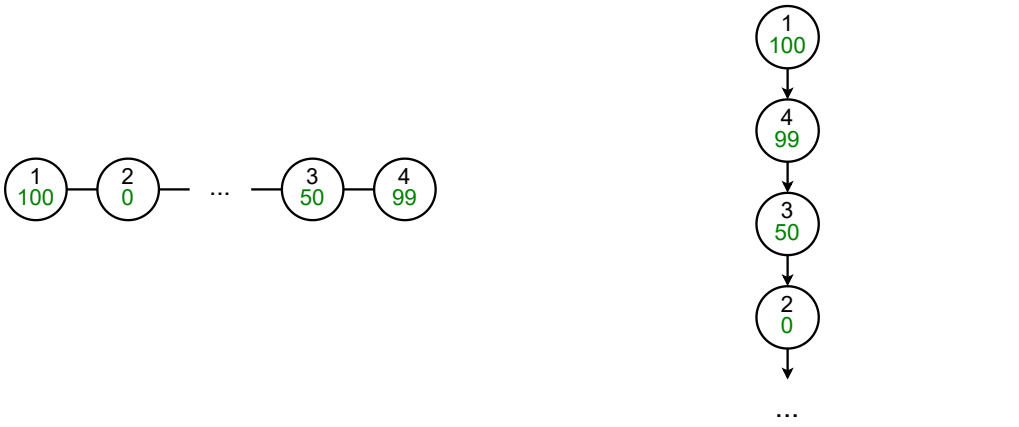
Then $D \cdot L \geq \frac{1}{2}n$, so $L^2 \cdot D^2 \geq \frac{1}{4}n^2$. So the running time is in $\mathcal{O}(L^2 \cdot D^2)$

□

If the search space is a path, this results in $\mathcal{O}(n^2)$ running time, which is the same as for the optimal binary search tree. So this algorithm can somewhat be seen as a generalization of Knuth's dynamic program. However, this bound is not optimal, since for a tree with $\mathcal{O}(n)$ diameter and leaves the algorithm will have $\mathcal{O}(n^4)$ running time, but we know that it is bounded by $\mathcal{O}(n^3)$. A better running time therefore is given by $\mathcal{O}(\min(L^2 \cdot D^2, n^3))$.

5.3 Knuth's trick further generalizations

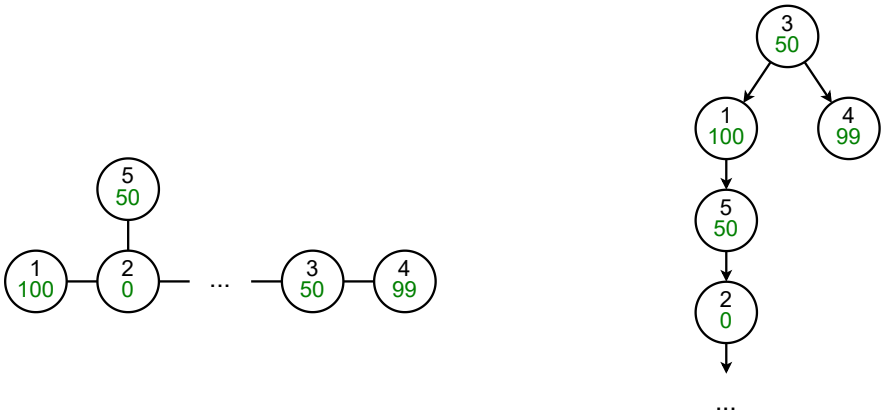
Knuth theorem does not generalize in the same way to optimal k-cut STT and general optimal STT's as it does to 2-cut STT's. For this, consider the search space in (Figure 5a). Here green numbers indicate the frequencies and black numbers the keys and the path indicated with "..." can be filled with an arbitrary amount of nodes with zero frequency. The optimal STT (unique for every node with non-zero frequency) with total cost 448 can be seen in Figure 5b



(a) search space (b) optimal STT

Figure 5: counter example together with optimal STT

Now assume we append another vertex with frequency 50 to the node with key 2 like in Figure 6a. The optimal STT (again unique for every non-zero frequency node) with cost 598 can be seen in Figure 6b



(a) search space with appended vertex (b) optimal STT

Figure 6: adapted search space together with optimal STT

This means that the root of a optimal STT is forced to change to 3 when vertex 5 is added. However 3 is not on a path from 1 to 5, which means that the root shifts into a sidebranch and can theoretically move away arbitrary from the original root.

It is still an open question if different kind of generalization are possible. For example, the root might not have to move away from the original root, while the original root remains on the path from the new root to the added node. We did not find a counter example for this yet, but the proof we used to show Theorem 18 does not seem to work in the same way, even if we only consider the case of appending a single vertex.

6 Experimental results and implementation

To validate the theoretical results with empirical evidence, we also implemented all algorithms for finding optimal 2-cut STT's described in this thesis. The code is written in python and to access the full implemenation, see [11]. First, a datastructure for storing input trees is given, which allows the following important operations:

1. *GetPathDirection(self, a, b)* calculates the next node on the path from a to b. For efficiency, results are stored in a private field called *pathDirects*.

2. *GetPath(self, a, b)* returns the path from a to b using the GetPathDirection function.

3. *IsInComponent(self, a, cuts)* calculates, if node a is inside the subtree defined by a given list of cuts. This function also uses the GetPathDirection function for efficiency.

4. *GetNodesFromCuts(self, cuts)* calculates all nodes inside the subtree defined by a given list of cuts. For efficiency, results are stored in a private variable called *cutNodes*, but only if the variable "cuts" contains one cut.

5. *GetFrequenciesFromCuts(self, cuts)* gets the total sum of frequencies in the subtree defined by a list of cuts. For efficiency, this function uses a similar method as described in Section 5.2.1.

Note, that we defined the datastructure in a dynamic way by storing previously calculated results interally. That way we do not need to do any precalculations manually. To visualize trees efficiently, we used a open source graph visualization software called "graphviz" (see [9]).

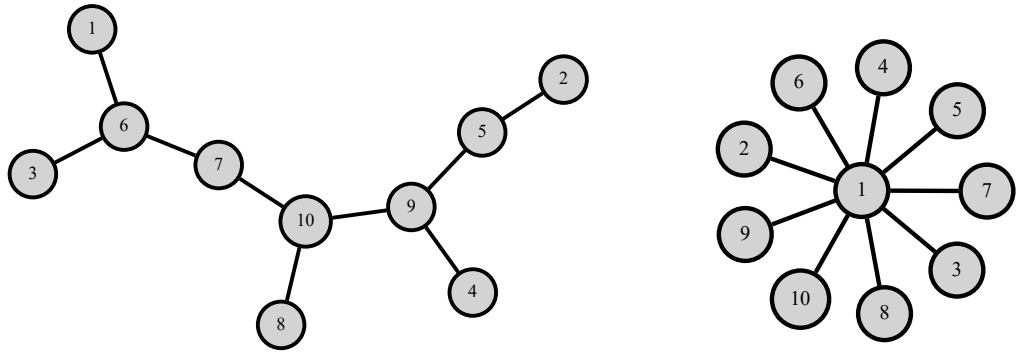
Further, five algorithms are given. All of them take a search space S as input and return the optimal cost together with the optimal tree.

1. *OPT2CutSTTBasic* calculates the optimal 2-cut STT without using any performance improvements.
2. *OPT2CutSTTKnuth* implements the algorithm described in Algorithm 2.
3. *OPT2CutSTTBest* implements the algorithm described in Algorithm 3.
4. *OPTKcutSTT* takes additional parameter k and calculates an optimal k-cut STT.
5. *ExactExponential* calculates an optimal STT on S. This algorithm first calculates the number of leaves L of S and calls the *OPTkcutSTT* algorithm with $L = k$, as suggested in Section 4.2.

6.1 Generating trees

To generate random trees, so called prüfer-codes where used[12]. A prüfer-code is a sequence of $n-2$ numbers, where each of the number in the sequence is in $\{1, ..., n\}$. According to prüfers original proof[4], this sequence can determine an unique labeled tree of size n. The prüfer-code for a given labeled tree can be obtained in the following way: Find the leaf with the smallest label and remove it. Append the label of the node it was attached to the prüfer-code. Repeat until there are only two nodes left. To generate trees from a given prüfer-code, we used the heap variation as described in [12], which takes $\mathcal{O}(n \log n)$ time.

This is a convenient way of generating trees for testing the performance of the proposed algorithms, since each tree has a unique prüfer-code. To generate random trees, we can just generate a random prüfer-code and build the corresponding tree. Also, to create a star tree of size n we can just use a prüfer-code only consisting of the same letter. In Figure 7 two trees generated from a given prüfer-code can be seen (visualized by graphviz).



(a) prüfer-code (6, 5, 6, 9, 9, 7, 10, 10) (b) prüfer-code (1, 1, 1, 1, 1, 1, 1, 1)

Figure 7: trees generated from a prüfer-code

These trees can then be used as inputs for the implemented algorithms. In Figure 8 you can see the output of *OPT2CutSTTBest* for the tree from Figure 7a. The green numbers indicate the used random frequencies.

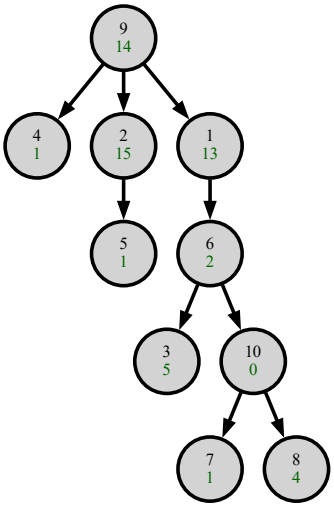


Figure 8: An optimal 2-cut STT with cost 126

6.2 Testing setup

In the implementation a test function is given, which executes a set of algorithms on random trees with increasing number of nodes while measuring the time each algorithm takes. Also, the function compares the resulting costs calculated by the algorithms and throws an error, if they differ. This can reveal incorrect implementations. Time is measured using the *time.time*() function from the standard python library "time" and all tests where performed on a 12th Gen Intel(R) Core(TM) i5-12400F 2.50 GHz cpu on a system with 16,0 GB of ram. The parameters of the test function are defined as follows:

1. *algorithms* defines a list of algorithms used for testing.
2. *treefunc* is a function that generates a random tree of a certain type. This function takes a parameter n and is supposed to generate a tree with n nodes. This function can be customized by the user.
3. *min_ n* and *max_ n* denote the minimum and maximum number of nodes that will be tried during testing
4. *stepsize* describes how much we increase n in each step
5. *samples* describes how many random trees we generate in each step to test the algorithms. The algorithms are tested for the same set of random trees in each step. At every step the average time each algorithm takes is measured and stored.
6. *plot* and *latexTable* are truth values and are used for visualizing the test results. If *plot* is set to true, the function will show the plot using the matplotlib library. If *latexTable* is set to true, the function will generate a latex table containing the test results.

6.3 Test results

We used the test function described in above section on trees generated by random prüfer-codes, on star trees from Figure 3, the tree from the counter example from Figure 4 and others. For all tests we used random values from between 0 and 100 as search frequencies.

The first test was a basic performance test for all algorithms on trees for random prüfer-codes (Section 6.3). In both tests we used $min\_n = 10$, $max\_n = 200$, $stepsize = 10$ and $samples = 50$. One interesting observation is, that the optimized versions seem to take around $\mathcal{O}(n^2)$ time on average for each step, while the naive implementation takes $\mathcal{O}(n^3)$ time. However this observation depends on the distribution of trees generated by random prüfer-codes. For example, a star tree is very unlikely to be generated, as there are only n possible prüfer-codes. However, a path tree is much more likely, since there are $\mathcal{O}(n!)$ possible prüfer-codes. And we know from previous discussions, that path trees are generally good inputs for the optimized algorithms. One could also try out different distributions, for example adding nodes to the tree one by one and attaching them somewhere random. Another idea is to generate from a uniform distribution over all unlabeled trees. However, there seems to be no easy way to do this.

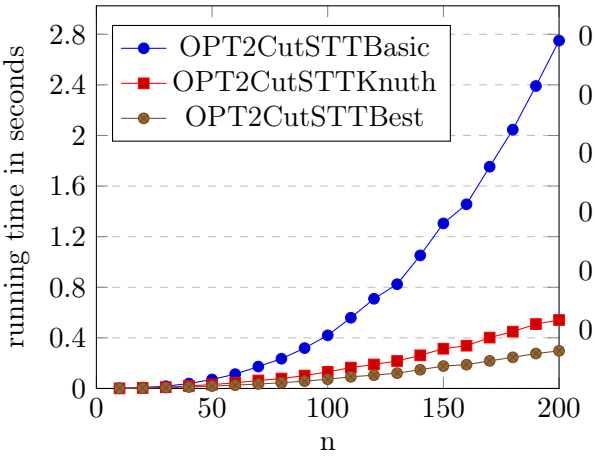


Figure 9: random trees

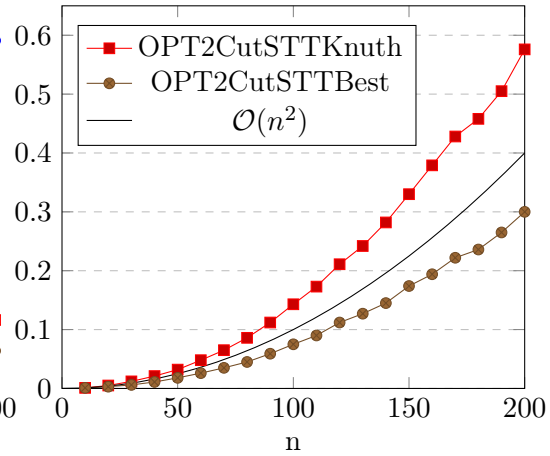


Figure 10: random trees improved

We proceeded to test out different kinds of star trees. For the normal star tree (see Figure 3) we used $min\_n = 10$, $max\_n = 200$, $stepsize = 10$ and $samples = 10$ as parameters for the testfunction. For the tree from Figure 4 (let us call it "star path tree") we tested values for k from 10 up to 67. One can observe, that the running time for the normal star tree is much better for the algorithm with improved cost-check (see Figure 11). As expected, for the star path tree the performance only seems to be better by a constant factor (Figure 12). This validates our previous findings.

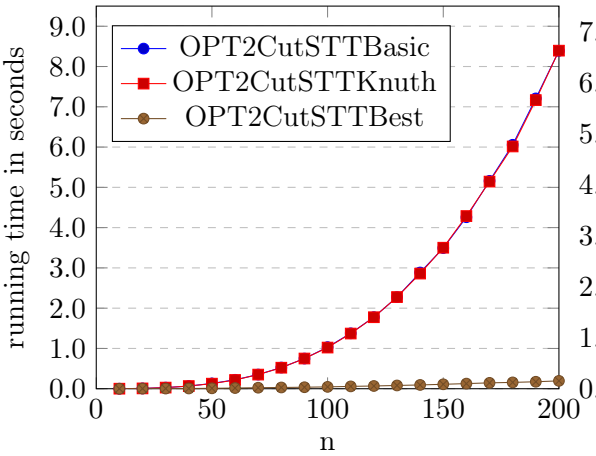


Figure 11: random star trees

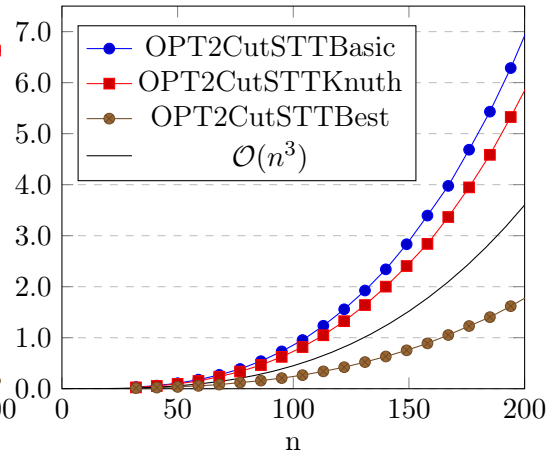


Figure 12: random star path trees

For the last test we generated random binary trees and random paths as input. We used $min\_n = 10$, $max\_n = 200$, $stepsize = 10$ as parameters with $samples = 10$ for the path and $samples = 20$ for the binary trees. As expected, the running time of the basic algorithm is much worse than the improved algorithms for the path trees (Figure 13). Surprisingly, the running time also seems to be really good for the binary trees (Figure 14). Using our previous analysis, the running time should be around $\mathcal{O}(n^2 \cdot \log^2 n)$, since a binary tree has $\mathcal{O}(n)$ leaves and a diameter of $\log(n)$. The implementation seems to be slightly better asymptotically (see Figure 15).

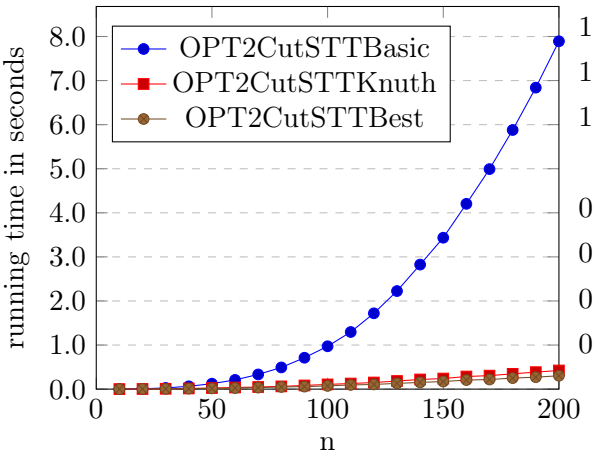


Figure 13: random paths

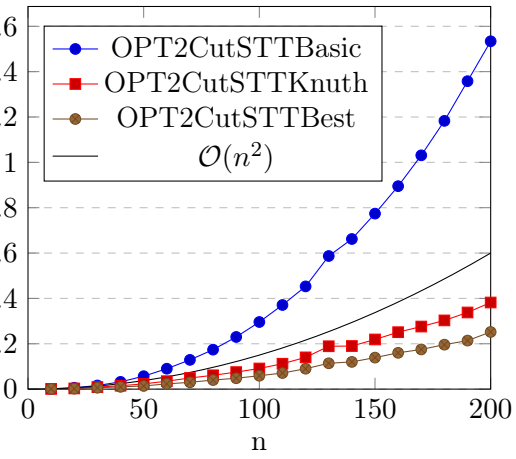


Figure 14: random binary trees

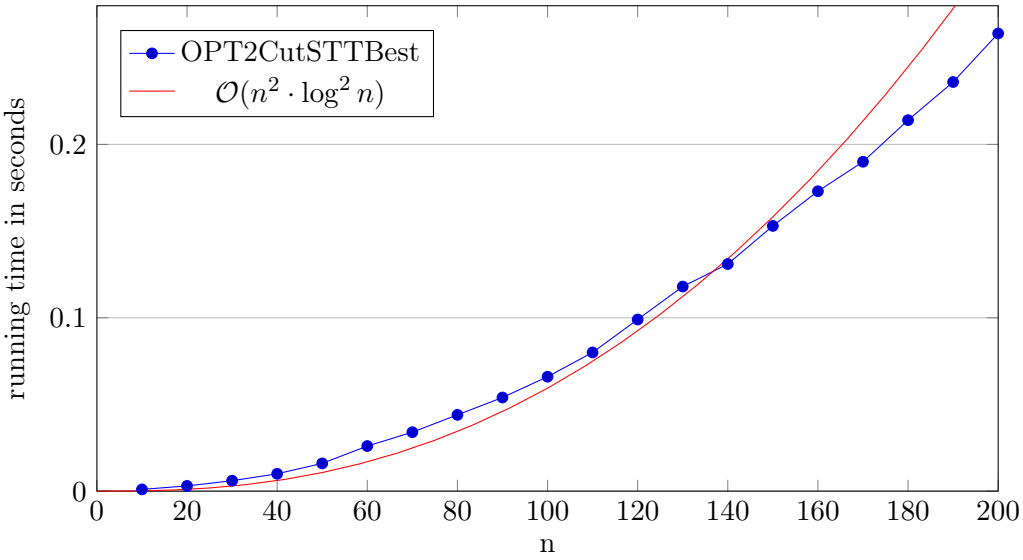


Figure 15: comparison with theoretical bound for random binary trees

References

[1] Benjamin Aram Berendsohn, Ishay Golinsky, Haim Kaplan, and László Kozma. *Fast approximation of search trees on trees with centroid trees*. 2022. DOI: 10.48550/ARXIV.2209.08024. URL: https://arxiv.org/abs/2209.08024.

[2] Benjamin Aram Berendsohn and László Kozma. "Splay trees on trees". In: *CoRR* abs/2010.00931 (2020). arXiv: 2010.00931. URL: https://arxiv.org/abs/2010.00931.

[3] Jean Cardinal, Arturo Merino, and Torsten Mütze. "Efficient generation of elimination trees and graph associahedra". In: *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 2128–2140. DOI: 10.1137/1.9781611977073.84. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611977073.84.

[4] Arthur Cayley. *A theorem on trees*. Vol. 23. Cambridge University Press, 1889, pp. 376–378. DOI: 10.1017/CBO9780511703799.010.

[5] Michael L. Fredman. "Two Applications of a Probabilistic Search Technique: Sorting X+Y and Building Balanced Search Trees". In: *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*. STOC '75. Albuquerque, New Mexico, USA: Association for Computing Machinery, 1975, pp. 240–244. ISBN: 9781450374194. DOI: 10.1145/800116.803774. URL: https://doi.org/10.1145/800116.803774.

[6] D. E. Knuth. "Optimum binary search trees". In: *Acta Informatica* 1.1 (1971), pp. 14–25. ISSN: 1432-0525. DOI: 10.1007/BF00264289. URL: https://doi.org/10.1007/BF00264289.

[7] Lawrence L Larmore. "A subquadratic algorithm for constructing approximately optimal binary search trees". In: *Journal of Algorithms* 8.4 (1987), pp. 579–591. ISSN: 0196-6774. DOI: https://doi.org/10.1016/0196-6774(87)90052-6. URL: https://www.sciencedirect.com/science/article/pii/0196677487900526.

[8] Kurt Mehlhorn. "Nearly optimal binary search trees". In: *Acta Informatica* (Dec. 1975). DOI: 10.1007/BF00264563. URL: https://doi.org/10.1007/BF00264563.

[9] *open source graph visualization software Graphviz*. URL: https://graphviz.org/.

[10] *Optimal binary search trees*. URL: https://en.wikipedia.org/wiki/Optimal_binary_search_tree.

[11] Johannes Voderholzer. *optimal 2-cut STT's implementation*. Accessed: February 28, 2023. 2023. URL: https://git.imp.fu-berlin.de/voderhoj00/optimal-2-cut-stts.

[12] Xiaodong Wang, Lei Wang, and Yingjie Wu. "An Optimal Algorithm for Prufer Codes". In: *JSEA* 2 (Jan. 2009), pp. 111–115. DOI: 10.4236/jsea.2009.22016.