Optimal Path-Decomposition of Tries

by

Alexandre Daigle

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2016

Abstract

In this thesis, we consider the path-decomposition representation of prefix trees. We show that given query probabilities for every word in the prefix tree, the *heavy-path* strategy produces the optimal trie with respect to the number of node accesses. We show how to implement the *heavy-path* strategy in $\mathcal{O}(N)$ time for a trie containing n words with total length N. To prove this result, we show a complete characterization of the choices made by the optimal decomposition strategy. Using this characterization, we describe how to efficiently support dynamic operations on the path-decomposed trie while preserving the optimality in $\mathcal{O}(\sigma |w|)$ time for an alphabet size of σ and a word length of $|w|$. We also give entropy-based bounds of the node accesses per query for their respective probabilities. Finally, we show theoretical and experimental results on the performance of *heavy-path* versus *max-score*, another popular path-decomposition strategy.

Acknowledgements

I would like to thank my supervisors Professor J. Ian Munro and Professor Alejandro López-Ortiz for guidance. I would also like to thank my colleague and friend Oliver Grant for many discussions and improvements on the thesis. And finally, I would like to thank my readers Professor Timothy M. Chan and Professor Therese Biedl.

Table of Contents

List of Tables

List of Figures

Nomenclature

$\mathcal{W}$	Set of words in the dictionary.
$\Sigma_{\mathcal{W}}$	The set of letters from the alphabet.
$n, \lvert\mathcal{W}\rvert$	The number of words.
$\sigma, \lvert\Sigma_{\mathcal{W}}\rvert$	The size of the alphabet.
N	The sum of the length of all words, i.e. $\sum_{w \in \mathcal{W}} \lvert w\rvert$.
$\mathcal{W}_p$	Subset of words of $\mathcal{W}$ starting with the prefix p.
ε	The empty prefix.
$\lvert w\rvert$	Length of the word w with $w \in \mathcal{W}$.
$f(w)$	Weight of the word w.
$f(\mathcal{W}_p)$	Weight of all words in $\mathcal{W}_p$, i.e. $\sum_{w \in \mathcal{W}_p} f(w)$.
$depth(w, \tau)$	The depth of a word w in a path-decomposed trie τ.
$\mathcal{P}(w)$	The set of prefixes of the word w, including the empty prefix ε.
$\mathcal{P}(\mathcal{W})$	The set of all prefixes of the words in $\mathcal{W}$, i.e. $\bigcup_{w \in \mathcal{W}} \mathcal{P}(w)$.
$\arg\max\limits_{s \in S} f(s)$	Function that returns the argument $s \in S$ having the highest valuation of $f(s)$.
$k\text{-}\arg\max\limits_{s \in S} f(s)$	Function that returns the set of the k arguments $s \in S$ having the highest valuation of $f(s)$.

Chapter 1

Introduction

Auto-complete or word completion is a feature that has many applications on mobile phones, in search engines and in text editors. The idea is that the software can predict the word that the user wants to write from the first few letters typed. Data structures have a preponderant influence on the efficiency of the completions and the fine tuning of those structures can have important impacts on the performance. Our interest in this thesis is in a specific representation of tries to allow fast word completion.

A *trie* is a digital tree that represents a classical data structure used for this problem, the prefix tree. There is a wealth of research focusing on various trade-off of compactness [5, 12, 13, 20, 22] and time complexity [3, 13, 17] of tries in various representation. In particular, we can name radix trees, Patricia trees, compressed prefix trees, ternary search trees or even suffix trees which are all data structures to solve similar word completion problems.

One representation of tree and graph structures that has been successful for improving compression and inferring meaningful structure from the representation is *path decomposition* [11, 13, 14] (or *cycle decomposition* in graphs [4, Chapter 2.8]). The idea is to convert a path in the original tree into a single node and recurse in each subtree dangling from this path. The structure becomes entirely different, but when a path has a meaningful representation, it helps reduce the number of nodes and regroup the relevant information together. In the setting that we are interested in, a path in a prefix tree represents a complete word and is thus providing a relevant structure to our path-decomposed representation.

A different approach in data structures is to optimize the layout of the internal structure based on the usage pattern. Moreover, if we have some idea on the access probabilities of the nodes in the tree, we can utilize this information to improve the structure for those access probabilities. The access pattern can be known a priori, estimated from a probability distribution or discovered upon requests for instances. In the context of text documents for example, the word frequency is known to follow a power-law distribution [18, 19] and we can use this information to optimize the layout of the data structure.

We apply these two ideas, path decomposition and optimizing the internal layout of data structures, together to produce optimal path-decomposed tries, for some definition of optimality based on the probability of accessing a word from the corpus.

1.1 Main Contribution

Our main contribution is to describe how to compute the optimal path-decomposed trie structure for given words and probabilities of access. We show that the well-known *heavy-path* decomposition strategy is the optimal layout and can be computed in $\mathcal{O}(N)$ time for a word count of n and a sum of length of all words of N. Previous results only guaranteed logarithmic depth of the trie when using the *heavy-path* strategy. We also prove a characterization of this optimal layout that we use to allow dynamic operations in $\mathcal{O}(\sigma |w|)$ time for a word of length $|w|$ in the path-decomposed trie while conserving the optimality condition. Finally, we provide upper and lower bounds on the optimal cost of the trie based on the entropy of the probability distribution and we compare both theoretically and practically the performance of the optimal path-decomposed trie to those created by the *max-score* heuristic.

1.2 Thesis Outline

In chapter 2, we present some background on trie and path-decomposition as well as background on optimal layout of these data structures. We formally define the path-decomposed tries and the notation that we use. We also present previously used strategies for the path selection, namely *heavy-path* and *max-score* and show the justification of their usage in their respective contexts.

We then present the various criteria of optimality that we consider in this thesis in chapter 3. We show a naïve solution to compute the optimal path decomposed trie for a set of words and their access probabilities. Next, we improve the construction of the optimal trie by a factor of up to n in some cases through a theorem showing locality of change for certain operations on path-decomposed tries.

In chapter 4, we give a complete characterization of the optimal choices to construct the optimal trie for the expected number of node accesses. This characterization leads to the optimality of the *heavy-path* strategy. We also show how to maintain the optimality of the trie for dynamic operations in $\mathcal{O}(\sigma \log n)$ time. Moreover, we provide bounds of the cost function based on the entropy of the distribution.

Finally, in chapter 5 we compare the theoretical performance of the optimal path-decomposed trie with a suboptimal strategy to get an approximation ratio. Furthermore, we give experimental results on the practical performance using an implementation of the heuristic with respect to the optimal. We use corpuses such as the English dictionary and protein sequences with various probability distributions.

In chapter 6 we present our conclusion and some ideas for future work.

Chapter 2

Background and Related Work

In this chapter, we review the problem of word completion, define it formally, and explain the structure that we improve upon in this thesis. We also describe common techniques to construct the path-decomposed trie data structure and how to execute queries on this structure.

2.1 Problem Statement

The problem of word completion is to give a list of complete words that are completions of a prefix given as an input. The list can contain one or many completions and the length of the list is often given as a parameter of the query. The result is a deterministic outcome of the highest probabilities of all the completions of the prefix. We consider the problem without any context to the surrounding text. This is a simplification, but we argue that the context-aware version is heavily based on the non-aware version and therefore, our work could be extended to more general models. An example of completion query with the empty word ε as a prefix input and a requested list length of one would be the most common word in the dictionary—for English that would be the word '*the*'. The completion of the prefix '*b*' with the English corpus would be the word '*be*'.

When the number of completions to return is specified, the problem is named *top-k completions*, where k is the number of results to return. We usually refer to this generalization as *completion queries* as they cover most use cases of auto-completion queries with various values of k. In order to simplify the relative probabilities of each word, we assign them a numeric *weight* that we use to determine the most likely completion. The weights can be converted to probabilities by a simple normalization.

2.2 Path-Decomposed Tries

The path-decomposed trie is the data structure that is the primary concern of this thesis. In particular, we improve the internal layout of its nodes. We first explain prefix trees and then apply a path-decomposition strategy to obtain a corresponding path-decomposed trie.

Prefix trees, whose structure was invented by de la Briandais [1959] referring to each level of the tree in terms of a table, are a hierarchical tree data structure in which the root represents the empty word and each edge stores a character or sequence of characters leading to a different node. Words are formed by the concatenation of those characters along a path to a leaf node. Any such path represents a word in the corpus. There can only be one edge per node with the same first character so that for any represented word, there is only one path to its corresponding leaf. Prefix trees or *tries* are very useful to represent dictionaries and various refinements to this data structure—whether more space efficient or faster—have been explored over the years.

Definition A *Path-Decomposed Trie* (PDT) is a trie constructed by transforming a prefix tree. We select a first complete path, from root to leaf, and transform this path into a single node (containing the concatenation of the characters of the path in the prefix tree). We repeat the same process for each subtrees branching off that path recursively and link those with an edge to the parent node in the PDT.

Various ordering strategies can be used for the children of a node. If using the representation of Grossi and Ottaviano [13], which encodes the trie topology using DFUDS[1] on a balanced parenthesis data structure, the children are listed from the top to the bottom of the path using the lexicographic ordering as the tie-breaker. An example of the first step of a path-decomposition is depicted in figure 2.1.

A path decomposition is completely defined by the strategy used to choose the path in the prefix tree. We present two strategies for path decomposition in sections 2.2.1 and 2.2.2. A third example of strategy could be to select the left-most node of the prefix tree which leads to a lexicographic depth-first ordering of the words of the original prefix tree. We use the following terminology to denote the edges of the path-decomposed trie.

[1] *Depth-first unary degree sequence* [2] is a representation of trees that enumerates the nodes by depth-first traversal where at each node, it appends an opening parenthesis for each child followed by a single closing parenthesis. The resulting sequence turns out to be balanced by prepending an extra opening parenthesis.

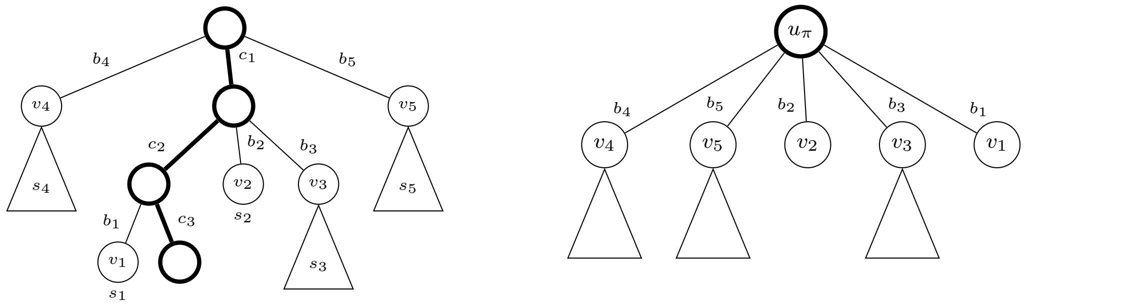


Figure 2.1: On the left, a prefix tree with the decomposition path π highlighted. On the right, the corresponding root node u_π of the path-decomposed trie after the first step. The node u_π represents the word $c_1 c_2 c_3$, the concatenation of characters along the path π.[2]

Definition A *branching position* of a node (or word, as there is a one-to-one mapping) is the smallest index of the character of its word at which another word branches out. The *branching character* is the character that causes the branching and is the first non-matching character of the branching word. A *branch* is the combination of a branching position and a branching character and leads to a subtrie.

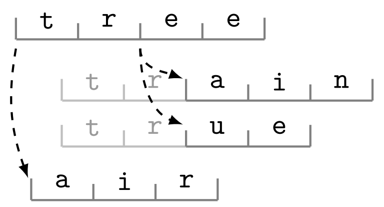


Figure 2.2: The nodes of the words '*train*' and '*true*' branches from the 3rd branching position of the root with branching characters '*a*' and '*u*'. '*air*' branches at the 1st position with the character '*a*'.[3]

With the previous definition, we can impose an ordering on the branches of our PDT which consists of the numerical ordering of the branching positions with tie-breaking on an arbitrary, but fixed, ordering of the branching characters. For the remainder of this thesis, we use the alphabetical order of the Latin alphabet as the tie-breaking rule. The same character can only occur at most once per branching position, making it a total order.

Since PDTs are built upon prefix trees, they inherit one important structural property that we formulate as the following invariant.

Property (Invariant) In a path-decomposed trie, a word belongs in a subtrie if and only if it shares a common prefix up to the branching position of that subtrie, including the branching character.

In other words, the invariant ensures that words sharing a common prefix also share a common subtrie. The property is true independently of the strategy used for path-decomposition.

[2]Figure and notation are inspired by Grossi and Ottaviano [13].

[3]Note that the branching character is usually stored in the edge leading to the next word.

Given a prefix query, we denote the first node matching the prefix from the root as the *locus node*. This notation is commonly used for discussing enumeration algorithms. For instance, in the trie matching the configuration of the words pictured in figure 2.2, the locus node of '*t*' or '*tr*' is the root node but the locus node of '*tra*' is the node matching the word '*train*'.

We now explore two common path-decomposition strategies used in the literature.

2.2.1 ***Heavy-Path*** Decomposition

The *heavy-path* decomposition is a greedy strategy that picks the subtree in the prefix tree with the highest total weight recursively until reaching a leaf node. The resulting path is named the *heavy path*, and the corresponding word is used as the root of the PDT. The same strategy is then used for each subtree branching from the selected path. Each of those dangling subtrees corresponds to a subrange of words and the strategy selects a local root from the heavy path of that subrange.

This strategy has been explored in [13] and results in a trie with height bounded by $\mathcal{O}(\log n)$. The strategy is also called the *centroid path decomposition*.

aa	2
aaa	1
aaab	1
ab	2
ba	3
baa	1
ca	1

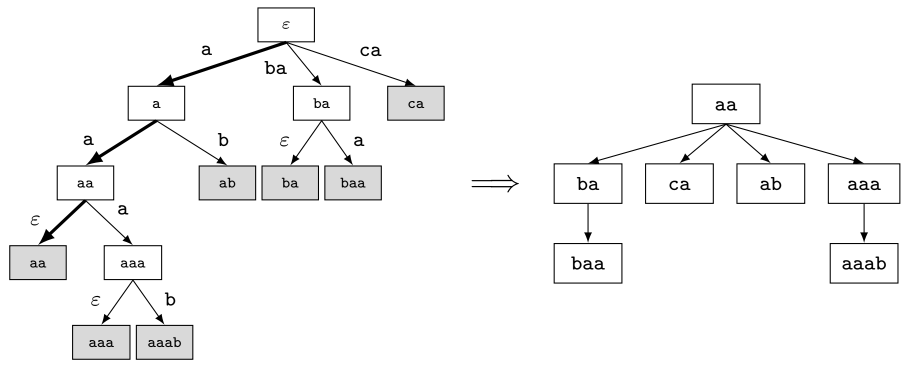


Figure 2.3: In the table on the left, the list of words with their associated weights. In the centre, the corresponding prefix tree with leaves filled in light gray. The concatenation of edge labels is written in the nodes. On the right, the path-decomposed trie created using the *heavy-path* strategy. The first heavy path is highlighted in bold in the prefix tree.

The above figure is an example of a trie created with the *heavy-path* strategy. We describe the selection process for the first heavy path. The heaviest branch from the root of the prefix tree is the one labeled '*a*' as its total weight is $2+1+1+2=6$, we thus follow its edge and re-evaluate the heaviest branch from there. The next heaviest branch is then '*a*' again with a total weight of $2+1+1=4$. The last choice is tied with a weight of 2 for either branch ('ε' or '*a*') and we arbitrarily select the first one: 'ε'. The path (represented by thicker edges in the figure) thus represents the word '*aa*' and is placed at the root. There are then four subtrees branching from that initial path corresponding to the set of words: {'*aaa*', '*aaab*'}, {'*ab*'}, {'*ba*', '*baa*'} and {'*ca*'} and we execute the *heavy-path* strategy on each of these subsets to get the final path-decomposed trie on the right.

2.2.2 *Max-Score* Decomposition

The *max-score* decomposition is also a greedy strategy that picks the word with the highest weight as the root for each subrange. Ties can be broken arbitrarily. Figure 2.4 shows the result of the max-score strategy applied to the same set of words of figure 2.3. The first choice is thus obviously '*ba*' as its cost is the highest with a weight of 3 and is placed at the root. In the branch with words starting with '*a*', we arbitrarily select '*aa*' as it is tied with '*ab*' for the highest weight and we continue processing the tree in the same fashion.

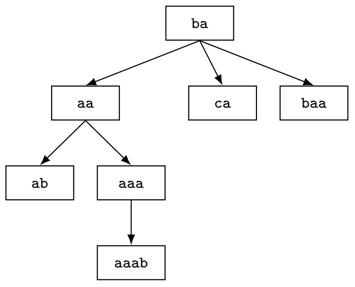


Figure 2.4: The *max-score* PDT using the word set of figure 2.3.

This strategy is analyzed in [14] as it has the appealing property that the locus node of a prefix **is** the most likely completion of that prefix. The paper explores the compact representation of the trie and its performance in both space and time in contrast to two other data structures: completion tries and RMQ tries[4].

2.3 Top-k Completions Enumeration

In this section, we show how we can implement completion enumeration queries. We first comment on the requirements of the query operation and then present two different ways to support such queries.

When searching for a prefix in a PDT, we find the locus node u whose subtries contain all possible words starting with the queried prefix. The query then consists of reporting the k words with the highest weights in the subtries branching after the end of the queried prefix in u. In general, the locus node does not have the highest weight of the subtrie, but

[4]RMQ tries use data structures designed to handle *range minimum queries* in constant time [9]. To support completion queries, we let the 'minimum' be the **highest** score and we get the k highest using a priority queue and exploring the left and right subranges of the highest score recursively. For full detail, we refer to the paper which gives pseudo-code of the top-k completion operation on RMQ tries.

one can choose a path-decomposition strategy to have this property (such as *max-score* from section 2.2.2). A path-decomposed trie for which the locus node of any prefix is the top-1 completion is said to have the *heap property*[5]. If the strategy does not enforce the heap property, we can avoid an exhaustive search of the words in the subtries by storing additional information in the nodes. We present two techniques: one to use in the general case and one to use with the heap property.

1. **Maximum weight in edges.** We first need to store the maximum weight in each subtrie in the edge leading to that subtrie. With this extra information, it becomes easy to follow the path containing the heaviest element as we simply follow the edge that has the biggest weight at each node. The algorithm to enumerate the k heaviest completions of a prefix is thus the following:

    a. Find the locus node of the queried prefix.

    b. Create a priority queue with a capacity of k elements and insert every edge branching from the locus node using the maximum weight as the ordering key.

    c. Remove the heaviest element of the queue and shrink its capacity by one.

    d. Follow that heavy edge of the element just removed and the following heavy edge recursively to its corresponding word (or any word having this heaviest weight). Along the way, keep adding to the priority queue every edge that isn't recursed into. Whenever the priority queue is full we replace its lowest value by the new element if the new element is bigger[6].

    e. Return the word at the current node and re-execute step c. to e. until the k heaviest words are returned.

    The algorithm can be implemented in $\mathcal{O}(kfd \log k)$ time for a maximum fan-out of f and a maximum depth of d. As k is likely to be small, f is bounded by the alphabet size and the length of the words, and the depth d can be kept small either in average case or worst case by a proper choice of decomposition strategy, we consider this time bound to be reasonably good to answer queries. The main benefit of this approach is the lack of explicit dependency on n—although the depth can still be bounded by some function of n in the worst-case.

[5]All nodes are greater than or equal to (or less or equal for a min-heap) each of its children, according to a comparison predicate defined for the heap [25].

[6]To find the lowest element in a priority queue implemented as a max-heap, we can store an additional min-heap with mutual pointers from identical edges in each heap. The min-heap helps us find the lowest edge quickly and we simply need to synchronize the heaps when executing operations.

2. **Heap enumeration.** Alternatively, if the strategy produces a trie with the heap property, we can find the locus node, return its word, and add its children in a priority queue ordered by weight. We then iteratively take the highest element of the priority queue and add its children. This strategy is taken from a previous paper [14] and has a running time of $\mathcal{O}(lk \log(lk))$ where l is the average length of the completions returned minus the prefix length $|p|$.[7]

In both cases, the sum of the depth of each accessed word is an upper bound to the number of pointers accessed. In the first case, the bound is tight for top-1 completion, but any additional completion restarts the search from the search path instead of the root. The second case uses the heap property so that top-k completions only use pointers to find the locus node and then use $k-1$ additional pointers to list the completions, assuming that the weights are stored in the edges and do not require pointer accesses.

[7]Note that this technique assumes constant-time implementation of many succinct data structures.

Chapter 3

Optimal Path-Decomposed Tries

With the previously described structure of PDTs, we are interested in selecting the path-decomposition strategy that produces an *optimal* representation. We use the expected number of node accesses in the trie as the objective function to minimize since it is closely related to the expected cost of the queries. The number of node accesses is bounded above by the depth of each completion and we also use this metric as an approximation. Throughout our research, we see that this definition entails consequences that further encourage this choice.

3.1 Cost Function

In order to provide a well-defined cost function, we give each word a *weight* such that the normalized weights represents the probabilities of writing each word with no context awareness. We assume the weights to be static, but we look into dynamic operations on the trie in chapter 4. For a word set $\mathcal{W}$, we denote by $f\colon \mathcal{W} \to \mathbb{R}^+$ the function mapping words to their associated weight. Let $\mathcal{T}_\mathcal{W}$ be the set of all possible path-decomposed tries for the words in $\mathcal{W}$; we use the function *depth*: $\mathcal{W} \times \mathcal{T}_\mathcal{W} \to \mathbb{N}$ to map the depth of a word in a particular trie[1]. With these notations, we define various cost functions formally.

Definition The *word-optimal* strategy is to minimize the depth of each word weighted by its frequency. The cost function of the word-optimal strategy is:

$$c(\mathcal{W}, \tau) = \sum_{w \in \mathcal{W}} f(w) \cdot \mathit{depth}(w, \tau). \tag{3.1}$$

[1]We assign a depth of 1 to the root instead of 0 as it simplifies the cost function.

We choose to measure the optimality in terms of the depth of a node in the PDT even though more than a constant number of operations can occur in a single node. We observe that the operations in a single node are done in consecutive memory (i.e. in a string corresponding to a word), whereas following an edge to a different node is likely to incur a cache miss. Minimizing the depth is thus going to minimize the number of cache miss (assuming no previous state of the memory) and give fast access time in practice. Additionally, minimizing the depth also increases the number of matching characters in early levels of the trie, which also makes better use of common prefixes in the dictionary.

With the previous argument, we can see that the word-optimal strategy makes sense when the inputs are complete words, but when the inputs are prefixes of words from the dictionary, the likelihood of word completions is distorted and, as such, the weights should be adapted. We give an alternative cost function to consider the cost per prefix completion, but first we give notation from formal languages to define prefixes. Let $\Sigma_{\mathcal{W}}$ be the alphabet of the word set $\mathcal{W}$ and $\Sigma_{\mathcal{W}}^*$ be the set of words made from any sequence of characters of $\Sigma_{\mathcal{W}}$.

As we now deal with prefixes, we need two notations: one to describe the prefixes of a word, and one to denote the words starting with a given prefix. We use $\mathcal{P}(w)$ to denote the first (set of prefixes of the word w) and $\mathcal{W}_p$ to denote the latter (subset of words of $\mathcal{W}$ having the prefix p). We describe formally these definition and give some small extension:

<u>Prefixes of words ($\mathcal{P}(w)$):</u>

We denote the concatenation of two characters c_1 and c_2 by c_1c_2. The decomposition of a word in its characters is $w = w_1w_2\cdots w_{|w|}$ and we define the prefix function as follows:

$$\mathcal{P}(w) = \left\{\varepsilon,\ w_1,\ w_1w_2,\ w_1w_2w_3,\ \ldots,\ w_1w_2\cdots w_{|w|}\right\} = \{\varepsilon\} \cup \left(\bigcup_{i=1}^{|w|} \{w_1\cdots w_i\}\right).$$

We denote by ε the empty word which is a prefix of every word. An example of the prefix function with the word 'cat' is $\mathcal{P}(\text{'}cat\text{'}) = \{\varepsilon, \text{'}c\text{'}, \text{'}ca\text{'}, \text{'}cat\text{'}\}$. We also generalize this function $\mathcal{P}(w)$ for set of words by simply taking the union of each prefixes: $\mathcal{P}(\mathcal{W}) = \bigcup_{w\in\mathcal{W}} \mathcal{P}(w)$, such that $\mathcal{P}(\{\text{'}cat\text{'}, \text{'}cow\text{'}\}) = \{\varepsilon, \text{'}c\text{'}, \text{'}ca\text{'}, \text{'}cat\text{'}, \text{'}co\text{'}, \text{'}cow\text{'}\}$.

<u>Words starting with a prefix ($\mathcal{W}_p$):</u>

This simply represents a subset of $\mathcal{W}$ that starts with p. We formally define it as such:

$$\mathcal{W}_p = \left\{ w \in \mathcal{W} \;\middle|\; |w| \geq |p| \text{ and } w_i = p_i \quad \forall i \in \{1, \ldots, |p|\} \right\}.$$

Finally, we also define arg $\max_{x \in X} f(x)$ as the argument x maximizing $f(x)$ over the domain X. Intuitively, the *max* function returns the maximal $f(x)$ values and *arg max* returns the associated argument x.

We now get back to defining a weight function based on the prefixes of every word of $\mathcal{W}$ (i.e. prefixes in the set $\mathcal{P}(\mathcal{W})$). We thus consider $f_{\text{prefix}} \colon \mathcal{P}(\mathcal{W}) \to \mathbb{R}^+$, the weight function of prefixes and we distribute the cost of every word $w \in \mathcal{W}$ uniformly between their $|w|+1$ prefixes, including the empty prefix noted ε.

$$f_{\text{prefix}}(p) = \sum_{w \in \mathcal{W}_p} \frac{f(w)}{|w|+1}$$

Consequently, the value $f_{\text{prefix}}(p)$ represents the probability that p is the queried prefix.

Definition The *top-1 prefix-optimal* strategy is to minimize the depth of the most likely completion of each prefix weighted by the probability of querying that prefix. The objective function of the prefix-optimal strategy is:

$$c_{\text{top-1}}(\mathcal{W}, \tau) = \sum_{p \in \mathcal{P}(\mathcal{W})} f_{\text{prefix}}(p) \cdot \textit{depth}\left(\arg\max_{w \in \mathcal{W}_p} f(w), \tau\right). \tag{3.2}$$

For convenience, we extend the definition of f to sets of words as such: $f(\mathcal{W}) = \sum_{w \in \mathcal{W}} f(w)$. We now make two observations regarding prefix-optimality.

First, a completion query consists of returning the most probable words to occur knowing the prefix given. This means that we are considering the conditional probability of writing a word knowing its prefix p, $\mathbb{P}\left[w \in \mathcal{W}_p\right] = f(w) \cdot \frac{f(\mathcal{W})}{f(\mathcal{W}_p)}$ which simplifies to $\frac{f(w)}{f(\mathcal{W}_p)}$ for normalized weights because $f(\mathcal{W}) = 1$. When we compare the probability of the words in $\mathcal{W}_p$ to take the highest, each one is scaled by the same factor $1/f(\mathcal{W}_p)$ from the above formula. Thus it is sufficient to simply take the highest probability of the words $f(w)$ with no regards to the conditional clause as it applies uniformly to each candidate of $\mathcal{W}_p$. This observation shows that we can simply compare the likelihood of the completion of a word by the direct weight given by the function f.

Next, we also observe that the cost function is actually shifting the probability of a prefix to its top-1 completion. In Lemma 1, we refine the second observation to describe how each prefix contributes to its most likely completion which allows us to generalize the cost function of top-1 prefix-optimality.

Lemma 1 *Given a set of words $\mathcal{W}$ with probabilities given by $f \colon \mathcal{W} \to \mathbb{R}^+$, the top-1 prefix-optimal cost function is equivalent to the word-optimal cost function with adjusted weights for each top-1 completion of prefixes.*

Proof The top-1 prefix-optimal cost function distributes the weight of every prefix to its best completion. An alternative formulation of the cost is to regroup the prefixes having the same top-1 completion by using an indicator function. We use the notation $\mathbb{1}_S(x)$ for the indicator function which has value 1 if $x \in S$ and 0 otherwise. Thus, the alternative formulation is:

$$c_{\text{top-1}}(\mathcal{W}, \tau) = \sum_{w \in \mathcal{W}} \left(\sum_{p \in \mathcal{P}(w)} f_{\text{prefix}}(p) \cdot \mathbb{1}_{\left\{\arg\max_{w' \in \mathcal{W}_p} f(w')\right\}}(w) \right) \cdot \mathit{depth}(w, \tau).$$

The equality holds because a given prefix can contribute to one and only one word in a top-1 completion (for any arbitrary tie-breaker). This form makes it obvious that a simple perturbation of the weights allows us to achieve top-1 prefix-optimality by solving the word-optimal cost function on the weights given by $\tilde{f}_{\text{top-1}}$:

$$\tilde{f}_{\text{top-1}}(w) = \sum_{p \in \mathcal{P}(w)} f_{\text{prefix}}(p) \cdot \mathbb{1}_{\left\{\arg\max_{w' \in \mathcal{W}_p} f(w')\right\}}(w)$$

We can thus use the word-optimal cost function (3.1) with the perturbed weights given by $\tilde{f}_{\text{top-1}}(w)$ to achieve the same optimization objective. $\square$

We can also extend the prefix-optimal strategy to contribute to top-k completions for fixed k or even to all completions. However, since the exact number of nodes accessed to make a top-k enumeration does not only depend on the absolute depth of each completion, but rather on their relative position to each other as mentioned in Section 2.3, we simply give upper and lower bounds on that value. For *top-*∞, the cost function contributing to all completions of a prefix, we need to explore the whole subtrie, so the cost can be computed exactly.

To generalize equation (3.2) to compute the *top-k prefix optimal*, we use k-max and k-arg max: the generalized max function (resp. arg max) that returns a set of the k maximum values (resp. arguments). The function $a(S_p, \tau)$ is used to denote the number of node accesses per completion which is dependent on the set of top-k completions S_p for the prefix p and the trie structure τ.

$$c_{\text{top-}k}(\mathcal{W}, \tau) = \sum_{p \in \mathcal{P}(\mathcal{W})} f_{\text{prefix}}(p) \cdot a\Big(k\text{-}\arg\max_{w \in \mathcal{W}_p} f(w), \tau \Big)$$

Computing $a(S_p, \tau)$ can be done exactly by simulating a query and would lead to an inefficient optimization algorithm. Precisely, we can count the pointer accesses for the completion of each prefixes and multiply by the weight of that prefix. This would still lead to a polynomial time algorithm with, for instance, the enumeration algorithm described in Section 2.3. Unfortunately, the optimization process is harder to solve as we cannot simply use the depth of the nodes in order to compute the cost. For that reason, we bound the exact number of node accesses by the depth of the completions to give a more manageable cost function. A lower bound on the number of node accesses is to simply access the locus node and get each completion in one additional access each. We have to subtract one if the locus node is one of the top-k completions. An upper bound on the number of node accesses is the full path from the root to each of the top-k completions. This specific upper bound is selected because it matches the word-optimal cost function.

$$\min_{w \in \mathcal{W}_p} depth(w, \tau) + |S_p| - 1 \leq a\left(S_p, \tau\right) \leq \sum_{s \in S_p} depth\left(s, \tau\right)$$

Using this upper bound on $a\left(S_p, \tau\right)$ allows us to write an explicit alternative weight function per prefix for a top-k completion.

$$\tilde{f}_{\text{top-}k}(w) = \sum_{p \in \mathcal{P}(w)} f_{\text{prefix}}(p) \cdot \mathbb{1}_{k\text{-arg max}_{w' \in \mathcal{W}_p} f(w')}(w)$$

For tied values in the last x ranks of the top-k selection, we can uniformly distribute the weight among all l tied values. This gives a fractional contribution of $^{x}/_{l}$ to those l completions. The actual values returned by the top-k enumeration is still limited to k and can proceed by random selection among the l tied values.

Finally, this last cost function attributes the weight of each word to all possible completions of the prefixes, not just a fixed value k. We call it *top-∞ prefix-optimal.*

$$c_{\text{top-}\infty}(\mathcal{W}, \tau) = \sum_{p \in \mathcal{P}(\mathcal{W})} f_{\text{prefix}}(p) \cdot \left(\min_{w \in \mathcal{W}_p} depth(w, \tau) + |\mathcal{W}_p| - 1 \right) \tag{3.3}$$

From that last equation, the expanded term $\sum_{p \in \mathcal{P}(\mathcal{W})} f_{\text{prefix}}(p) \left(|\mathcal{W}_p| - 1\right)$ is not relevant for the optimization problem as it is not dependent on the trie layout. For that reason, we consider the optimization problem on $c^{\star}_{\text{top-}\infty}(\mathcal{W}, \tau) = \sum_{p \in \mathcal{P}(\mathcal{W})} f_{\text{prefix}}(p) \cdot \min_{w \in \mathcal{W}_p} depth(w, \tau)$. Although, we cannot derive an exact alternate cost function because of the min operand on the depth, we provide an alternative optimization function in the next section to solve the optimal layout for the top-∞ cost function.

Corollary 1 *An upper bound of the top-k prefix-optimal cost functions is equivalent to the word-optimal cost function with the adjusted weights given by $\tilde{f}_{\text{top-}k}$.*

Lemma 1 and its corollary show that it is sufficient to have an algorithm to compute the word-optimal path-decomposed trie and other optimality metrics are simply obtained by first perturbing the weights of the words appropriately and running the algorithm. In Section 3.2.2, we consider the performance of this upper bound in terms of the average case. We now consider the various models for which we can consider an alternative number of node accesses.

3.1.1 Alternative Models

When searching for the locus node of a prefix in the trie, we can either read some characters from the current node or follow an edge to another node. A few properties of the path-decomposed structure are relevant to our optimality function and have an impact on the choice of the model. We list them here and investigate their various interpretations.

Property a. The number of character comparisons necessary in order to find the locus node is at most the length of the prefix.

Property b. The number of edges that need to be followed to find the locus node is at most the length of the prefix.

Both properties come from the fact that whenever a non-matching character is found, we branch out at the current branching position at the branching character (i.e. by a hashtable lookup for instance) directly. This branching out matches at least one additional character: the branching character used from the edge taken. Therefore, whether a character comparison succeed or not always end up matching at least one character from the prefix; and the two properties follow immediately.

Now in terms of cost, we consider that reading a character is 'inexpensive' and that following an edge is 'expensive' in terms of computational time. We come to this conclusion because the characters of a word are likely to be stored in contiguous memory and accessing memory with high spatial locality is much faster due to cache access and prefetching in modern CPUs [10]. On the other hand, pointers are likely to cause a cache miss which requires to load the page where the word is located into a higher level of cache. As such, the cost functions we defined are based on the number of node accesses in the trie, thus counting the pointer accesses. We now describe the various models for which we can specialize the cost function.

- **Standard branch:** The standard branch model is the one described in the previous section. In that model, we pay one pointer access for each branch, independent of the type of branch.

- **Free empty branches:** In this model, we pay one pointer access to access any branch, except empty branches: those whose edge are labeled with ε. We consider these empty branches free as there is no need to follow a pointer to a node that is essentially empty.
- **Free terminal branches:** This last model charges only for accessing non-terminal branches. We consider branches to be terminal when they consist of a subtrie containing only one word.

Under the last two models, for two words $w_1, w_2 \in \mathcal{W}$ such that $w_1 \in \mathcal{P}(w_2)$, we can always construct the optimal trie such that w_2 is parent of w_1. This scheme is optimal because the cost of accessing w_1 from the parent w_2 is free in either model, and since w_1 has more opportunity to branch to other words, we prefer this configuration. However, it is true that, for the last model only, if there are no other words sharing the prefix w_1 and $f(w_1) = f(w_2)$, then the choice of w_1 or w_2 as the parent produces the same cost, but for consistency, we can apply the above rule in all cases.

3.2 Properties of Optimal PDTs

In this section we explore properties necessary for the creation of the algorithm that we present in section 3.3. These properties help restrict what constitutes an optimal trie so that the algorithm can be described without considering such ineligible cases.

3.2.1 Dummy Words

Suppose we could add *dummy words* to the word set, each with probability of access zero (because they are not real words), in the hope to improve the layout of the PDT. We show that building a trie that contains any of these words cannot have a lower cost than the optimal trie. As such they are better off simply removed from the trie as they are not part of the word set[2].

Lemma 2 *The word-optimal PDT contains only nodes consisting of words that are part of the word set $\mathcal{W}$ with non-zero weight. Any other words should be left out or placed as leaves.*

[2]Of course, if these words have probability of being returned null, but are still meaningful for testing existence in the trie, we can leave them as leaves of the trie which doesn't contradict the optimality.

Proof If a dummy word is added as a leaf, then it doesn't change the cost of the trie and we can remove or leave it.

Let z be a dummy word at an internal node such that some words are branching from it. We replace that dummy word with any words branching at the furthest branching position, which reduces the cost of the trie by one level per words from that subtrie. It also leaves the dummy word branching from the replaced word as a leaf node. Since whenever a dummy word exists, we can either remove it or change the structure of the trie and then remove it to make it more optimal, we conclude that the optimal trie does not contains any dummy words. $\square$

3.2.2 Optimality for Average Case

As we noted in section 3.1, the cost of the top-k prefix-optimal does not exactly represent the number of pointer accesses as we have no way of having this exact value other than to simulate a query on a trie to count the node accesses. In contrast, the other cost functions depend only on the weights of the words and have no dependency on the structure of the trie. In that section, we argue that using the sum of depths of each word (being an upper bound) is a good approximation, one that we consider sufficient. Now we show that this approximation produces the optimal number of pointer accesses per completion on average.

The idea is that for a single top-k completion query with results $S_p = k\text{-arg}\max_{w \in \mathcal{W}_p} f(w)$, the average depth of the completions is

$$\frac{1}{|S_p|} \sum_{w \in S_p} \mathit{depth}(w, \tau) = \mathit{depth}(p, \tau) + \frac{1}{|S_p|} \sum_{w \in S_p} \mathit{depth}(w, \tau_{\mathcal{W}_p})$$

where $\tau_{\mathcal{W}_p}$ is the subtrie of τ containing the subset of words $\mathcal{W}_p$. This expression gives the average cost per completion in a single top-k query. When taking the weighted average per completion we get an alternative cost function that we refer to by $c_{\text{avg-top-}k}(\mathcal{W}, \tau)$. Minimizing this objective function gives the minimal number of pointer accesses per completion (as opposed to per query) in the average case.

We can achieve a similar weight adjustment as the worst-case top-k cost function in order to solve the problem using the word-optimal cost function. To do so, we simply divide the probability of each top-k completion by the actual number of completion returned from the word set. Using the set S_p as before gets us the following alternate cost function:

$$\tilde{f}_{\text{avg-top-}k}(w) = \sum_{p \in \mathcal{P}(w)} \frac{f_{\text{prefix}}(p) \cdot \mathbb{1}_{S_p}(w)}{|S_p|}.$$

3.3 Dynamic Programming Algorithm

Up to this point, we have examined optimality metrics and shown a convergence of these to a single one: word-optimal. In this section, we first give a naïve algorithm to compute the optimal path-decomposed trie. It uses a dynamic programming approach to solve for increasing instances of $\mathcal{W}_{[i,j]}$, the subset of $\mathcal{W}$ which contains the i^{th} to j^{th} words in sorted order. We use the set of sorted words $\mathcal{W}$ as it facilitates finding ranges of words sharing a common prefix. Our dynamic approach is similar to Knuth's optimal binary tree recurrence [15].

In order to process the cost of each branch, we use a function $B(\mathcal{W}, w)$ on a word set $\mathcal{W}$ and a specific word $w \in \mathcal{W}$. We can either let B return a prefix p for each corresponding range of words branching from each position, i.e. a range defined by $\mathcal{W}_p \subseteq \mathcal{W}$, or return a pair of integers representing the range (beginning and end of the range in the sorted word set). We use the variant with indices to define the dynamic programming algorithm as it is easier to index words into array while working with the dynamic table, but we use the variant with prefixes to explain the idea as it is conceptually simpler.

$B(\cdot,\cdot)$ returns a set of prefixes.

Let $B(\mathcal{W}_p, w)$ for $w \in \mathcal{W}_p$ for some prefix p be the set of prefixes matching the branches of the word w over the set $\mathcal{W}_p$. For example, if $\mathcal{W} = \{\text{‘}car\text{’}, \text{‘}cart\text{’}, \text{‘}cat\text{’}, \text{‘}cow\text{’}, \text{‘}dog\text{’}\}$, then $B(\mathcal{W}_\varepsilon, \text{‘}cat\text{’}) = \{\text{‘}car\text{’}, \text{‘}co\text{’}, \text{‘}d\text{’}\}$ because both ‘*car*’ and ‘*cart*’ diverge from ‘*cat*’ at the third letter with the prefix ‘*car*’, ‘*cow*’ at the second letter with prefix ‘*co*’, and ‘*dog*’ at the first letter with prefix ‘*d*’.

$B(\cdot,\cdot)$ returns a set of pairs of indices.

In this case, we let $B(\mathcal{W}_{[i,j]}, w)$ for $w \in \mathcal{W}_{[i,j]}$ be the set of ranges of the words in all the branches of w. For the above ordered set $\mathcal{W}$, we have the corresponding example: $B(\mathcal{W}_{[0,4]}, \text{‘}cat\text{’}) = \{[0,1],[3,3],[4,4]\}$.

Using the definition of $B(\cdot,\cdot)$, we transform the cost function $c(\mathcal{W}, \tau)$ into a dynamic recurrence[3] for any subset of words $\mathcal{W}_p$ (or $\mathcal{W}_{[i,j]}$). To simplify the notation, we consider the function $c(\cdot)$ on the (sub)set of words only and let the trie structure be defined by each selection of word by the minimization operation.

$$c(\mathcal{W}_p) = f(\mathcal{W}_p) + \min_{w \in \mathcal{W}_p} \left(\sum_{p' \in B(\mathcal{W}_p, w)} c(\mathcal{W}_{p'}) \right) \tag{3.4}$$

[3] We use *dynamic recurrence* to specifically refer to the recurrence of a dynamic programming algorithm.

$$c\left(\mathcal{W}_{[i,j]}\right) = f(\mathcal{W}_{[i,j]}) + \min_{w \in \mathcal{W}_{[i,j]}} \left(\sum_{(i',j') \in B(\mathcal{W}_{[i,j]},w)} c\left(\mathcal{W}_{[i',j']}\right) \right). \tag{3.5}$$

Lemma 3 *Equations (3.4) and (3.5) are equivalent to the word-optimal cost function.*

Proof First note that (3.4) and (3.5) are equivalent because they only differ by the alternative definition of $B(\cdot,\cdot)$. We now prove that (3.5) is equivalent to (3.1).

The case when $i = j$ is trivial.

For the general case, the first summation counts the total contribution of every word for the first level. The second term, minimizes the cost of every subtrie for any possible selected word. Thus, a word's weight stops contributing to the total cost exactly when it is selected as the minimum by the second term because the partition of $B(\mathcal{W}_{[i,j]}, w)$ never includes w. Therefore, the weight of any word is added one time for each level where it isn't selected as the subroot, making the recurrence equivalent to the previous definition of $c(\mathcal{W}, \tau)$ based on the depth (3.1). □

By the correctness of the dynamic recurrence, we know that the optimization process consists of choosing an optimal root and then optimizing each subtrie individually. This is what is expressed by the minimization term of equation (3.4)[4]

We now develop a dynamic programming algorithm from the above dynamic recurrence (3.4). We do so by filling a table of size $n \times n$, where the cell at coordinate (i,j) contains the result of $c(\mathcal{W}_{[i,j]})$. We start filling the cells from the main diagonal of the table, those of coordinate (i,i), and process each diagonal with the precomputed results of the previous rows until we finish with the top-right corner which contains the solution for the complete word set $\mathcal{W}$. We know the dynamic programming algorithm gives the optimal solution because each step minimizes the cost of the subset of words it covers. The following lemma explore the time complexity of the algorithm.

Lemma 4 *There is a dynamic programming algorithm that can compute the word-optimal path-decomposed trie in $\mathcal{O}\left(n^3 \cdot b_{max}\right)$ time where b_{max} is the maximal number of branches for any node using $\mathcal{O}\left(n^2\right)$ space.*

Proof We first compute a partial sum data structure of the weights of the lexicographically sorted words $\mathcal{W}$. The idea is to execute the above recurrence and complete a two-dimensional array for all subranges $i \leq j$ of $\mathcal{W}$. Each cell takes $\mathcal{O}\left(\sum_{w \in \mathcal{W}_{[i,j]}} \left|B(\mathcal{W}_{[i,j]}, w)\right|\right)$

[4]For the remainder of this section, we use equation (3.4) and equation (3.5) interchangeably to denote the dynamic recurrence, as the same results holds by lemma 3.

$\subseteq \mathcal{O}\left((j-i)\cdot b_{\max}\right)$ time to compute as each of the evaluations of the min operation requires one lookup in the dynamic table per branch of $B(\mathcal{W}_{[i,j]}, w)$. We can also write down the selected root or tied-roots (the arg min value of the dynamic recurrence (3.5)) as $R_{i,j}$ at each step to reconstruct the trie.

Once the whole table is complete, we construct the optimal trie by setting the root to any optimal root $r \in R_{0,n}$ and let its children be any of the optimal roots of $R_{i,j}$ for each $(i,j) \in B(\mathcal{W}_{[0,n]}, r)$ and so on recursively, each on the subsets $\mathcal{W}_{[i,j]}$. $\square$

We can thus compute any of the prefix-optimal tries in $\mathcal{O}\left(n^3 \cdot b_{\max}\right)$ time using the same dynamic programming algorithm given that we compute the perturbed weights beforehand. We examine the cost of this preprocessing in the next subsection.

3.3.1 Complexity of the Preprocessing

The dynamic programming algorithm of Section 3.3 finds the optimal trie using the word-optimal strategy. As we proved earlier in Lemma 1 and corollary 1, we can transform the original weights of words to obtain alternative weights (denoted by the $\tilde{f}$ notation) and apply to the word-optimal cost function. We explain how to compute those alternative weights and prove the time complexity of this preprocessing.

The idea is to use a hash table with prefixes as keys and store the weight f_{prefix} along with the number of completions encountered so far. We then execute the algorithm in three steps:

1. For each word w, add the value $\frac{f(w)}{|w|+1}$ to each of the entry associated with the prefixes of w in the hash table.
2. Sort the words by weight in descending order.
3. Take the sorted words one by one and for each of its prefixes do the following:
    (a) If the count of the number of completions encountered so far doesn't exceed k, add the prefix's weight to the word's alternative weight and increase the number of completions encountered of the prefix by one.

The first step fills the hash table with the values of f_{prefix}. The second step allows us to limit the contribution of a prefix to its k best completions as we iterate over the words in the third step. The threshold k can be arbitrary (i.e. 1 for top-1 or none for top-∞). A pseudo-code sample is given in algorithm 1.

Assuming $\mathcal{O}(1)$ behavior of hash table in the average case, we calculate the computation time of the algorithm. Steps 1 and 3 take $\mathcal{O}(N)$ time where $N = \sum_{w\in\mathcal{W}} |w|$ and sorting in step 2 can be done in $\mathcal{O}(N)$ time by building a prefix-tree of the words and then enumerating the words in depth-first, left-to-right order of the branches of the prefix tree.

Claim 1 *The running time of the preprocessing step to compute the alternative weights of top-1, the upper bound of top-k or top-∞ prefix-optimality is bounded by $\mathcal{O}(N)$ time in average case.*

Algorithm 1 Preprocessing($\mathcal{W}, f, k$)

```
$h \leftarrow$ hashtable
for $w \in \mathcal{W}$ do
    for $p \in \mathcal{P}(w)$ do
        $h[p].weight \leftarrow h[p].weight + \frac{f(w)}{|w|+1}$
sort $\mathcal{W}$ by weight
$\tilde{f} \leftarrow$ hashtable
for $w \in \mathcal{W}$ do
    for $p \in \mathcal{P}(w)$ do
        if $h[p].count < k$ then
            $\tilde{f}[p] \leftarrow \tilde{f}[p] + h[p].weight$
            $h[p].count \leftarrow h[p].count + 1$
return $\tilde{f}$
```

3.3.2 Dynamic Recurrence for top-∞

The top-∞ cost function is based on the depth of the locus node, the rest is a straightforward enumeration of every nodes matching the prefix. We can refer to the depth of the locus node as $\min_{w\in\mathcal{W}_p} depth(w,\tau)$ for a prefix p which we simplify by extending the depth function on prefixes: $depth(p,\tau)$. Ultimately, the main difference from the previous cost function is that the further contributions of the prefixes of a word already selected should not be considered in the next levels of the trie.

Although it is possible to define a cost function in terms of the ranges of words in $\mathcal{W}$, it is easier to work with range of words having a certain prefix as with equation (3.4).

$$c(\mathcal{W}_p) = \left[\sum_{\substack{p'\in\mathcal{P}(\mathcal{W}_p) \\ |p'|>|p|}} f_{\text{prefix}}(p') \right] + \min_{w\in\mathcal{W}_p} \sum_{p'\in B(\mathcal{W}_p,w)} c(\mathcal{W}_{p'}) . \tag{3.6}$$

Clearly, in this equation, a prefix stops contributing to additional degree of depths whenever it is selected since the first terms only considers the prefixes longer than the current range. This equation thus computes the depth of the locus node of each prefix, resulting in the optimization function $c^\star_{\text{top-}\infty}(\tau)$ of equation (3.3).

3.3.3 Alternative Models

To support the *free empty branches* model, when we list the branching position $B(\mathcal{W}_{[i,j]}, w)$, we simply ignores branches with one element where the next character is the empty word ε. For the *free terminal branches* model, we do the same process as with free empty branches but ignore the cost whenever a branch contains only one element.

This change in the dynamic programming algorithm does not affect the asymptotic running time as it only requires one additional condition to evaluate for each word evaluation by the min operation.

3.4 Tightening the Dynamic Recurrence

We can tighten the results of the dynamic recurrence (3.5) by showing that a local weight increase in a subtrie can only provoke a change of the ancestor nodes to an optimal choice in that local subtrie. In this section, we formalize the property and prove that it applies to PDTs.

3.4.1 Characterization of Candidates to Update

In this section we verify that PDTs have a specific property on certain operations. The operations that we consider on PDTs are insertion of words, increase the weight of words, decrease the weight of words, and finally, deletion of words. It is important to note that we consider increases in weight and decreases in weight differently as they can have different algorithm assigned to them.

First, it is useful to recognize the following corollary that follows from the dynamic recurrence used in the dynamic programming algorithm from the previous section.

Corollary 2 *The optimal trie for a word set $\mathcal{W}$ given a fixed word $w \in \mathcal{W}$ at the root is composed of the optimal tries of each subset of words in each branch imposed by the choice of the root.*

We now define the concept of tied-optimal roots to generalize over the (possibly) multiple choices of roots that are optimal.

Definition We define the set of *tied-optimal roots* as the set of nodes having a minimum word-optimal cost in a trie when placed at its root.

The property of interest can now be defined as follows using the previous definition.

Property (Locality of change) An operation on an tree structure has *locality of change* for a specific optimality metric if applying this operation to any optimal (sub)tree τ can only change its root to one of its tied-optimal roots or to one of the tied-optimal roots of the subtree where the operation has been applied.

An operation having this property is likely to be more efficient to execute as it reduces the number of candidates needed to be checked in order to maintain the optimality of the tree after the operation. It also suggests independence between different branches of the tree with respect to the concerned operation. This independence strengthens the implicit structure of the stored values induced by the representation.

Theorem 1 *Insertions and updates increasing the weight of a word in a path-decomposed trie have the locality of change property for the word-optimal cost function.*

Proof From lemma 2, we observe that the insert operation can be executed as an insertion of the word as a leaf node with weight zero, followed by an update of the weight. We also note that the initial insertion with weight zero does not break the optimal representation of the trie since the total contribution of that new word is null. We can thus only consider the update operations to prove the property.

First, we define some notation. Let τ_r be the word-optimal trie with an optimal root r. We consider the candidates for the new root of the optimal trie after updating the weight of an arbitrary word x by δ_x, the positive weight increase. We denote the direct subtrie of r in τ_r where x falls into by $S_{\tau_r,x}$. Furthermore, $S^-_{\tau_r,x}$ and $S^+_{\tau_r,x}$ are the sets of subtries branching from previous and subsequent branches of r — using the ordering defined in 2.2 — and S_{τ_r} is the set of all the immediate subtries of r as pictured in figure 3.1. We let $\mathcal{W}(\tau)$, for any (sub)trie τ, be the set of words included in τ. We use the cost function from the word-optimal definition as follows:

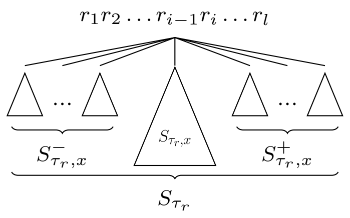


Figure 3.1: Notation of the subtries of τ_r.

$$f\Big(\mathcal{W}(\tau_r)\Big) = \sum_{w\in\mathcal{W}(\tau_r)} f(w) \qquad\qquad c(\tau_r) = f\Big(\mathcal{W}(\tau_r)\Big) + \sum_{\tau_i\in S_{\tau_r}} c(\tau_i).$$

Finally, we denote the word-optimal trie obtained *after* the update of x's weight with a prime notation: τ'_z for the **fixed** root z. Note that in the optimal trie τ'_z with fixed root

z, we do **not** consider z to be the optimal root choice, but rather a specific choice with all of its subtries optimized. Considering this exact situation allows us to compare the cost of the trie having a specific root z with the cost of the trie keeping the previously optimal root r at its root **after** the update of the weight of x.

We now prove by induction that the optimal root after the update cannot be any other choice than a tied-optimal root of τ_r or a tied-optimal root of $S_{\tau'_r,x}$. The induction hypothesis is that the locality of change property holds for $S_{\tau_r,x}$ on update operations increasing the weight over the word-optimal metric.

Base case:

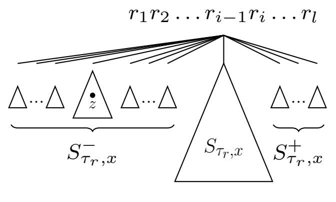
In the base case, x is the only node in $S_{\tau_r,x}$. Its total contribution to the cost of the trie τ_r is $f(x) \cdot \mathit{depth}(x, \tau_r) = 2f(x)$, which can only be reduced by placing x at the root by the invariant of the structure of the trie. Hence, the locality of change is verified in the base case.

Induction step:

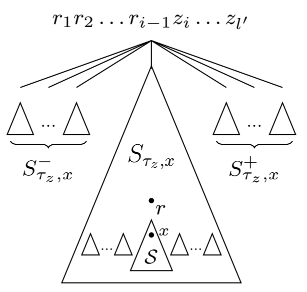
For the induction step, we prove by contradiction that any other choice of root z has a strictly bigger cost than the tied-optimal roots described by the property.

Case 1: **Any fixed z in a subtrie preceding $S_{\tau_r,x}$ (i.e. $S_{\tau_r,z} \in S^-_{\tau_r,x}$).**

The intuition for this case is that by taking a word branching before $S_{\tau_r,x}$ as the root, the updated node x may only occur deeper in τ'_z than in τ'_r, thus contributing more to the total cost.



(a) Trie τ_r with $z \in S^-_{\tau_r,x}$.



(b) Trie τ_z where $S_{\tau_r,z} \in S^-_{\tau_r,x}$.

The situation from the first case can be represented by the figures 3.2a and 3.2b. The strategy is to show that $c(\tau'_r) \leq c(\tau'_z)$ and we know that $c(\tau_r) \leq c(\tau_z)$ by the optimality of τ_r.

We characterize the cost of the trie after the modification by the previous optimal cost added with the minimal increase cost $\Delta_x(\tau)$ in the (sub)trie τ containing x. We use the minimal increase cost as we consider the optimal trie after the modification, we thus execute the modication having the lowest added cost.

$$\begin{aligned} c(\tau'_r) &= c(\tau_r) + \Delta_x(\tau_r) \\ c(\tau'_z) &= c(\tau_z) + \Delta_x(\tau_z). \end{aligned} \tag{3.7}$$

Now, consider the prefix p leading to the subtrie $S_{\tau_r,x}$. The same set of words, albeit with maybe additional words, can be found by searching for that prefix in τ_z. We rename the set of words $\mathcal{W}(S_{\tau_r,x})$ as $\mathcal{W}$ as we refer to it for both tries. From corollary 2, we know that the optimization process for the words of $\mathcal{W}$ is exactly the same in both tries. This result comes directly from the dynamic programming solution. As such, the cost of the subtrie containing $\mathcal{W}$, relative to its local root, in τ_r and in τ_z is the same. However, the total absolute cost of those words are not the same, since the depth of the subtrie containing $\mathcal{W}$ in τ_z is at least the same depth as the subtrie containing the same words in τ_r. This last affirmation comes from the fact that the subtrie containing $\mathcal{W}$ in τ_r is at depth exactly 2 by definition of $\mathcal{W}$ but the subtrie containing the same words in τ_z is **at least** at depth 2 since it cannot be at the root, as the root $z \notin \mathcal{W}$. Therefore, we bound the minimal cost of increase in both trie: $\Delta_x(\tau_r) \leq \Delta_x(\tau_z)$.

We then use the equations (3.7) to conclude that the only possible words of $S^-_{\tau_r,x}$ that can be selected are exactly the tied-optimal roots of τ_r because they would then share the exact same cost before the update, $c(\tau_r) = c(\tau_z)$, and the increase in cost could still be equal in certain cases. If z is not a tied-optimal root, then the above argument proves that z cannot be the optimal root after the modification on x's weight. As such, the result holds in this case.

Case 2: **Any fixed z in a subtrie succeeding $S_{\tau_r,x}$ (i.e. $S_{\tau_r,z} \in S^+_{\tau_r,x}$).**

By taking a new root z branching after x's branch, $S_{\tau_r,x}$ is the same as $S_{\tau_z,x}$ by the optimality of that subtrie (i.e. they contains the same set of words under the same structure). Therefore, the cost increase of adding x to both tries is the same, but the previous cost of τ_r is optimal. We get that τ_z costs at least as much as τ_r and can share the same optimal cost only if z is a tied-optimal root in τ_r. The result holds in this case too.

Case 3: **Any fixed z in $S_{\tau_r,x}$, but not one of its tied-optimal roots.**

The subtries of $S^-_{\tau_r,x}$ are part of $S^-_{\tau_z,r}$ in the same configuration as they were already optimal and the same branches occur in z as in r. Two cases occur for the subtries of $S^+_{\tau_r,x}$:

Case a: A subtrie branches at the same branching position, but on a subsequent lexicographic letter as x from r in τ_r. In this case, the subtrie stays at the same position in τ_z, causing no change in the cost of the trie.

Case b: A subtrie branches from a subsequent branching position. Then, the words of that subtrie need to be placed in $S_{\tau_z,r}$. Their new position is thus at least at depth 2, whereas before they were place exactly at depth 2.

Combining these two cases, the change in the optimal cost considering only the subsequent subtries can only be greater. This, in addition to the strictly suboptimal choice of the root in $S_{\tau_r,x}$, shows that z cannot be the root of the optimal trie.

Therefore, since we show that every choice other than the tied-optimal roots of τ_r and the tied-optimal roots of $S_{\tau_r,x}$ are suboptimal, the only candidates to improve the cost are one of these two. It thus proves the induction step and confirms the property. $\square$

We do not have the same result for deletions and updates reducing the weights since the words sharing a prefix with the decreased word can be replaced as root of a subtrie by the root of competing subtries at the same branching position. [5]

Since the dynamic programming construction of the path-decomposed trie is adding words one by one, we can exploit the property of locality of change and only consider the tied-optimal roots stored for each cell. This improvement reduces the number of values to check in the second term of the dynamic recurrence (3.5).

Claim 2 *The improved version of the dynamic recurrence reduces the* best-case time *to compute the word-optimal path-decomposed trie to* $\mathcal{O}\left(n^2 \cdot b_{max}\right)$ *where* b_{max} *is the maximal number of branches for any node.*

The worst-case is still $\mathcal{O}\left(n^3 \cdot b_{\max}\right)$ because the algorithm needs to check every tied-optimal root, which could include all candidates in some instances.

[5]Actually, a dual property apply to those operations: a word deletion or an update decreasing the weight of a node in an optimal (sub)trie τ can only change its root to one of its tied-optimal roots or to one of the tied-optimal roots of the subtries of τ where the operation is **not** applied to. We do not give a proof of this dual property because it isn't of much use to our dynamic optimality, but the proof would follow the same ideas as the previous.

Chapter 4

Better Characterization & *Heavy-Path*

The previous section explains which candidate can replace an optimal root when adding words to the dictionary. This idea leads to an improvement on the efficiency of the dynamic programming algorithm. In section 4.1, we investigate an even stronger characterization by considering exactly *when* one of those candidates needs to replace the root. Knowing both the candidate nodes and a condition on their weights allows us to even further improve the construction of the optimal PDT and leads us to other known construction techniques. We also use the complete characterization in section 4.2 to maintain the optimal trie over dynamic operations.

4.1 Update Condition

Theorem 1 shows which candidates can replace the root. We now show exactly when a candidate can replace the root, and therefore, completely characterize the optimal root selection. For the following theorem, we use the notation $w_{[i,j]}$ to represent the substring $w_i \cdots w_j$ for a word $w = w_1 w_2 \cdots w_{|w|}$.

We associate a subtrie with a prefix based on the invariant of PDTs defined in section 2.2. As such the *associated subtrie of a prefix p in τ* is the subtrie where the root is the locus node of a search of p in τ restricted with only the branches following the last branching position of characters of p in the locus node. Therefore, all the words having prefix p are located in that subtrie.

Theorem 2 *For any prefix $p = p_1 \dots p_l$ in a word-optimal trie τ, the root of its associated subtrie in τ has the prefix $p\sigma$ with the highest value $f(\mathcal{W}_{p\sigma})$ for all $\sigma \in \Sigma_{\mathcal{W}}$.*

Proof Since the theorem applies to any prefix and each selection of root is dependent on the selection of its subtrie, we prove the result by induction on the size of the associated subtrie of $\mathcal{W}_p$. We can consider every complete word to be a non-prefix of other words by appending the empty character \$ at the end of every word. The complete prefix of a word, including the last \$, is uniquely completing the queried word such that $|\mathcal{W}_p| = 1$.

Base case:

A subtrie of size one contains only one word, thus the theorem holds trivially.

Induction step:

We prove this by contradiction, i.e. we show that not taking the highest valued subtrie invariably leads to a suboptimal cost for the whole trie. Suppose that $\mathcal{S}$ is the associated subtrie of prefix p and that the subtries branching after the last letter of p are $S_{\sigma_1}, \dots, S_{\sigma_{|\Sigma_{\mathcal{W}}|}}$ (some of which can be empty). Let S_{σ_i} be the highest valued subtrie for the common prefix p, i.e. $f(\mathcal{W}_{p\sigma_i}) \geq f(\mathcal{W}_{p\sigma_j})$ for all $j \in \{1, \dots, |\Sigma_{\mathcal{W}}|\}$. Each of these subtries are assumed optimal by the induction step, which means that they have a root that minimizes their cost $c(S_{\sigma_j})$. Including the empty character in the alphabet $\Sigma_{\mathcal{W}}$, we compute the total cost. The total cost consists of the contribution of every character that doesn't match the one selected as the root plus the cost of each subtries:

$$\begin{aligned} c(\mathcal{S}) &= \sum_{j=1}^{|\Sigma_{\mathcal{W}}|} c(S_{\sigma_j}) + \min_{i \in \{1,\dots,|\Sigma_{\mathcal{W}}|\}} \sum_{j \in \{1,\dots,i-1,i+1,\dots|\Sigma_{\mathcal{W}}|\}} f(\mathcal{W}_{p\sigma_j}) \\ &= \sum_{j=1}^{|\Sigma_{\mathcal{W}}|} c(S_{\sigma_j}) + f(\mathcal{W}_p) - \max_{i \in \{1,\dots,|\Sigma_{\mathcal{W}}|\}} f(\mathcal{W}_{p\sigma_i}). \end{aligned}$$

From the minimized term, we find that choosing the highest valued subtrie gives the lowest cost, which proves the result. □

Suppose we have an optimal path-decomposed trie and we want to update the weight of a word. We can use the *update condition* defined by Theorem 2 to know when a node need to replace it's parent based on the cost of the two alternatives. To do so, we check if $f(\mathcal{W}_{p\sigma_{\text{branch}}})$ (for the branching character to the current node σ_{branch}) is heavier than $f(\mathcal{W}_{p\sigma_{\text{parent}}})$ (for the non-matching character of the parent σ_{parent}) and replace the parent if this is true. We support dynamic insertion of words in a similar way by adding the new word as a leaf in the trie and move it up according to the update condition of Theorem 2.

On the other hand, if we know the list of all words beforehand and we want to compute the optimal trie, we can simply find the heaviest subtrie starting with a first letter and recursively continue the process until a single word is selected. This node becomes the root of the trie and we can repeat the complete construction of the trie recursively for each subrange of words from the branches imposed by the selected root. This strategy is known as the *heavy-path* decomposition (as explained in section 2.2.1) and the following corollary follows from this argument obtained by applying Theorems 1 and 2.

Corollary 3 Heavy-path *creates a word-optimal path-decomposed trie.*

We now consider the running time to construct a PDT using the greedy *heavy-path* strategy:

Preprocessing:

Insert every word of $\mathcal{W}$ into a prefix tree while keeping track of the total weight at every prefix (node). We achieve this accounting by adding the weight of the inserted word to each node encountered in the insertion. The total construction can be done in $\mathcal{O}(N)$ time.

Heavy-Path Selection:

(a) Define an empty list l that will contain the branches to recurse into.

(b) Starting with the root, compute the heavy path by following the heavy letter (i.e. the one with the highest total weight stored in the node) among the (up to σ) choices at each level of the prefix tree. We also place the node of each non-selected letter along the heavy path into the list l. This process eventually selects a word w and we place w in the path-decomposed trie τ.

(c) Recurse from step b but with each node in l instead of the root. Fill out a new list l for each of these different execution.

(d) When the whole prefix tree is processed, return the path-decomposed trie τ.

Creating a prefix tree is known to take $\mathcal{O}(N)$. The exploration steps obtaining the heavy-path observe each node only once and the number of nodes is exactly $N + 1$ in the prefix tree (one more for the root). Accounting for all of these costs, we get the following claim.

Theorem 3 *Construction of the word-optimal path-decomposed trie takes $\mathcal{O}(N)$ time, where $N = \sum_{w \in \mathcal{W}} |w|$ using the* heavy-path *strategy.*

Comparing this result with the improved dynamic programming algorithm of Claim 2, there could be bad examples of dictionaries (particularly with very long words, e.g. $\mathcal{O}(n^2)$ length of words) where the dynamic programming algorithm would be favorable. However, the preprocessing step presented in Section 3.3.1 also shares the same bound on $\mathcal{O}(N)$.

4.2 Dynamic Optimal PDTs

Using the greedy *heavy-path* strategy, we can describe dynamic operations on the trie while still maintaining the optimality condition. Again, we regroup the operations as updates that increment the weight of a word, and as updates that decrement the weight of a word. These two types of operation include insertions and deletions by adding the word as a leaf nodes or removing a leaf node respectively.

4.2.1 Incrementing a Weight

Using the previously described properties, we can implement the increment operation while still maintaining optimality efficiently. The idea is to 'flip' the nodes on the path to the locus node with their parents whenever theorem 2 applies. We give the following description of the algorithm.

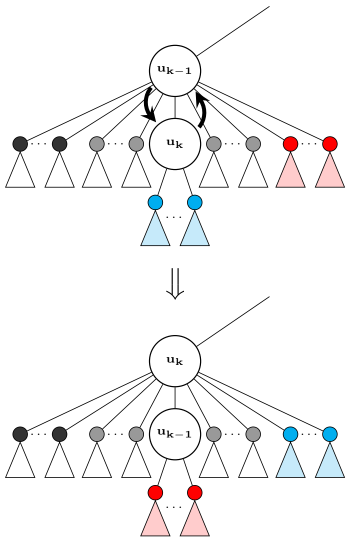


Figure 4.1: Flip of u_k and u_{k-1}. Black subtries have branching position preceding u_k, gray subtries have the same, and blue and red have subsequent branching position of u_k and u_{k-1} respectively.

1. Find the node matching the word to update and increase its weight.
2. Apply the following flip operation in a bottom-up iterative approach on the nodes of the path to the updated word. We denote the node of the processed subtrie by u_k and its parent by u_{k-1}. We also denote their respective prefixes by p_k and p_{k-1}.
    (a) Let l be the length of the longest common prefix of p_k and p_{k-1}. If $f(\mathcal{W}_{{p_k}_{[0,l+1]}}) > f(\mathcal{W}_{{p_{k-1}}_{[0,l+1]}})$, flip u_{k-1} and u_k by replacing their subtries subsequent to the branching position of u_k from u_{k-1}. Place u_{k-1} in lexicographic order among the subtries at the same branching position. The flip operation is illustrated in figure 4.1.

The algorithm is correct because theorem 1 assures us that only the new root of the bottom-most change can move higher, and theorem 2 shows the cost threshold to make the flip. The latter theorem also proves that the choice of tied-optimal root is arbitrary since they all reduce the cost by the same amount.

Claim 3 *Incrementing the weight of a word w in a PDT containing the words $\mathcal{W}$ takes $\mathcal{O}\left(|\Sigma_{\mathcal{W}}| \cdot |w|\right)$ time.*

Proof To increment the weight of a word, we first find the node in $\mathcal{O}(|w|)$ time. We then take the branching character leading to w, σ_w, and the one of its parent σ_{parent}, and compare the value of $f(\mathcal{W}_{p\sigma_w})$ against $f(\mathcal{W}_{p\sigma_{\text{parent}}})$ for their common prefix p. If the former is heavier, we *flip* the parent with w, adjust the labels on their parent edges, and move all corresponding branches (those that branched before w) to the flipped node w as in figure 4.1. We then re-execute the same process for w with its new parent or stop if w is at the root. Otherwise, we leave the parent as-is and continue checking the condition on the path to the root.

The time complexity is thus bounded by the number of branches to switch during each flip operation. This number is upper bounded by the total number of branches branching before or at the branch to w. This is also again upper bounded by the maximal number of branches possible: $|\Sigma_{\mathcal{W}}| \cdot (|w| + 1)$. Computing $f(\mathcal{W}_{p\sigma})$ requires a partial sum data structure to be done in $\mathcal{O}(1)$ time and either $\mathcal{O}(\log n)$ search time to find the range of $\mathcal{W}_{p\sigma_w}$ in the sorted array containing $\mathcal{W}$; or $\mathcal{O}(1)$ time if we store those ranges in the edges directly (doing so is a simple modification of the preprocessing step of the algorithm related to theorem 3). Using the latter, we get a total running time of $\mathcal{O}\left(|\Sigma_{\mathcal{W}}| \cdot |w|\right)$.□

4.2.2 Decrementing a Weight

We can do decrements of weight in a similar way. Of course, since Theorem 1 does not apply, we cannot simply consider the decremented node as the only candidate, but we need to consider the up to $|\Sigma_{\mathcal{W}}| - 1$ other candidates. As we see with the following proof, we get the same time complexity, albeit with a slightly higher constant hidden by the big-O notation.

Claim 4 *Decrementing the weight of a word w in a PDT containing the words $\mathcal{W}$ takes $\mathcal{O}\left(|\Sigma_{\mathcal{W}}| \cdot |w|\right)$ time.*

Proof To decrement a word, we use the same process as the increment operation, but we also compare the up to $|\Sigma_{\mathcal{W}}| - 1$ other branches at the same branching position from the parent. We take the heaviest branch $f(\mathcal{W}_{p\sigma})$ for $\sigma \in \Sigma_{\mathcal{W}}$ and replace the parent with it, continuing upward up to the root.

The running time of this operation is also $\mathcal{O}\left(|\Sigma_{\mathcal{W}}| \cdot |w|\right)$ because the additional $\mathcal{O}\left(|\Sigma_{\mathcal{W}}|\right)$ comparisons can be done in $\mathcal{O}\left(|\Sigma_{\mathcal{W}}|\right)$ time using the same precomputation of ranges in the edges and the change of root is executed only to the heaviest of those. □

4.3 Entropy Bound

With the comprehensive construction of the optimal trie explained in section 4.1, we further describe the optimal cost using other values. The goal is to better characterize the cost function chosen and give a better incentive to our definition of optimality. In this section, we give bounds on the depth of a node based on its weight and prove bounds based on the entropy of the path-decomposed trie.

4.3.1 Bounding the Depth

The goal is to bound the cost function with some measure of entropy. We use the definition of entropy on the probability of words such that for normalized weights given by f, the entropy of the probability distribution of f is $\mathcal{H}_f(\mathcal{W}) = \sum_{w \in \mathcal{W}} f(w) \log_2 {}^1\!/_{f(w)}$.

As the cost function is defined by the depth of the words, giving an upper bound of the depth based on the weight of the word would lead to the concept of entropy. Some related results are known on the height of the trie using *heavy-path* decomposition, namely, the height is bounded by $\lceil \log_2 |\mathcal{W}| \rceil$ and the average depth is $\mathcal{O}(\log_\sigma |\mathcal{W}|)$ [11]. We get an explicit upper bound on the depth of a word based on its weight with the following lemma.

Lemma 5 *Let w be a word in the path-decomposed trie τ constructed using the optimal* heavy-path *strategy. The depth of w in τ is bounded by the minimum of the logarithm of its inverse probability and its length, i.e.*

$$\mathit{depth}(w, \tau) \leq \min \left\{ |w| \,,\ \log_2 {}^1\!/_{f(w)} \right\}.$$

Proof First we prove a general result on the depth of any path-decomposed trie and then we investigate the specialized bound of the *heavy-path* strategy. It is by combining both that we prove the main lemma.

Lemma 6 *The worst-case depth of a word w in any path-decomposed trie is $|w|$.*

Proof When searching for a word w, each time w branches from a node in a trie τ, we reach a level one deeper than the previous, but we also select one branching character as the branch contains a character matching w. Therefore, a word with length $|w|$ can only branch from $|w|$ nodes. □

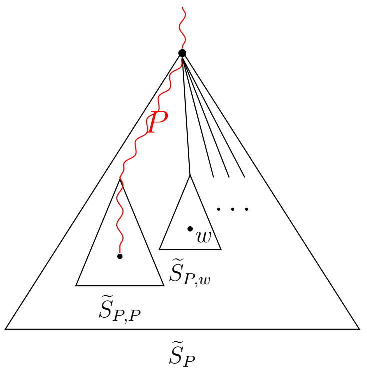


Figure 4.2: Notation of the subtrees of the heavy path P in the prefix tree containing the words $\mathcal{W}$.

The specialized bound for the *heavy-path* strategy comes from the fact that a word is at depth d in τ if and only if d heavier subtrees branch from the path of the root to w in the prefix tree. We define a notation for the subtrees along this path to give the desired bound. Let P be a *heavy path*, a path selected by the *heavy-path* strategy, in a subtree of the prefix tree; at some level along the path, w branches in a different subtree. We denote the subtree of the prefix tree where this occurs as $\widetilde{S}_P$, the *heavy subtree* where the path continues as $\widetilde{S}_{P,P}$, and the subtree containing w as $\widetilde{S}_{P,w}$. Equation (4.1) gives a bound when w is **not** in the heavy subtree.

$$f\left(\mathcal{W}(\widetilde{S}_{P,w})\right) \leq \frac{f\left(\mathcal{W}(\widetilde{S}_P)\right)}{2}. \tag{4.1}$$

If equation (4.1) wasn't true, the heavy path would go toward w, so that subtree must contains less than half of the weight of the parent.

Let $P_1, P_2, \dots, P_d$ be the d heavy paths before reaching w. The above bound holds for each of them and we can link them together using $f\left(\mathcal{W}(\widetilde{S}_{P_i})\right) \leq f\left(\mathcal{W}(\widetilde{S}_{P_{i-1},w})\right)$ simply because $\mathcal{W}(\widetilde{S}_{P_i}) \subseteq \mathcal{W}(\widetilde{S}_{P_{i-1},w})$. We get the following bound of the weight of w:

$$f(w) \leq f\left(\mathcal{W}(\widetilde{S}_{P_d,w})\right) \leq \frac{f\left(\mathcal{W}(\widetilde{S}_{P_d})\right)}{2} \leq \frac{f\left(\mathcal{W}(\widetilde{S}_{P_{d-1},w})\right)}{2} \leq \dots \leq \frac{f\left(\mathcal{W}(\widetilde{S}_{P_1})\right)}{2^d} \leq \frac{1}{2^d}.$$

By solving for d, we get that the depth is at most $\log_2 {}^1/_{f(w)}$. To further improve the bound, we can take the minimum between the upper bound of the heuristic and the maximum height of the word in general, to get the result. □

4.3.2 Entropy Results

As a consequence of the depth relation, we can give an upper bound on the cost function with the entropy of the trie as follows.

Theorem 4

$$\frac{\mathcal{H}_f(\mathcal{W})}{\log_2\left(\sigma \cdot \max_{w \in \mathcal{W}} |w|\right)} \leq c(\tau) \leq \mathcal{H}_f(\mathcal{W})$$

Proof The lower bound follows from an information theoretic result due to Shannon [24]. Our path-decomposed trie has maximum fan-out of $\sigma \cdot \max_{w \in \mathcal{W}} |w|$, thus the source coding theorem for symbol codes gives the lower bound.

The upper bound comes directly from the our cost function using the upper bound on the depth from lemma 5:

$$c(\tau) = \sum_{w \in \mathcal{W}} f(w)\, depth(w, \mathcal{T}) \leq \sum_{w \in \mathcal{W}} f(w) \log_2 \left({}^{1}\!/_{f(w)}\right) = \mathcal{H}_f(\mathcal{W}). \qquad \square$$

Tightness of upper bound

Consider a set of words where every word is a prefix of the subsequents words such as: ‘a’, ‘aa’, ‘aaa’ and so on. The prefix tree of these words forms a path for which a leaf sticks out at each level. We set the weight of each of these words to be ${}^{1}\!/_{2^l}$ where l is the level. The *heavy-path* strategy will then pick the heaviest subtree, arbitrarily choosing the leaf sticking out first. The total cost is thus $c(\tau) = \sum_{i=1}^{n} \frac{i}{2^i} \leq 2$. We can see that the upper bound based on the entropy is tight: $\mathcal{H}_f(\mathcal{W}) = \sum_{i=1}^{n} \frac{1}{2^i} \log_2 2^i = \sum_{i=1}^{n} \frac{i}{2^i} = c(\tau)$. We can also see that this result is not only specific for trie of arity 2; we can simply add leaves sticking out of the trie at any position with very small weight, say ϵ, and this will not perturb the result by very much as $\lim_{\epsilon \to 0} \epsilon \log_2 {}^{1}\!/_{\epsilon} = 0$.

Chapter 5

Heuristics & Experimental Results

In this chapter, we compare the performance of the *max-score* strategy against the optimal trie created by the *heavy-path* strategy. We look in particular at the ratio of their cost functions. In the first section, we examine the theoretical worst-case performance and in the second section, we examine the total cost of the trie composed with both approaches under various corpuses and weights.

5.1 Approximation Ratio of the *Max-Score* Strategy

The *max-score* path-decomposition strategy can be viewed as a heuristic for computing the word-optimal trie. In this section, we bound the weighted average cost of the heuristic in comparison with the optimal cost.

Theorem 5 *There exists a word set $\mathcal{W}$ of arbitrary size n with frequencies from $f\colon \mathcal{W} \to \mathbb{R}^+$ such that the approximation ratio of* max-score *to the word-optimal cost is at least $\Omega\left(n \cdot \frac{\min_{w\in\mathcal{W}} f(w)}{\max_{w\in\mathcal{W}} f(w)}\right)$ in the worst-case.*

Proof The example that we use is $\mathcal{W} = \{\text{‘}a\text{’}, \text{‘}aa\text{’}, \ldots, \text{‘}a^{(n)}\text{’}\}$ with priorities in decreasing order such that $f(\text{‘}a\text{’}) > f(\text{‘}aa\text{’}) > \cdots > f(\text{‘}a^{(n)}\text{’})$ and $f(\text{‘}a^{(i)}\text{’}) < \sum_{j=i+1}^{n} f(\text{‘}a^{(j)}\text{’})$ for $i \in \{1, \ldots, n-2\}$.

The *max-score* strategy returns a path containing all words since they are all prefixes of the following words. On the other side, the word-optimal trie build with *heavy-path* for a specific set of weights following the above constraints selects the second to last word ($a^{(n-1)}$) as a root and places every other word on the second level as a direct child of the root. The cost of each is:

$$c_{\max}(\mathcal{W}, f) = \sum_{i=1}^{n} i \cdot f(w_i)$$
$$c_{\text{opt}}(\mathcal{W}, f) = \sum_{i=1}^{n} 2f(w_i) - f(w_{n-1}).$$

We compute a lower bound of the cost of the *max-score* strategy and a upper bound on the cost of the optimal trie.

$$c_{\max}(\mathcal{W}, f) \geq f(w_n) \cdot \frac{n(n+1)}{2}$$
$$c_{\text{opt}}(\mathcal{W}, f) \leq 2nf(w_1)$$

Therefore, this counter-example provides the following lower bound to the worst case.

$$\max_{\mathcal{W}', f'} \frac{c_{\max}(\mathcal{W}', f')}{c_{\text{opt}}(\mathcal{W}', f')} \geq \frac{c_{\max}(\mathcal{W}, f)}{c_{\text{opt}}(\mathcal{W}, f)} \geq \frac{\frac{n(n+1)}{2} f(w_n)}{2nf(w_1)} = \frac{(n+1)f(w_n)}{4f(w_1)}$$

The approximation ratio is thus at least in $\Omega\left(n\frac{f(w_n)}{f(w_1)}\right)$ for the worst case. $\square$

Now, we consider a distribution of the frequency of words following Zipf's law where the word at rank k has frequency $f(k; s; n) = \frac{1}{k^s H_{n,s}}$ for a parameter of the distribution $s > 1$, a total number of word n, and the generalized harmonic number of order n of s: $H_{n,s}$. We use the Zipf's law as the frequency of word in English is known behave similarly [18]. We can recompute the ratio above with the specific f to obtain the corollary that follows.

$$\max_{\mathcal{W}', f'} \frac{c_{\max}(\mathcal{W}', f')}{c_{\text{opt}}(\mathcal{W}', f')} \geq \frac{c_{\max}(\mathcal{W}, f)}{c_{\text{opt}}(\mathcal{W}, f)} \geq \sum_{k=1}^{n} \frac{\frac{k}{k^s H_{n,s}}}{\frac{2}{k^s H_{n,s}}} = \sum_{k=1}^{n} \frac{k}{2} = \frac{n(n+1)}{4}.$$

Corollary 4 *For Zipf distributed frequencies with parameter $s > 1$ and with n so that $\frac{1}{k^s} < \sum_{i=k+1}^{n} \frac{1}{k^s}$* [1]*, the approximation ratio of* max-score *can be as bad as $\Omega(n^2)$.*

[1]Note that any value of s above a certain threshold (in the range $(1.728, 1.729)$) cannot respect this property. The results still holds for smaller values of s.

5.2 Experimental Results

In this section, we compare the performance of the *max-score* strategy and the *heavy-path* strategy on real-world examples in terms of the cost function, i.e. the average number of pointer accesses per query. We do so as we do not consider (nor improve) the actual representation of the nodes in the trie in this thesis, but rather the layout of those nodes. As such, we do no provide any results based on time per query or bits per string. We expect the time performance to improve proportionally to the improvement in the number of pointers accessed. For experimental results on the time complexity, we refer to Grossi and Ottaviano [13] which uses the *heavy-path* decomposition to give results based on a practical representation.

Based on our optimality result, we know that the ratio is always superior or equal to one. These experimental results illustrate the actual improvement we can expect on real-world corpuses. We apply the techniques on the following datasets:

- **`english`:** We use the dictionary from WinEdt [26] and apply weights as follows:
  - `english-books`: We use a sanitized subset of books from the Project Gutenberg [16, 1]. This collection contains 3036 English books written by 142 authors. The weights are the normalized frequency of words of the whole collection[2].
  - `english-twitter`: We use the tweets from a dataset used for sentiment analysis [23]; we only use the content of the tweets to generate frequencies. The corpus contains 1.6 million tweets and since it includes many misspelled words and acronyms, we add those only after a threshold of 20 occurrences.
  - `english-uniform`: We assign uniforms weights to every word in the dictionary.
  - `english-exp`: We use the dictionary and assign weights based on an exponential distribution with rate of 1. We average the result of the cost function over 10 random samplings of the distribution.
- **`proteins`:** We use protein sequences to experiment on a different set of words. We take our data set from the RCSB Protein Data Bank [21] which contains the proteins represented in their IUB/IUPAC codes. The dataset contains over 345 000 sequences on an alphabet of 25 symbols with 3 additional special characters and the average

[2]As the collection contains older books, some words are archaic and are not present in our dictionary. In those cases, we add the word to the dictionary if its usage is somewhat common, using a threshold of 20 occurrences in the whole corpus. We find that this produces a good balance between accepting real words, although potentially archaic, and accepting gibberish from parsing errors.

length of sequences is 230. We only use uniform and exponential distributions to assign weights with the same parameters as with the English corpus.

We compute the two strategies on all of those datasets with respect to both the number of pointer accesses per query (table 5.1) and per completion (table 5.2). For the cost function of top-k completion queries, we use the upper bound based on the depth as mentioned in Section 3.1.

dataset	$c_{\text{top-1}}$			$c_{\text{top-5}}$			$c_{\text{top-10}}$			$c_{\text{top-20}}$			$c_{\text{top-}\infty}$		
	opt	max	ϕ	opt	max	ϕ	opt	max	ϕ	opt	max	ϕ	opt	max	ϕ
`english-books`	2.373	2.378	+0.23%	15.117	15.180	+0.42%	31.138	31.363	+0.72%	62.916	63.531	+0.98%	137803.591	149470.693	+8.47%
`english-twitter`	2.341	2.342	+0.03%	14.406	14.452	+0.32%	29.894	30.196	+1.01%	59.397	59.945	+0.92%	67216.976	73483.290	+9.32%
`english-uniform`	3.905	4.015	+2.83%	17.087	17.772	+4.01%	30.560	31.308	+2.45%	53.511	59.015	+10.29%	91645.962	92614.343	+1.06%
`english-exp`	2.159	2.201	+1.94%	8.165	8.244	+0.97%	14.042	14.230	+1.34%	24.574	24.940	+1.49%	18266.653	18646.656	+2.08%
`proteins-uniform`	4.526	4.919	+8.68%	8.133	8.630	+6.12%	10.536	11.023	+4.62%	14.287	14.779	+3.45%	7388.779	7782.138	+5.32%
`proteins-exp`	1.113	1.199	+7.70%	1.702	1.809	+6.28%	2.115	2.216	+4.76%	2.815	2.985	+6.04%	668.101	715.175	+7.05%

Table 5.1: Cost per query for the *max-score* and *heavy-path* strategy.

We notice that the ratio seems to be better when considering the completions instead of the queries. The difference can be explained by the fact that taking the average of a top-k that contains less than k completions gives a higher proportion of the weight to those completions compared to the non-weighted total sum. Also note that since the weights used for the optimization process are different, the tries can be different too.

dataset	$c_{\text{avg-top-1}}$			$c_{\text{avg-top-5}}$			$c_{\text{avg-top-10}}$			$c_{\text{avg-top-20}}$			$c_{\text{avg-top-}\infty}$		
	opt	max	ϕ	opt	max	ϕ	opt	max	ϕ	opt	max	ϕ	opt	max	ϕ
`english-books`	2.373	2.378	+0.23%	3.213	3.235	+0.70%	3.444	3.494	+1.43%	3.673	3.748	+2.05%	4.989	5.578	+11.79%
`english-twitter`	2.341	2.342	+0.03%	3.070	3.084	+0.46%	3.333	3.390	+1.71%	3.508	3.559	+1.46%	4.710	5.169	+9.75%
`english-uniform`	3.905	4.015	+2.83%	4.495	4.680	+4.10%	4.660	4.842	+3.92%	4.760	5.154	+8.28%	5.329	5.847	+9.72%
`english-exp`	2.159	2.201	+1.94%	2.441	2.492	+2.06%	2.511	2.587	+3.03%	2.569	2.664	+3.73%	2.871	3.187	+11.00%
`proteins-uniform`	4.526	4.919	+8.68%	4.694	5.152	+9.75%	4.716	5.220	+10.69%	4.729	5.283	+11.72%	4.760	5.845	+22.80%
`proteins-exp`	1.113	1.199	+7.70%	1.139	1.243	+9.14%	1.142	1.254	+9.77%	1.145	1.266	+10.62%	1.158	1.316	+13.60%

Table 5.2: Average cost per completion for the *max-score* and *heavy-path* strategy.

These results show that the improvements can go from very modest to significant depending on the corpus. The experiment also suggests that the improvement is better with weights following uniform or exponential distribution that those found from empirical frequencies. The data also shows that the improvement is significantly better for the protein corpus than English's. To conclude, we note that although the practical improvement can go from very low to modest, there is no additional cost from the standpoint of the asymptotic running time complexity to execute the *heavy-path* strategy.

Chapter 6

Conclusion and Future Work

To conclude this thesis, we reiterate that we have provided a proof showing the optimality of the *heavy-path* decomposition on the average number of node accesses to answer top-k completion queries. This main result allows to construct the optimal path-decomposed trie with the greedy *heavy-path* technique which significantly reduces the running time from the naïve dynamic programming approach. We also show dynamic operations on the optimal PDT with complexity based on the length of the modified word.

6.1 Future Work

We use the number of pointer accesses or the depth of a node in the path-decomposed trie as the cost to access that node. Although this assumption makes sense for a simplified model, it ignores some parts of the actual computations necessary to execute queries. By refining the model, we could examine a more accurate model of computing and achieve a PDT that is optimal in regards to this advanced model. Examples of refinements to the cost function include:

- Considering the number of character comparisons made by a search. We already have an upper bound of $|p|$ character comparisons for a prefix p, but we don't take this cost into account in the cost function.
- Counting the exact number of node accesses for top-k enumeration.
- Taking into account the exact cost for a specific representation of the trie.

Another approach to consider is the optimality of the trie in regards to the memory hierarchy. The actual packing of the trie in memory can be made such that accessing part of the structure contains others parts that are relevant for the rest of the query. This would allow the system to reduce the number of copies of pages to higher levels of memory as the relevant information can be packed in the same pages. In our approach, we disregarded the actual representation of the path-decomposed trie as some previous research [13] already considered this problem, although with no explicit attention to the optimality of the layout or to memory hierarchy considerations.

Lastly, we could look into further refinements of the algorithms used to maintain the optimal trie over dynamic operations. They contain a factor on the alphabet size ($|\Sigma_{\mathcal{W}}|$) that we believe could be improved.

References

[1] Project Gutenberg, 2016. URL https://www.gutenberg.org.

[2] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, December 2005. ISSN 0178-4617. doi: 10.1007/s00453-004-1146-6. URL http://dx.doi.org/10.1007/s00453-004-1146-6.

[3] Iwona Bialynicka-Birula and Roberto Grossi. *String Processing and Information Retrieval: 12th International Conference, SPIRE 2005, Buenos Aires, Argentina, November 2-4, 2005. Proceedings*, chapter Rank-Sensitive Data Structures, pages 79–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-32241-2. doi: 10.1007/11575832_10. URL http://dx.doi.org/10.1007/11575832_10.

[4] A. Bondy and U.S.R. Murty. *Graph Theory*. Graduate Texts in Mathematics. Springer London, 2011. ISBN 9781846289699. URL https://books.google.ca/books?id=HuDFMwZOwcsC.

[5] Nieves R. Brisaboa, Rodrigo Cánovas, Miguel A. Martínez-Prieto, and Gonzalo Navarro. Compressed string dictionaries. *CoRR*, abs/1101.5506, 2011. URL http://arxiv.org/abs/1101.5506.

[6] David Richard Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Waterloo, Ont., Canada, Canada, 1998. UMI Order No. GAXNQ-21335.

[7] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009. doi: 10.1137/070710111. URL http://dx.doi.org/10.1137/070710111.

[8] René de la Briandais. File searching using variable length keys. In *Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), pages 295–298, New York, NY,

USA, 1959. ACM. doi: 10.1145/1457838.1457895. URL http://doi.acm.org/10.1145/1457838.1457895.

[9] Stephane Durocher. A simple linear-space data structure for constant-time range minimum query. *CoRR*, abs/1109.4460, 2011. URL http://arxiv.org/abs/1109.4460.

[10] N. Faria, R. Silva, and J.L. Sobral. Impact of data structure layout on performance. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 116–120, Feb 2013. doi: 10.1109/PDP.2013.24.

[11] Paolo Ferragina, Roberto Grossi, Ankur Gupta, Rahul Shah, and Jeffrey Scott Vitter. On searching compressed string collections cache-obliviously. In *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '08, pages 181–190, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-152-1. doi: 10.1145/1376916.1376943. URL http://doi.acm.org/10.1145/1376916.1376943.

[12] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1):4:1–4:33, nov 2009. ISSN 0004-5411. doi: 10.1145/1613676.1613680. URL http://doi.acm.org/10.1145/1613676.1613680.

[13] Roberto Grossi and Giuseppe Ottaviano. Fast compressed tries through path decompositions. *CoRR*, abs/1111.5220, 2011. URL http://arxiv.org/abs/1111.5220.

[14] Bo-June Paul Hsu and Giuseppe Ottaviano. Space-efficient data structures for top-k completion. In *WWW 2013*. ACM, May 2013. URL http://research.microsoft.com/apps/pubs/default.aspx?id=201369.

[15] D.E. Knuth. Optimum binary search trees. *Acta Informatica*, 1(1):14–25, 1971. ISSN 0001-5903. doi: 10.1007/BF00264289. URL http://dx.doi.org/10.1007/BF00264289.

[16] Shibamouli Lahiri. Complexity of word collocation networks: A preliminary structural analysis. *CoRR*, abs/1310.5111, 2013. URL http://arxiv.org/abs/1310.5111.

[17] Guoliang Li, Jiannan Wang, Chen Li, and Jianhua Feng. Supporting efficient top-k queries in type-ahead search. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '12, pages

355–364, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1472-5. doi: 10.1145/2348283.2348333. URL http://doi.acm.org/10.1145/2348283.2348333.

[18] W. Li. Random texts exhibit Zipf's-law-like word frequency distribution. *Information Theory, IEEE Transactions on*, 38(6):1842–1845, Nov 1992. ISSN 0018-9448. doi: 10.1109/18.165464. URL http://dx.doi.org/10.1109/18.165464.

[19] Michael Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Internet Mathematics*, 1(2):226–251, 2004. doi: 10.1080/15427951.2004.10129088. URL http://dx.doi.org/10.1080/15427951.2004.10129088.

[20] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. *CoRR*, abs/cs/0610001, 2006. URL http://arxiv.org/abs/cs/0610001.

[21] Research Collaboratory for Structural Bioinformatics. RCSB Protein Data Bank – RCSB PDB, 2016. URL http://www.rcsb.org/pdb/home/home.do#Subcategory-download_sequences.

[22] Kunihiko Sadakane and Gonzalo Navarro. Fully-functional static and dynamic succinct trees. *CoRR*, abs/0905.0768, 2009. URL http://arxiv.org/abs/0905.0768.

[23] Sanders Analytics. Twitter Sentiment Corpus, 2011. URL http://www.sananalytics.com/lab/index.php.

[24] C. E. Shannon. A mathematical theory of communication. *Bell system technical journal*, 27, 1948.

[25] Wikipedia. Heap (data structure) — Wikipedia, the free encyclopedia, 2015. URL https://en.wikipedia.org/wiki/Heap_(data_structure).

[26] WinEdt. Dictionaries WinEdt.org, 2016. URL http://www.winedt.org/Dict/. Last visited on 08/2/2016.

[27] Guoqing Zhang, Mei Rong, and Guangquan Zhang. A succinct string dictionary index in external memory. *International Journal of Database Theory and Application*, 7(3):13–22, 2014.