СОФИЙСКИ УНИВЕРСИТЕТ
"СВ. КЛИМЕНТ ОХРИДСКИ"

ФАКУЛТЕТ ПО МАТЕМАТИКА И ИНФОРМАТИКА

Стефан Владимиров Герджиков

# ЕФЕКТИВЕН АЛГОРИТЪМ ЗА ПРИБЛИЖЕНО ТЪРСЕНЕ В РЕГУЛЯРНИ МНОЖЕСТВА

## ДИСЕРТАЦИЯ

**за придобиване на образователната и научна степен "доктор"**
**в професионално направление 4.5 "Математика"**
**по научната специалност 01.01.01 "Математическа логика"**

Научен ръководител
**доцент д-р Стоян Михов**

**СОФИЯ, септември 2013**

# Резюме

В настоящия труд разглеждаме проблема за ефективно намиране на думите от даден (регулярен) език, които са близки до дадена дума. Близостта в това изследване е ортографска близост, тоест всяко отклонение в последователността или вида на буквите от оригинала се наказва с цена, която е цяло положително число. Конкретните ортографски замени могат да бъдат произволни, стига техният брой да е краен и те, както и техните цени да са предварително зададени. Проблемът е да се намерят всички думи в даден регулярен език, които са близки до дадена дума, $V$, с цена, ненадхвърляща $q|V|$, където $q \in (0; 1)$ е рационално число, а $|V|$ е дължината на думата $V$.

В настоящия труд е изложено ефективно алгоритмично решение на поставения по-горе проблем. В своята същност то представлява метод от класа разделяй и владей като първоначалната заявка, думата $V$, се разделя на по-къси поддуми, които от своя страна определят заявки от същия вид. По-късите заявки се оказват по-лесни за разрешаване, а след това могат да се комбинират, така че да дадат решение и на първоначалната задача.

С цел осъществяването на тази идея, в настоящата работа е разработен прост алгебричен апарат, който позволява изследването на свойствата на дадено ортографско разстояние и как то се отразява при търсенето на близки до дадена дума думи от даден език. Резултатите от този подход са основата за разработването на скицирания по-горе алгоритъм. Те позволяват оптималното решение на всяка от възникващите подзадачи в случая на краен език. Това означава, че времето за изпълнението на алгоритъма е пропорционално на резултатите, които той трябва да генерира. В общия случай, когато езикът е произволен регулярен, алгоритъмът е почти оптимален. Излишеството се състои в това, че в края на решаването на определена подзадача, думите генерирани в това решение, трябва да бъдат прочетени още веднъж.

Ефективността на изложения алгоритъм е обоснована при известни предположения за регулярния език, ортографското разстояние и рационалния параметър $q$. За получаването на такъв резултат се използва комбинаторно-вероятностен подход, който използва метода на пораждащите функции. Той дава оценка отгоре за очакваната сложност на получения алгоритъм, която е линейна функция относно дължината на заявката с параметри, зависещи от структурата на езика, ортографското разстояние и параметъра $q$. Въпреки че при определени ситуации тези параметри може да не са състоятелни, тоест да бъдат $+\infty$, показани са достатъчни условия, които осигуряват съществуването на тези параметри и тяхната крайност.

В последната глава на настоящата работа предлагаме нов подход за определяне на близост между думи. Неговата основна идея е да се отчетат типичните операции, определящи разликите между търсената и дадената дума, според структурата на думите в цялост и тяхната контекстна зависимост. Предложена е практическа реализация на такъв метод, която позволява ефективно търсене на най-близката до дадена дума и адекватността на тази реализиция е потвърдена експериментално.

# Резюме на получените резултати и декларация за оригиналност на труда

Авторът смята, че основни приноси на дисертационния труд са следните резултати:

1. **Общ апарат за изучаване на свойствата на подравнявания от думи и ефективното генериране на кандидати за корекция**. Този апарат, основаващ се на концепцията за множествата от подравнявания и списъци от разстояния, е разработен в Глава 4.

2. **Нов алгоритъм за приближено търсене в произволно регулярно множество от думи.** Това е алгоритъмът по схемата разделяй и владей, който е представен подробно в Глава 5.

3. **Нов алгоритъм за приближено търсене за произволно ортографско разстояние.** Това е алгоритъмът представен в Глава 6, който използва схемата разделяй и владей и решава проблема за приближено търсене за произволно ортографско разстояние.

4. **Практическа ефективност на предложените алгоритми.** Предложеният алгоритъм има свойството, че постепенно увеличава допустимото разстояние. По този начин броят на генерираните кандидати се поддържа разумно малък. Допълнителен аргумент за практическата ефективност на алгоритъма са и резултатите от Твърдения 5.3.6 и 5.3.15, които казват, че всеки кандидат за корекция в този алгоритъм се генерира веднъж, по-точно $O(1)$ пъти.

5. **Теоретична ефективност на предложения алгоритъм.** Предложен е вероятностен подход, който теоретично аргументира ефективността на алгоритъма, Твърдение 7.2.2 и 7.2.7.

6. **Нов подход за дефинирине на близост между думи.** Глава 8 представя нов подход за дефиниране на близост между думи, който отчита цялостната структура на езика, а също и контекстната информация в рамките на самите думи. Концептуалните предимства на този подход са потвърдени експериментално, а ефективността на разработения алгоритъм е оценена в Твърдение 8.2.9.

Идеята, която стои зад Резлутатите 1–5 развива предишни идеи на Myers, [47], Baeza-Yates и Navarro, [49], Mihov и Schulz, [42]. За нейната реализация са използвани класически резултати от теорията на крайните автомати и частично предишни резултати на Blumer et al., [11, 12], Ko и Aluru, [34]. За постигането на Резултат 5 е използван алгебричен подход, предложен от Eilenberg, [17], класически резултати от линейната алгебра, теория на графите, теория на крайните автомати.

Резултат 6 е следствие от оригинална математическа интерпретация на структурата, предложена от Blumer et al., [12]. За нейната алгоритмична

реализация са приложени идеи на Aho и Corasick, [7], а също и общ метод, предложен от Hart et al., [24, 25], от който новият подход съумява да се възползва.

Резултатите 1–6 са отразени в една самостоятелна и няколко статии в съавторство със Стоян Михов, Петър Митанкин, Klaus Schulz и Владислав Ненчев:

1. **Some algebraic properties of alignments of words, S. Gerdjikov, Comptes rendu de l'Academie bulgare des Sciences, 65(10):1311–1319, 2012**,

   Тази статия е самостоятелна. Тя представя основните стъпки, които водят до Резултати 1 и 5. По-точно, тази статия въвежда и описва основните свойства на списъците от разстояния и множества от подравнявания, които са в основата на Твърдения 5.3.6 и 5.3.15. Тя също представя Лема 7.1.6, която се използва съществено при доказателството Твърдения 7.2.2 и 7.2.7.

2. **WallBreaker - overcoming the wall effect in similarity search, S. Gerdjikov, S. Mihov, P. Mitankin, and K. U. Schulz, ACM Proceedings of the 2013 Joint EDBT/ICDT Workshops, 2013,**

   и нейната пълна версия:

   **Good parts first - a new algorithm for approximate search in lexica and string databases, S. Gerdjikov, S. Mihov, P. Mitankin, and K. U. Schulz, ArXiv, 2013 e-prints:http://adsabs.harvard.edu/abs/2013arXiv1301.0722G.**

   Тези две статии са в съавторство със Петър Митанкин, Стоян Михов, и Klaus Schulz. Алгоритъмът за приближено търсене, описан в статиите, е резултат, постигнат в рамките на семинара, воден от Стоян Михов, където Петър Митанкин и авторът едновременно работеха и представяха своите идеи за решаването на този проблем. Така, заедно, те достигнаха до две еквивалентни решения в случая на Левенщайн разстояние. Различията бяха в линейните структури от данни, които се използваха за представянето на множеството от думи.

   Понататъшните изследвания на Петър Митанкин доведоха до окончателния вариант, представен в тези статии, в който се използва още по-компактна структура, именно тази от [12, 29].

   В същото време дисертантът обобщи метода за произволно ортографско разстояние, Резултат 3.

   Тези части от статиите [20] и [21] представляват основата за постигането на Резултати 2, 3 и 4, а експерименталните резултати в [20] и независимото оценяване по време на форума Workshop on Scalable Similarity Search Strings/Join, Genoa, 2013, емпирично потвърждават Резултат 4.

3. **Extraction of spelling variations for noisy text correction. S. Gerdjikov, S. Mihov, and V. Nenchev, In Proceedings of 12th International Conference on Documents Analysis and Recognition 2013, 2013, p.324–328.**

   Тази статия е в съавторство със Стоян Михов и Владислав Ненчев. В нея авторът има основен принос при създаването, разработването и конкретната реализация на описания в статията подход, който е всъщност Резлутат 6.

Някои от тези резултати бяха представени на международни форуми:

- **WallBreaker - overcoming the wall effect in similarity search, S. Gerdjikov, S. Mihov, P. Mitankin, and K. U. Schulz, on the EDBT/ICDT Workshop for Scalable String Similarity Search/Join, Genoa, Italy, 2013.** (презентация на С. Герджиков)

  При тази изява авторът представи основните идеи от статиите [21] и [20] с акцент върху Резултата 4 – практическата ефективност на алгоритъма за приближено търсене.

- **Extraction of spelling variations for noisy text correction. S. Gerdjikov, S. Mihov, and V. Nenchev on the 12th International Conference on Documents Analysis and Recognition, Washington, DC, USA, 2013.** (постер, представен от С. Герджиков)

  С този постер авторът илюстрира резултатите от [22]. В дискусиите с експерти, с научни интереси в обработката на исторически текстове и OCR-корекция, авторът представи Резултата 6 от различни гледни точки, като по този начин мотивира неговата широка приложимост.

- **On Modernisation of Historical Texts. S. Gerdjikov, Computability in Europe 2012, Cambridge, UK, 2012.** (презентация С. Герджиков)

  В тази презентация авторът представи Резултат 6 в контекста на нормализация на английски текстове от 17 век.

Съществени части от Резултати 5 и 6 бяха представени на Пролетните сесии на Факултета по математика и информатика на Софийския Университет:

- **Доколко са регулярни правописните промени в българския език от 19 век до днес? С. Герджиков, Пролетна сесия на Факултета по математика и информатика на Софийския Университет, 2012** (презентация пред катедра Математическа логика и приложенията ѝ)

  В тази презентация авторът представи Резултат 6 върху конкретната задача за нормализация на български език от 19 век.

- **Комбинаторен етюд: "Подравнявания на думи"С. Герджиков, Пролетна сесия на Факултета по математика и информатика на Софийския Университет, 2011** (презентация пред катедра Математическа логика и приложенията ѝ)

  В тази презентация авторът представи комбинаторния резултат от Lemma 7.1.6, който води до Резултат 5.

Въз основа на предишните параграфи, авторът заявява, че настоящата дисертация е оригинален научен труд. Употребата на предишни резултати е отразена по честен начин като съответните източници са цитирани според условията на авторските права на техните автори, и/или издатели и/или други притежатели на конкретните авторски права.

S O F I A  U N I V E R S I T Y
"St. K L I M E N T  O H R I D S K I"

F A C U L T Y  F O R  M A T H E M A T I C S  A N D  I N F O R M A T I C S

**Stefan Vladimirov Gerdjikov**

# EFFICIENT ALGORITHM FOR THE APPROXIMATE SEARCH PROBLEM IN REGULAR SETS

## DISSERTATION

in partial fulfilment of the requirements of the degree

**Doctor of Philosophy in Mathematics**

Supervisor
**Dr. Stoyan Mihov**

**Sofia, September 2013**

# Contents

# List of Figures

# Introduction

The Levenshtein edit-distance originally arose in the context of information transmission in noisy channels and coding theory where its combinatoric properties were studied by Sellers, [56], Levenshtein, [37], etc. In this model a sequence of characters, i.e. a word, should be sequentially transmitted through a channel. However, due to noise, the resulting word can differ from the original one in several ways: (i) some of the original characters may have been *substituted* with a different character, (ii) some of the original characters may have been *deleted*, (iii) new characters may have been *inserted* by the channel. In this scenario, the noisy channel is considered as a "black box" so we do not know which processes have taken place during the transmission and we can observe only the resulting word as a whole. The general problem is to reconstruct the original word from the corrupted one that we see.

Stated in this way the problem is too general to get a reasonable answer. In order to become more sensible it requires some further specification.

Firstly, it is natural to assume that the original word is not an arbitrary one but rather belongs to some domain, $\mathcal{D}$. In different application areas this may mean different things. For instance, in the natural languages processing, it is often the case that the word is a single word in a specific language or a sentence in some language. Thus, the original word is not simply a jumble of characters but a sequence of characters that forms a word in a dictionary, or a sequence of such words. The noisy channel then arises in a natural way by identifying it for example with an OCR-engine, that strives to recognise optical characters and partially fails. In the bioinformatics, the original word can be a sequence of nucleotide bases that originates from a specific genome, or a sequence of alpha-amino-acids that represent a certain protein from a collection of proteins. In this scenario the biological mutations can be modelled as a noisy channel. Other applications may ask for different domains and different models of a noisy channel.

Secondly, it seems highly improbable that the noisy channel would corrupt the entire word. Or, more correctly, we would be extremely powerless to reconstruct the original word given that the noisy channel may have corrupted it completely. Hence, the second reasonable assumption is that the noise introduced by the channel is bounded by some threshold, say $b$. Now we can restate our original problem as follows. *Given the domain of the original word $\mathcal{D}$, the corrupted word, $V$, and the (noise-)threshold, $b$, which are the possible original*

*words?*

On a very high level, this is the *approximate search problem*. Still, it rises some questions. What does "given the domain" mean? What is the appropriate value for the (noise-)threshold? Finally, "the possible original words" is not quite the same as "the reconstructed word", one would expect to obtain. With this last remark we cannot argue and in the general framework described above it is hardly manageable to do better than list *all* the possible original words and then verify them by some other means to obtain *the one*.

"Given the domain" assumes a finite representation of the domain, $\mathcal{D}$. One possibility is that $\mathcal{D}$ is simply a finite set of words, i.e. a single word, [60], a set of infixes of a long text,[47, 9, 49, 15], finite set of words,[16]. An alternative is that $\mathcal{D}$ is given as a finite state automaton, [50, 36, 55, 42, 52]. This second case clearly extends the class of finite set of words.

The choice of the threshold, $b$, also varies and different scenarios are considered. Some models assume that $b$ is a general predetermined constant, [16, 15, 13]. On the other extremity, $b$ can be considered as specific for the resulting word, [50, 36, 55, 42], and thus it can be arbitrary. A compromise between these two scenarios is to determine the threshold $b$ as a fixed ratio, $q$, of the length of the resulting word, which can vary, [47, 9, 49]. Since the threshold $b$ essentially determines the number of the possible original words, the bigger $b$ the more possible candidates must be verified afterwards. This suggests that $b$ should not be too big for practical needs. However, fixing $b$ might be also a bad choice, since this constraint might be vulnerable to miss the original word we are looking for. Thus, the middle way, where $b$ depends on a parameter, $q \in (0, 1)$ and the length of the corrupted word, seems to achieve a favourable trade-off.

With respect to this variety of *approximate search problems* arising from different assumptions for the domain, $\mathcal{D}$, and the threshold, $b$, we focus on the following which we call *approximate search problems in regular sets*. It is specified by a domain (or regular language) $\mathcal{D}$ represented as finite state automaton and a threshold $b$ that is determined as $q|V|$ where $q \in (0; 1)$ is fixed and $V$ is the resulting corrupted word. The problem is to retrieve all the words $U \in \mathcal{D}$ that are possible original words for the corrupted word, $V$.

Although the approximate search problem is easier than the problem of reconstruction of the original word based on the corrupted word, it is not a simple one. Though, some special instances of this problem have an efficient solution. The case when $\mathcal{D}$ is a singleton can be solved in time $O(b|V|)$ and space $O(b)$ via a standard programming algorithm, [60]. The case when $\mathcal{D}$ is the set of infixes of a long text, $T$, and $b \leq 1$ can be solved $O(|V| + occ + log|T|loglog|T|)$ and space $O(|T|)$ by an algorithm of Chan et al., [15], where *occ* stays for the number of the possible original words.

However, in general, when the threshold $b$ is not given in advance or the set of words is big or even arbitrary regular set such favourable results and efficient solutions are not described yet. In the sequel we shall summarise some of the approaches developed in order to solve the problem in practice. Although our aim is only to give the essence of these algorithms and avoid the technical details,

we shall need some terms, like *finite state automata*, *alignment*, (generalised and Levenshtein) *edit-distance*, *Ukkonen's algorithm*, that are formally introduced in Chapter 1.

## Forward Algorithm

In [50], Offlazer considers the following version of the approximate search problem, that is also similar to [33, 36] and [67]. Given a finite set of words $\mathcal{D}$ and a query word $V$ *which are all the words* $U \in \mathcal{D}$ which are similar to $V$. The formal restatement of the problem is rather straightforward:

```
Given:D a finite set of words
Input: V ∈ Σ*, b ∈ ℕ
Output: {U ∈ D | d_L(U,V) ≤ b}.
```

Here $d_L$ stays for the Levenshtein edit-distance. Note, that in this framework we distinguish between *given* and *input*. The reason is that $\mathcal{D}$ is interpreted as a dictionary of some language which is known in advance. The query then consists in recognising the input word $V$ in the context of this language. For these reasons the set $\mathcal{D}$ is regarded as something stable which is rarely updated, whereas there is freedom for the query word, $V$, and the threshold, $b$.

Oflazer represents the dictionary $\mathcal{D}$ as a deterministic finite state automaton $\mathcal{A} = \langle Q, \Sigma, s, \delta, F \rangle$. To answer the query, his algorithm combines the ideas of Ukkonen's algorithm, [60], and the straightforward left to right traversal of the automaton. The traversal of the automaton essentially generates the words of $\mathcal{D}$. The Ukkonen's algorithm is used during the traversal in order to cut the prefixes of words which are too far away from any prefix of $V$.

```
 Flag(d, b)
@1    return d ≤ b

 DFSearch(A, q, U, j, V, d, b)
@ 1    if q ∈ F and j = |V| then
@ 2       return U
@ 3    for ∀ σ ∈ Σ !δ(q, σ) do
@ 4       p = δ(q, σ)
@ 5       if σ = V_{j+1} then
@ 6          DFSearch(A, p, U ∘ σ, j + 1, V, d, b)
@ 7       else
@ 8          if Flag(d, b) then
@ 9             DFSearch(A, p, U ∘ σ, j + 1, V, d + 1, b)
@ 10       if Flag(d, b) then
@ 11          DFSearch(A, p, U ∘ σ, j, V, d + 1, b)
@ 12    if Flag(d, b) then
@ 13       DFSearch(q, U, j + 1, V, d + 1, b)

 Search(A, V, b)
@ 1   DFSearch(A, s, ε, 0, V, 0, b)
```

Note that in lines 3 to 9 the algorithm handles possible substitutions, in line 10 it considers the possible insertions and in line 12 it checks for deletions.

Having the more general concept of what an operation is, one can easily describe all these cases homogeneously and also reflect the possibility of different costs. This can be done by introducing a single loop on the admissible operations and checking which of them are compatible with the query word and with the threshold.

```
 GeneralDFSearch(A, q, U, j, V, d, b)
@ 2    if  q ∈ F  and  j = |V| then
@ 3       return U
@ 4    for  ∀ op ∈ Op do
@ 5       if  r(op) = V[j + 1..j + |r(op)|]) and Flag(d + c(op), b) then
@ 6          p = δ*(q, l(op))
@ 7          if  p is defined  then
@ 8             GeneralDFSearch(A, p, U ∘ l(op), j + |r(op)|, d + c(op), b)
@ 9          fi
@ 10      fi
@ 11   done

 GeneralSearch(A, V, b)
@ 1   GeneralDFSearch(A, s, ε, 0, V, 0, b)
```

Observe that the algorithm will always terminate, because the recursion is invoked with $d \leq b$ and $j \leq |V|$. On the other hand since each operation with $r(op) \neq \varepsilon$ is of positive cost ($c(op) \geq 1$) each successive invocation has either a greater parameter $j$, or the parameter $d$ strictly increases. Hence no viciousness will take place and the algorithm halts.

Depth first search is only one possibility for generating the entries of the dictionary. One can equally well profit from the breadth first search. It provides a better semantics of the algorithm but it suffers from greater space requirements for its implementation in practice. In this specific context we shall see that whereas the depth first search will report one and the same word several times, the breadth first search suppresses this effect. More importantly, it allows us to relate its efficiency with the number of alignments determined by the query word $V$ and the successful candidates in $\mathcal{D}$.

In [52], Reffle describes how a breadth first search can be realised in order to solve the approximate search problem for generalised edit-distance. Except the search strategy, Reffle also describes a nice application of the Aho-Corasick algorithm [7] that allows him to efficiently select the operation applicable at each step of the algorithm. The idea is essentially to keep track of the longest suffix of the currently processed word that is also a prefix of a right hand side of an operation. This fits nicely in the Aho-Corasick tree-structure and following the 'links' one deduces in constant time per operation all the operations that match the suffix of the currently processed word.

## Forward-backward algorithm

The bottleneck in the Oflazer's algorithm, regardless of its implementation, is that it generates all prefixes $U_i$ of words in the dictionary (or regular language) which are at edit-distance less than or equal to the threshold $b$ to some prefix $V_j$ of the query $V$. Provided that there are a lot of different prefixes $U_i$ represented in the automaton $\mathcal{A}$ and that the edit-operations $Op$ allow the flexibility of a large scope of alignments as it is in the Levenshtein edit-operations, the number of generated strings $U_i$ increases exponentially with $b$. On the other hand, even if $b$ is big, the size of the output, i.e. the number of words $U \in \mathcal{D}$ that are close to the query word $V$ can be small.

This feature of the approximate search problem is known as the *wall-effect*. This is one of the main differences between the problem considered by Ukkonen and that addressed by Oflazer. The wall-effect is not experienced for small sets, since their size is a more favourable upper bound than the mere exponent predicted by $b$. Once the dictionary $\mathcal{D}$ and the exponent $\exp(b)$ are comparable the initial blow-up caused by *unnecessary generated* prefixes becomes inevitable.

In order to deal with this problem, different approaches were considered. Some of them rely on a precomputed index. The algorithms of Cole et al., [16], and of Chan et al., [15], are tailored as to reduce the total size of $\mathcal{D}$ during the search phase. The idea is to represent each word not by a single path as it is in the deterministic automaton but by several paths which branch at certain points. In this way wild-cards, i.e. characters which can be substituted or deleted, are explicitly encoded in the branching points. Hence, each path encodes rather a class of alignments compatible with the word it reads and all possible other words compatible with this sample. This keying is then used in order to reduce the number of paths which are considered during the search. As a result the algorithm of Cole et al., [16], achieves worst-case bounds $O(3^b(\log N)^b)$ where $N = |V|$ is the length of the query word, $b$ is a fixed threshold, and it requires $O((\log|\mathcal{D}|)^b)$ additional storage space. Similar approaches were described by Mor and Fraenkel [46] and Boytsov [13]. Unfortunately, these approaches are sensitive to the threshold $b$ as the space requirements indicate. Since storage space is dependent on the size of the set $\mathcal{D}$, they cannot be applied for infinite sets.

An alternative approach was suggested by Mihov and Schulz, [55, 42]. They explore the following idea. If there are $b$ errors distributed on the entire word $V$, then either a prefix of $V$ contains less than $\frac{b}{2}$ errors or its complementary suffix obeys this property. This observation allows to suppress the initial wall-effect caused by the error-tolerance $b$ to the wall-effect induced by $\frac{b}{2}$. However this case distinction requires to traverse the dictionary twice. Once, as in the forward-method which corresponds to first align the prefixes, and once again, but this time traversing the dictionary and the query word backwards in reverse manner, this time considering the suffixes' alignments first. As we explained earlier the wall-effect depends exponentially on $b$, thus replacing $b$ by $\frac{b}{2}$ reduces its magnitude significantly and the disadvantage to traverse the dictionary twice is ignorable.

Since the contribution of this work generalises on this method we give some more details about it in the special case of Levenshtein edit-distance. Assume that $\omega$ is an alignment of $U$ against $V$ with cost $c(\omega) \leq b$. Suppose also that $V = V_0 \circ V_1$ where $V_0$ and $V_1$ are of (almost) equal lengths. Because each operation, substitution, deletion or insertion concerns at most one character of $V$, we can decompose $\omega = \omega_0 \circ \omega_1$ such that $r(\omega_i) = V_i$. On the other hand $c(\omega) = c(\omega_0) + c(\omega_1)$ and therefore $c(\omega_i) \leq \frac{b}{2}$ either for $i = 0$ or for $i = 1$. Now it should be clear that $U = l(\omega_0) \circ l(\omega_1)$. These considerations suggest the following modification of Oflazer's algorithm. If $j \leq \frac{|V|}{2}$, then $d$ must be less than or equal to $\frac{b}{2}$. After the first run of the modified algorithm we apply it once again for $\mathcal{D}^{rev}$ and $V^{rev}$, but instead of $U$, which this time belongs to $\mathcal{D}^{rev}$, one has to report $U^{rev}$. Again the dictionaries $\mathcal{D}$ and $\mathcal{D}^{rev}$ are represented via deterministic finite state automata, $\mathcal{A} = \langle \Sigma, Q, s, \delta, F \rangle$ and $\mathcal{A}^{rev} = \langle \Sigma, Q^{rev}, s^{rev}, \delta^{rev}, F^{rev} \rangle$, respectively.

```
 DFSearch2(A, q, U, j, V, d, b)
@ 1    if  q ∈ F  and  j = |V| then
@ 2       return U
@ 3    for  ∀ σ ∈ Σ !δ(q, σ) do
@ 4       p = δ(q, σ)
@ 5       if σ = V_{j+1}  then
@ 6          DFSearch(p, U ∘ σ, j + 1, V, d, b)
@ 7       else
@ 8          if  Flag(d, b) and (Flag(d, b/2) or j + 1 > |V|/2) then
@ 9             DFSearch(p, U ∘ σ, j + 1, V, d + 1, b)
@ 10       if Flag(d, b) and (Flag(d, b/2) or j > |V|/2) then
@ 11          DFSearch(p, U ∘ σ, j, V, d + 1, b)
@ 12    if Flag(d, b) and (Flag(d, b/2) or j + 1 > |V|/2) then
@ 13       DFSearch(q, U, j + 1, V, d + 1, b)

 Search2(A, A^rev, V, b)
@ 1    DFSearch2(A, s, ε, 0, V, 0, b)
@ 2    DFSearch2(A^rev, s^rev, ε, 0, V, 0, b)
```

One can easily transform the depth first search into a breadth first search as we already clarified this issue for Oflazer's algorithm. It is also not difficult to overcome the constraint of Levenshtein edit-distance. We shall have the opportunity to highlight the details which this generalisation entails further on.

## Myers's Algorithm

Mihov and Schulz were not the first to apply the *divide and conquer* strategy for an approximate search problem. The same idea was already utilised by Myers [47] for the following particular case of the approximate search problem:

`Given:` $T \in \Sigma^*$, $q \in (0; 1)$

`Input:` $V \in \Sigma^*$
`Output:` $\{U \mid U \in Inf(T) \ \ \& \ d_L(U,V) \leq q|V|\}$

The approximate search problem considered by Myers can be restated in terms of the Oflazer's model by setting:

$$\begin{aligned} \mathcal{D} &= Inf(T) \\ b &= q|V|. \end{aligned}$$

To solve a single query, Myers makes the following observation which is very similar to that of Hirschberg [26]. Imagine that the input $V$ is decomposed into a prefix $V_0$ and a suffix $V_1$, such that $V = V_0 \circ V_1$. Assume that some word $U \in \mathcal{L}$ can be obtained from $V$ with no more than $q|V|$ insertions, deletions and (proper) substitutions. Then either no more than $q|V_0|$ operations concern the prefix $V_0$, or no more than $q|V_1|$ operations concern the suffix $V_1$. But this means that $U$ can be decomposed into $U = U_0 \circ U_1$ such that the sum of edit-operations to transform $V_i$ into $U_i$ does not exceed $q|V|$ and fewer than $q|V_i|$ operations are needed to transform $U_i$ into $V_i$ either for $i = 0$, or for $i = 1$.

Formally this can be stated as (compare with the Lemma on page 3 in [47]):

**Lemma 0.0.1** *If $V = V_0 \circ V_1$ is a decomposition of the query word $V$ and $U \in \mathcal{L}$ with:*

$$d_L(U,V) \leq q|V|,$$

*then there exists a decomposition $U = U_0 \circ U_1$ such that:*

$$\begin{aligned} &d_L(U_0,V_0) + d_L(U_1,V_1) = d_L(U,V) \text{ and} \\ &d_L(U_0,V_0) \leq q|V_0| \text{ or } d_L(U_1,V_1) \leq q|V_1|, \end{aligned}$$

*where $d_L$ is the Levenshtein edit-distance induced by $(Op_L, c_L)$ (see Remark 1.3.3.)*

In case when $\mathcal{L}$ is the set of all infixes of a word (text) $T$, as it is in the problem of interest for Myers, $U_0, U_1 \in \mathcal{L}$ whatever the decomposition of $U \in \mathcal{L}$ might be. This suggests a straightforward divide and conquer approach to process the query. Namely one splits the initial query $V$ into two smaller determined by $V_0$ and $V_1$. After having computed the solutions for both of them recursively, we try to extend the solutions $U_0$ for $V_0$ *to the right* so that they find their appropriate suffixes $U_1$, and we extend the solutions $U_1$ generated by $V_1$ *to the left* so that they match with appropriate prefixes $U_0$. In this way the threshold for the recursive invocation is $q|V_i|$ and the original threshold $q|V|$ comes into play at the very last stage when only fewer candidates need to be explicitly verified.

Myers implements this idea in an algorithm that runs on average in $O(q|V||T|^{\varepsilon(q)} \log |T|)$ time where $\varepsilon(q) < 1$ is a convex function with $\varepsilon(0) = 0$. This means that if the text is a result of a sequence of independent Bernoulli trials and the query $V$ of length $|V| > \log^2 |T|$ is fixed, the expected value for the running time of the algorithm grows sublinearly w.r.t. the length of the text $T$.

To achieve this result Myers' algorithm uses two more ingredients besides Lemma 0.0.1. Schematically it computes an appropriate index for shorter words.

Then, instead of applying Lemma 0.0.1 recursively, it goes in a bottom-up fashion. At the extension steps, it processes subqueries simultaneously by the means of a dynamic programming similar to the Ukonnen's algorithm, [60].

In more details, the Myers' algorithm precomputes for each word $V$ of length $\lceil \log |T| \rceil$ all the *hits* in the text $T$. Essentially, a hit for $V$ is (a representation of) an infix of $T$ that is at edit-distance $\leq q|V|$ from $V$. This sort of indexing the text $T$ allows an immediate access to all the candidates for short words. The result of this index applied at the first step becomes a basis for the rest of the search which successively applies Lemma 0.0.1 in a bottom-up fashion.

Thus, given a query word, it is split into subwords of length approximately $\lceil \log |T| \rceil$ and the hits for these subwords are retrieved. The subsequent steps of the algorithm correspond to the extension-merge phases of the recursive algorithm we looked at above. It unifies the candidates sharing common positions in the text together and uses dynamic programming technique, [60], to verify which of the candidates can be successfully extended.

The core difference between the Myers' algorithm and the other algorithms, e.g. the forward or forward-backward method, is the following. The latter algorithms use a fixed number of values for a threshold of the magnitude of the original bound $b$. Commonly the filter is either the Ukkonen's dynamic programming scheme, [60], or a universal Levenschtein automaton, [42], and this imposes that insertions, deletions and substitutions of number $O(b)$ are allowed at each stage of the algorithm. This results in the wall-effect. Contrary to this concept, Myers applies a system of filters for subwords, $V_\alpha$, of $V$. Crucially, the threshold for the filter is a ratio $\frac{|V_\alpha|}{|V|}$ of the original bound $b$. Hence, the shorter the query $V_\alpha$ less number of errors are allowed in the candidates $U$. The original filter that allows $q|V|$ errors is therefore postponed for the very last stage of the algorithm when the candidates $U$ have already restricted the size of the searching space within the text $T$.

There are several problems which hinder the immediate application of Myers' algorithm in case of arbitrary regular language, $\mathcal{L}$. Firstly, $\mathcal{L}$ should not be closed under infixes. Hence, the homogeneity of the subproblems would be lost. To overcome this obstacle we shall rather use an automaton-based representation for the set of infixes of $\mathcal{L}$ which encodes in terms of terminal states those infixes which are actually words of $\mathcal{L}$. Secondly, the capacity to extend an infix into either direction (left or right) is crucial for the Myers' algorithm. Note, that we mean not an *arbitrary* extension of an infix but only such an extension which results into an infix of the language $\mathcal{L}$. In case when $T$ is a text and $\mathcal{L}$ constitutes of the infixes of $T$, this is easily achievable by considering $T$ as an array and each infix as a subarray $T[j..k]$. Hence, extending $T[j..k]$ to the left means simply look at position $T[j-1]$ into $T$ and extending to the right means to have a look at position $T[k+1]$. In case of an arbitrary regular language $\mathcal{L}$ this is no more the case, since the common way to think of $\mathcal{L}$ is as an automaton and it does not provide information about the extensions to the *left*. Finally, and probably this is the most essential problem, the index required by the Myers' algorithm grows with the size of the text $T$. Hence, it is inapplicable for infinite languages.

Another practical issue concerning the efficiency of the Myers' algorithm, is the naive representation of the generated candidates. Infixes spelling the same word but occurring at distinct positions in the text are generated. To overcome this problem, Baeza-Yates and Navarro, [9, 49], propose an algorithm that uses suffix trees, [23, 61]. Thus, different occurrences of the same infix are already represented uniformly. Unfortunately, the divide and conquer algorithm suffers from the usage of this data structure. Actually, it is only in the first step that one can start with an exact match or a reasonable error threshold. Immediately after the detection of the initial matches, the algorithm switches to the maximal threshold allowed by the query. Thus, it loses the nice feature of the Myers' algorithm to increase the threshold smoothly with the increase of the length of the candidates. Nevertheless, the same authors, Baeza-Yates and Navarro, show in [49] that using an appropriate index and under significant constraints on the threshold bounds, the proposed algorithm has a good theoretical complexity on average.

In the next chapters we are going to consider the *approximate search problem in regular sets* and generalising the ideas from [55, 42] and [47, 49] we shall describe an efficient algorithm for its solution.

Chapter 1 and Chapter 2 have preliminary character.

In Chapter 1, we give some basic definitions and formally state the approximate search problem in regular sets.

In Chapter 2, we shall present linear size representations of the infixes of finite sets of words.

In Chapter 3, we illustrate the basic concept of our approach on a small example. The main contributions of the current work are presented in Chapters 4 to 8.

In Chapter 4, we develop a formal framework of *alignments sets* and *edit-distance lists*. These notions allow us to formalise our approach. Intuitively, the alignments sets naively represent all possible ways the noisy channel may have processed the original word in order to obtain the corrupted word. They do not account for the specificity of the domain and may or may not reflect the threshold requirements. Thus they are impractical but mathematically convenient. The edit-distance lists represent alignment sets with respect to a specific domain (language). They take into account the domain and eliminate the redundancies introduced by the alignment sets reducing them only to the information that is relevant for the search.

In Chapter 5, we present our algorithm in a special case that extends the Levenshtein edit-distance, and estimate its complexity with respect to the number of generated candidates, including the false ones. We also provide a faster solution for the case of finite set of words that based on the data structures in Chapter 2 requires linear space. In the general case the space requirement is the space needed for the representation of *the infixes of the given language* (and not the language itself).

In Chapter 6, we generalise the approach described in Chapter 4, so that it captures the case of general edit-distance. This allows us to extend the algorithm developed in Chapter 4 and prove that it has similar characteristics as the basic one. Yet, the extended algorithm may require some additional bookkeeping.

In Chapter 7 we investigate a general framework that allows us to argue the efficiency of our algorithm. The assumption that it relies on is that the corrupted word is a result of a sequence of independent Bernoulli trials. We prove a combinatorial Lemma that relates the expected number of generated candidates with the structure of the original language. Using the analysis of the running time of the algorithm from Chapter 5 and Chapter 6 this allows us to conclude linear running time on average under certain constraints for the given language. Finally, we give sufficient conditions of quite a general kind that assert the consistency of our result.

Finally, in Chapter 8 we address the challenging problem to reconstruct the original word from an observed noisy word. It uses a finite (multi)set of examples saying that a particular noisy word stays for a particular original word, and a dictionary describing the set of original words. Based on these data we develop general approach that explores the structure of the examples and

the structure of the dictionary in order extract the operations typical for the source of noise, from the structure of a finite set. In a quite standard way we define the likeliness for a particular noisy word to stay for a particular original word. As a result we arrive at a practical algorithm that generates a ranked list of correction candidates for an arbitrary noisy word. The adequacy of this approach is defended empirically.

In the Appendix we add a further note to the additional index structure that may be needed in a specific case of the algorithm's parameters.

The results from Chapter 4, Chapter 6 and Chapter 7 were published in [19] and the algorithm from Chapter 5 was essentially described in [20, 21]. The approach in Chapter 8 was published in [22].

I am grateful to Stoyan Mihov for the proposed research topic and for the leadership of a great seminar in Natural Language Processing. I express my acknowledgement to the participants in this seminar for prolific discussions, for the opportunity they give me to informally expose new ideas and for sharing their experience. Most of the results presented in the current work are due to preparation for the sessions of this seminar or to discussions held during these sessions. Other results obtained in the same fruitful academic atmosphere found other appropriate venues. Yet, the most interesting sessions remain those that do not yield any particular result, but conclude with another question; question that motivates further research, conjectures and consideration – a good reason for anticipating the next seminar session and its discussions.

# Chapter 1

# Preliminaries

In this Chapter we introduce the basic notions for regular languages, alignments and some basic linear algebra notions. We also formally state the approximate search problem in Section 1.6 that we shall consider in more details in the next Chapters.

## 1.1 Words and Languages

An *alphabet* $\Sigma$ is a finite set whose elements are called *characters*. We denote with $\Sigma^*$ the set of all finite sequences of characters of $\Sigma$. Hence a *word* $W$ over $\Sigma$ is simply $W \in \Sigma^*$. We use $U, V, W, \ldots$ to denote *words* over the alphabet $\Sigma$. For a word $U$ the *length* of $U$ is determined as the number of characters in $U$. We denote the length of $U$ with $|U|$. There is a unique word of length 0 over an alphabet $\Sigma$ which is called the *empty word*. We denote the empty word with $\varepsilon$.

The *concatenation* of words $U = u_1 u_2 \ldots u_m$ and $V = v_1 v_2 \ldots v_n$ is the word $U \circ V = u_1 u_2 \ldots u_m v_1 v_2 \ldots v_n$. A word $U$ is a *prefix* of a word $V$ if there exists a word $W$ with $V = U \circ W$. Dually, $U$ is a *suffix* of a word $V$ if there exists a word $W$ with $V = W \circ U$. We say that $U$ is an *infix* of a word $V$ if there are words $W_1, W_2$ with $V = W_1 \circ U \circ W_2$. We denote the infix of $V$ starting at position $i$ and terminating at position $j$ with $I_i^j(V)$, i.e.:

$$I_i^j(V) = v_i \circ v_{i+1} \ldots v_j.$$

Assuming that $\Sigma = \{\sigma_1, \ldots, \sigma_{|\Sigma|}\}$ is an alphabet which induces the natural order on its characters $\sigma_1 < \sigma_2 < \cdots < \sigma_{|\Sigma|}$, we say that a word $U = u_1 \ldots u_m$ is *lexicographically smaller* than a word $V = v_1 \ldots v_n$ if and only if:

$$U = I_1^m(V) \text{ and } m < n \text{ or } \exists (k < m, n)[I_1^k(U) = I_1^k(V) \text{ and } u_{k+1} < v_{k+1}].$$

In this case we shall write $U \prec_{lex} V$. We use $U \preceq_{lex} V$ as a short hand for:

$$U = V \text{ or } U \prec_{lex} V.$$

It should be clear that $\preceq$ is a linear ordering on words.

**Definition 1.1.1** Given an alphabet $\Sigma$, a language over $\Sigma$ is an arbitrary set $\mathcal{L} \subseteq \Sigma^*$.

Next we list some basic operations on languages.

**Definition 1.1.2** Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be languages over an alphabet $\Sigma$, then:

1. union of $\mathcal{L}_1$ and $\mathcal{L}_2$ is the language $\mathcal{L}_\cup = \mathcal{L}_1 \cup \mathcal{L}_2$.
2. concatenation of $\mathcal{L}_1$ with $\mathcal{L}_2$ is the language:
$$\mathcal{L}_\circ = \mathcal{L}_1 \circ \mathcal{L}_2 = \{U_1 \circ U_2 \,|\, U_1 \in \mathcal{L}_1, \text{ and } U_2 \in \mathcal{L}_2\}.$$
3. for an integer number $n \in \mathbb{N}$ the power $\mathcal{L}_1^n$ is defined recursively as:
$$\mathcal{L}_1^0 = \{\varepsilon\}$$
$$\mathcal{L}_1^{n+1} = \mathcal{L}_1^n \circ \mathcal{L}_1.$$
4. the iteration of the language $\mathcal{L}_1$ is the language $\mathcal{L}_*$ defined as:
$$\mathcal{L}_* = \mathcal{L}_1^* = \cup_{n=0}^{\infty} \mathcal{L}_1^n.$$

With this definition it is easy to see that $\Sigma^n$ is the set of all words $U \in \Sigma^*$ with $|U| = n$. For a word $U = u_1 u_2 \dots u_n$ with $u_j \in \Sigma$, the *reverse* of $U$ is $U^{rev} = u_n \dots u_2 u_1$.

The focus of our research is the class of *regular languages*. There are different ways to introduce them [27, 17]. One of them is the following:

**Definition 1.1.3** Given an alphabet $\Sigma$, the set of regular languages $Reg(\Sigma)$ is the least set of languages over $\Sigma$ which is closed under union, concatenation and iteration and such that:

$$\emptyset \in Reg(\Sigma) \text{ and } \forall \sigma \in \Sigma[\{\sigma\} \in Reg(\Sigma)].$$

**Definition 1.1.4** Let $\mathcal{L}$ be a language, then we define:

1. the set of prefixes of $\mathcal{L}$ as:
$$Pref(\mathcal{L}) = \{U \in \Sigma^* \,|\, \exists V[U \circ V \in \mathcal{L}]\}$$
2. the set of suffixes of $\mathcal{L}$ as:
$$Suf(\mathcal{L}) = \{V \in \Sigma^* \,|\, \exists U[U \circ V \in \mathcal{L}]\}$$
3. the set of infixes of $\mathcal{L}$ as:
$$Inf(\mathcal{L}) = \{W \in \Sigma^* \,|\, \exists U, V[U \circ W \circ V \in \mathcal{L}]\}$$
4. the reverse language of $\mathcal{L}$ as:
$$\mathcal{L}^{rev} = \{U^{rev} \,|\, U \in \mathcal{L}\}.$$

In the particular case when $\mathcal{L} = \{W\}$ is a singleton, we shall use $Pref(W)$, $Inf(W)$ and $Suf(W)$ instead of $Pref(\{W\})$, $Inf(\{W\})$ and $Suf(\{W\})$, respectively.

## 1.2 Finite State Automata

The finite state automata are syntactic devices which provide a finite representation of regular languages [17, 27, 32]. In this section we refresh some of the basic definitions which are necessary for their understanding.

**Definition 1.2.1** A finite state automaton is $\mathcal{A} = \langle \Sigma, Q, I, \Delta, T \rangle$ where $\Sigma$ is an alphabet, $Q$ is a finite set of states, $I \subseteq Q$ and $T \subseteq Q$ are initial and terminal states and $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation of $\mathcal{A}$. $\mathcal{A}$ is deterministic if $|I| = 1$ and $\Delta$ is the graph of a partial function $\delta : Q \times \Sigma \to Q$.

**Definition 1.2.2** Let $\mathcal{A} = \langle \Sigma, Q, I, \Delta, T \rangle$. For a transition $t = \langle p', a, p'' \rangle \in \Delta$ we denote with $\iota(t) = p'$ the initial state of the transition, with $\tau(t) = p''$ the terminal state of the transition and with $\lambda(t) = a$ the label of the transition. A path $\pi$ in an automaton $\mathcal{A} = \langle \Sigma, Q, I, \Delta, T \rangle$ is a finite sequence of transitions $\pi = t_1 t_2 \dots t_n$ where $t_k \in \Delta$ and $\tau(t_k) = \iota(t_{k+1})$ for each $1 \leq k < n$. We denote with $|\pi| = n$ the length of the path and we use the notions:

$$\iota(\pi) = \iota(t_1), \quad \tau(\pi) = \tau(t_n),$$
$$\lambda(\pi) = \lambda(t_1) \circ \lambda(t_2) \circ \dots \circ \lambda(t_n)$$

for the initial and terminal state of $\pi$ and the label of $\pi$, respectively. We denote the set of all paths in the automaton $\mathcal{A}$ with $\Pi(\mathcal{A})$.

**Definition 1.2.3** A path $\pi$ in an automaton $\mathcal{A} = \langle \Sigma, Q, I, \Delta, T \rangle$ is called accepting if and only if $\iota(\pi) \in I$ and $\tau(\pi) \in T$. We denote with $Acc(\mathcal{A})$ the set of the accepting paths in $\mathcal{A}$. The language $\mathcal{L}(\mathcal{A})$ of the automaton $\mathcal{A}$ is thus the set of labels of all accepting paths in $\mathcal{A}$.:

$$\mathcal{L}(\mathcal{A}) = \{\lambda(\pi) \,|\, \pi \in Acc(\mathcal{A})\}$$

Kleene's characterisation, [32], of the class of regular languages states that this is the class of languages recognisable with finite state automata:

**Theorem 1.2.4** *If $\Sigma$ is an alphabet, a language $\mathcal{L}$ is regular over $\Sigma$ if and only if there is an automaton $\mathcal{A}$ over $\Sigma$ with $\mathcal{L}(\mathcal{A}) = \mathcal{L}$.*

Using this characterisation it is not difficult [27] to constructively prove that for each regular language $\mathcal{L} \in Reg(\Sigma)$, the languages $Pref(\mathcal{L})$, $Suf(\mathcal{L})$, $Inf(\mathcal{L})$ and $\mathcal{L}^{rev}$ are regular. In particular we have the following results:

**Lemma 1.2.5** *Let $\mathcal{A} = \langle \Sigma, Q, I, \Delta, T \rangle$ be a finite state automaton, then $\mathcal{A}_{\mathcal{I}} = \langle \Sigma, Q, Q, \Delta, Q \rangle$ is a finite state automaton with $\mathcal{L}(\mathcal{A}_{\mathcal{I}}) = Inf(\mathcal{L}(\mathcal{A}))$.*

□

**Lemma 1.2.6** *Let $\mathcal{A} = \langle \Sigma, Q, I, \Delta, T \rangle$ be a finite state automaton, then $\mathcal{A}^{rev} = \langle \Sigma, Q, T, \Delta^{rev}, I \rangle$ where $\Delta^{rev} = \{\langle q, a, p \rangle \,|\, \langle p, a, q \rangle \in \Delta\}$ is a finite state automaton with $\mathcal{L}(\mathcal{A}^{rev}) = \mathcal{L}^{rev}(\mathcal{A})$.*

□

It is also a well known fact, [17, 27], that for each finite state automaton there is a finite state deterministic automaton recognising the same language:

**Lemma 1.2.7** *Let $\mathcal{A} = \langle \Sigma, Q, I, \Delta, T \rangle$ be a finite state automaton, then there is a deterministic automaton $\mathcal{A}_D = \langle \Sigma, Q_D, I, \delta, T_D \rangle$ where $Q_D \subseteq 2^Q$ and $\delta$ is a graph of a function.*

□

A special case of deterministic automata are *tries* for a finite set of words $\mathcal{D} \subseteq \Sigma^*$. They will be useful for the representation of the finite subsets of the original language $\mathcal{L}$ which we shall be investigating:

**Definition 1.2.8** Let $\mathcal{D}$ be a finite language over $\Sigma$. A trie for $\mathcal{D}$ is an automaton $\mathcal{T}_\mathcal{D} = \langle Q_\mathcal{D}, \Sigma, I_\mathcal{D}, \Delta_\mathcal{D}, T_\mathcal{D} \rangle$ where:

$$\begin{aligned} Q_\mathcal{D} &= \{p_W \,|\, W \in Pref(\mathcal{D})\} \cup \{p_\varepsilon\} \\ I_\mathcal{D} &= \{p_\varepsilon\} \\ \Delta_\mathcal{D} &= \{\langle p_W, \sigma, p_{W \circ \sigma} \rangle \,|\, \sigma \in \Sigma \text{ and } W \circ \sigma \in Pref(D)\} \\ T_\mathcal{D} &= \{p_W \,|\, W \in \mathcal{D}\}. \end{aligned}$$

It follows from the definition that $\mathcal{T}_\mathcal{D}$ is a deterministic automaton and furthermore it recognises the language $\mathcal{L}(\mathcal{T}_\mathcal{D}) = \mathcal{D}$. To keep the notion simpler in this case we shall denote the trie for $\mathcal{D}$, $\mathcal{T}_\mathcal{D} = \langle Q_\mathcal{D}, \Sigma, s_\mathcal{D}, \delta_\mathcal{D}, T_\mathcal{D} \rangle$ where $s_\mathcal{D} = p_\varepsilon$ and $\delta_\mathcal{D}$ is the function with graph $\#\delta_\mathcal{D} = \Delta_\mathcal{D}$.

## 1.3 Operations and Edit-distance

Let $\Sigma$ be an alphabet. The identity set (of operations) $Id \subseteq \Sigma^* \times \Sigma^*$ is:

$$Id = \{(\sigma, \sigma) \,|\, \sigma \in \Sigma\}$$

**Definition 1.3.1** A set of operations[1] $Op$ is a finite subset of $\Sigma^* \times \Sigma^*$ of the form:

$$Op = Id \cup \mathcal{U}$$

where there is no operation $(X, Y) \in \mathcal{U}$ with $X = Y$.

Given a set of operation $Op$ and an operation $op = (X, Y)$ in $Op$ the *left* (*right*) side of $op$ is $X$ ($Y$, respectively). We use the notations $l(op) = X$ and $r(op) = Y$ to denote the left and right side of an operation, respectively.

[1] Although the general terms *alphabet, character and word*, properly describe the terms *the set of operations, operation and alignment*, respectively, we prefer the latter in order to better reflect the specificity of the problem of interest.

**Definition 1.3.2** A generalised (Levenshtein) edit-distance is a pair $(Op, c)$ where $Op$ is a set of operations and $c : Op \to \mathbb{N}$ is a function with the property:

$$\forall op \in Op[c(op) = 0 \iff op \in Id].$$

**Remark 1.3.3** In the particular case when $Op_L = (\Sigma \cup \{\varepsilon\})^2 \setminus \{(\varepsilon, \varepsilon)\}$ we obtain the Levenshtein operations, that is the *insertions* – the operations of the form $(\varepsilon, \sigma_i)$ with $\sigma_i \in \Sigma$; the *deletions* – the operations of the form$(\sigma_i, \varepsilon)$ with $\sigma_i \in \Sigma$; and the *substitutions* – the operations of the form $(\sigma_i, \sigma_j) \in \Sigma \times \Sigma$. Supplying $Op_L$ with the cost function $c_L : Op_L \to \mathbb{N}$ defined with:

$$c_L(op) = \begin{cases} 0, & \text{if } op \in Id \\ 1, & \text{otherwise.} \end{cases}$$

we appear the usual Levenshtein edit-distance $(Op_L, c_L)$.

**Definition 1.3.4** Given a generalised edit-distance $(Op, c)$, an alignment is an arbitrary word of operations, $\omega \in Op^*$ and cost of the alignment $\omega = op_1 op_2 \dots op_n$ is the sum of individual costs of the operations constituting the alignment, i.e.:

$$c(\omega) = \sum_{k=1}^{n} c(op_k).$$

The left and right sides of $\omega$ are defined canonically as:

$$l(\omega) = l(op_1) \circ l(op_2) \circ \cdots \circ l(op_n)$$
$$r(\omega) = r(op_1) \circ r(op_2) \circ \cdots \circ r(op_n).$$

We say that $\omega$ aligns a word $U$ against a word $V$, or equivalently $\omega$ is an alignment of $U$ against $V$ if:

$$l(\omega) = U \text{ and } r(\omega) = V.$$

We should stress that the empty alignment contains no operations. Still it aligns the empty word with the empty word and is the unique alignment with this property. It should be also clear that its cost is 0.

**Definition 1.3.5** An edit-distance (on words) induced by the generalised edit-distance $(Op, c)$ is the mapping $d : \Sigma^* \times \Sigma^* \to \mathbb{N} \cup \{\infty\}$ which maps every pair of words $U, V \in \Sigma^*$ to the cost of a cheapest alignment of $U$ and $V$ induced by $Op$, i.e.:

$$d(U, V) = \min_{\omega \in Op^*} \{c(\omega) \,|\, l(\omega) = U \,\&\, r(\omega) = V\}.$$

## 1.4 Dynamic Programming Algorithms

Although the edit-distance between two words is defined in a quite unintuitive way, it can be computed in an efficient way via a standard dynamic programming

scheme, [23]. In this Section we are going to consider the problems of computing $d(U,V)$ and verifying $d(U,V) \leq b$ for a given threshold $b$.

We start with the problem of computing the Levenshtein edit-distance between two words, $U, V \in \Sigma^*$. That is to find the minimal number of *substitutions, deletions, insertions* which transform the word $U$ into the word $V$:

`Given:` $U, V \in \Sigma^*$
`Output:` $d_L(U,V)$.

Assume that $U = u_1 \circ u_2 \ldots u_m$ and $V = v_1 \circ v_2 \cdots \circ v_n$ are the input words. The idea which stays behind the Levenshtein's solution of this problem is to compute the edit distance between each pair of prefixes of $U$ and $V$. Formally, let us set $U_i = u_1 \circ u_2 \cdots \circ u_i$ and $V_j = v_1 \circ v_2 \ldots v_j$. Here $U_0 = V_0 = \varepsilon$. The objective is to determine the values:

$$D_{i,j} = d_L(U_i, V_j).$$

Since $U = U_m$ and $V = V_n$ the answer of the query will be $D_{m,n} = d(U_m, V_n)$. Some of the values of the matrix $D$ can be easily filled in. Namely, there is a unique alignment $\omega$ with $l(\omega) = \varepsilon$ and $r(\omega) = V_j$ and similarly there is a unique alignment $\omega$ with $l(\omega) = U_i$ and $r(\omega) = \varepsilon$. Thus it becomes apparent that:

$$D_{i,0} = i \text{ and } D_{0,j} = j$$

for all $i \leq m$ and $j \leq n$.

In order to compute the remaining values of the matrix $D$, let us consider an arbitrary alignment $\omega$ of $U_i$ and $V_j$ for $i > 0$ and $j > 0$. Let $\nu$ be the last operation of $\omega$, i.e. $\omega = \omega' \circ \nu$. Due to the specific type of operations, there are three cases for $\nu$:

- **substituition,** $\nu = (x, y)$**.** Since $\omega$ aligns $U_i$ against $V_j$ we deduce that $u_i = x$ and $v_j = y$. Therefore $c(\omega) = c(\omega') + c((u_i, v_j))$ and $\omega'$ is an alignment of $U_{i-1}$ against $V_{j-1}$.
- **deletion,** $\nu = (x, \varepsilon)$**.** In this case $\nu = (u_i, \varepsilon)$ and consequently $c(\omega) = c(\omega') + c((u_i, \varepsilon))$ and $\omega'$ aligns $U_{i-1}$ against $V_j$.
- **insertion,** $\nu = (\varepsilon, y)$**.** Hence $y = v_j$ and $\omega'$ aligns $U_i$ against $V_{j-1}$ and $c(\omega) = c(\omega') + c((\varepsilon, v_j))$.

Note that $c(\nu) = 1$ unless $\nu = (u_i, v_j)$ and $u_i = v_j$ when $c(\nu) = 0$. This observation allows us to derive the following recurrence.

$$D_{i,j} = \min \begin{cases} D_{i-1,j-1} + \text{ if } (u_i = v_j) \text{ then 0 else 1} \\ D_{i-1,j} + 1 \\ D_{i,j-1} + 1 \end{cases}$$

Since the values $D_{i',j'}$ required in order to define $D_{i,j}$ satisfy $i' + j' < i + j$, we can compute the values $D_{i,j}$ in increasing order of $i + j$. Since we need

$O(1)$ operations to determine $D_{i,j}$ based on the previously determined values the total time of the algorithm is $O(mn)$.

One can easily generalise this approach for arbitrary set of operations $Op$. Specifically consider the problem:

`Given:` $U, V \in \Sigma^*$
`Output:` $d(U, V)$

where $d = (Op, c)$. Now we can define $D_{i,j}$ in the same way as above:

$$D_{i,j} = d(U_i, V_j).$$

Clearly $D_{0,0} = 0$. Next let $i + j > 0$ and consider an alignment $\omega$ of $U_i$ against $V_j$. Since $|\omega| > 0$ it can be decomposed as $\omega = \omega' \circ \nu$ where $\nu \in Op$. We can further constrain $\nu$ to the set of those operations with $l(\nu)$ being a suffix of $U_i$ and $r(\nu)$ – suffix of $V_j$. Consequently $c(\omega) = c(\omega') + c(\nu)$ which let us conclude that:

$$D_{i,j} = \min\{D_{i-|l(\nu)|,j-|r(\nu)|} + c(\nu) \mid \nu \in Op \text{ and } l(\nu) \text{ is a suffix of } U_i \text{ and } r(\nu) \text{ is a suffix of } V_j\}.$$

Again, since $|l(\nu)| + |r(\nu)| > 0$ we have that the values $D_{i',j'}$ involved in the computation of the minimum satisfy $i' + j' < i + j$. Finally for a fixed pair $(i, j)$ we investigate each operation $\nu \in Op$ and exhaustively verify whether $l(\nu)$ is a suffix of $U_i$ and $r(\nu)$ is a suffix of $V_j$. This can be performed in time $\sum_{\nu \in Op}(|l(\nu)| + |r(\nu)|)$ which is considered to be constant in the sense that it is fixed and known in advance and does not depend in any way on the input $U$ and $V$.

Hence we can compute the edit distance $d(U, V) = D_{m,n}$ in time $O(mn)$. Furthermore one does not need to store the entire table $D$ in order to compute the edit-distance $d(U, V)$. Actually, in the computation of $D_{i,j}$ only values from the rows $i - |l(\nu)|$ are involved. Thus, it suffices to store only the last few rows $i - \rho, i - \rho + 1, \ldots, i$ before the current row $i$ where $\rho$ is the maximum over all possible lengths $|l(\nu)|$.

The Ukkonen's problem slightly differs from the original Levenshtein's concept:

`Given:` $U, V \in \Sigma^*$, $b \in \mathbb{N}$
`Output: Yes if` $d_L(U, V) \leq b$ `and No otherwise.`

The idea of this model is to consider the edit-distance as a characteristic of similarity rather than as a distance. Any two words $U$ and $V$ determine a quantity $d_L(U, V)$ but it does not mean nothing more than a single integer. The idea of Ukkonen encoded in the threshold $b$ is the concept of similarity. Namely it exploits the comparison of $d_L(U, V)$ with $b$ as an indicator which postulates that $U$ and $V$ are similar, i.e. if someone has typed $V$ and $d_L(U, V) \leq b$, then indeed (s)he might have intended to write $U$, instead. But if $d_L(U, V) > b$, then definitely it is not reasonable to assume that $V$ is meant to stay for $U$.

Beyond this novel viewpoint on the Levenshtein edit-distance, the Ukkonen's problem has not contributed for a new approach towards its solution. Actually

it still uses the same dynamic programming scheme and computes the table $D_{i,j}$ as before. The additional feature of the problem, one can benefit of is that there is no need to compute the entire table in order to answer the query. Indeed, it is a simple observation that:

$$D_{i,j} \geq |j - i|$$

and additionally if the value $D_{i,j}$ is involved in someway with the estimation of $D_{k,l}$, then $D_{k,l} \geq D_{i,j}$. This two remarks show that there is no need to compute any of the entries $D_{i,j}$ with $|j - i| > b$ since they do not influence the result of the query. Hence the recurrence of $D$ can be modified to:

$$\begin{aligned} D'_{i,0} &= D'_{0,i} = \begin{cases} i \text{ if } i \leq b \\ \neg! \text{ otherwise} \end{cases} \\ D'_{i,j} &= \neg! \text{ if } |i-j| > b \end{aligned}$$

and if $|i - j| \leq b$ and $\min(i, j) > 0$

$$D'_{i,j} = \min \begin{cases} D'_{i-1,j-1} + \text{ if } (u_i = v_j)0 \text{ else } 1 \\ D'_{i-1,j} + 1 \\ D'_{i,j-1} + 1 \end{cases}$$

where the minimum operation extends only on the defined values and ignores the undefined ones.

Since no time is spent in filling in values in the cells with $|i - j| > b$, and we spare $O(1)$ for each of the remaining cells, we deduce that the running time of this procedure is $O(b \min(|U|, |V|))$. Hence, if $b << \max(|U|, |V|)$ the time-complexity $O(b \min(|U|, |V|))$ outperforms the straightforward application of the Levenshtein's algorithm with a linear factor.

## 1.5 Generalised Levenshtein Automata

In the dynamic programming schemes considered in the previous Section no attempt is made to avoid applying incompatible operations. At each possible pair of positions $\langle i, j \rangle$ the entire set of operations $Op$ is exhaustively checked without taking into account that only few of them will be compatible with the left contexts of $U[i]$ and $V[j]$. The framework of the finite state automata allows to encode these features in the states and to control how they progress in time by appropriately defined transitions. We give the flavour of this approach in the sequel.

The general idea is to use the states of the automaton in order to encode the edit-distance between the currently processed strings as well an appropriate left context in order to control the search. With this remark, it is easy to devise a nondeterministic automaton that achieves this goal. Each of the states stays for a particular edit-distance $d$ which does not exceed a predetermined threshold $b$. The transitions of the automaton are labelled with operations in a way that a state number $i$ is connected with a transition labelled $op$ to a state numbered

$i + c(op)$. In this way the paths in this automaton reflect alignments and the cost of a particular alignment is given as the difference between the terminal and initial state of the path. Formally we have:

**Definition 1.5.1** Let $(Op, c)$ be an edit-distance and $b \in \mathbb{N}$ be a threshold. A generalised Levenshtein automaton is:

$$\mathcal{A} = \langle Q_b, Op, \{0\}, \Delta_{Op}, Q_b \rangle$$

where:

$$\begin{aligned} Q_b &= \{0, 1, \ldots, b\} \\ \Delta_{Op} &= \{\langle j, op, j + c(op) \rangle \mid j, j + c(op) \in Q_b, \text{ and } op \in Op\}. \end{aligned}$$

From the above discussion it should be clear that the Levenshtein automata have the following property:

**Lemma 1.5.2** *Let $(Op, c)$ be an edit-distance and $b \in \mathbb{N}$ be a threshold. The generalised Levenshtein automaton $\mathcal{A} = \langle Q_b, Op, \{0\}, \Delta_{Op}, Q_b \rangle$ has language $\mathcal{L}(\mathcal{A}) = \{\omega \in Op^* \mid c(\omega) \leq b\}$. Furthermore for arbitrary word $V \in \Sigma^*$ it holds:*

$$d(U, V) \leq b \Rightarrow d(U, V) = \arg\min\{\delta^*(0, \omega) \mid l(\omega) = U,\ r(\omega)V\}.$$

□

Unfortunately this automaton, though being deterministic with respect to the operations, is in general nondeterministic with respect to $(\Sigma \times \Sigma)^*$. That is for one and the same pair of words, $\langle U, V \rangle$ there may be more than one path that corresponds to an alignments between $U$ and $V$.

In order to deal with this problem one has to determinise the automaton. Although the general subset construction is applicable in theory it results in a lot of redundancies which prohibitively increase the size of the resulting automaton in practice. In [55] Mihov and Schulz describe how this undesired effect can be reduced. Their construction achieves also the interesting property that the automaton and its size do not depend explicitly on the specific alphabet, $\Sigma$, rather only on the threshold, $b$, and the sample structure of the operations, $Op$. Using bit vectors of appropriate length ($O(b)$) one can encode which operations are locally applicable on a particular position, say of the word $U$. The states of the automaton reflect both the edit distance of the currently processed words and the sufficient left and right context that allows to proceed the traversal deterministically. It is in preprocessing step that the pair of words $\langle U, V \rangle$ is mapped to a sequence of bit vectors of length $\max(|U|, |V|)$. Afterwards the search can be executed in time $O(\max(|U|, |V|))$. Although the preprocessing step requires $O(b \max(|U|, |V|))$, the universal Levenshtein automata achieve a factor gain of several times over the Ukkonen's algorithm. However the space requirements of the universal Levenshtein automata still grows exponentially with the increase of the threshold $b$.

In [43] Mitankin, Mihov and Schulz extend this approach and present necessary and sufficient conditions for the structure of the operations and their

costs (that do not need to be integers any more but can belong to an arbitrary semiring) that enables the construction of such kind of automata. In the same paper they also present the synchronised Levenshtein automata that achieve faster running time in order to decide $d(U,V) \leq b$ for fixed $b$. The core idea is to transcribe the bit vectors from the Universal Levenshtein automata to pairs of characters from $\Sigma \times \Sigma$. This results in a deterministic (two-tape) automaton over $\Sigma \times \Sigma$ which tests whether $d(U,V) \leq b$ in time $O(|U|+|V|)$ which is a factor of $b$ faster than the algorithms described in the previous paragraphs. However the size of these automata increases exponentially with $b$ and the size of the alphabet, $\Sigma$.

## 1.6 Approximate Search Problem

The approximate search problem can be described in the following way. One disposes on a finite representation of a language $\mathcal{L}$ and on a generalised edit-distance $(Op, c)$. The task is to answer queries which have as an input a single word $V \in \Sigma^*$ and a natural number $b \in \mathbb{N}$. The output is the set of all words $U \in \mathcal{L}$ which are at edit-distance $d$ less than or equal to $b$ from $V$.

Formally this can be stated as:

`Given:` $\mathcal{L} \subseteq \Sigma^*$, $(Op, c)$
`Input:` $V \in \Sigma^*$, $b \in \mathbb{N}$
`Output:` $\{U \in \mathcal{L} \mid d(U,V) \leq b\}$.

Typically $\mathcal{L}$ is assumed to be a finite set of words, e.g. a single word [60], the set of infixes of a long text [47], a dictionary [50]. We generalise this concept by allowing $\mathcal{L}$ to be an arbitrary regular language which is represented by a finite state automaton.

Next there are different scenarios for the ways the threshold $b$ is determined. One possible way is to assume that $b$ is known in advance and thus it is constant for all queries [16, 15, 55]. Hence the problem has rather the structure:

`Given:` $\mathcal{L} \subseteq \Sigma^*$, $(Op, c)$, $b \in \mathbb{N}$
`Input:` $V \in \Sigma^*$
`Output:` $\{U \in \mathcal{L} \mid d(U,V) \leq b\}$.

Since $b$ is considered to be a constant one can precompute some appropriate information (an index) which can be used to speed-up the processing of the query.

On the other extreme $b$ can be given as an input parameter, i.e. $b$ constitutes an essential part of the query [60, 50]. The first approach is clearly very restrictive when considering words of arbitrary length. The second one is too flexible and may also turn to be irrelevant, e.g. if $b$ is too big, the list of words satisfying the query would be enormous and thus it would be of not much worth if one should analyse it afterwards. A compromise between these two extremities is to

model $b$ as a ratio $q \in (0,1)$ of the length of the input word $V$ [47]. The ratio $q$ is assumed to be given in advance and thus the query is entirely specified by the input word $V$. Nevertheless the threshold $b$ varies with the length of the input.

In our research we use the latter approach to model the approximate search problem.

**Definition 1.6.1** Given a regular language $\mathcal{L}$, a generalised edit-distance $(Op, c)$ and a rational number $q \in (0;1)$, one has to process queries of the form:

`Input:` $V \in \Sigma^*$
`Output:` $\{U \in \mathcal{L} \mid d(U,V) \leq q|V|\}$.

## 1.7 Norm of Matrices

Let $\mathcal{M}_{n,n}(\mathbb{R})$ be the algebra of real valued matrices. We recall the definition [28] for a *norm* on matrices:

**Definition 1.7.1** A norm on the algebra of real-valued matrices $\mathcal{M}_{n,n}(\mathbb{R})$ is an operator $||.|| : \mathcal{M}_{n,n}(\mathbb{R}) \to \mathbb{R}$ which satisfies the following conditions:

$$\begin{array}{rl} ||A|| & \geq 0 \text{ with } ||A|| = 0 \iff A = O \\ ||\nu A|| & = |\nu|||A|| \text{ for each } \nu \in \mathbb{R} \\ ||A+B|| & \leq ||A|| + ||B|| \\ ||AB|| & \leq ||A||||B|| \end{array}$$

for arbitrary matrices $A, B \in \mathcal{M}_{n,n}(\mathbb{R})$.

In our considerations we shall use the $||.||_\infty$:

$$||A||_\infty = \max_j \sum_{k=1}^{n} |a_{j,k}|$$

which defines a norm [28] on $\mathcal{M}_{n,n}(\mathbb{R})$.

## 1.8 FSAs Algebraically

The finite state automata represent languages, i.e. sets of words. An interesting question is how to measure the size of a language, $\mathcal{L}$, represented by a finite state automaton $\mathcal{A} = \langle \Sigma, Q, I, \Delta, T \rangle$. Of course, the answer of these question seems obvious in the case when $\mathcal{L}$ is finite. In this case we can use $|\mathcal{L}|$ as such a measure. However this approach is impractical if we have to deal with infinite languages.

A general approach to formalise this notion is presented by Eilenberg in [17]. Here we shall only summarise those results that are of interest for our research

and we present them in a way that is convenient for our purposes thus deviating from the formal concepts in [17].

Let $\Sigma$ be an alphabet with characters $\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}$. Next notion defines the number of characters $\sigma_i$ in a word $U \in \Sigma^*$.

**Definition 1.8.1** For a word $U \in \Sigma^*$ with $U = u_1 u_2 \ldots u_n$ where $u_j \in \Sigma$ we determine the $i$-type of $U$, $||U||_i$ as:

$$||U||_i = |\{j \,|\, u_j = \sigma_i\}|.$$

And we set the type of the word $U$ to be the vector $||U|| \in \mathbb{N}^{|\Sigma|}$ defined as:

$$||U|| = (||U||_1, ||U||_2, \ldots, ||U||_{|\Sigma|}).$$

Thus $||U||$ encodes the quantitative information about the characters involved in a word $U$ and discards their specific order. The measure that we shall define for a regular language $\mathcal{L}$ will account for the types of the words and not for the words themselves. Essentially for each possible type we shall count the number of words $U \in \mathcal{L}$ of this type.

**Definition 1.8.2** Let $\mathcal{L}$ be a language and $\boldsymbol{u} \in \mathbb{N}^{|\Sigma|}$ be a vector of natural numbers. We denote with $\mathcal{L}(\boldsymbol{u}) = |\{U \in \mathcal{L} \,|\, ||U|| = \boldsymbol{u}\}|$ the number of words in $\mathcal{L}$ of type $\boldsymbol{u}$.

Next we turn our attention to the finite state automata and associate with each transition $\langle p, \sigma_i, q\rangle$ a variable $x_i$. This allows us to consider each finite state automaton as an adjacency matrix whose entries are polynomials of $x_1, x_2, \ldots, x_{|\Sigma|}$ and more precisely these would be polynomials of degree one (or $-\infty$) and coefficients zeroes and ones.

For notational convenience till the end of this Section we assume that the set of states, $Q$, of a finite state automaton is $Q = \{1, 2, \ldots, |Q|\}$. The general case can be easily reduced to this one by simply renaming the states $Q$.

**Definition 1.8.3** Let $\mathcal{A} = \langle \Sigma, Q, I, \Delta, T\rangle$ be a finite state automaton. We introduce the $|Q| \times |Q|$ matrix, $M_{\mathcal{A}}(\boldsymbol{x})$, whose entries are determined by:

$$M_{\mathcal{A}}(k, l; \boldsymbol{x}) = \sum_{i:\langle k,\sigma_i,l\rangle \in \Delta} x_i$$

for every two states $k, l \in Q$ and $\boldsymbol{x}$ is $|\Sigma|$-dimensional vector.

**Lemma 1.8.4** *Let $\mathcal{A} = \langle \Sigma, Q, I, \Delta, T\rangle$ be a finite state automaton and $n \in \mathbb{N}$ be a natural number. Then for every two states $k, l \in Q$ the following equality holds:*

$$M^n_{\mathcal{A}}(k, l; \boldsymbol{x}) = \sum_{\pi \in \Pi(\mathcal{A}):|\pi|=n, \iota(\pi)=k, \tau(\pi)=l} \boldsymbol{x}^{||\lambda(\pi)||}$$

*where $\boldsymbol{x} = (x_1, x_2, \ldots, x_{|\Sigma|})$, $\boldsymbol{x}^{||U||} = \prod_{i=1}^{|\Sigma|} x_i^{||U||_i}$ and $M^n_{\mathcal{A}}(k, l; \boldsymbol{x})$ is the $(k, l)$-entry of the $n$-th power of the matrix $M_{\mathcal{A}}(\boldsymbol{x})$, .*

*Proof.* Let $\mathbf{1}$ and $\mathbf{0}$ be the polynomials that identically equal to 1 and 0, respectively. The proof proceeds by induction on $n$. For $n = 0$ we have that $M^0_{\mathcal{A}}(\boldsymbol{x}) = I$ where $I$ is the matrix with entries $\mathbf{1}$ along the main diagonal and $\mathbf{0}$ outside it. For each $k \in Q$ there is a unique path $\pi$ of length 0 and $\iota(\pi) = k$, namely the trivial path $\pi = (k)$. Its label is clearly $\lambda(\pi) = \varepsilon$ and since the type of $\varepsilon$ is $||\varepsilon|| = (0, 0, \ldots, 0)$ we conclude that:

$$\sum_{\substack{\pi\in\Pi(\mathcal{A}):|\pi|=n,\\ \iota(\pi)=k,\tau(\pi)=l}} \boldsymbol{x}^{||U||} = \begin{cases} \boldsymbol{x}^{||\varepsilon||} = \mathbf{1}, \text{ if } k = l \\ \mathbf{0}, \text{ if } k \neq l \end{cases}$$

which implies that the claim of the Lemma is valid for $n = 0$.

Let us assume that the claim of the Lemma holds for some $n \in \mathbb{N}$. We prove that in this case it is also true for $n + 1$. For $k, l \in Q$ we have :

$$\begin{aligned}
M^{n+1}_{\mathcal{A}}(k, l; \boldsymbol{x}) &= \sum_{j=1}^{|Q|} M^n_{\mathcal{A}}(k, j; \boldsymbol{x}) M_{\mathcal{A}}(j, l; \boldsymbol{x}) \\
(\text{ by the definition of } M_{\mathcal{A}}(\boldsymbol{x})) &= \sum_{j=1}^{|Q|} M^n_{\mathcal{A}}(k, j; \boldsymbol{x}) \sum_{i:\langle j,\sigma_i,l\rangle\in\Delta} x_i \\
(\text{ by the induction hypothesis }) &= \sum_{j=1}^{|Q|} \sum_{\substack{\pi\in\Pi(\mathcal{A})\\ |\pi|=n\iota(\pi)=k,\tau(\pi)=j}} \boldsymbol{x}^{||\lambda(\pi)||} \sum_{i:\langle j,\sigma_i,l\rangle\in\Delta} x_i \\
&= \sum_{j=1}^{|Q|} \sum_{\substack{\pi\in\Pi(\mathcal{A})\\ |\pi|=n,\iota(\pi)=k,\tau(\pi)=j}} \sum_{i:\langle j,\sigma_i,l\rangle\in\Delta} \boldsymbol{x}^{||\lambda(\pi)||} x_i \\
&= \sum_{j=1}^{|Q|} \sum_{\substack{\pi\in\Pi(\mathcal{A})\\ |\pi|=n,\iota(\pi)=k,\tau(\pi)=j}} \sum_{i:\langle j,\sigma_i,l\rangle\in\Delta} \boldsymbol{x}^{||\lambda(\pi\langle j,\sigma_i,l\rangle)||} \\
&= \sum_{\substack{\pi\in\Pi(\mathcal{A})\\ |\pi|=n+1,\iota(\pi)=k,\tau(\pi)=l}} \boldsymbol{x}^{||\lambda(\pi)||}.
\end{aligned}$$

The last equality follows by the observation that each path of length $n + 1$ starting at $k$ and terminating at $l$ can be considered as path of length $n$ starting at $k$ and terminating at some state $j$ followed by a single transition from $j$ to $l$, and vice versa. □

In the particular case when the finite state automaton $\mathcal{A}$ is deterministic, we have a one-to-one correspondence between the words $U \in \mathcal{L}(\mathcal{A})$ and the successful paths in $\mathcal{A}$, i.e. those that start at the initial state and terminate at a terminal state. Hence we get the following result.

**Corollary 1.8.5** *Let $\mathcal{A} = \langle \Sigma, Q, \{s\}, \Delta, T \rangle$ be a deterministic automaton with language $\mathcal{L} = \mathcal{L}(\mathcal{A})$. Let $\boldsymbol{u} \in \mathbb{N}^{|\Sigma|}$ be a vector of integers and $n = \sum_{i=1}^{|\Sigma|} u_i$. Then it holds:*

$$\mathcal{L}(\boldsymbol{u}) = [\boldsymbol{x}^{\boldsymbol{u}}] \sum_{f \in T} M_{\mathcal{A}}^{n}(s, f; \boldsymbol{x})$$

*where $[\boldsymbol{x}^{\boldsymbol{u}}]$ means the coefficient of the polynomial before the term $\boldsymbol{x}^{\boldsymbol{u}}$.*

□

In the general case when $\mathcal{A}$ should not be deterministic there is a one-to-many correspondence between the words $U \in \mathcal{L}(\mathcal{A})$ and the successful paths in $\mathcal{A}$. Hence we cannot claim equality but rather inequality:

**Corollary 1.8.6** *Let $\mathcal{A} = \langle \Sigma, Q, I, \Delta, T \rangle$ be a finite state automaton with language $\mathcal{L} = \mathcal{L}(\mathcal{A})$. Let $\boldsymbol{u} \in \mathbb{N}^{|\Sigma|}$ be a vector of integers and $n = \sum_{i=1}^{|\Sigma|} u_i$. Then it holds:*

$$\mathcal{L}(\boldsymbol{u}) \leq [\boldsymbol{x}^{\boldsymbol{u}}] \sum_{s \in I, f \in T} M_{\mathcal{A}}^{n}(s, f; \boldsymbol{x})$$

*where $[\boldsymbol{x}^{\boldsymbol{u}}]$ means the coefficient of the polynomial before the term $\boldsymbol{x}^{\boldsymbol{u}}$.*

□

With respect to Corollary 1.8.6 and taking into account a standard construction of an infix automaton, see Lemma 1.2.5, we get:

**Corollary 1.8.7** *Let $\mathcal{A} = \langle \Sigma, Q, I, \Delta, T \rangle$ be a finite state automaton with language $\mathcal{L} = \mathcal{L}(\mathcal{A})$ and let $Inf_{\mathcal{L}} = Inf(\mathcal{L})$. Then for every type $\boldsymbol{u} \in \mathbb{N}^{|\Sigma|}$ with $n = \sum_{i=1}^{|\Sigma|} u_i$ it holds:*

$$Inf_{\mathcal{L}}(\boldsymbol{u}) \leq [\boldsymbol{x}^{\boldsymbol{u}}] \sum_{k,l \in Q} M_{\mathcal{A}}^{n}(k, l; \boldsymbol{x}).$$

□

Since the performance of our algorithm depends on the characteristics of the $Inf(\mathcal{L})$ rather than the original language, $\mathcal{L}$, we define a "measure" of a finite state automaton with respect to Corollary 1.8.7:

**Definition 1.8.8** Let $\mathcal{A} = \langle \Sigma, Q, I, \Delta, T \rangle$ be a finite state automaton, then a generating function $g_{\mathcal{A}}(\boldsymbol{x})$ is the power series:

$$g_{\mathcal{A}}(\boldsymbol{x}) = \sum_{\pi \in \Pi(\mathcal{A})} \boldsymbol{x}^{||\lambda(\pi)||}.$$

By Lemma 1.8.4 we can equivalently define $g_{\mathcal{A}}$ as:

$$g_{\mathcal{A}}(\boldsymbol{x}) = \sum_{n=0}^{\infty} \sum_{k,l \in Q} M_{\mathcal{A}}^{n}(k, l; \boldsymbol{x}).$$

The function $g_{\mathcal{A}}$ reflects all the infixes in the language $\mathcal{L}(\mathcal{A})$. In some cases we shall be interested only in the long enough infixes. For this reason for a rational number $n_1 \in \mathbb{Q}$ we also introduce the generating function $g_{\mathcal{A},n_1}$:

**Definition 1.8.9** Let $\mathcal{A} = \langle \Sigma, Q, I, \Delta, T \rangle$ and $n_1 \in \mathbb{Q}$ be a finite state automaton, then a generating function $g_{\mathcal{A},n_1}(\boldsymbol{x})$ is the power series:

$$g_{\mathcal{A},n_1}(\boldsymbol{x}) = \sum_{\pi \in \Pi(\mathcal{A}):|\pi| \geq n_1} \boldsymbol{x}^{||\lambda(\pi)||}.$$

By Lemma 1.8.4 we can equivalently define $g_{\mathcal{A},n_1}$ as:

$$g_{\mathcal{A},n_1}(\boldsymbol{x}) = \sum_{n \geq n_1} \sum_{k,l \in Q} M^n_{\mathcal{A}}(k, l; \boldsymbol{x}).$$

We point out that in the particular case when the vector $\boldsymbol{x}$ is interpreted as a real-valued vector $\boldsymbol{t} = (t_1, t_2, \ldots, t_{|\Sigma|})$ we get the power series:

$$g_{\mathcal{A}}(\boldsymbol{t}) = \sum_{n=0}^{\infty} \sum_{k,l \in Q} M^n_{\mathcal{A}}(k, l; \boldsymbol{t})$$

and also:

$$g_{\mathcal{A},n_1}(\boldsymbol{t}) = \sum_{n \geq n_1} \sum_{k,l \in Q} M^n_{\mathcal{A}}(k, l; \boldsymbol{t})$$

The resulting power series may converge, may not converge, this is a matter that depends on the properties of the matrix $M_{\mathcal{A}}$.

# Chapter 2

# Bidirectional Infix Structures for Finite Sets

In this chapter we recall some classical data structures which can be used in order to represent the set of infixes of a finite set of words.

Suffix trees [66, 40, 61], affix trees [57, 39, 58], directed word acyclic graphs (DWAG) [11, 12, 44] and compressed directed word acyclic graphs (CDWAG) [29, 30] are only part of the data structures developed in order to compactly represent the set of infixes of large set of words or a single long text. They were conceived in relation with the pattern matching problem:

`Given: text` $\mathcal{T}$
`Input: pattern` $P \in \Sigma^*$
`Question:` $P \in Inf(T)$ `?`

Driven by this identity-based query problem the research in this area has attained to different ingenious structures which allow to answer the query in time $O(|P|)$ after an $O(|\mathcal{S}|)$ preprocessing and require $O(|\mathcal{S}|)$ storage.

Our purpose is to highlight a property of some of these data structures which plays a crucial role for the Myers' algorithm and will be of interest for our generalised algorithm. Namely one can use DWAGs and CDWAGs not only to process a word from left to right as usually but also to extend a word in either direction left or right, arbitrary.

This point is of no significant interest for the pattern matching problem where it does not influence neither the result nor the time complexity of the algorithm. However it is exactly this feature which yields the efficiency of the approximate search.

## 2.1 Blumers' Construction

If we want to construct a minimal DFA for a language $\mathcal{L}$, the Myhill-Nerode Theorem [17, 27] for $\mathcal{L}$ states that we have to determine the equivalence classes

of the relation:

$$X \sim_{\mathcal{L}} Y \iff \forall U \in \Sigma^*(X \circ U \in \mathcal{L} \iff Y \circ U \in \mathcal{L}).$$

Put in this way, it seems inevitable to consider all the infixes of $\mathcal{S}$ in order to determine the equivalence classes. So this would require quadratic time. However a rather straightforward observation made by Blumer et al. gives a way around. The idea is to consider not *infixes* but *suffixes*, instead. That is to say, we consider the relation:

$$X \sim_{Suf(\mathcal{S})} Y \iff \forall U \in \Sigma^*(X \circ U \in Suf(\mathcal{S}) \iff Y \circ U \in Suf(\mathcal{S})).$$

It is rather easy to show that $\sim_{Suf(\mathcal{S})}$ refines the relation $\sim_{Inf(\mathcal{S})}$ for every (finite) set of words $\mathcal{S}$. Indeed the assumption that $X \sim_{Suf(\mathcal{S})} Y$ implies that for every word $U$:

$$X \circ U \in Suf(\mathcal{S}) \iff Y \circ U \in Suf(\mathcal{S}).$$

Now for every word, say $V$, the word $X \circ V \in Inf(\mathcal{S})$ is an infix of $\mathcal{S}$ if and only if we can find a word $Z$ such that $X \circ V \circ Z \in Suf(\mathcal{S})$. Since $X$ and $Y$ are equivalent with respect to the suffixes of the set $\mathcal{S}$, we conclude that $Y \circ V \circ Z \in Suf(\mathcal{S})$. But this implies that $Y \circ V$ is an infix in the set $\mathcal{S}$. Substituting $X$ with $Y$ and vice versa in the above argument we also obtain that: each time $Y \circ V$ is an infix $\mathcal{S}$, $X \circ V$ is also an infix in $\mathcal{S}$. Thus we have establish the following property:

**Lemma 2.1.1** *For arbitrary words $X$ and $Y$ and a set $\mathcal{S}$, it holds:*

$$X \sim_{Suf(\mathcal{S})} Y \Rightarrow X \sim_{Inf(\mathcal{S})} Y.$$

This idea was in the origin for Blumer et al. [11, 12, 44] DAWG's algorithm for a single word $W$. In [11] the authors define the mapping $end{-}pos_W$ for a word $W = w_1 \circ w_2 \dots w_N$ for each word $X \in \Sigma^*$ as:

$$end{-}pos_W(X) = \{i \mid w_{i-|X|+1} \circ w_{i-|X|+2} \cdots \circ w_i = X\}.$$

Next they introduce $X \equiv_W Y$ if and only if $end{-}pos_W(X) = end{-}pos_W(Y)$. However one can easily recognise that $end{-}pos_W(X) = end{-}pos_W(Y)$ is equivalent to $X \sim_{Suf(W)} Y$. The reason for this is that each position, say $i$, in $W$ encodes a suffix $w_{i+1} \circ \cdots \circ w_N$. Since suffixes starting at different positions in $W$ being of different length are distinct, we have that $end{-}pos_W(X)$ encodes the set of distinct suffixes of $W$ that can follow $X$ in a unique way. This is the reason for $\equiv_W$ being the same relation as $\sim_{Suf(W)}$.

The case when $\mathcal{S} = \{W\}$ is a singleton allows us to give the nice characterisation of the equivalence classes of $\sim_{Suf(W)}$. Namely for arbitrary words $X$ and $Y$, the sets $end{-}pos_W(X)$ and $end{-}pos_W(Y)$ are either disjoint or one of them is a subset of the other. Indeed if $i$ is a common element for $end{-}pos_W(X)$ and $end{-}pos_W(Y)$, then both $X$ and $Y$ have an occurrence in $W$ terminating

at position $i$. Thus one of them, say $Y$, is a suffix of the other, say $X$. Now at every position, $j$, where $X$ terminates will witness that $Y$ also terminates at position $j$. Therefore $end{-}pos_W(X) \subseteq end{-}pos_W(Y)$. This consideration shows that not only $end{-}pos_W(X)$ form a laminar family, but also whenever $Y$ is a suffix of $X$, $end{-}pos_W(X) \subseteq end{-}pos_W(Y)$. Thus we have the following result, [11]:

**Lemma 2.1.2** *For every word $W \in \Sigma^*$ and $X, Y in \Sigma^*$ we have:*

$$\begin{array}{rcl} end{-}pos_W(X) \subseteq end{-}pos_W(Y) & or & end{-}pos_W(Y) \subseteq end{-}pos_W(X) \\ or\ end{-}pos_W(X) \cap end{-}pos_W(Y) & = & \emptyset \end{array}$$

*Further $end{-}pos_W(X) \cap end{-}pos_W(Y) \neq \emptyset$ and $|X| < |Y|$ implies that $X$ is a suffix of $Y$ and if $X$ is a suffix of $Y$, then $end{-}pos_W(Y) \subseteq end{-}pos_W(Y)$.*

These properties of the mapping $end{-}pos$ yield a nice characterisation of the equivalence classes of $\sim_{Suf(W)}$. It turns out that the *important* infixes of $W$ that determine the relation $\sim_{Suf(W)}$ are those that are either prefixes of $W$ or occur in two different immediate left contexts, $a \in \Sigma$ and $b \in \Sigma$ with $a \neq b$. To realise this, Blumer et al. define the *representative* of a equivalence class $[X]_{\equiv_W}$ with $end{-}pos_W(X) \neq \emptyset$ to be the longest word $X_0 \in [X]_{\equiv_W}$. There are two possible cases:(i) either $X_0$ is a prefix of $W$ or (ii) $X_0$ is not a prefix of $W$. Let us consider case (ii). Since $X_0$ is an infix of $W$ but not a prefix of $W$, then there is some character, say $a \in \Sigma$, such that $a \circ X_0$ is also an infix in $W$. Now we use that $X_0$ is the longest word in the equivalence class $[X_0]_{\equiv_W}$. In particular $a \circ X_0 \notin [X_0]_{\equiv_W}$ and since $X_0$ is a suffix of $a \circ X_0$ by Lemma 2.1.2 we conclude that $end{-}pos_W(a \circ X_0) \subsetneq end{-}pos_W(X_0)$. This implies that there is a position $i \in end{-}pos_W(X_0) \setminus end{-}pos_W(a \circ X_0)$. Since $X_0$ is not a prefix of $W$ $X$ does not span the first $|X_0|$ characters of $W$ which shows that $i > |X_0|$. Now since $w_{i-|X_0|+1} \circ \cdots \circ w_i = X_0$ but $i \notin end{-}pos_W(a \circ X_0)$ we obtain that $w_{i-|X_0|} \neq a$. Now $a \circ X_0$ and $w_{i-|X_0|} \circ X_0$ are both infixes of $W$ and therefore $X_0$ occurs in two different immediate left contexts. With these remarks we can easily establish that the representatives of $\equiv_W$ are exactly the prefixes of $W$ and those infixes of $W$ that occur in (at least) two different left contexts, [11]:

**Lemma 2.1.3** *An infix $X$ of $W$ is a representative for $\equiv_W$ if and only if one of the following two conditions hold:*

1. *$X_0$ is a prefix of $W$.*
2. *there exist distinct characters $a \neq b$ such that both $a \circ X_0$ and $b \circ X_0$ are infixes of $W$.*

*Proof.* If $X_0$ is a representative w.r.t. $\equiv_W$ and $X_0$ is not a prefix of $W$, then there is a character $a$ such that $a \circ X_0$ is an infix of $W$. Form the discussion above we deduce that there is another character $b \neq a$ such that $b \circ X_0$ is an infix of $W$.

Conversely, if $X_0$ is a prefix of $W$, then $end{-}pos_W(X_0)$ contains position $i = |X_0|$. It should be clear that no word, $Y$, longer than $X_0$ could have the

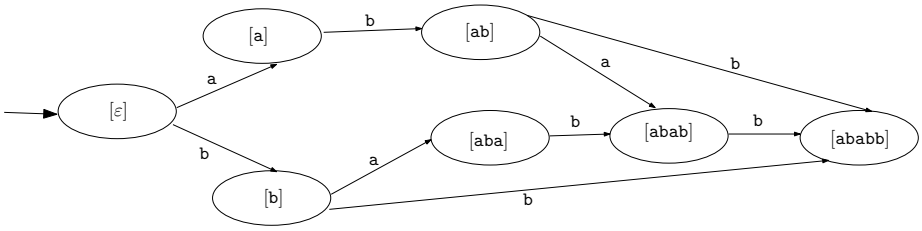



Figure 2.1: The Blumer et Blumer automaton for the word `ababb`

property that $i \in end{-}pos_W(Y)$. Finally if $X_0$ satisfies the second condition and $a \neq b$ are such that $a \circ X_0$ occurs at position $i$ and $b \circ X_0$ occurs at position $j$ in $W$, then there could not be word $Y$ longer than $X_0$ such that it has occurrences in $W$ terminating at positions $i$ and $j$, respectively. This is to say that if such a word, $Y$, existed, then $w_{i-|X_0|} = a$ and $w_{j-|X_0|} = b$ must have been equal to the last but $|X_0|$-th character of $Y$ which is impossible for $a \neq b$. □

This characterisation already allows to bound the size of the minimal deterministic automaton corresponding to $\equiv_W = \sim_{Suf(\mathcal{S})}$. Actually following the Myhill-Nerode Theorem, we can determine the minimal automaton for $Suf(W)$, see Figure 2.1:

$$\mathcal{A}_W = \langle Q_W, \Sigma, [\varepsilon]_{\equiv_W}, \delta_W, T_W \rangle,$$

with states the equivalence classes of $\equiv_W$ induced by an infix of $W$. The initial state is the class corresponding to the empty word, and the transitions are determined as:

$$\delta_W([X]_{\equiv_W}, a) = \begin{cases} [X \circ a]_{\equiv_W} \text{ if } X \circ a \in Inf(W) \\ \neg! \text{ otherwise.} \end{cases}$$

With this notion it is clear that the language recognised by the automaton $\mathcal{A}_W$ is exactly the set:

$$\mathcal{L}(\mathcal{A}_W) = \cup T_W.$$

Thus if we set $T_W = \{[X]_{\equiv_W} \mid X \in Suf(W)\}$ we obtain an automaton recognising the suffixes of $W$, and if we set $T_W = Q_W$ we obtain an automaton recognising the set of infixes of $W$.

The automaton $\mathcal{A}_W$ allows to traverse the infixes of $W$ from left to right. In order to achieve the traversal from right to left we take advantage from the tree structure of the representatives of the relation $\equiv_W$, see Figure 2.2. That is we define the tree $\mathcal{T}_W = \langle Q_W, E_W \rangle$ where the edges are defined as:

$$\begin{aligned} \langle [X]_{\equiv_W}, [P]_{\equiv_W} \rangle \in E_W \quad \Longleftrightarrow \quad & end{-}pos_W(X) \subset end{-}pos_W(P) \& \\ & \nexists Y (end{-}pos_W(X) \subset end{-}pos_W(Y) \subset end{-}pos_W(P)) \end{aligned}$$

or equivalently if the representative, $P_0$, of $[P]_{\equiv_W}$ is the longest suffix of $X$ that is not equivalent to $X$.

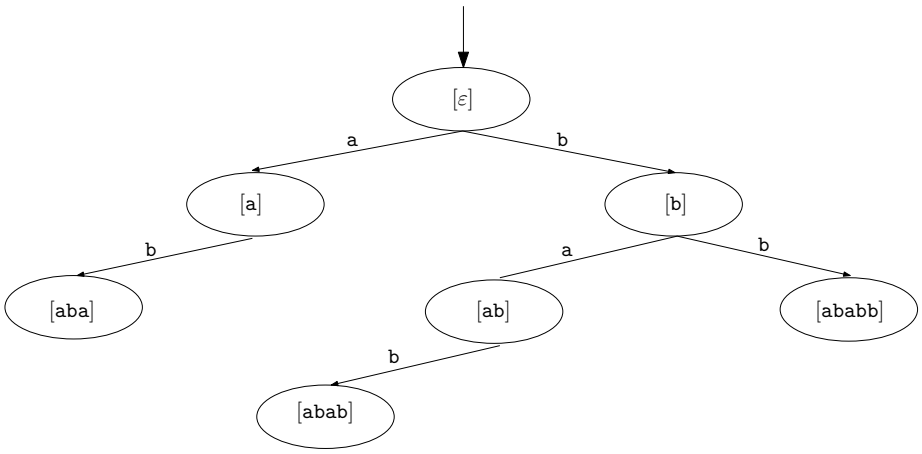



Figure 2.2: The tree structure of the representatives of the word `ababb`.

Introducing the tree $\mathcal{T}_W$ essentially answers the question what the asymptotic size of $Q_W$ is,[11]:

**Lemma 2.1.4** *The number of equivalence classes $Q_W$ is bounded by $2|W|-1$ for $|W|>2$.*

*Proof.* We consider the case when $W$ contains at least two different characters. We associate to each equivalence class $[X]_{\equiv_W}$ its representative $X_0$ and we bound the number of representatives rather the number of equivalence classes. Let $L_0$ be the set of leaves of $\mathcal{T}_W$, $L_1$ be the set of nodes of $\mathcal{T}_W$ with exactly one child and $L_{\geq 2}$ be the set of all other nodes. Since $\mathcal{T}_W$ is a tree we have that:

$$|L_0| + |L_1| + |L_{\geq 2}| = |Q_W| = |E_W| + 1.$$

On the other hand the number of edges, $E_W$, can be regarded as the set of pairs, node and its child. Hence each node of type $L_0$ is charged with no edges, each node of type $L_1$ is responsible for exactly one edge, and each node of type $L_{\geq 2}$ is associated with at least two edges. Therefore:

$$|L_0| + |L_1| + |L_{\geq 2}| = |E_W| + 1 \geq |L_1| + 2|L_{\geq 2}| + 1.$$

This implies that $|L_{\geq 2}| \leq |L_0| - 1$. From the characterisation of the representatives we that only non-prefix representatives have at least two children, thus they are of type $L_{\geq 2}$. Thus the nodes of type $L_0 \cup L_1$ are at most the number of prefixes in $W$. The empty word also belongs to the set $L_{\geq 2}$. Therefore $|L_0 \cup L_1| = |L_0| + |L_1| \leq |W|$. Now a straightforward computation shows that:

$$|Q_W| = |L_0| + |L_1| + |L_{\geq 2}| \leq |L_0| + |L_1| + |L_0| - 1 \leq 2(|L_0| + |L_1|) - 1 = 2|W| - 1.$$

This proves the Lemma in the general case. Finally if $W = a^N$ for some $a \in \Sigma$ and $N \geq 2$, then the number of different infixes is clearly $N + 1 \leq 2N - 1$ for $N \geq 2$. □

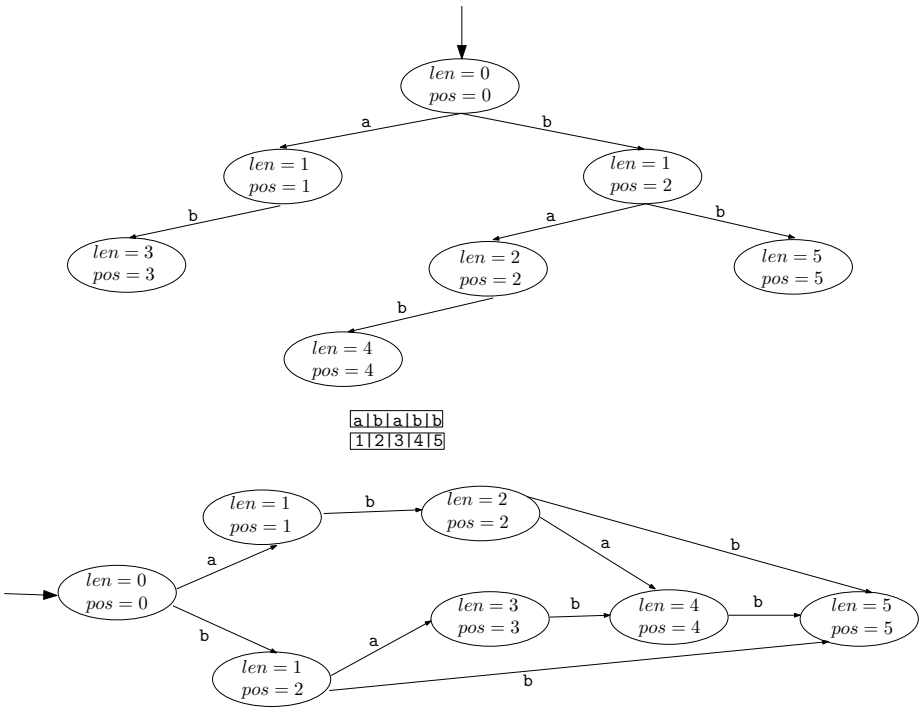



Figure 2.3: A linear size structure allowing the interchanging left and right traversal of the infixes of `ababb`.

Now in order to incorporate the left traversal of infixes of $W$, we supply the tree $\mathcal{T}_W$ with the following information, see Figure 2.3. First we keep the word $W$ as an array. Next for each node $[X]_{\mathcal{W}}$ of $\mathcal{T}_W$ we store:

1. a position $pos = \min end{-}pos_W(X)$.

2. the length $len$ of the representative $X_0$ of $[X]_{\equiv_W}$.

Finally we label each edge $\langle [X]_{\equiv W}, [P]_{\equiv_W} \rangle \in E_W$ with a character, $a \in \Sigma$ such that $aP_0 \equiv_W X$ where $P_0$ is the representative of $[P]_{\equiv_W}$. This data structure can be stored in $O(|W|)$ space since $|\mathcal{T}_W| = O(|W|)$ and we store constant number of data per node and per edge.

We point out that this structure can be considered as CDWAG but it is not an automaton in general. Therefore the traversal of this structure requires some care. Imagine that we have already successfully processed an infix, say $X$, of $W$. Thus we have reached the node $[X]_{\mathcal{W}}$ in the tree $\mathcal{T}_W$. We also assume that we know the length of $X$, say $l$. The left extension of $X$ with a character $a$ can be carried out as follows:

1. Check whether $l < len$ where $len$ is the length stored in $[X]_{\equiv_W}$.

2. If yes, then check whether $W[pos - l] = a$ where *pos* is the position stored in $[X]_{\equiv_W}$. If the answer is yes, then $a \circ X \equiv_W X$ and its length $l' = l + 1$.

3. if $l \not< len$, then since $X \equiv_W X_0$ we have actually that $X = X_0$. We check whether there is an edge labelled $a$ from $X_0$ to some of its children. If there is one, then it is unique, say $C$, and $a \circ X \equiv_W C$. In this case we set $l' = l + 1$. If there is no such child, then $a \circ X$ is not an infix of $W$.

The point of this case distinction is the following. Either $X$ is a proper suffix of its representative, $X_0$, in which case $X$ can be uniquely extended to the left as to remain an infix of $W$, or $X = X_0$ in which case the feasible left extensions are encoded in the labels of the edges of $\mathcal{T}_W$ that outgo from $X_0$.

In [11] the authors describe an $O(|W|)$ on-line algorithm which constructs $\mathcal{A}_W$ and $\mathcal{T}_W$. The additional data required for the left traversals can be easily incorporated in the algorithm or be inserted at a later stage.

## 2.2 Blumer et Blumer for finite set of words, $\mathcal{S}$

In order to represent the infixes of a finite set of words, $\mathcal{S}$ so that left and right extensions are accessible in constant time and linear space, we build on the observations from the previous Section. The crucial point was the characterisation of the representatives

Thus for a finite set of words, $\mathcal{S}$, we define:

$$X \equiv_{\mathcal{S}} Y \iff \forall W \in \mathcal{S}(X \equiv_W Y).$$

At this point we should stress that this equivalence relation is a refinement of $\sim_{Suf(\mathcal{S})}$. Indeed if $X \equiv_{\mathcal{S}} Y$ and $X \circ U \in Suf(\mathcal{S})$, then there is a word $W \in \mathcal{S}$ with $W = V \circ X \circ U$ for an appropriate word $V$. Now since $X \equiv_W Y$ we know that $W = V_1 \circ Y \circ U$ and therefore $Y \circ U \in Suf(\mathcal{S})$. Repeating this argument with $X$ and $Y$ playing the reverse roles, we deduce that $\equiv_{\mathcal{S}}$ is a refinement of $\sim_{\mathcal{S}}$. It is also straightforward that $\equiv_{\mathcal{S}}$ is right invariant. Thus knowing the equivalence classes of $\equiv_{\mathcal{S}}$ we can define a finite state automaton recognising $Suf(\mathcal{S})$ or $Inf(\mathcal{S})$ as desired.

Again, the important notion, is the one of a representative of an equivalence class $[X]_{\equiv_{\mathcal{S}}}$. And for $X \in Inf(\mathcal{S})$ this is the longest member, $X_0$, of the equivalence class. We have similar characterisation of the representatives as in the case of a single word. Namely, a representative is an infix $X \in Inf(\mathcal{S})$ that is either a prefix of a word in $\mathcal{S}$, or there are distinct characters $a \neq b$ such that both $a \circ X$ and $b \circ X$ are infixes of $\mathcal{S}$. The following result from [12] can be proven analogously as Lemma 2.1.3

**Lemma 2.2.1** *An infix $X \in Inf(\mathcal{S})$ is a representative for $\equiv_{\mathcal{S}}$ if and only if one of the following conditions is satisfied:*

1. *$X$ is a prefix of $\mathcal{S}$,*

2. *there exist distinct characters, $a \neq b$ such that both $a \circ X, b \circ X \in Inf(\mathcal{S})$.*

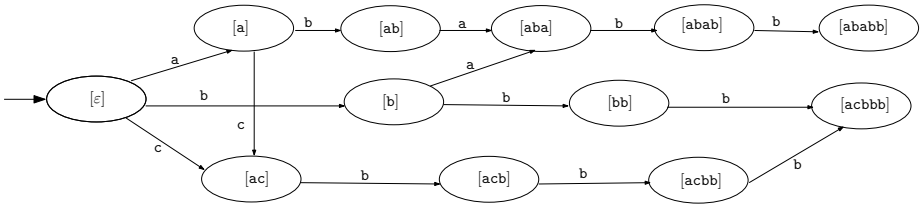



Figure 2.4: A linear size automaton recognising exactly the suffixes of {ababb, acbbb}.

Furthermore since each of equivalence relations $\equiv_W$ is right invariant, we deduce that equivalence relation $\equiv_{\mathcal{S}}$ is also right invariant. This implies that we can use the equivalence classes, or equivalently the representatives, in order to construct a deterministic finite state automaton recognising the set of infixes $Inf(\mathcal{S})$ or the set of suffixes, $Suf(\mathcal{S})$, see Figure 2.4. To this end we set:

$$Q_{\mathcal{S}} = \{[X]_{\equiv_{\mathcal{S}}} \mid X \in Inf(\mathcal{S})\}$$

and define the transition function $\delta_{\mathcal{S}} : Q_{\mathcal{S}} \times \Sigma \to Q_{\mathcal{S}}$ as:

$$\delta_{\mathcal{S}}([X]_{\equiv_{\mathcal{S}}}, a) = \begin{cases} [X \circ a]_{\equiv_{\mathcal{S}}} \text{ if } X \circ a \in Inf(\mathcal{S}) \\ \neg! \text{ otherwise.} \end{cases}$$

Thus the automaton $\mathcal{A}_{\mathcal{S}} = \langle Q_{\mathcal{S}}, \Sigma, [\varepsilon]_{\equiv_{\mathcal{S}}}, \delta_{\mathcal{S}}, Q_{\mathcal{S}} \rangle$ recognises the set of infixes, $Inf(\mathcal{S})$, of the set $\mathcal{S}$.

Now the right extensions of the infixes of the set $\mathcal{S}$ can be easily processed in constant time. Indeed, as in the case where $\mathcal{S}$ was a singleton containing a single word, given the state $[X]_{\equiv_{\mathcal{S}}}$ of an infixes $X \in Inf(\mathcal{S})$ and a character $a \in \Sigma$, the question: *Is $X \circ a$ an infixes of $Inf(\mathcal{S}$?* can be answered with a single look-up in the transition function:

$$\delta_{\mathcal{S}}([X]_{\equiv_{\mathcal{S}}}, a).$$

Thus the answer is positive if and only if the transition function is defined for $[X]_{\equiv_{\mathcal{S}}}$ and the character $a \in \Sigma$. Furthermore in this case it also provides the representation, $[X \circ a]_{\equiv_{\mathcal{S}}}$ of $X \circ a$. Thus this process can be iterated as long as required.

The left extensions are realised by the means of a tree structure similar to that of $\mathcal{T}_W$ considered in the previous Section, see Figure 2.5. Namely, we define the tree $\mathcal{T}_{\mathcal{S}} = \langle Q_{\mathcal{S}}, E_{\mathcal{S}} \rangle$ with edges:

$$\langle [X]_{\equiv_{\mathcal{S}}}, [P]_{\equiv_{\mathcal{S}}} \rangle \in E_{\mathcal{S}} \iff P_0 \in Suf(X_0) \text{ and } \forall Y \in Suf(X_0)[|Y| > |P_0| \Rightarrow Y \equiv_{\mathcal{S}} X_0].$$

This means that we set edges between distinct equivalence classes, $[X]_{\equiv_{\mathcal{S}}}$ and $[P]_{\equiv_{\mathcal{S}}}$, such that the representative $P_0$ of the equivalence class $[P]_{\equiv_{\mathcal{S}}}$ is the longest suffixes of the representative, $X_0$, of $[X]_{\equiv_{\mathcal{S}}}$ that is not equivalent to $X_0$.

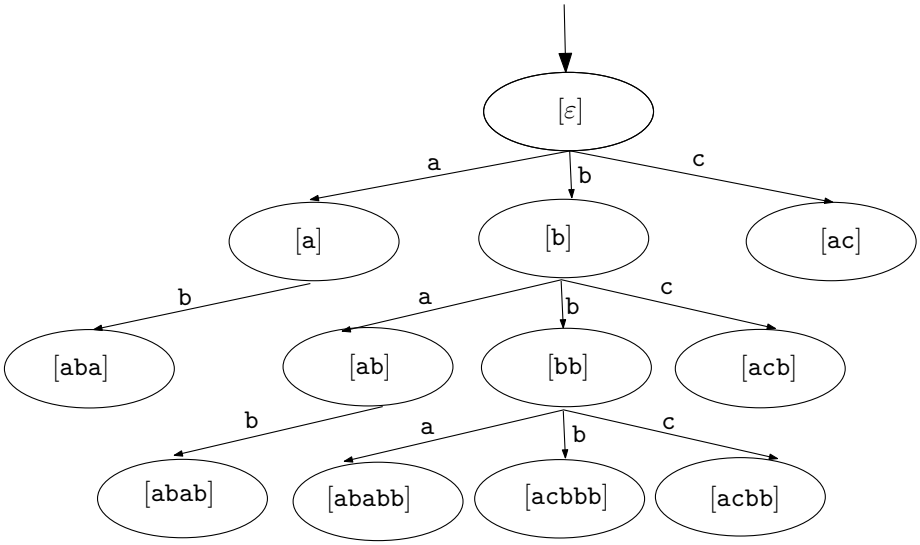



Figure 2.5: A linear size tree structure for the representatives induced by the set $\{\mathtt{ababb}, \mathtt{acbbb}\}$.

Put in this way the definition is the same as the definition of the edges in $\mathcal{T}_W$. However the equivalence relations are different.

Now the characterisation of the representatives of $\equiv_{\mathcal{S}}$ renders similar results as the results for $\equiv_W$ and the tree $\mathcal{T}_W$ we derived in the previous Section. Namely from the characterisation of the representatives of $\equiv_{\mathcal{S}}$ we have that each representative $X_0$ that is not a prefix of none of the words $W \in \mathcal{S}$ occurs in at least two distinct left contexts, $a \neq b$, s.t. $a \circ X_0, b \circ X_0 \in Inf(\mathcal{S})$. Hence the equivalence classes $[a \circ X_0]_{\equiv_{\mathcal{S}}}$ and $[b \circ X_0]_{\equiv_{\mathcal{S}}}$ will be children of $[X_0]_{\equiv_{\mathcal{S}}}$ in $\mathcal{T}_{\mathcal{S}}$. Therefore each node of $\mathcal{T}_{\mathcal{S}}$ can be associated with a prefix in $\mathcal{S}$ or has at least two children in $\mathcal{T}_{\mathcal{S}}$. Thus the same arguments we used for the proof of Lemma 2.1.4 yield that:

**Lemma 2.2.2** *The number of equivalence classes $Q_{\mathcal{S}}$ is bounded by:*

$$|Q_{\mathcal{S}}| \leq 2|Pref(\mathcal{S})| - 1 \leq 2 \sum_{W \in \mathcal{S}} |W| - 1.$$

Lemma 2.2.2 asserts that the space requirements for the representation of the automaton $\mathcal{A}_{\mathcal{S}}$ and the tree structure $\mathcal{T}_{\mathcal{S}}$ are linear in terms of the size of the input, that is the total size of the words contained in $\mathcal{S}$.

In order achieve the left extensions of infixes in $\mathcal{S}$ we proceed as in the case of a single word, see Figure 2.6. First we concatenate all the words $W \in \mathcal{S}$ in a single word $W_{\mathcal{S}}$ of length:

$$|W_{\mathcal{S}}| = \sum_{W \in \mathcal{S}} |W|.$$

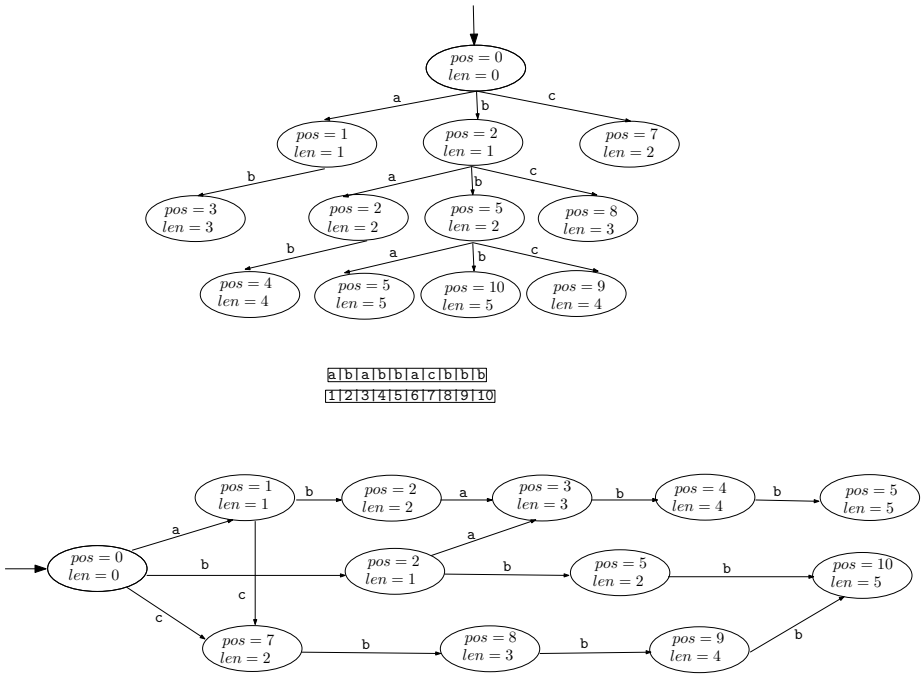



Figure 2.6: A linear size structure allowing the interchanging left and right traversal of the infixes of the set $\{\texttt{ababb}, \texttt{acbbb}\}$.

We store $W_{\mathcal{S}}$ as an array so that we have a constant-time access to an arbitrary character to each character in $W_{\mathcal{S}}$. Next each node in $\mathcal{T}_{\mathcal{S}}$ stores a constant amount of information that encodes the representative associated with this node. Specifically for a node $[X]_{\equiv_{\mathcal{S}}}$ with representative $X_0$ we store:

1. a position *pos* in $W_{\mathcal{S}}$ where $X_0$ terminates.

2. $len = |X_0|$, the length of the representative.

Finally with each edge, $\langle [X]_{\equiv_{\mathcal{S}}}, [P]_{\equiv_{\mathcal{S}}} \rangle$, is labelled with the unique character $a \in \Sigma$ such that $[X]_{\equiv_{\mathcal{S}}} = [a \circ P_0]_{\equiv_{\mathcal{S}}}$ where $P_0$ is the representative of $[P]_{\equiv_{\mathcal{S}}}$.

Clearly this representation of $\mathcal{T}_{\mathcal{S}}$ amounts to $O(|Q_{\mathcal{S}}|)$ space, thus linear in terms of the size of the input $\mathcal{S}$. With this data structure at hand and given an infix $X' \in Inf(\mathcal{S})$ of length $l$ and knowing that $X' \in [X]_{\equiv_{\mathcal{S}}}$ we can easily decide whether $a \circ X'$ is an infix of $\mathcal{S}$ in constant time and arbitrary character $a \in \Sigma$.

Indeed we can proceed as follows:

1. If $l' < len(= |X_0|)$, then $a \circ X'$ is an infix of $\mathcal{S}$ if and only if $a \circ X'$ is a suffix of $X_0$. Knowing that $X'$ is a suffix of $X_0$ this reduces to check whether $a = W_{\mathcal{S}}[pos - l']$ where $i = \min end{-}pos_{W_{\mathcal{S}}}(X_0)$. Since we have an immediate access to *pos* and to each entry of $W_{\mathcal{S}}$ we can test for this property in constant time. If the answer is positive, we have that $a \circ X' \equiv_{\mathcal{S}} X_0$ does occur as an infix in $\mathcal{S}$ and its length is $l' + 1$, otherwise it does not.

2. If $l' = len(= |X_0|)$, thus $X' = X_0$. Then $a \circ X'$ is an infix in $\mathcal{S}$ if and only if $[a \circ X_0]_{\equiv_{\mathcal{S}}}$ is a child of $[X_0]_{\equiv_{\mathcal{S}}}$ and we can check if this is the case with a single look up in the edges outgoing from $[X_0]_{\equiv_{\mathcal{S}}}$. If there is such a child, say $[C]_{\equiv_{\mathcal{S}}}$, then $a \circ X_0 \equiv_{\mathcal{S}} C$ and the length of $a \circ X_0$ is determined as $l' + 1$. Otherwise $a \circ X_0$ does not occur in $Inf(\mathcal{S})$.

It is important to stress that we not only decide whether $a \circ X'$ is an infix of $\mathcal{S}$ or not, but in case that it is we also obtain the node $[a \circ X']_{\equiv_{\mathcal{S}}}$ that represents $a \circ X'$ in the tree $\mathcal{T}_{\mathcal{S}}$ and its length $|a \circ X'| = 1 + l'$. Therefore we can successively extend to the left each infix of $\mathcal{S}$ until we hit a word that is not an infix in $\mathcal{S}$ or no further extensions to the left are needed.

The automaton $\mathcal{A}_{\mathcal{S}}$ and the tree structure $\mathcal{T}_{\mathcal{S}}$ can be constructed on-line in time $O(\sum_{W \in \mathcal{S}} |W|)$ by the means of the algorithm described in [12], [44]. The additional amendments of the tree structure $\mathcal{T}_{\mathcal{S}}$ can be easily incorporated in this algorithm without deteriorating its efficiency or they can be performed on a second preprocessing stage.

**Remark 2.2.3** Another linear space representation of the set of infixes of a single word and a finite set of words was proposed by Inenaga, [29, 30]. Essentially the ideas used [29, 30] naturally extend the ideas of Blumer et al. and in general achieve better compression than the structure of Blumer et Blumer. In [29, 30] Inenaga provides linear on-line algorithms that construct this structure that can

be adapted for the left/right extensions essentially in the same way as described above. For the purposes of our research it suffices that such a structure can be efficiently constructed and stored.

## 2.3 Suffix Arrays

We conclude this Chapter with a data structure which is tightly related with the representation of suffixes and infixes of a single word, $W$. Whereas the automata techniques we considered in the previous Sections provide an efficient traversal of infixes of a finite set of words, $\mathcal{S}$, the suffix arrays can be used to provide the lexicographical order of these infixes. The definition of a suffix array is the following:

**Definition 2.3.1** For a word $W = a_1 \circ a_2 \circ \cdots \circ a_N$, we denote with $W_i = a_i \circ \ldots a_N$ the suffix starting at position $i$. A suffix array, $AW$ for the word $W$ is an array of size of $N$ that presents a permutation of the numbers $\{1, 2, \ldots, N\}$ and has the following property:

$$W_{AW[i]} \prec_{lex} W_{AW[i+1]} \text{ for all } i < N.$$

Suffix array of a word $W$ of length $N$ can be computed in time $O(N)$, using a suffix tree [61], or directly [31, 34]. For a taxonomy of various algorithms constructing a suffix array for a given word we refer the reader to [51].

An important observation that we shall use in our algorithm is the following. Assume that $W = W_{\mathcal{S}}$ is the concatenation of all words in $\mathcal{S}$. We shall refer to the $i$-th character of $W_{\mathcal{S}}$ as $W_{\mathcal{S},i}$. Next let $A$ be a suffix array for $W_S$. Then if two distinct infixes $U, V \in Inf(\mathcal{S})$ are of equal lengths, $U$ starts at position $i$ and $V$ starts at position $j$ in $W_{\mathcal{S}}$, then:

$$U \prec_{lex} V \iff W_{\mathcal{S},i} \prec_{lex} W_{\mathcal{S},j}.$$

This follows by the fact that $U \neq V$. Thus, since $U$ and $V$ are of equal lengths, there is a first position, say $k$, where $U$ and $V$ are distinct. This would be also the first position where $W_{\mathcal{S},i}$ and $W_{\mathcal{S},j}$ are distinct and therefore $U \prec_{lex} V$ will be equivalent to $W_{\mathcal{S},i} \prec_{lex} W_{\mathcal{S},j}$.

Therefore if we compute the function $f$ such that:

$$A[f(i)] = i \text{ for } i \in \{1, 2 \ldots |W_{\mathcal{S}}|\}$$

we can easily check whether $W_{\mathcal{S},i} \prec_{lex} W_{\mathcal{S},j}$ by simply comparing the values $f(i)$ and $f(j)$. Indeed $f(i) < f(j)$ if and only if the value $i$ occurs before $j$ in the array $A$ which according to the definition of the suffix array is equivalent to $W_{\mathcal{S},i} \prec_{lex} W_{\mathcal{S},j}$.

# Chapter 3

# Example

This chapter is an informal presentation of our algorithm. Its main purpose is to give the flavour of what we desire to obtain. Along the intuitive approach, we shall come across questions we are going to discuss in details in the next chapters.

Let us consider the following simple example. Assume that we are given the language $\mathcal{L} = \{read, lead, ear\}$. We would like to process misspelled words $V$ and suggest their corrected variants $V$ from the list $\mathcal{L}$. The errors that we allow are *insertion*, *deletion* and *proper substitution* of a single character. It should be clear that using these operations we can correct an arbitrary word $V$ to each of the words $U \in \mathcal{L}$. However our intuition opposes to the fact that each word can be considered as a misspelling of *read*, *lead* and *ear*. In order to suppress this degenerated case, we impose a constraint on the number of unit corrections that can be done. For instance we can require that this number should not exceed 40% of the length of the query word $V$.

Let us see how this framework applies if the query word is $V = dread$. The length of $V$ is $|V| = 5$ and hence we are allowed to perform at most $2 = 40\%.5$ unit corrections. It is clear that if we *delete* the $d$ in the beginning we shall obtain *read*. Thus *read* is a misspelling variant of *dread* under these assumptions. If we further substitute the $r$ with $l$ we also obtain *lead*. Hence *lead* is also a misspelling variant of *dread*. However *ear* is not a correction variant of *dread*. Indeed since *dread* is of length 5 and *ear* is of length 3, we need at least 2 deletions for such a correction. So the characters of *ear* should not be involved in any correction and thus they have to occur in the same order in *dread*. This is obviously not the case.

The above naive approach is clearly inefficient since it requires to consider all the words in the language $\mathcal{L}$ before answering the query. Let us have a second glance at the problem. Think of $V = dread$ as a word which contains at most $2 = 40\%.5$ errors (which subsequently require a unit correction). Now if we split $V$ into $V_0 = dre$ and $V_1 = ad$, then at least in one of this words the number of errors is at most 40% of the length of the corresponding word. Indeed otherwise if the number of errors in $V_0$ was greater than $40\%.3$ and the number of errors

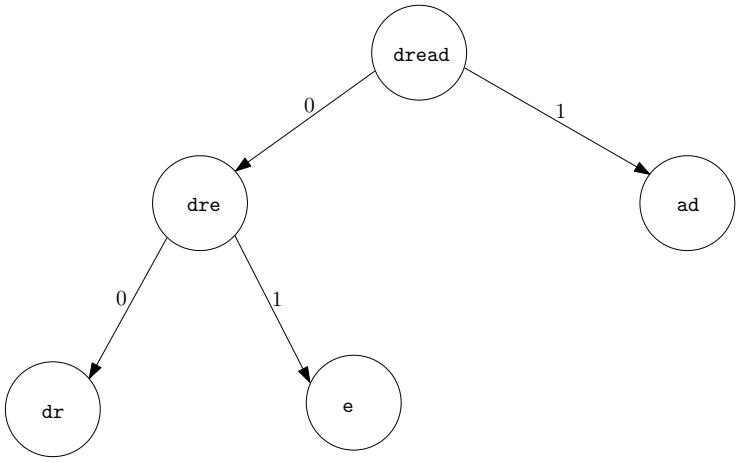



Figure 3.1: The search tree we build for the query word $V = dread$.

in $V_1$ was greater than 40%.2, then the total number of errors in $V$ would be more than $40\%(2+3) = 40\%.5$ which is a contradiction. Now the length of $V_0$ is $|V_0| = 3$ and the length of $V_1$ is $|V_1| = 2$. Assume that the first case applies, that is $V_0 = dre$ contains at most $40\%.3 = 1.2$ errors. We iterate our procedure and split $V_0$ into two $V_{00} = dr$ and $V_{01} = e$. As above we see that either in $V_{00}$ there are at most 40%.2 errors or in $V_{01}$ there are at most 40%.1 of errors, see Figure 3.1

At this stage we are at a situation where we know that either $V_{00} = dr$ or $V_{01} = e$ or $V_1 = ad$ contains at most 40% of its length errors. However $40\%.1 < 40\%.2 < 1$ and this means some of the subwords $V_{00} = dr$, $V_{01} = e$ or $V_1 = ad$ contains no errors at all. Now we look at our language $\mathcal{L}$ in order to see which of these three cases is consistent. Obviously, the combination $dr$ does not occur in the words *read*, *lead*, *ear*, thus we abandon this possibility. The two other words $V_{01} = e$ and $V_1 = ad$ occur as *subwords* in $\mathcal{L}$, so they are possible candidates which we have to consider in the further steps. It is crucial that at this step we do not need to consider all the words in the language $\mathcal{L}$ but only the infixes of the query word $V$. Using a suitable infix structure, see Chapter 2, this step can be handled in an efficient way.

We proceed with the hypothesis about $V_0 = dre$, see Figure 3.2. Could it contain fewer than $40\%.3 < 2$ errors? Looking at $V_{00}$ and $V_{01}$ we recognise, that the only possibility is the result $e$ for $V_{01}$ which occurs in the words of $\mathcal{L}$. In order to verify this hypothesis, we search for an extension of $e$ to the *left* in the words *read*, *lead*, *ear*, so that at most 1 error is induced to *dre*. There are three possibilities: (i) we extend $e$ to $re$ (read); (ii) we extend $e$ to $le$ (lead); (iii) we extend $e$ with blank (ear). In case (i) $re$ is a correction of $dre$ with 1 operation (deletion of $d$), the two other cases (ii) and (iii) are not consistent since they require at least 2 operations. To facilitate these steps it makes sense to use an appropriate filter which is able to prune the false extensions as soon as possible.

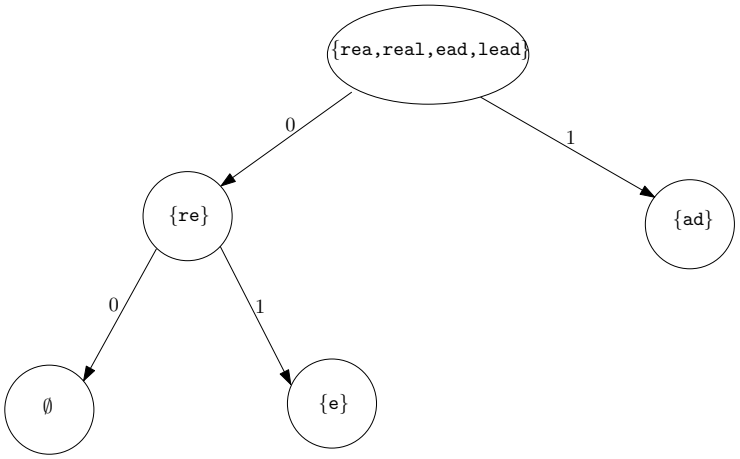



Figure 3.2: Solving the queries induced by $V = dread$ in a bottom-up fashion.

Finally we turn our attention to our query word $V = dread$. Can it contain at most $40\%.5 = 2$ errors? As in the previous situation, the answer of this question is reduced to $V_0 = dre$ and $V_1 = ad$.

First, we look at $V_0 = dre$. It permits a unique correction to $re$ with 1 error. Hence we look for *right* extensions of $re$ in the set of words, *read*, *lead*, *ear* which provoke at most 1 more error in the part $V_1 = ad$ of $V$. Clearly, we can extend $re$ to: (i) re; (ii) rea; or (iii) read. The first case (i) forces deletion of $ad$ so it causes 2 more errors to $V$ and thus is rejected. The two other possibilities are legible. Case (ii) causes a deletion of the last $d$ in $V_1$ and case (iii) requires no edition.

Next, we consider $V_1 = ad$. It permits unique correction to $ad$ with 0 errors. Hence we look for *left* extensions of $ad$ in the set of words *read*, *lead*, *ear* with credit of 2 errors in the part $V_0 = dre$. Here we obtain four cases: (i) ad; (ii) ead; (iii) read; (iv) lead. Clearly, all but the first one are valid corrections with at most 2 errors.

In this way we obtain that $V = dread$ can be corrected with at most 2 operations to one of the following: (i) rea; (ii) read; (iii) ead;(iv) lead. This is still not the answer of our original query, for it may contain subwords of $\mathcal{L}$ which are not words in $\mathcal{L}$ themselves. So we need to filter those which are among *read*, *lead* and *ear* and we obtain that only candidates (ii) and (iv) are valid words $\mathcal{L}$. Thus we report *read* and *lead*, the same as our naive approach suggested to be true.

In the next chapters we are going to formalise the intuitive concept described in Chapter 3. First, we shall consider the case when all the operations have right side of length at most 1. This includes the Levenshtein type operations but also extends to operations like $(\mathtt{ab}, \mathtt{c})$, or $(\mathtt{abcd}, \varepsilon)$. However transpositions like $(\mathtt{ab}, \mathtt{ba})$ or split operations like $(\mathtt{a}, \mathtt{ab})$ do not belong to this class, since $\mathtt{ba}$ in the first case and $\mathtt{ab}$ in the second case are both of length 2.

In order to distinguish between the class of operation of Levenshtein type and those that contain operations like transpositions or split operations, we introduce the following characteristic of operation sets.

**Definition 3.0.2** Let $Op$ be a set of operation, then $\rho = \rho(Op)$ is defined as:

$$\rho = \max\{|r(op)| \,|\, op \in Op\}.$$

Hence, the Levenshtein operations, $Op_L$, have characteristic $\rho(Op_L) = 1$. In the sequel we shall consider sets of operations, $Op$, with $\rho(Op) \geq 1$.

For sake of completeness, we should point out that the case $\rho(Op) = 0$ is a degenerated one where nothing interesting happens. Indeed, $\rho(Op) = 0$ means that the right side of each operation $op \in Op$ is $r(op) = \varepsilon$ and hence each alignment $\omega \in Op^*$ has right side $r(\omega) = \varepsilon$. Therefore only an input $V = \varepsilon$ may cause a nontrivial output. However if $V = \varepsilon$, then for every $q \in (0; 1)$ we have $q|V| = 0$. Hence only the empty word $U = \varepsilon$ satisfies $d_{Op}(U, V) \leq q|V|$. This shows that the case $\rho(Op) = 0$ is indeed a trivial one. Unless $V = \varepsilon$ we return the empty set. Otherwise if the empty word is in the (regular) language $\mathcal{L}$ we return the set $\{\varepsilon\}$ and else the empty set.

Hence, $\rho(Op) = 1$, is the next natural candidate to consider. Although restrictive it might appear, this case is quite representative and will allow us to present the essence of our approach. Once we have seen the formal solution in this basic case, it will be easy to realise how it can be adjusted for the general situation, $\rho(Op) \geq 1$. In large extent this will reduce to rather technical details that we postpone to Chapter 6.

# Chapter 4

# Alignments and Edit-Distance Lists

The purpose of this chapter is to improve our understanding about the alignments and their influence on the searching process in a language $\mathcal{L}$. This will help us to formalise the idea we presented in the previous chapter and to turn it into an efficient algorithm. To this end we describe some simple atomic operations on sets of alignments which reveal how an alignment set evolves in time. In certain sense these constructions illustrate how to generate alignments, i.e. sequences of operations, in a systematic stepwise manner. In addition the algebraic approach to the alignments allows us to formally specify and generalise the divide and conquer technique we introduced in Chapter 3.

The simplicity of the atomic operations on alignments sets allows transmitting them in a straightforward way to the searching process in a language $\mathcal{L}$. Since many distinct alignments may have the same impact on the search, one has to be careful in order to suppress the repetitive computations. In the second part of this chapter we tailor the *edit-distance lists* in a way that achieves this goal while preserving the necessary information which guarantees the correctness of the output. They generate each intermediate result only once and in this sense they are optimal. However they crucially rely on an efficient representation of the language $\mathcal{L}$ which determines their applicability in practice.

The approach outlined in this chapter is based on [19].

## 4.1 Some Basic Properties of the Alignments

Let us consider the following alignment of `lead` and `dread` over the Levenshtein operations,

$$\omega = (\varepsilon, d)(l, r)(e, e)(a, a)(d, d)$$

of total cost $c(\omega) = 2$. Clearly we can consider this alignment as a concatenation of the alignments:

$$\omega_0 = (\varepsilon, d)(l, r)(e, e) \text{ and } \omega_1 = (a, a)(d, d)$$

and thus as an alignments of `le` and `dre`, and `ad` and `ad`, respectively. This was the fact we utilised in our Example in Chapter 3 to argue that we can split the query `dread` into `dre` and `ad`. Each alignment of the former word can be viewed as a concatenation of alignments of the two shorter words.

**Lemma 4.1.1** *Let $Op$ be a set of operations with $\rho(Op) = 1$. If $V = V_0 \circ V_1$ is a word and $\omega \in Op^*$ is an alignment with right side $r(\omega) = V$, then there exist alignments $\omega_0$ and $\omega_1$ such that:*

$$\omega = \omega_0 \circ \omega_1 \text{ and } r(\omega_i) = V_i, \text{ for } i = 0, 1.$$

*Proof.* Let $\omega = \omega = op_1 \circ op_2 \circ \cdots \circ op_N$. We define $\omega^{(m)} = op_m \circ op_{m+1} \circ \cdots \circ op_N$ for $1 \leq m \leq N$. We set $\omega^{(N+1)} = \varepsilon$. Since each operation $op_m$ has right side $r(op_m)$ of length at most one it follows that $|r(\omega^{(m+1)})| + 1 \geq |r(\omega^{(m)})|$. Now $|r(\omega^{(1)}| = |r(\omega)| = |V|$ and $|r(\omega^{(N+1)}| = |\varepsilon| = 0$ and since $0 \leq |V_1| \leq |V|$ we deduce that for some $m$ we must have an equality $|r(\omega^{(m)})| = |V_1|$. Let $m_0$ be an instance of such an $m$ that $|r(\omega^{(m)})| = |V_1|$. We set $\omega_1 = \omega^{(m_0)}$ and $\omega_0 = op_1 \circ op_2 \circ \cdots \circ op_{m_0-1}$. It is now straightforward to complete the proof. Both $r(\omega_1)$ and $V_1$ are suffixes of $V$, and they are of equal lengths thus $r(\omega_1) = V_1$. Similarly, $r(\omega_0)$ and $V_0$ are prefixes of $V$ and an easy computation shows that they are of equal lengths:

$$|r(\omega_0)| = |r(\omega)| - |r(\omega_1)| = |V| - |V_1| = |V_0|.$$

Thus $r(\omega_0) = V_0$. Finally, by the definition of $\omega_0$ and $\omega_1$ we see that $\omega_0$ is constituted of the first $m - 1$ and $\omega_1$ by the last $N - m + 1$ operations of $\omega$. Thus $\omega = \omega_0 \circ \omega_1$. □

Intuitively, Lemma 4.1.1 tells us that we can express the alignments of a word $V = V_0 \circ V_1$ as a concatenation of alignments of the words $V_0$ and $V_1$. Here the point is that $V_0$ and $V_1$ are determined in advance and refer to all the alignments of $V$. Our next step is to relate these decomposition with appropriate cost-bounds of the alignments. This can be achieved by a simple application of the Pigeonhole Principle. We sketched it in Chapter 3, but it is useful to revisit it at this stage once again. Let $V =$ `dread`, $V_0 =$ `dre` and $V_1 =$ `ad`. We consider the alignment:

$$\omega = (\varepsilon, d)(l, r)(e, e)(a, a)(d, d)$$

of total cost $c(\omega) = 2$. Now we can argue by Lemma 6.1.1 that

$$\omega_0 = (\varepsilon, d)(l, r)(e, e) \text{ and } \omega_1 = (a, a)(d, d)$$

exist and we have $c(\omega) = c(\omega_0) + c(\omega_1)$. Since $c(\omega) \leq 2$ it is impossible that $c(\omega_0) > 1$ and $c(\omega_1) > 1$ simultaneously. Thus either $c(\omega_0) \leq 1$ or $c(\omega_1) \leq 1$.

This being the general idea we present it formally in the sequel. We start with a variant of the lemma on page 3 in [47].

**Corollary 4.1.2** *Let $b_0, b_1$ be nonnegative rational numbers and $b = b_0 + b_1$. Let $Op$ be a set of operations with $\rho(Op) = 1$ and $V = V_0 \circ V_1$ be a word. Then an alignment $\omega \in Op^*$ with right side $r(\omega) = V$ is of cost $c(\omega) \leq b$ only if there exist alignments $\omega_0, \omega_1$ such that $\omega = \omega_0 \circ \omega_1$, $r(\omega_i) = V_i$ for $i = 0, 1$ and:*

1. *either $c(\omega_0) \leq b_0$,*
2. *or $c(\omega_1) \leq b_1$.*

*Proof.* Let $\omega$ with $r(\omega) = V$ be of cost $r(\omega) \leq b$. We consider the alignments $\omega_0$ and $\omega_1$ determined by Lemma 4.1.1. Then clearly $\omega_0 \circ \omega_1 = \omega$ and $r(\omega_i) = V_i$ for $i = 0, 1$. For the sake of contradiction assume that $c(\omega_i) > b_i$ for both $i = 0$ and $i = 1$. Then $c(\omega) = c(\omega_0) + c(\omega_1) > b_0 + b_1 = b$. This is a contradiction. Thus either $c(\omega_0) \leq b_0$ or $c(\omega_1) \leq b_1$ as required. □

In our algorithm we are going to apply Corollary 4.1.2 for $b = q|V|$, $b_0 = q|V_0|$ and $b_1 = q|V_1|$. Then since $|V| = |V_0| + |V_1|$ we obtain that $b = b_0 + b_1$ and the assumptions of the Corollary 4.1.2 hold. We need only very little to reverse the implication of this statement. However we postpone these details until later and we move our attention from single alignments to sets of alignments.

## 4.2 Sets of Alignments

Our example from Chapter 3 suggests that we follow many candidates at a time. This is why we need a formal framework that allows to deal with all of them simultaneously in a homogeneous way. In order to feel what sort of primitives would be suitable for our purposes, we start our analysis with considerations about set of alignments.

The presence of threshold $b \in \mathbb{Q}^+$ suggests in a natural way the following definition for a set of alignments:

**Definition 4.2.1** Let $\mathfrak{A} \subseteq Op^*$ be a set of alignments and $b \in \mathbb{Q}^+$. Then:

$$\mathfrak{A}^{\leq b} = \{\omega \in \mathfrak{A} \,|\, c(\omega) \leq b\}$$

The second notion that we introduce is closely related with the way we generate candidates. The problem arises with the operations from the set $Op$ which can be applied in different order without changing the right side of the alignment. Formally, we define:

**Definition 4.2.2** For a set of operations $Op$, $\Lambda = \Lambda(Op)$ is defined as:

$$\Lambda = \{op \in Op \,|\, r(op) = \varepsilon\}.$$

From practical point of view it might be that it is not that important how exactly the operations $\Lambda$ would be applied (simulated). However, this does matter in order to study the efficiency of our algorithm.

Very generally and superficially, our idea is to generate shorter candidates first and than longer. We are going to be much more explicit in the next chapters and provide the necessary mathematical arguments.

At the current stage we simply give the following definition:

**Definition 4.2.3** Let $\mathfrak{A} \subseteq Op^*$ be a set of alignments and $j \in \mathbb{N}$ be an integer, then:

$$\mathfrak{A}[j] = \{\omega \in \mathfrak{A} \,|\, |l(\omega)| = j\}.$$

In the sequel we summarise some simple properties of these two notions. First of all we remark that for every set of alignments $\mathfrak{A}$ we have the equality:

$$\mathfrak{A} = \cup_{j=0}^{\infty} \mathfrak{A}[j].$$

The reason for this is that every alignment $\omega \in \mathfrak{A}$ has a particular left side length, $|l(\omega)|$, which uniquely determines a $j = |l(\omega)|$ such that $\omega \in \mathfrak{A}[j]$. The reverse inclusion is straightforward since each of the sets $\mathfrak{A}[j]$ is a subset of $\mathfrak{A}$ by definition.

Next lemma gives a recursive definition of the set $\mathfrak{A} \circ \Lambda^*$:

**Lemma 4.2.4** *Let $\mathfrak{A}$ be a set of alignments, $b \in \mathbb{Q}^+$ be a nonnegative rational number. If $\mathfrak{B} = \mathfrak{A} \circ \Lambda^*$, then for each $j \in \mathbb{N}$ it holds:*

$$\mathfrak{B}[j] = \mathfrak{A}[j] \cup \textstyle\bigcup_{op \in \Lambda} \mathfrak{B}[j - |l(op)|] \circ op$$
$$\mathfrak{B}^{\leq b}[j] = \mathfrak{A}^{\leq b}[j] \cup \textstyle\bigcup_{op \in \Lambda} \left(\mathfrak{B}^{\leq b}[j - |l(op)|] \circ op\right)^{\leq b}.$$

*Proof.* It should be clear that $\mathfrak{B} = \mathfrak{A} \cup \mathfrak{A}\Lambda^+ = \mathfrak{A} \cup \mathfrak{B} \circ \Lambda$. Now the first equality can be verified as follows. An alignment $\omega \in \mathfrak{B}[j]$ is an alignment $\omega \in \mathfrak{B}$ with $|l(\omega)| = j$. Therefore $\omega \in \mathfrak{A}$ or $\omega \in \mathfrak{B} \circ \Lambda$. In the former case we have $\omega \in \mathfrak{A}[j]$. In the latter one $\omega = \omega' \circ op$ with $op \in \Lambda$ and $\omega' \in \mathfrak{B}$. Since $|l(\omega)| = |l(\omega')| + |l(op)|$ we deduce that $|l(\omega')| = j - |l(op)|$. Therefore $\omega' \in \mathfrak{B}[j - |l(op)|]$. Hence $\omega \in \mathfrak{A}[j] \cup \bigcup_{op \in \Lambda} \mathfrak{B}[j - |l(op)|] \circ op$.

Conversely each alignment $\omega \in \mathfrak{A}[j] \cup \bigcup_{op \in \Lambda} \mathfrak{B}[j - |l(op)|] \circ op$ has the properties $|l(\omega)| = j$ and $\omega \in \mathfrak{A} \cup \mathfrak{B} \circ \Lambda$. Thus $\omega \in \mathcal{B}$ and $|l(\omega)| = j$ which shows $\omega \in \mathfrak{B}[j]$.

The second part of the lemma can be derived from the first. Namely we have:

$$\mathfrak{B}^{\leq b}[j] = \left(\mathfrak{A}[j] \cup \bigcup_{op \in \Lambda} \mathfrak{B}[j - |l(op)|] \circ op\right)^{\leq b}.$$

And since every alignment of cost less than or equal to $b$ has only subalignments which fulfil this constraint too it follows that

$$\begin{aligned} \mathfrak{B}^{\leq b}[j] &= \left(\mathfrak{A}[j] \cup \bigcup_{op \in \Lambda} \mathfrak{B}[j - |l(op)|] \circ op\right)^{\leq b} \\ &= \mathfrak{A}^{\leq b}[j] \cup \bigcup_{op \in \Lambda} \left(\mathfrak{B}^{\leq b}[j - |l(op)|] \circ op\right)^{\leq b}. \end{aligned}$$

□

A special case of alignments which naturally arise in the context of our problem is the set of alignments with fixed right side.

**Definition 4.2.5** Let $V$ be a word, $Op$ be a set of operations, then:

$$\mathfrak{A}_{Op}(V) = \mathfrak{A}(V) = \{\omega \in Op^* \mid r(\omega) = V\}.$$

It is now clear that the answer of a query induced by a word $V$ is closely related with the structure of the alignment set $(\mathfrak{A}(V))^{\leq b}$ with $b = q|V|$.

In the sequel we shall consider the set $\mathfrak{A}(V)$ and derive some simple still useful facts about them. In this section we shall consider the simple case when $\rho(Op) = 1$, i.e. each operation has right side of length at most 1. The general case will be described in Chapter 6.

At a first step we shall obtain the set $\mathfrak{A}(V)$ as a result of an atomic operation applied on a simpler set. Next lemma gives such a characterisation:

**Lemma 4.2.6** *Let $Op$ be a set of operations with $\rho(Op) = 1$, $V = V' \circ \sigma$ be a word and $\sigma \in \Sigma$ be a single character. Then:*

$$\mathfrak{A}(V) = \left( \bigcup_{op=(U,\sigma)\in Op} \mathfrak{A}(V') \circ op \right) \circ \Lambda^*.$$

*Proof.* Let us first consider an alignment $\omega \in \mathfrak{A}(V)$. This means that $r(\omega) = V = V' \circ \sigma$. Let $op \in Op$ be the last operation in $\omega$ that has a nonempty right side, i.e. $r(op) \neq \varepsilon$. Thus $\omega = \omega' \circ op \circ \omega_\varepsilon$ where all of the operations involved in $\omega_\varepsilon$ have empty right side. Hence $\omega_\varepsilon \in \Lambda^*$ or equivalently $r(\omega_\varepsilon) = \varepsilon$. Now we have:

$$V' \circ \sigma = V = r(\omega) = r(\omega') \circ r(op) \circ (\omega_\varepsilon) = r(\omega') \circ r(op).$$

Since $|r(op)| \leq 1$ and $|r(op)| \neq 0$ we obtain that $|r(op)| = 1$ and therefore $r(op) = \sigma$. Thus $op$ is of the form $(U, \sigma) \in Op$ for an appropriate word $U$. It is also straightforward that $r(\omega') = V'$ and hence $\omega' \in \mathfrak{A}(V')$. This readily shows that:

$$\omega = \omega' \circ op \circ \omega_\varepsilon \in \left( \bigcup_{op=(U,\sigma)\in Op} \mathfrak{A}(V') \circ op \right) \circ \Lambda^*.$$

The inclusion from right to left is straightforward. Concatenating an alignment with right side $V'$ with an operation with right side $x$ clearly results in an alignment, $\omega$, with right side $V' \circ \sigma = V$. Concatenating $\omega$ with operations $op \in \Lambda$ does not change the right side of $\omega$, since $r(op) = \varepsilon$. □

Next lemma generalises the result from Corollary 4.1.2 to alignment sets and provides the missing reverse inclusion part:

**Lemma 4.2.7** *Let $Op$ be a set of operations with $\rho(Op) = 1$ and $V = V_0 \circ V_1$ for some nonempty words $V_0, V_1 \in \Sigma^*$ with lengths $n_0$ and $n_1$, respectively. Let $b = b_0 + b_1$ be rational numbers $b_0, b_1 \in \mathbb{Q}^+$. Then:*

$$\mathfrak{A}^{\leq b}(V) = \left(\mathfrak{A}^{\leq b_0}(V_0) \circ \mathfrak{A}(V_1)\right)^{\leq b} \cup \left(\mathfrak{A}(V_0) \circ \mathfrak{A}^{\leq b_1}(V_1)\right)^{\leq b}$$

*Proof.* We first prove the inclusion from left to right. To this end we consider an arbitrary alignment $\omega \in \mathfrak{A}^{\leq b}(V)$. By Lemma 4.1.2, $\omega = \omega_0 \circ \omega_1$ with $\omega_i \in \mathfrak{A}(V_i)$ for $i = 0, 1$ and:

$$c(\omega_0) \leq b_0 \text{ or } c(\omega_1) \leq b_1.$$

In the former case we have that $\omega_0 \in \mathfrak{A}^{\leq b_0}(V_0)$ and since $\omega_1 \in \mathfrak{A}(V_1)$ we get:

$$\omega \in \left(\mathfrak{A}^{\leq b_0}(V_0) \circ \mathfrak{A}(V_1)\right)^{\leq b}.$$

Symmetrically, in the latter case we have that $\omega_1 \in \mathfrak{A}^{\leq b_1}(V_1)$ and using that $\omega_0 \in \mathfrak{A}(V_0)$ we get:

$$\omega \in \left(\mathfrak{A}(V_0) \circ \mathfrak{A}^{\leq b_1}(V_1)\right)^{\leq b}.$$

The inclusion from right to left is straightforward. Each alignment $\omega \in \mathfrak{A}(V_0) \circ \mathfrak{A}(V_1)$ clearly satisfies $r(\omega) = V_0 \circ V_1 = V$ and thus $\omega \in \mathfrak{A}(V)$. The constraint $^{\leq b}$ on the out most level then guarantees that $\omega \in \mathfrak{A}^{\leq b}(V)$.

□

With respect to the result of the previous lemma we introduce the following notion.

**Definition 4.2.8** Let $V = V_0 \circ V_1$ be words and $b_0, b_1 \in \mathbb{Q}^+$ be nonnegative rational numbers with sum $b = b_0 + b_1$. Then we introduce the alignment sets $\mathfrak{A}(V_0, b_0 \rightarrow V, b)$ and $\mathfrak{A}(V, b \leftarrow V_1, b_1)$ as:

$$\begin{aligned} \mathfrak{A}(V_0, b_0 \rightarrow V, b) &= \left(\mathfrak{A}^{\leq b_0}(V_0) \circ \mathfrak{A}(V_1)\right)^{\leq b}. \\ \mathfrak{A}(V, b \leftarrow V_1, b_1) &= \left(\mathfrak{A}(V_0) \circ \mathfrak{A}^{\leq b_1}(V_1)\right)^{\leq b} \end{aligned}$$

Now, using Lemma 4.2.6 we can derive an iterative description of the set:

$$\mathfrak{A}(V_0, b_0 \rightarrow V, b).$$

Since similar result can be dually obtained, see Section 4.4, for the set:

$$\mathfrak{A}(V, b \leftarrow V_1, b_1)$$

we actually get a constructive description of $\mathfrak{A}^{\leq b}(V)$ in the terms of the simpler $\mathfrak{A}^{\leq b_i}(V_i)$ for $i = 0, 1$.

**Lemma 4.2.9** *Let $Op$ be a set of operations with $\rho(Op) = 1$ and $V = V_0 \circ V_1$ be a word such that $|V_i| = n_i \geq 1$ and let $b = b_0 + b_1$ be nonnegative rational numbers. Let $\overrightarrow{\mathfrak{A}}^j$ be the alignment sets:*

$$\overrightarrow{\mathfrak{A}}^j = \mathfrak{A}(V_0, b_0 \rightarrow V_0 \circ I_1^j(V_1), b) = \left(\mathfrak{A}^{\leq b_0}(V_0) \circ \mathfrak{A}(I_1^j(V_1))\right)^{\leq b}.$$

*Then the following recurrence holds:*

$$\begin{aligned} \overrightarrow{\mathfrak{A}}^0 &= \left(\mathfrak{A}^{\leq b_0}(V_0) \circ \Lambda^*\right)^{\leq b} \\ \overrightarrow{\mathfrak{A}}^{j+1} &= \left(\left(\bigcup_{op=(U, I_{j+1}^{j+1}(V_1)) \in Op} (\overrightarrow{\mathfrak{A}}^j \circ op)^{\leq b}\right) \circ \Lambda^*\right)^{\leq b} \end{aligned}$$

*for* $j < n_1$.

*Proof.* For $j = 0$ we have that $I_1^0(V_1) = \varepsilon$ and therefore:

$$\mathfrak{A}(I_1^0(V_1)) = \mathfrak{A}(\varepsilon) = \Lambda^*.$$

This clearly implies the desired equality.

$$\mathfrak{A}^0 = \left(\mathfrak{A}^{\leq b_0}(V_0) \circ \mathfrak{A}(I_1^0(V_1))\right)^{\leq b} = \left(\mathfrak{A}^{\leq b_0}(V_0) \circ \Lambda^*\right)^{\leq b}$$

Next we use Lemma 4.2.6 in order to express $\mathfrak{A}(I_1^{j+1}(V))$ in terms of $\mathfrak{A}(I_1^j(V))$. Let $\sigma_{j+1}$ be the $(j+1)$-st character of $V_1$ then we deduce that: $I_1^{j+1}(V_1) = I_1^j(V_1) \circ \sigma_{j+1}$. Applying Lemma 4.2.6 we obtain:

$$\mathfrak{A}(I_1^{j+1}(V_1)) = \bigcup_{op=(U,\sigma_{j+1})\in Op} \mathfrak{A}(I_1^j(V_1)) \circ op \circ \Lambda^*.$$

Concatenating both sides with $\mathfrak{A}^{\leq b_0}(V_0)$ and using the distributive law of the union over the concatenation we get:

$$\mathfrak{A}^{\leq b_0}(V_0) \circ \mathfrak{A}(I_1^{j+1}(V_1)) = \bigcup_{op=(U,\sigma_{j+1})\in Op} \mathfrak{A}^{\leq b_0}(V_0) \circ \mathfrak{A}(I_1^j(V_1)) \circ op \circ \Lambda^*.$$

Finally, introducing the constraint $^{\leq b}$ for all the alignments we obtain the result:

$$\begin{aligned}
\overrightarrow{\mathfrak{A}}^{j+1} &= \left(\left(\bigcup_{op=(U,I_{j+1}^{j+1}(V_1))\in Op} ((\mathfrak{A}^{\leq b_0}(V_0) \circ \mathfrak{A}(I_1^j(V_1)))^{\leq b} \circ op)^{\leq b}\right) \circ \Lambda^*\right)^{\leq b} \\
&= \left(\left(\bigcup_{op=(U,\sigma_{j+1})\in Op} (\overrightarrow{\mathfrak{A}}^j \circ op)^{\leq b}\right) \circ \Lambda^*\right)^{\leq b}.
\end{aligned}$$

□

## 4.3 Edit-Distance Lists

The notion of alignment is only a tool which defines the edit-distance between words. In this sense many different alignments may witness for a particular edit-distance. For instance along with the alignment $\omega' = (\mathtt{l},\mathtt{d})(\varepsilon,\mathtt{r})(\mathtt{e},\mathtt{e})(\mathtt{a},\mathtt{a})(\mathtt{d},\mathtt{d})$ we also have the alignment $\omega'' = (\varepsilon,\mathtt{d})(\mathtt{l},\mathtt{r})(\mathtt{e},\mathtt{e})(\mathtt{a},\mathtt{a})(\mathtt{d},\mathtt{d})$ which has the same cost as $\omega'$, $c(\omega'') = c(\omega') = 2$. In practice we are ignorant about the concrete alignments as far as they witness for the edit-distance of the words. In our case we are interested that $d(\mathtt{lead},\mathtt{dread}) = 2$ and we care little about $\omega'$ and $\omega''$.

Furthermore, the alignment $(\mathtt{l},\mathtt{d})(\varepsilon,\mathtt{r})(\mathtt{e},\varepsilon)(\varepsilon,\mathtt{e})(\mathtt{a},\mathtt{a})(\mathtt{d},\mathtt{d})$ which is of cost 3 is redundant since it is a negative witness for the edit-distance $d(\mathtt{lead},\mathtt{dread}) = 2$.

These three and many other alignments would be elements of $\mathfrak{A}^{\leq 3}(dread)$. In this section we consider a model which removes these redundancies but preserves the expressiveness of the alignment sets with respect to unions and concatenations with a single operation.

Our idea is to compactly represent the answers of the following kind of queries:

`Given:regular language` $\mathcal{L}$, $q \in (0;1)$
`Query:` $V \in \Sigma^*$, $b = q|V|$
`Answer:` $\{l(\omega) \in Inf(\mathcal{L}) |\ \omega \in \mathfrak{A}^{\leq b}(V)\}$

In the special case when $\mathcal{L} = \Sigma^*$, the answer of the query would correspond to the removal of the redundancies in $\mathfrak{A}^{\leq b}(V)$. In the other extreme, when $\mathcal{L}$, is the particular (regular) language of interest for the approximate search problem, the solution of this query can be easily filtered so that we obtain the answer of the approximate search problem.

We formally define the *edit-distance lists* as (partial) functions which map words to integers:

**Definition 4.3.1** An edit-distance list is a partial function $L : \Sigma^* \to \mathbb{N}$.

In general the edit-distance lists have little to do with alignments operations. However, in view of the above query, every language and every alignment set define a unique edit-distance list. Exactly these edit-distance lists will be in the focus of this section and will play an important part in our algorithm.

**Definition 4.3.2** Let $\mathcal{L}$ be a language, $\mathfrak{A}$ be a set of alignments. We say that the edit-distance list $L : \Sigma^* \to \mathbb{N}$ $\mathcal{L}$-represents $\mathfrak{A}$ if and only if:

$$Dom(L) = Inf(\mathcal{L}) \cap \{l(\omega) \,|\, \omega \in \mathfrak{A}\}$$
$$L(U) = \min\{c(\omega) \,|\, \omega \in \mathfrak{A} \text{ and } l(\omega) = U\}.$$

In the special case when $\mathfrak{A} = \mathfrak{A}^{\leq b}(V)$ the edit-distance list $L$ which $\mathcal{L}$-represents $\mathfrak{A}$ gives the precise answer of the query defined above and additionally provides the edit-distance $d(U, V) = L(U)$ for every answer of the query.

In the previous section we expressed the alignment sets $\mathfrak{A}(V)$ recursively by the means of concatenation with a single operation and unions of alignment sets. Hence it is interesting whether the edit-distance lists inherit these properties of alignment sets. In the sequel we shall see that the answer is in a certain sense affirmative. We say 'in certain sense' because the language $\mathcal{L}$ is given at this stage in a non-constructive manner and the decision problem about $Inf(\mathcal{L})$ is something we cannot argue about. But abstracting the decision procedure gives us a efficient method which simulates union of alignment sets and concatenation of alignment sets with a single operation on edit-distance lists.

Next two lemmata summarise these results:

**Lemma 4.3.3** *Let $\mathcal{L}$ be a language, $\mathfrak{A}$ be an alignment set and $op \in Op$ be an operation, $op = (X, Y)$. If $L : \Sigma^* \to \mathbb{N}$ $\mathcal{L}$-represents $\mathfrak{A}$, then the edit-distance*

*list* $L' : \Sigma^* \to \mathbb{N}$ *defined as:*

$$\begin{aligned} Dom(L') &= Dom(L) \circ X \cap Inf(\mathcal{L}) \\ L'(U \circ X) &= \begin{cases} L(U) + c(op) \text{ if } U \circ X \in Inf(\mathcal{L}) \\ \neg! \text{ else} \end{cases} \end{aligned}$$

$\mathcal{L}$*-represents the alignment set* $\mathfrak{A} \circ op$.

*Proof.* Since $l(op) = X$ each alignment $\omega \in \mathfrak{A} \circ op$ has left side $l(\omega) \in \Sigma^* \circ X$. We first prove that:

$$Dom(L') = \{l(\omega) \,|\, \omega \in \mathfrak{A} \circ op \text{ and } l(\omega) \in Inf(\mathcal{L})\}.$$

First consider a word $U' \in Dom(L')$. Then $U' \in Dom(L) \circ X$. Since $L$ $\mathcal{L}$-represents $\mathfrak{A}$, we deduce that $U' = l(\omega) \circ X$ with $\omega \in \mathfrak{A}$. Since $op = (X, Y)$ we obtain that $\omega \circ op \in \mathfrak{A} \circ op$ and $l(\omega \circ op) = l(\omega) \circ X = U'$. Finally $U' \in Inf(\mathcal{L})$ and therefore $l(\omega \circ op) \in Inf(\mathcal{L})$.

Next, let $\omega' \in \mathfrak{A} \circ op$ and $l(\omega') \in Inf(\mathcal{L})$. Therefore, $\omega' = \omega \circ op$ for some alignment $\omega \in \mathfrak{A}$. Let $U = l(\omega)$. Since $op = (X, Y)$ we have that $l(\omega') = U \circ X \in Inf(\mathcal{L})$. It remains to verify that $U \circ X \in Dom(L) \circ X$ which is equivalent to show that $U \in Dom(L)$. But $U \circ X \in Inf(\mathcal{L})$ and therefore the subword $U \in Inf(\mathcal{L})$. We also have that $l(\omega) = U$ with $\omega \in \mathfrak{A}$. Since $L$ $\mathcal{L}$-represents $\mathfrak{A}$ this implies that $U \in Dom(L)$.

The last step of the proof is to show that the values attained by $L'$ are the same as those taken by the edit-distance list which $\mathcal{L}$-represents $\mathfrak{A} \circ op$. This follows by a straightforward computation. Let $U \circ X \in Dom(L')$, then:

$$\begin{aligned} L'(U \circ X) &= L(U) + c(op) = \min\{c(\omega) \,|\, \omega \in \mathfrak{A}, l(\omega) = U\} + c(op) \\ &= \min\{c(\omega) + c(op) \,|\, \omega \in \mathfrak{A}, l(\omega) = U\} \\ &= \min\{c(\omega \circ op) \,|\, \omega \circ op \in \mathfrak{A} \circ op, l(\omega) \circ l(op) = U \circ X\} \\ &= \min\{c(\omega') \,|\, \omega' \in \mathfrak{A} \circ op, l(\omega') = U \circ X\} \end{aligned}$$

as required by the definition of the edit-distance list $\mathcal{L}$-representing $\mathfrak{A} \circ op$ □

Next property of the edit-distance lists shows the relationship between the edit-distance lists representing an alignment set $\mathfrak{A}$ and the edit-distance list representing the alignment set $\mathfrak{A}^{\leq b}$:

**Property 4.3.4** *Let* $L$ *be an edit-distance list which* $\mathcal{L}$*-represents the alignment set* $\mathfrak{A}$ *and let* $b \in Q^+$ *be a threshold. Then the edit-distance list* $L^{\leq b} : \Sigma^* \to \mathbb{N}$ *defined as:*

$$L^{\leq b}(U) = \begin{cases} L(U) \text{ if } L(U) \leq b \\ \neg! \text{ else} \end{cases}$$

$\mathcal{L}$*-represents* $\mathfrak{A}^{\leq b}$.

*Proof.* Indeed, a word $U \in Dom(L^{\leq b})$ if and only if $U \in Dom(L)$ and $L(U) \leq b$. Since for $U \in Dom(L)$:

$$L(U) = \min\{c(\omega) \,|\, \omega \in \mathfrak{A} \text{ and } l(\omega) = U\},$$

the constraint $L(U) \leq b$ implies that:

$$L(U) = \min\{c(\omega) \leq b \,|\, \omega \in \mathfrak{A} \text{ and } l(\omega) = U\} = \min\{c(\omega) \,|\, \omega \in \mathfrak{A}^{\leq b} \text{ and } l(\omega) = U\}.$$

If $L(U) > b$, then every alignment $\omega \in \mathfrak{A}$ such that $l(\omega) = U$ is of cost $c(\omega) > b$ and hence $\omega \notin \mathfrak{A}^{\leq b}$. Therefore, such words $U$ do not belong to the domain of the edit-distance list $\mathcal{L}$-representing $\mathfrak{A}^{\leq b}$. □

**Corollary 4.3.5** *Let $\mathcal{L}$ be a language, $\mathfrak{A}$ be a set of alignments, $op = (X, Y)$ be an operation and $b \in \mathbb{Q}^+$ be a threshold. If an edit-distance list $L$ $\mathcal{L}$-represents $\mathfrak{A}$, then the edit-distance list $L'' : \Sigma^* \to \mathbb{N}$ defined as:*

$$\begin{aligned} Dom(L'') &\subseteq Dom(L) \circ X \cap Inf(\mathcal{L}) \\ L''(U \circ X) &= \begin{cases} L(U) + c(X) \text{ if } L(U) + c(X) \leq b \text{ and } U \circ X \in Inf(\mathcal{L}) \\ \neg! \text{ else} \end{cases} \end{aligned}$$

*$\mathcal{L}$-represents the alignment set $(\mathfrak{A} \circ op)^{\leq b}$.*

*Proof.* Immediately by Lemma 4.3.3 and Remark 4.3.4

□

The constructions described by Lemma 4.2.6 and Corollary 4.3.5 reflect the concatenation of alignment sets with a single operation as an operation on edit-distance lists which can $\mathcal{L}$-represents the given alignment set. Furthermore, if we can efficiently solve the decision problem for the language $\mathcal{L}$, the construction in Lemma 4.3.3 and Corollary 4.3.5 amount to a single go through the domain of the input edit-distance list.

Our next goal is to describe a similar construction which reflects the union of alignment sets. This can be achieved quite easy. This time we do not even need to solve the decision problem for the language $\mathcal{L}$. The idea is that a word, $U$, represented by the union of two alignment sets must be represented by at least one of them. The reverse is also true. And in this case the value of the edit-distance list at the word $U$ is also easily computed using only the values of the initial edit-distance lists at $U$, if defined. The formal statement is the following:

**Lemma 4.3.6** *Let $\mathcal{L}$ be a language and $\mathfrak{A}_1$ and $\mathfrak{A}_2$ be alignment sets. If $L_1$ and $L_2$ are edit-distance lists $\mathcal{L}$-representing $\mathfrak{A}_1$ and $\mathfrak{A}_2$, respectively, then the edit-distance list $L : \Sigma^* \to \mathbb{N}$ defined as:*

$$L(U) = \begin{cases} \min\{L_1(U), L_2(U)\} \text{ if } U \in Dom(L_1) \cap Dom(L_2) \\ L_1(U) \text{ if } U \in Dom(L_1) \setminus Dom(L_2) \\ L_2(U) \text{ else} \end{cases}$$

*$\mathcal{L}$-represents the alignment set $\mathfrak{A} = \mathfrak{A}_1 \cup \mathfrak{A}_2$.*

*Proof.* By the definition of the edit-distance list $L$ we have that:

$$Dom(L) = Dom(L_1) \cup Dom(L_2) = \{l(\omega) \,|\, \omega \in \mathfrak{A}_1, l(\omega) \in Inf(\mathcal{L})\} \cup \{l(\omega) \,|\, \omega \in \mathfrak{A}_2, l(\omega) \in Inf(\mathcal{L})\}$$

where the last equality follows since $L_i$ $\mathcal{L}$-represents the alignment set $\mathfrak{A}_i$ for $i = 1, 2$. Therefore:

$$Dom(L) = \{l(\omega) \,|\, \omega \in \mathfrak{A}_1 \cup \mathfrak{A}_2, l(\omega) \in Inf(\mathcal{L})\} = \{l(\omega) \,|\, \omega \in \mathfrak{A}, l(\omega) \in Inf(\mathcal{L})\}.$$

Therefore to prove that $L$ $\mathcal{L}$-represents the alignment set $\mathfrak{A}$ it suffices to show that:

$$L(U) = \min\{c(\omega) \,|\, \omega \in \mathfrak{A} \text{ and } l(\omega) = U\}$$

for every $U \in Dom(L)$. Then at least one of the sets $\{\omega \in \mathfrak{A}_1 \,|\, l(\omega) = U\}$ and $\{\omega \in \mathfrak{A}_2 \,|\, l(\omega) = U\}$ is nonempty. Consequently:

$$\begin{aligned} &\min\{c(\omega) \,|\, \omega \in \mathfrak{A} \text{ and } l(\omega) = U\} = \min\{c(\omega) \,|\, \omega \in \mathfrak{A}_1 \cup \mathfrak{A}_2 \text{ and } l(\omega) = U\} \\ &= \min\{\min\{c(\omega) \,|\, \omega \in \mathfrak{A}_1 \text{ and } l(\omega) = U\}, \min\{c(\omega) \,|\, \omega \in \mathfrak{A}_2 \text{ and } l(\omega) = U\}\} \end{aligned}$$

Now if $U \in Dom(L_1) \cap Dom(L_2)$ the last minimum is equivalent to $\min\{L_1(U), L_2(U)\} = L(U)$. If $U \in Dom(L_1) \setminus Dom(L_2)$, the minimum is $L_1(U)$ and finally if $U \in Dom(L_2) \setminus Dom(L_1)$, then it is $L_2(U)$. In either case we obtain that:

$$L(U) = \min\{c(\omega) \,|\, \omega \in \mathfrak{A} \text{ and } l(\omega) = U\}$$

for $U \in Dom(L)$ as required. □

## 4.4 Reversing Alignments

Reconsidering Lemma 4.2.4, Lemma 4.2.6 and Lemma 4.3.3 there is an evident advantage that we give to *right* concatenation. And we never mentioned that symmetric properties and constructions are valid for concatenations on the *left*. Namely how about $\Lambda^* \circ \mathfrak{A}$, or $\mathfrak{A}(V) \circ \mathfrak{B}$? Do we have similar formulae and can we reflect them via edit-distance lists? The answers to all these questions are affirmative. But instead of arguing by symmetry, we first describe the symmetry which is expressed by the reversal of the alignments.

Intuitively, given the alignment, $\omega = (\mathtt{l}, \mathtt{d})(\varepsilon, \mathtt{r})(\mathtt{e}, \mathtt{e})(\mathtt{a}, \mathtt{a})(\mathtt{d}, \mathtt{d})$, the reverse alignment should: (i) reverse the order of the individual operations, (ii) align the reverse words, i.e. $\mathtt{lead}^{rev} = \mathtt{dael}$ and $\mathtt{dread}^{rev} = \mathtt{daerd}$. In order to achieve this we give the following definition:

**Definition 4.4.1** Let $(Op, c)$ be a set of operations supplied with a cost function. For an operation $op = (X, Y) \in Op$, we define $op^{rev} = (X^{rev}, Y^{rev})$. We define $(Op^{rev}, c^{rev})$ as:

$$\begin{aligned} Op^{rev} &= \{op^{rev} \,|\, op \in Op\} \\ c^{rev}(op^{rev}) &= c(op). \end{aligned}$$

First we notice that $(op^{rev})^{rev} = op$ because of the corresponding property of the reverse operation on words. Next in case of $X, Y \in \Sigma$, we have that $X^{rev} = X$, $Y^{rev} = Y$. In particular we have that $Id^{rev} = Id$ and therefore $(Op^{rev}, c^{rev})$ is again a set of operations supplied with a cost function[1] $c^{rev}$.

With this notion we can also reverse alignments. However the reverse alignment would be not an alignment over $Op$ but an alignment over $Op^{rev}$. Here are the details:

**Definition 4.4.2** Let $(Op, c)$ be a set of operations with a cost function $c$. For an alignment $\omega = op_1 \circ op_2 \cdots \circ op_N$ in $Op^*$ we define $\omega^{rev}$ to be the alignment over $Op^{rev}$ given by:

$$\omega^{rev} = op_N^{rev} \circ op_{N-1}^{rev} \circ \cdots \circ op_1^{rev}.$$

For instance if $\omega = (\mathtt{l}, \mathtt{d})(\varepsilon, \mathtt{r})(\mathtt{e}, \mathtt{e})(\mathtt{a}, \mathtt{a})(\mathtt{d}, \mathtt{d})$, then:

$$\omega^{rev} = (\mathtt{d}, \mathtt{d})(\mathtt{a}, \mathtt{a})(\mathtt{e}, \mathtt{e})(\varepsilon, \mathtt{r})(\mathtt{l}, \mathtt{d})$$

and thus $\omega^{rev}$ aligns `dael` against `deard` exactly as our intuition suggested it should be.

It is straightforward to see that $(\omega^{rev})^{rev} = \omega$ since:

$$(\omega^{rev})^{rev} = \left(op_N^{rev} \circ op_{N-1}^{rev} \circ \cdots \circ op_1^{rev}\right)^{rev} = op_1 \circ op_2 \cdots \circ op_N = \omega.$$

Furthermore, it should be clear that the costs of alignments is preserved under reverse. Indeed:

$$c(\omega) = \sum_{i=1}^{N} c(op_i) = \sum_{i=1}^{N} c^{rev}(op_i^{rev}) = c^{rev}(\omega^{rev}).$$

It should be also clear that for any two alignments $\omega_1$ and $\omega_2$ over $(Op, c)$, the reverse alignment of their concatenation $\omega = \omega_1 \circ \omega_2$ is the same as the concatenation of the reversed alignments $\omega_2$ and $\omega_1$, where the order is interchanged:

$$(\omega_1 \circ \omega_2)^{rev} = \omega_2^{rev} \circ \omega_1^{rev}.$$

The properties of reversed alignments naturally transfer to set of alignments. Thus we have that $(Op^{rev})^{rev} = Op$ and $(\mathfrak{A}^{rev})^{rev} = \mathfrak{A}$ for any set of alignments $\mathfrak{A}$ over $(Op, c)$. Furthermore if $\mathfrak{A}$ and $\mathfrak{B}$ are sets of alignments, then $(\mathfrak{A} \circ \mathfrak{B})^{rev} = \mathfrak{B}^{rev} \circ \mathfrak{A}^{rev}$.

We can extend the reverse operation on distance-sets in a natural way:

**Definition 4.4.3** Let $L : \Sigma^* \to \mathbb{N}$ be an edit-distance list, then $L^{rev}$ is the distance list defined as:

$$L^{rev}(U) = L(U^{rev}).$$

[1] $c^{rev}(op^{rev}) = 0$ if and only if $op \in Id$, that is $op^{rev} \in Id$.

Now if an edit-distance list $\mathcal{L}$-represents an alignment set $\mathfrak{A}$, then $L^{rev}$ will be defined on the reversed domain of $L$, i.e.:

$$Dom(L^{rev}) = (Dom(L))^{rev} = \{l(\omega) \,|\, l(\omega) \in Inf(\mathcal{L}) \text{ and } \omega \in \mathfrak{A}\}^{rev}.$$

Therefore a word $U$ belongs to $Dom(L^{rev})$ if and only if $U = l^{rev}(\omega)$ belongs to $Inf(\mathcal{L}^{rev})$ and $\omega \in \mathfrak{A}$. But this is equivalent to:

$$U \in \{l(\omega) \,|\, l(\omega) \in Inf(\mathcal{L}^{rev}) \text{ and } \omega \in \mathfrak{A}^{rev}\}.$$

Using that the reverse operation on alignments preserves the cost of alignments, we obtain the following result:

**Lemma 4.4.4** *Let $L$ be an edit-distance list which $\mathcal{L}$-represents the alignment set $\mathfrak{A}$. Then $L^{rev}$ $\mathcal{L}^{rev}$-represents $\mathfrak{A}^{rev}$.*

Taking into account that applying the reverse operation twice we obtain the identity both for words and alignments, we deduce that:

**Corollary 4.4.5** *An edit-distance list $L$ $\mathcal{L}$-represents $\mathfrak{A}$ if and only if $L^{rev}$ $\mathcal{L}^{rev}$-represents $\mathfrak{A}^{rev}$.*

Finally, we shall use the reverse operation on alignment sets in order to derive a characterisation of the alignment sets $\mathfrak{A}(V, b \leftarrow V_1, b_1)$ from Section 4.2. The following property is obvious:

**Lemma 4.4.6** *If $V \in \Sigma^*$ is a word, $Op$ is a set of operations, then:*

$$\mathfrak{A}^{rev}(V) = \mathfrak{A}_{Op^{rev}}(V^{rev}).$$

*where the subscript $Op^{rev}$ indicates that the alignment sets are considered over the set of operations $Op^{rev}$ and not $Op$.*

*Proof.* The equality follows by the fact that every alignment $\omega \in \mathfrak{A}(V)$ has the property that $r(\omega^{rev}) = V^{rev}$ and $\omega^{rev}$ itself is an alignment over $Op^{rev}$.

□

Now we can prove the following analogue of Lemma 4.2.9.

**Lemma 4.4.7** *Let $Op$ be a set of operations with $\rho(Op) = 1$ and $V = V_0 \circ V_1$ be a word with lengths $|V_i| = n_i$ and let $b = b_0 + b_1$ be nonnegative rational numbers. Let $\overleftarrow{\mathfrak{A}}^j$ be the alignments:*

$$\overleftarrow{\mathfrak{A}}^j = \mathfrak{A}(I^{n_0}_{n_0-j+1}(V_0) \circ V_1, b \leftarrow V_1, b_1) = \left(\mathfrak{A}(I^{n_0}_{n_0-j+1}(V_0)) \circ \mathfrak{A}^{\leq b_1}(V_1)\right)^{\leq b}$$

*Then:*

$$\begin{aligned}
\overleftarrow{\mathfrak{A}}^0 &= (\Lambda^* \circ \mathfrak{A}^{\leq b_1}(V_1))^{\leq b} \\
\overleftarrow{\mathfrak{A}}^{j+1} &= \left(\Lambda^* \circ \left(\bigcup_{op=(U, I^{n_0-j}_{n_0-j}(V_0)) \in Op} op \circ \overleftarrow{\mathfrak{A}}^j\right)^{\leq b}\right)^{\leq b}
\end{aligned}$$

*for $j < n_0$.*

*Proof.* Let $W_0 = V_1^{rev}$ and $W_1 = V_0^{rev}$. Therefore position $l$ in $W_1$ is position $l$ in $V_0^{rev}$ and thus it holds the same character as the character at position $n_0 - l + 1$ in $V_0$. With this remark it is rather straightforward that $(I^{n_0}_{n_0-j+1}(V_0))^{rev} = I^j_1(W_1)$. Now using Lemma 4.4.6 we see that:

$$\begin{aligned} \mathfrak{A}^{rev}(V_1) &= \mathfrak{A}_{Op^{rev}}(W_0) \\ \overleftarrow{\mathfrak{A}}^{rev}(I^{n_0}_{n_0-j+1}(V_0)) &= \overrightarrow{\mathfrak{A}}_{Op^{rev}}(I^j_1(W_1)). \end{aligned}$$

This means that:

$$\mathfrak{A}^{rev}(I^n_{n_0-j+1}(V_0) \circ V_1, b \leftarrow V_1, b_1) = \mathfrak{A}_{Op^{rev}}(W_0, b_1 \rightarrow W_0 \circ I^j_1(W_1), b)$$

Now we apply Lemma 4.2.9 to the sets $\mathfrak{A}_{Op^{rev}}(W_0, b_1 \rightarrow W_0 \circ I^j_1(W_1), b)$, then we use the properties of the reverse operations in order to complete the proof. □

As a by-product of the above proof we obtain:

**Corollary 4.4.8** *In the notions of Lemma 4.4.7 it holds:*

$$\overleftarrow{\mathfrak{A}}^{j,rev} = \left( \left( \bigcup_{op=(U, I^j_j(V_0^{rev})) \in Op^{rev})} (\overleftarrow{\mathfrak{A}}^{j-1,rev} \circ op)^{\leq b} \right) \circ (\Lambda^{rev})^* \right)^{\leq b}.$$

That is essentially $\mathfrak{A}^{j,rev}$ obeys the same recurrence as $\mathfrak{A}^j$ from Lemma 4.2.9 with the only difference that the alignment sets are considered over $(Op^{rev}, c^{rev})$ and not over $(Op, c)$.

# Chapter 5

# Approximate Search in Regular Sets, $\rho(Op) = 1$

In this chapter we formally describe how to transform the idea we presented in Chapter 3 into an algorithm. In this chapter we still consider the case when $\rho(Op) = 1$ and thus we can rely on the theoretical preliminaries from Chapter 4. We start with a brief overview which sets the framework of our approach to solving the approximate search problem:

`Given:` $\mathcal{L} \subseteq \Sigma^*$ `regular language,`
$d = (Op, c)$ `an edit-distance,`
$q \in (0; 1)$ `a threshold parameter`
`Input:` $V \in \Sigma^*$
`Output:` $\{U \in \mathcal{L} \mid d(U, V) \leq q|V|\}$.

Although our solution is conceptually based on the results presented in Chapter 4 we need to take care of the technical details on which the efficient algorithm depends.

The main ideas in this chapter were described in [21] and in more details in [20].

## 5.1 Algorithm Overview

Given a query word $V \in \Sigma^*$ of length $N$, it specifies a query with threshold $qN$. The task is to determine all words $U \in \mathcal{L}$ such that $d(U, V) \leq qN$. In order to apply the idea from Chapter 3 we first slightly generalise the query. In particular we solve:

`Given:` $\mathcal{L} \subseteq \Sigma^*$ `regular language,`
$d = (Op, c)$ `an edit-distance,`
$q \in (0; 1)$ `a threshold parameter`
`Input:` $V \in \Sigma^*$

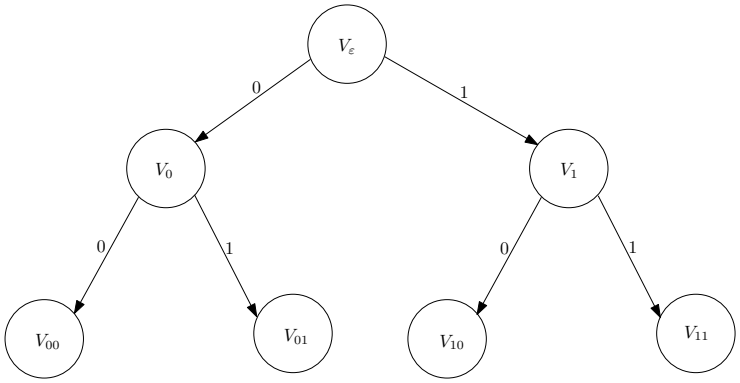



Figure 5.1: The structure of a searching tree in the case when $V$ is split into 4 subwords.

`Output:` $\{U \in Inf(\mathcal{L}) \,|\, d(U,V) \leq q|V|\}$.

The difference constitutes in the fact that we search for all the *infixes* of the language $\mathcal{L}$ which satisfy the query rather than entire words.

This generalisation allows us to split the initial query specified by $V$ into shorter which we represent as a *binary tree*, see Figure 5.1. The initial word $V$ is partitioned into a prefix $V_0$ and a suffix $V_1$ of almost equal lengths. Then we apply the same procedure to $V_0$ and $V_1$ recursively. The process terminates when we obtain a word $V_\alpha$ which is short enough, i.e. $q|V_\alpha| < 1$. In this way we obtain a binary tree, $\mathcal{T}(V)$, whose nodes, $\alpha \in \{0,1\}^*$, are labelled with queries $V_\alpha$.

Our next goal is to solve each of these queries:

`Given:`$\mathcal{L} \subseteq \Sigma^*$ `regular language,`
$d = (Op, c)$ `an edit-distance,`
$q \in (0;1)$ `a threshold parameter`
`Input:` $V_\alpha \in \Sigma^*$
`Output:` $\{U \in Inf(\mathcal{L}) \,|\, d(U,V_\alpha) \leq q|V_\alpha|\}$.

To this end we start with the queries assigned to the leaves of $\mathcal{T}(V)$. For each such query we have that $q|V_\alpha| < 1$ which suggests that only the exact match can be an answer to this query. Afterwards we extend the answers of the queries $V_{\alpha 0}$ and $V_{\alpha 1}$ in order to compute the answers for the query $V_\alpha$. In this way we are capable to propagate the answers from the leaves towards the root of the search tree $\mathcal{T}(V)$. The *extension steps* are based on Lemma 4.2.7 and the representation of alignment sets via edit-distance lists which provides the correctness of the approach. In order to make this technique applicable in practice we use the following resources:

1. edit-distance lists which $\mathcal{L}$-represent the alignment sets $(\mathfrak{A}(V_\alpha))^{\leq q|V_\alpha|}$.
2. an automata-based representations of the languages $Inf(\mathcal{L})$ and $Inf(\mathcal{L}^{rev})$.

Thus, on the one hand, the edit-distance lists which $\mathcal{L}$-represent the alignment sets $(\mathfrak{A}(V_\alpha))^{\leq q|V_\alpha|}$ describe exactly those infixes, $U$, with $U \in Inf(\mathcal{L})$ such that $U$ can be aligned with $V_\alpha$ at cost at most $q|V_\alpha|$ and hence $d(U, V_\alpha) \leq q|V_\alpha|$. Therefore the answers of the query will efficiently represented in the edit-distance lists. On the other hand, the deterministic automata for $Inf(\mathcal{L})$ and $Inf(\mathcal{L}^{rev})$ will provide an efficient procedure to extend each word from the domain of an edit-distance lists in arbitrary direction while controlling that the result remains a valid infix of $\mathcal{L}$ or $\mathcal{L}^{rev}$, respectively. Thus, in view of the constructions described in Chapter **??** we will be able to extend the solutions of the shorter queries in order to obtain results for the longer queries.

In case that the language $\mathcal{L}$ is finite, its infixes can be represented by the data-structures considered Chapter 2 and this will bring us to a more efficient algorithm. The result from the second step of our algorithm is that we have the answers for the query specified by $V = V_\varepsilon$:

`Given:` $\mathcal{L} \subseteq \Sigma^*$ `regular language,`
$d = (Op, c)$ `an edit-distance,`
$q \in (0; 1)$ `a threshold parameter`
`Input:` $V \in \Sigma^*$
`Output:` $\{U \in Inf(\mathcal{L}) \mid d(U, V) \leq q|V|\}$.

Since $\mathcal{L} \subseteq Inf(\mathcal{L})$ and $\mathcal{L}$ is regular we can easily filter these answers so that we obtain only those words $U \in \mathcal{L}$ which satisfy:

$$d(U, V) \leq q|V|.$$

This final step is effectuated by the means of an automaton for the language $\mathcal{L}$.

## 5.2 Initialisation Step

In this section we formally describe how we organise the searching process and how we initiate it when $\rho = \rho(Op) = 1$.

Recall that $\rho = 1$ means that every right hand side of an operation is either the empty word or a single-character-word. We start by defining the set of queries we are going to solve. To this end we first split the query word $V$ into subwords $V_\alpha$.

Given a word $V$, we decompose it recursively into shorter subwords $V_\alpha$ with $\alpha \in \{0, 1\}^*$ and we define a binary tree $\mathcal{T}(V)$ with nodes $\alpha$ as follows (see Figure 5.1):

1. $V_\varepsilon = V$ and $\varepsilon$ is the root of $\mathcal{T}(V)$.
2. if $V_\alpha$ and $\alpha$ are defined and $q|V_\alpha| \geq 1$ we define $V_{\alpha 0}$ and $V_{\alpha 1}$ such that:

$$V_\alpha = V_{\alpha 0} \circ V_{\alpha 1}$$
$$0 \leq |V_{\alpha 0}| - |V_{\alpha 1}| \leq 1.$$

We set $\alpha 0$ to be the left child of $\alpha$ and $\alpha 1$ to be the right child of $\alpha$.

For every node $\alpha$ we define the length, $N_\alpha = |V_\alpha|$, of the word $V_\alpha$ and the threshold $b_\alpha = qN_\alpha$. As we explained above we associate with every node $\alpha$ the query:

`Given:` $\mathcal{L} \subseteq \Sigma^*$ `regular language,`
$d = (Op, c)$ `an edit-distance,`
$q \in (0; 1)$ `a threshold parameter`
`Input:` $V_\alpha \in \Sigma^*$
`Output:` $\{U \in Inf(\mathcal{L}) \mid d(U, V_\alpha) \leq q|V_\alpha|\}$.

and in the node $\alpha$ we compute the graph of the edit-distance list $L(\alpha)$ which $\mathcal{L}$-represents the alignment set $\mathfrak{A}(V_\alpha)^{\leq b_\alpha}$.

**Lemma 5.2.1** *Given a leaf, $\alpha$, of the search tree $\mathcal{T}(V)$, we have that:*

$$L(\alpha) = \begin{cases} \{\langle U, 0\rangle\} \text{ if } U = V_\alpha \text{ and } V_\alpha \in Inf(\mathcal{L}) \\ \emptyset \text{ otherwise.} \end{cases}$$

*If we further dispose on a deterministic automaton $\mathcal{A}$ which recognises $Inf(\mathcal{L})$ we can compute the (graphs of the) edit-distance lists $L(\alpha)$ for all the leaves $\alpha$ in time $O(N)$.*

*Proof.* Indeed for every leaf $\alpha$ we have that $b_\alpha = qN_\alpha < 1$. Now since each nonidentity operation is of positive cost, we conclude that:

$$\mathfrak{A}(V_\alpha)^{\leq b_\alpha} = \{\omega_\alpha\}$$

where $\omega_\alpha \in Id^*$ is uniquely determined by the condition $r(\omega_\alpha) = V_\alpha$. It is then clear that $c(\omega_\alpha) = 0$ and $l(\omega_\alpha) = r(\omega_\alpha) = V_\alpha$. Hence, according to the definition of $\mathcal{L}$-representability we deduce:

$$L(\alpha) = \begin{cases} \{\langle U, 0\rangle\} \text{ if } U = V_\alpha \text{ and } V_\alpha \in Inf(\mathcal{L}) \\ \emptyset \text{ otherwise.} \end{cases}$$

This proves the first part of the lemma. The second part follows immediately by the first. To see this, consider the subwords, $V_\alpha$, of $V$ where $\alpha$ ranges over all the leaves of $\mathcal{T}(V)$. Using the automaton $\mathcal{A}$ we can check in time $O(N_\alpha)$ whether $V_\alpha \in Inf(\mathcal{L})$. Now the result follows by the fact that:

$$\sum_{\alpha \text{ leaf of } \mathcal{T}(V)} N_\alpha = N.$$

Hence the total time spent in the traversal of the automaton $\mathcal{A}$ is $O(N)$. □

`ApproximateSearchInitialise`$(\mathcal{IA}_r, V, N, q, \alpha)$
$N_\alpha \leftarrow N$
$V_\alpha \leftarrow V$

```
if qN < 1 then
   T(α) ←empty trie with root r
   s ←the initial state of IA_r
   st ← TraverseAutomaton(IA_r, s, V)
   L(α) ← ∅
   if st is defined then
      u ← TraverseTrie(T, r, V)
      st(u) ← st
      B[N] ←new empty bucket
      B[N].Insert(⟨u, 0⟩
      L(α).Append(B[N])
   fi
else
   N_left ← ⌈N/2⌉
   N_right ← ⌊N/2⌋
   V_left ← V[1..N_left]
   V_right ← V[N_left + 1..N]
   ApproximateSearchInitialise(IA, V_left, N_left, q, α0)
   ApproximateSearchInitialise(IA, V_right, N_right, q, α1)
```

## 5.3 Extension Steps

This section describes how the answers of the queries at the nodes $\alpha 0$ and $\alpha 1$ can be extended as to obtain the answers for the query specified by the node $\alpha$. Here we present two possible approaches. The first one is rather general and involves a standard still tedious indexing of the edit-distance lists. The second one is based on a tricky but natural order of the (graph of) the edit-distance lists which however is applicable only for finite languages $\mathcal{L}$.

### 5.3.1 Edit-Distance Lists Represented as Tries

Our goal is to find the edit-distance lists $L(\alpha)$ that $\mathcal{L}$-represent the alignment set $\mathfrak{A}^{\leq b_\alpha}(V_\alpha)$ for every node $\alpha$ of the query tree $\mathcal{T}(V)$. To this end we shall proceed in a bottom-up fashion and compute $L(\alpha)$ on the basis of $L(\alpha 0)$ and $L(\alpha 1)$. The idea is simply to use Lemma 4.2.7 from Chapter 4 for $V_\alpha = V_{\alpha 0} \circ V_{\alpha 1}$ and $b_\alpha = b_{\alpha 0} + b_{\alpha 1}$:

$$\mathfrak{A}^{\leq b_\alpha}(V_\alpha) \quad = \quad \mathfrak{A}(V_{\alpha 0}, b_{\alpha 0} \rightarrow V_\alpha, b_\alpha) \cup \mathfrak{A}(V_\alpha, b_\alpha \leftarrow V_{\alpha 1}, b_{\alpha 1}).$$

The framework of alignment sets and edit-distance lists representing alignment sets that we developed in Chapter 4 allow us to compute the desired result iteratively. In particular we are going to compute the representations of the sets $\mathfrak{A}(V_{\alpha 0}, b_{\alpha 0} \rightarrow V_\alpha, b_\alpha)$ and $\mathfrak{A}(V_\alpha, b_\alpha \leftarrow V_{\alpha 1}, b_{\alpha 1})$ in steps and afterwards we shall unite them.

To get the idea, let us consider the computation of the edit-distance lists representing $\mathfrak{A}(V_{\alpha 0}, b_{\alpha 0} \rightarrow V_\alpha, b_\alpha)$ for fixed $\alpha$. Since we have the edit-distance

list $L(\alpha 0)$ we know that we have appropriate representations of the alignment set:

$$\mathfrak{A}^{\leq b_{\alpha 0}}(V_{\alpha 0}).$$

We are going to compute the edit-distance lists $L_r(\alpha, j)$ that $\mathcal{L}$-represent the alignment sets $\overrightarrow{\mathfrak{A}}^j$ defined as:

$$\overrightarrow{\mathfrak{A}}^j = \mathfrak{A}(V_{\alpha 0}, b_{\alpha 0} \rightarrow V_{\alpha 0} \circ I_1^j(V_{\alpha 1}), b_\alpha)$$

Based on Lemma 4.2.9 we have that $\overrightarrow{\mathfrak{A}}^j$ can be expressed as:

$$\overrightarrow{\mathfrak{A}}^0 = (\mathfrak{A}^{\leq b_{\alpha 0}}(V_{\alpha 0}) \circ \Lambda^*)^{\leq b_\alpha}$$

$$\overrightarrow{\mathfrak{A}}^j = \left( \left( \bigcup_{(U, I_j^j(V_{\alpha 1})) \in Op} (\overrightarrow{\mathfrak{A}}^{j-1} \circ op)^{\leq b_\alpha} \right) \circ \Lambda^* \right)^{\leq b_\alpha}$$

for $j \geq 1$. In the latter case we proceed in two steps. In the first step given the edit-distance lists $L_r(\alpha, j-1)$ an $\mathcal{L}$-representation for the alignment set $\overrightarrow{\mathfrak{A}}^{j-1}$ for $j$ we first determine the edit-distance list $L'_r(\alpha, j)$ that $\mathcal{L}$-represents the alignment set:

$$\widetilde{\mathfrak{A}}^j = \bigcup_{(U, I_j^j(V_{\alpha 1})) \in Op} (\overrightarrow{\mathfrak{A}}^{j-1} \circ op)^{\leq b_\alpha}$$

In a second step we compute the list $L_r(\alpha, j)$ that $\mathcal{L}$-represents the alignment set:

$$\overrightarrow{\mathfrak{A}}^j = \left( \widetilde{\mathfrak{A}}^j \circ \Lambda^* \right)^{\leq b_\alpha}.$$

We refer to the first step as *extension step* and to the second step as *$\varepsilon$-closure step*. In this terms the computation of $\overrightarrow{\mathfrak{A}}^0$ is a particular case. It requires only a single $\varepsilon$-closure step applied on the $\mathfrak{A}^{\leq b_{\alpha 0}}(V_{\alpha 0})$.

The extension step is essentially based on Lemma 4.3.3 and Lemma 4.3.6. The *$\varepsilon$-closure step* additionally exploits Lemma 4.2.4 which allows to consider the $\varepsilon$-closure step as union of concatenations. In order to apply Lemma 4.3.3 we need a supervisory mechanism that filters the words that are not infixes of the regular language $Inf(\mathcal{L})$. To this end we use a deterministic finite state automaton, $\mathcal{IA}_r$ for the language $Inf(\mathcal{L})$.

Symmetrically, in order to find appropriate edit-distance lists representation for $\mathfrak{A}(V_\alpha, b_\alpha \leftarrow V_{\alpha 1}, b_{\alpha 1})$ we apply Lemma 4.4.7. Thus we need to compute edit-distance lists that $\mathcal{L}$-represent the alignment sets:

$$\overleftarrow{\mathfrak{A}}^j = \mathfrak{A}(I_{N_{\alpha 0 - j + 1}}^{N_{\alpha 0}}(V_{\alpha 0}) \circ V_{\alpha 1}, b_\alpha \leftarrow V_{\alpha 1}, b_{\alpha 1})$$

for $j$ varying from 0 to $N_{\alpha 0}$. However, according to Corollary 4.4.8 this problem reduces to the problem $\overrightarrow{\mathfrak{A}}^j$ by reversing both the alignment sets and the operation set. In particular we have that:

$$\overleftarrow{\mathfrak{A}}^{j,rev} = \mathfrak{A}^{rev}(V_{\alpha 1}^{rev}, b_{\alpha 1} \rightarrow V_{\alpha 1}^{rev} \circ I_1^j(V_{\alpha 0}^{rev}), b_\alpha)$$

On the other hand disposing on an edit-distance list $L_l^{rev}(\alpha, j)$ that $\mathcal{L}^{rev}$-represents the alignment set $\overleftarrow{\mathfrak{A}}^{j,rev}$ we can easily compute the edit-distance list $L_l(\alpha, j)$ that $\mathcal{L}$-represents the alignment set $\overleftarrow{\mathfrak{A}}^{j}$ due to Lemma 4.4.4. We do not need to do this for every single $j$. Since we are interested only in the final outcome $L_l(\alpha, N_{\alpha 0})$, we can apply the reverse operation once only for $L_l^{rev}(\alpha, N_{\alpha 0})$. In the sequel we shall develop a procedure that computes the edit-distance lists $L_r(\alpha, j)$ with respect to the edit-distance $(Op, c)$ and relies on an infix automaton $\mathcal{IA}_r$ with the language $Inf(\mathcal{L})$. Thus, in order to to compute the edit-distance lists $L_l^{rev}(\alpha, j)$ it will be enough to substitute $V_{\alpha 1}$ with $V_{\alpha 0}^{rev}$, $(Op, c)$ with $(Op^{rev}, c^{rev})$ and $\mathcal{IA}_r$ with an automaton $\mathcal{IA}_l$ for the language $Inf(\mathcal{L}^{rev})$.

In what follows we address these issues for the extensions and $\varepsilon$-closures. Finally we shall comment on the union of the edit-distance lists $L_r(\alpha 0, j)$ and $L_l(\alpha 1, j)$ so that we obtain the actual list of interest $L(\alpha)$ and its intact representation.

*Representation of Edit-Distance Lists.*

As we already mentioned the extension and $\varepsilon$-closure steps are supervised by a deterministic infix automaton $\mathcal{AI}_r$ with language the set of infixes of the given language $\mathcal{L}$, i.e. $\mathcal{L}(\mathcal{AI}_r) = Inf(\mathcal{L})$. Additionally, the domains $Dom(L_r(\alpha, j))$ will be stored by the means of a trie $\mathcal{T}$. Whereas many edit-distance lists will be computed during this step, a unique trie will be maintained and it will be extended appropriately. Each edit-distance list will be responsible to keep track of its domain within the trie $\mathcal{T}$.

With every node $u$ of the trie $\mathcal{T}$ we associate a label $label(u)$ which is the word spelled by the path form the root of $\mathcal{T}$ to the node $u$. In each node $u \in \mathcal{T}$ we store the following piece of information:

1. a state, $st(u)$, in the infix automaton $\mathcal{AI}_r$. The state $st(u)$ is the state reached from the initial state of $\mathcal{AI}_r$ with the label $label(u)$.

2. an integer $cost(u)$.

3. an active flag $act(u)$ which can be true or false.

Now the elements of the distance list $L_r(\alpha, j)$ are represented as a list of buckets $B_\alpha^j[0], B_\alpha^j[1], \ldots$, see Figure 5.2. In this example we have the representation of the edit-distance list for the subquery $(\texttt{abb}, 1)$ in the regular language $\{\texttt{ababbb}, \texttt{acbbb}\}^*$ with respect to the Levenshtein edit-distance. All the candidates are stored in a trie where the common prefixes are shared between the distinct words. Each bucket stores an appropriate representation of the candidates of corresponding length. For instance, the bucket $B[3]$ stores the representation of the candidates of length3. On our example, these are $\langle \texttt{abb}, 0\rangle$ and $\langle \texttt{aba}, 1\rangle$. This information about a single candidate is encoded in a pointer to an appropriate node in the trie and an integer value, see Figure 5.2. In the nodes of the trie we store a reference to the corresponding state in the infix automaton, $\mathcal{AI}_r$. Thus, for instance the state with label $S_6$ in the trie, encodes that with the word `aba` we reach from the initial state $S_0$ to the state $S_6$ in the

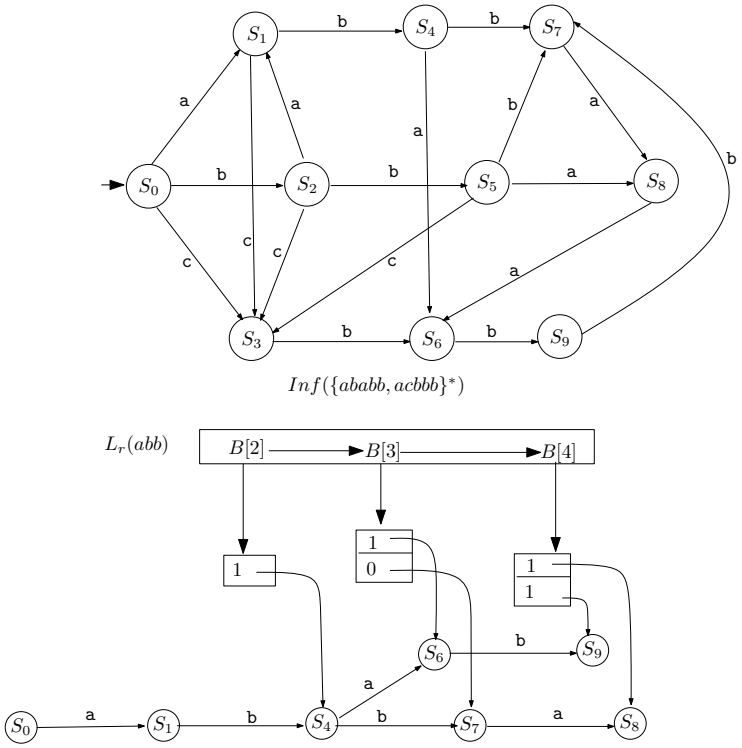



Figure 5.2: The representation of the edit-distance lists $L_r(\mathtt{abb}, 1)$ that $\mathcal{L}$-represents $(\mathfrak{A}^{\leq 0}(\mathtt{ab}) \circ \mathfrak{A}(b)))^{\leq 1}$.

infix automaton for $\{\mathtt{ababbb}, \mathtt{acbbb}\}^*$. Thus, each state of the trie is assigned to a unique state in the infix automaton. However the converse should not hold.

The representation for the edit-distance lists $L_l(\alpha, j)$ is similar.However the representation is reversed, i.e. it goes from right to left rather then to the natural left to right reading order. For our example, we use a deterministic infix automaton for the language $(\{\mathtt{ababbb}, \mathtt{acbbb}\}^*)^{rev}$ and the trie spells the candidates from $Dom(L_l(\mathtt{abb}, 2))$ in reverse fashion. Thus, for example, the exact match `abb` is spelled like `bba`. This is the only significant feature that distinguishes the representation of $L_l(\alpha, j)$ from the one for $L_r(\alpha, j)$.

In general the bucket $B^j_\alpha[n]$ stores (representations of) the elements[1] $\langle U, c_U \rangle \in L_r(\alpha, j)$ such that $|U| = n$. Hence the bucket with number $n$ is responsible uniquely for the $n$-th slice of the edit-distance list. Upon the time the edit-distance list $L_r(\alpha, j)$ will have been constructed no empty buckets will be present in its representation. However some empty buckets may arise during the construction. The list of buckets is ordered increasingly with $n$.

[1]For the better readability of the outline we use the same notation for the edit-distance lists $L_r(\alpha, j)$ and their graphs that we actually represent.

Every bucket $B^j_\alpha[n]$ is a list of pairs $\langle u, c(u)\rangle$ where:

1. $u$ is a node in the trie $\mathcal{T}$, such that $label(u) \in Dom(L_r(\alpha, j))$.
2. $c(u)$ is an integer.

During the construction of the bucket $B^j_\alpha$ some of the values $c(u)$ might be undefined but upon termination they will satisfy that:

$$\langle label(u), c(u)\rangle \in L_r(\alpha, j).$$

Finally we keep a list *Act* of distinct active nodes. At the beginning of each step $j$, *Act* is being emptied and is filled in during the step.

*Extension. Construction of the lists* $L'_r(\alpha, j)$.

We first describe how a representation of $\widetilde{\mathfrak{A}}^j$ is computed on the base of previously computed edit-distance lists. We call the resulting edit-distance list $L'_r(\alpha, j)$.

*Initialisation of the buckets.* For a positive integer $j > 0$, we consider the already constructed edit-distance list $L_r(\alpha, j-1)$. Let $\mu = \max\{|l(op)| \,|\, op \in Op\}$ be the maximal length of a left side of an operation. At the initialisation step we take care to create empty buckets that will represent the edit-distance list $L'_r(\alpha, j)$. Since the alignment $\omega'$ that are represented by $L'_r(\alpha, j)$ are a result from the concatenation of an alignment $\omega$ represented in $L_r(\alpha, j-1)$ and a single operation $op \in Op$, we know that $|l(\omega')| = |l(\omega)| + |l(op)|$. Hence given the list of nonempty buckets representing $L_r(\alpha, j-1)$, $B^{j-1}_\alpha[n_1], B^{j-1}_\alpha[n_2], \ldots, B^{j-1}_\alpha[n_{M_{j-1}}]$, we know that elements of $L'_r(\alpha, j)$ will be assigned only to buckets $B^j_\alpha[n]$ such that $n = n_k + l$ where $l \leq \mu$. Thus in the first step for each $l \leq \mu$ we determine the sets:

$$S(j-1, l) = \{n + l \,|\, B^{j-1}_\alpha[n] \text{ is a bucket}\}$$

Since the buckets $B^{j-1}_\alpha[n]$ are ordered increasingly within a list, each of the sets $S(j-1, l)$ can be easily obtained in increasing order in time proportional to the number of buckets of $L_r(\alpha, j-1)$. Next we can sort the sets $S(j-1, op)$ in a single increasingly sorted set $S(j-1) = \{s_1 < s_2 < \cdots < s_{|S|}\}$ and we create a list of empty buckets:

$$B^j_\alpha[s_1], B^j_\alpha[s_2] \ldots B^j_\alpha[s_{|S|}]$$

Finally, for every $n$ and every operation $l \leq \mu$ the bucket $B^{j-1}_\alpha[n_k]$, is assigned an $l$-pointer to the bucket $B^j_\alpha[n_k + l]$.

```
InitialiseBuckets(⟨Op, c⟩, η, j, L)
//L is an array of lists, e.g. L_r(α, k, .) or L_l(α, k, .)
   μ ← max{|l(op)| | op ∈ Op}
   for l = 0 to μ do
      S[l] ← ∅
      for B[n] ∈ L[j−1] in (increasing) order do// B[n] = B_α^{j−sign}[n]
         S[l].Insert(⟨n + l, n⟩)
      done
```

```
    done
    SOrd ← MergeSort(S, μ)
    last ← ⊥
    while SOrd ≠ ∅ do
       ⟨m, n⟩ ← SOrd.RemoveFirst()
       if m ≠ last then
          Bnew[m] ← new empty block
          L[j].Append(Bnew[m])
          last ← m
       fi
       l ← m − n
       B[n].pointer[l] ← Bnew[m]
    done


MergeSort(S, μ)
    H ← ∅ //empty heap of pairs ordered w.r.t. the first component
    SOrd ← ∅
    for l = 0 to μ do
       ⟨m, n⟩ ← S[l].First()
       if ⟨m, n⟩ is defined then
          H.InsertToHeap(⟨m, n⟩)
       fi
    done
    while H ≠ ∅ do
       ⟨m, n⟩ ← H.ExtractMin()
       SOrd.Append(⟨m, n⟩)
       l ← m − n
       ⟨m_next, n_next⟩ ← S[l].Next(⟨m, n⟩)
       if ⟨m_next, n_next⟩ is defined then
          H.InsertToHeap(⟨m_next, n_next⟩)
       fi
    done
    return SOrd
```

**Lemma 5.3.1** *Let $M_{j-1}$ be the number of buckets in $L_r(\alpha, j-1)$, then the creation of the buckets $B^j_\alpha[n]$ can be carried out in time $O\left(|M_{j-1}|(\mu+1)\log(\mu+1)\right)$. Furthermore, each bucket $B^{j-1}_\alpha[n_m]$ is assigned with an $l$-pointer to a valid bucket $B^j_\alpha[n_m + l]$.*

*Proof.* Clearly, we can determine each of the the sets $S(j-1, l)$ in time $O(M_{j-1})$ each. Therefore each of them contains at most $|M_{j-1}|$ elements. Since the buckets are ordered, each of the sets $S(j-1, l)$ is increasingly ordered. Hence, we can obtain the sorted set $S(j-1)$ using a merge sort of $\mu+1$ ordered sets at the total expense of $O\left(M_{j-1}(\mu+1)\log(\mu+1)\right)$ time. It is also straightforward to set the pointers during the merge sort. □

*Filling the buckets.* In the second step we consider the list $L_r(\alpha, j-1)$ and we look at the operations $op \in Op$ meeting the following conditions, see also procedure FillBuckets:

1. $|r(op)| = 1$.
2. $r(op) = I_j^j(V_{\alpha 1})$, i.e. the $j$-th character of $V_{\alpha 1}$.

For every such operation, $op$, we pass through the buckets $B_\alpha^{j-1}[n_k]$ and for each element $\langle u', c(u')\rangle \in B_\alpha^{j-1}[n_k]$ we proceed in the following way:

1. if $c(u') + c(op) > b_\alpha$ we continue with the next element in the bucket or in case that this was the last element of the bucket with the next bucket.
2. we follow the transitions in $\mathcal{AI}_r$ starting at $st(u')$ and reading $l(op)$. If we fail at some step we proceed with the next pair, else we define the last state of the chain of transitions to be $st$.
3. we traverse the trie $\mathcal{T}$ from $u'$ with label $l(op)$ and create each nonexisting state on the way. Let $u$ be the last state on this sequence.
4. if $u$ is a new node in the trie or $act(u)$ is false, then:
   (a) we set $act(u) = true$ and $st(u) = st$.
   (b) we insert $u$ to the list of active nodes $Act$ and we insert a pair $\langle u, -1\rangle$ to the bucket $B_\alpha^j[n_k + |l(op)|]$ (this is done by using the $|l(op)|$-pointer of the bucket $B_\alpha^{j-1}[n_k]$ we are currently investigating.)
   (c) we set $cost(u) = c(u') + c(op)$.
5. if $cost(u) > c(u') + c(op)$ then we reset $cost(u) = c(u') + c(op)$.

Finally, for each bucket $B_\alpha^j[n]$ and each element $\langle u, c\rangle \in B_\alpha^j[n]$ we set $c = cost(u)$.

```
FillBuckets(A, T, ⟨Op, c⟩, q, V, Act, α, η, L, j)
//L is an array of lists, e.g. L_r(α,.) or L_l(α,.)
   b_α ← q|V_α|
   pos ← if η = 1 then j else |V_α0| − j + 1
   for op ∈ Op do
      if r(op) = V_αη[pos] then
         for B[n] ∈ L[j − 1] do
            for ⟨u', c'⟩ ∈ B[n] do
               if c' + c(op) ≤ b_α then
                  st ← TraverseAutomaton(A, st(u'), l(op))
                     if st is defined then
                        u ← TraverseTrie(T, u', l(op))
                        if act(u) = false then
                           act(u) ← true
                           st(u) ← st
```

```
                                    cost(u) ← c' + c(op)
                                    Act.Insert(u)
                                    B[n].pointer[|l(op)|].Insert(⟨u, −1⟩)
//B[n].pointer[|l(op)|] is actually the bucket B[n + |l(op)|] of L[j])
                                fi
                                if cost(u) > c' + c(op) then
                                    cost(u) ← c' + c(op)
                                fi
                            fi
                        fi
                    fi
                done
            done
        fi
    done
    for B[n] ∈ L[j] do
        for ⟨u, c⟩ ∈ B[n] do
            c ← cost(u)
        done
    done
```

```
TraverseAutomaton(A, st, W)
/*δ_A is the transition function of a deterministic automaton A
st is a state of A and W is a word with characters W[i]
starting from i = 1. The result is δ*_A(st, W).*/
    for i = 1 to |W| do
        if δ_A(st, W[i]) is not defined
            return not defined
        fi
        st ← δ_A(st, W[i])
    done
    return st
```

```
TraverseTrie(T, st, W)
/* δ_T is the transition function of a trie T
st is a state of T and W is a word with characters W[i]
starting from i = 1. The trie is modified so that all the states
along  δ*_T(st, W) are defined and the result  is δ*_T(st, W).*/
    for i = 1 to |W| do
        if δ_T(st, W[i]) is not defined then
            st' ← new trie state
            act(st') = false; cost(st') = −1
            δ_T(st, W[i]) ← st'
        fi
        st ← δ_A(st, W[i])
    done
    return st
```

**Lemma 5.3.2** *Let $Size(j-1)$ be the size of the list $L_r(\alpha, j-1)$, i.e. the number of pairs in all of its buckets. Then the step of filling the buckets $B^j_\alpha$ requires $O(Size(j-1))$ time. Furthermore upon termination of this step the following two properties hold:*

1. *$u \in Act$ if and only if $act(u) = true$ if and only if $\langle u, c\rangle \in B^j_\alpha[n]$ with $n = |label(u)|$ and some $c$.*
2. *$L'_r(\alpha, j)$ $\mathcal{L}$-represents $\widetilde{A}^j$.*

*Proof.* The time bounds for this step should be clear. Indeed we need $O(||Op||)$ time[2] in order to determine which operations *op* meet the condition $|r(op)| = 1$ and $r(op) = I^j_j(V_{\alpha 1})$. Next each element of $B^{j-1}_\alpha[n_k]$, $\langle u', c(u')\rangle$, is considered once only and we do some work on it. First we traverse one the automaton $\mathcal{AI}_r$ starting from $st(u')$ with the label $l(op)$. Since we have an immediate access to $st(u')$, this step requires $O(|l(op)|)$ steps for the traversal. Next we may need to traverse and possibly update the trie $\mathcal{T}$, again at the cost $O(|l(op)|)$. And finally we have to do some work for the state $u$. Again each of these actions is an atomic one, i.e. insertion of an element in a list or resetting an integer value. The only subtle point is that we have an immediate access to the list $B^j_\alpha[n_k + |l(op)|$ which is globally provided by the list $B^{j-1}_\alpha[n_k]$. Thus we spend for each pair $\langle u', c(u')\rangle$ constant amount of time. This results in the claimed bound by summing over all the elements of $L_r(\alpha, j-1)$.

Next, consider the situation when $u$ becomes active. This happens at the same time when $u$ is inserted to the list *Act* and to some bucket $B^j_\alpha[n]$. Furthermore if the activity of $u$ was invoked by some pair $\langle u', c(u')\rangle \in B^{j-1}_\alpha[n_k]$ and some operation *op*, then $n = n_k + |l(op)|$, since $|label(u')| = n_k$. Using that $u$ results from $u'$ by traversal of $l(op)$ in the trie $\mathcal{T}$, we deduce that $|label(u)| = |label(u')| + |l(op)| = n_k + |l(op)| = n$ as claimed.

Finally, using Lemma 4.2.6 and Lemma 4.3.3 and Lemma 4.3.6 from Chapter 4 we can argue that the edit distance list $L'_r(\alpha, j)$ has the property to $\mathcal{L}$-represent the alignment set $\widetilde{\mathfrak{A}}^j$. This follows by the assumption that $L_r(\alpha, j-1)$ $\mathcal{L}$-represent the alignment set $\overrightarrow{\mathfrak{A}}^{j-1}$ and the recurrence:

$$\widetilde{\mathfrak{A}}^j = \bigcup_{(U, I^j_j(V_{\alpha 1}))\in Op} (\overrightarrow{\mathfrak{A}}^{j-1} \circ op)^{\leq b_\alpha}.$$

We remark that Step 5 guarantees that $cost(u)$ is minimal over all possible ways the node $u$ can be reached from some $u'$ w.r.t. the procedure. Therefore, upon termination the list *Act* and the edit-distance list $L'_r(\alpha, j)$ have the desired properties.

□

*$\varepsilon$-closure step.* Once we have a representation of $L'_r(\alpha, j)$ we need to obtain a representation of $L_r(\alpha, j)$ by considering the cases where the last operation in

[2]This can be done in a somewhat more efficient way by using some tricks described by Reffle, [52], but this is not of much significance for the analysis.

a successful alignment might be $op \in \Lambda$. We organise this process as prescribed by Lemma 4.2.4.

We consider the buckets $B^j_\alpha[n]$ in increasing order of $n$. During this step some buckets may be deleted and new ones might be created but their order within the list will always respect the order of their indices $n$. We proceed as follows, see also procedure EpsilonClosure:

1. if the list $B^j_\alpha[n]$ is empty, delete it and proceed with the next unconsidered bucket.
2. for every pair $\langle u, c\rangle \in B^j_\alpha[n]$ do the following:
   (a) set $c = cost(u)$.
   (b) for every operation $op \in \Lambda$ such that $cost(u) + c(op) \leq b_\alpha$ do:
      i. traverse $\mathcal{AT}_r$ starting from $st(u)$ with label $l(op)$. If all the states along the chain are defined, let $st$ be the last reached state, otherwise continue with the next operation.
      ii. traverse the trie $\mathcal{T}$ from $u$ with label $l(op)$. Create all nonexisting states on the way and let $u_1$ be the final one.
      iii. if $act(u_1) = false$ then:
         A. set $act(u_1) = true$, $st(u_1) = st$.
         B. insert $u_1$ to the list $Act$.
         C. go along the list of buckets starting from the current $B^j_\alpha[n]$ and find a bucket $B^j_\alpha[n + l(op)]$. If such a list does not exist, then create it appropriately in the list of buckets. Add $\langle u_1, -1\rangle$ to $B^j_\alpha[n + l(op)]$.
         D. set $cost(u_1) = cost(u) + c(op)$.
      iv. if $cost(u_1) > cost(u) + c(op)$, then reset $cost(u_1) = cost(u) + c(op)$.

```
EpsilonClosure(𝒜, 𝒯, ⟨Op, c⟩, q, V, Act, α, L, j)
  b_α ← q|V_α|
  for B[n] ∈ L[j] in increasing order of n do
    if B[n] = ∅ then
      L[j].Remove(B[n])
    else
      for ⟨u', c'⟩ ∈ B[n] do
        ⟨u', c'⟩ ← ⟨u', cost(u')⟩
        for op ∈ Op with r(op) = ε do // op ∈ Λ
          if c(op) + c' ≤ b_α then
            st ← TraverseAutomaton(𝒜, st(u'), l(op))
              if st is defined then
                u ← TraverseTrie(𝒯, u', l(op))
                if act(u) = false then
                  act(u) ← true
```

```
                                Act.Insert(u)
                                st(u) ← st
                                cost(u) ← c′ + c(op)
                                m ← n //a block counter along the list
                                do
                                   B[m′] ← L[j].Next(B[m])
                                   if B[m′] is not defined then
                                      m′ ← ∞
                                   fi
                                   if m′ ≤ n + |l(op)|
                                      m ← m′
                                   fi
                                while m′ < n + |l(op)|
                                if m < n + |l(op)| then
                                   B[n + |l(op)|] ← new empty block
                                   L[j].InsertAfter(B[m], B[n + |l(op)|])
                                   B ← B[n + |l(op)|]
                                else
                                    B ← B[m]
//In either case B is assigned to B^j_α[n + |l(op)|]
                                   B.Insert(⟨u, −1⟩)
                             fi
                             if cost(u) > c′ + c(op) then
                                cost(u) ← c′ + c(op)
                             fi
                          fi
                       fi
                    done
                 done
           done
```

**Lemma 5.3.3** *Let $Size(j)$ be the size of $L_r(\alpha, j)$ upon termination of the algorithm and $Z$ be the number of empty buckets at the end of the filling of buckets stage, then the above procedure can be executed in time $O(Size(j) + Z)$. Furthermore it correctly computes $L_r(\alpha, j)$.*

*Proof.* The correctness of the algorithm follows by Lemma 4.2.4 and the correctness of the construction of $L'_r(\alpha, j)$ as described in the previous lemma. Then indeed the algorithm applies Lemma 4.2.4 to obtain the correct slices $B^j_\alpha[n]$ of the edit distance list $L_r(\alpha, j)$. It also accounts for obtaining only valid infixes $label(u) \in Inf(\mathcal{L})$ which is effectuated by the traversal of the automaton $\mathcal{AI}_r$. The only difference between the proposed algorithm and the straightforward realisation of Lemma 4.2.4 is the look-ahead technique which the algorithm utilises. Thus, instead of computing the bucket $B^j_\alpha[n]$ on the basis of the previous results $B^j_\alpha[n']$ with $n' < n$, it adds pieces of information from earlier stages $B^j_\alpha[n']$ to the later stages $B^j_\alpha[n]$. At Step iv we update the minimum value $cost(u_1)$ while keeping a witness that it is attainable.

The time bounds can be argued in the following way. During this stage of the algorithm, if a new bucket is created this happens in Step C and immediately afterwards it is filled with an element. Thus no empty buckets occur. This shows that at most $Z$ buckets will be deleted in this stage of the algorithm. Next each element $\langle u, c(u)\rangle$ is considered once only because it can reside only in $B^j_\alpha[n]$ with $n = |label(u)|$. However the buckets are considered in order and an already considered bucket is never reconsidered again. In order to complete the analysis let us consider the work done per element $\langle u, c(u)\rangle \in B^j_\alpha[n]$. First we set its eventual value $c(u) = cost(u)$ which requires constant time per element. Next we consider at most $|\Lambda|$ operations $op \in \Lambda$. For each of them we have at most two traversals with $l(op)|$ – one in the infix automaton and possibly a second one in the trie, $\mathcal{T}$. Each of them is executed at the cost of $O(|l(op)|)$ and hence can be considered for constant.

If a node $u_1 \in \mathcal{T}$ has to be considered as a result of a particular operation $op$, then the amount of work it would require is clearly $O(1)$ save for the Step C where we need to search for a specific bucket $B^j_\alpha[n+|l(op)|]$. However, this bucket lies in at most $|l(op)|$ elements apart from the current bucket, $B^j_\alpha[n]$. Therefore, we can follow the list of buckets, starting from the current one in $O(|l(op)|)$ time and as a result we shall either find the required bucket $B^j_\alpha[n + |l(op)|]$ or establish that such a bucket does not exist. In the latter case we can determine the position where a bucket $B^j_\alpha[n + |l(op)|]$ must be inserted within the same time-bounds, $O(|l(op)|)$, and insert it at the cost of $O(1)$ time.

Summing up we spend $O(\sum_{op\in\Lambda} |l(op)|)$ time per element $\langle u, c(u)\rangle$ in the edit-distance list $L_r(\alpha, j)$ and therefore the total amount of work the algorithm requires is $O(Size(j) + Z)$ as claimed.

□

Now, given an edit-distance list $L(\alpha 0)$ we can easily compute $L_r(\alpha, |V_{\alpha 1}|)$, see procedure RightExtension. We start with an EpsilonClosure step of the candidates generated in $L(\alpha 0)$ and then iterate the main steps extension and $\varepsilon$-closure. In the step extension we initialise the buckets and fill them in. To control these steps we use a deterministic infix automaton, $\mathcal{IA}_r$, for the language $Inf(\mathcal{L})$.

Similarly to the right extensions, given the edit-distance list $L^{rev}(\alpha 1)$ we can compute the edit-distance list, $L^{rev}_l(\alpha, |V_{\alpha 0}|)$, see procedure LeftExtension. The only difference here is that we have to process the word $V_{\alpha 0}$ backwards, i.e. from right to left. This is the reason to start counting from $|V_{\alpha 0}| + 1$, using the reverse edit-distance, $(Op^{rev}, c^{rev})$, and a deterministic automaton, $\mathcal{IA}_l$, for the $Inf(\mathcal{L}^{rev})$.

`RightExtension(`$\mathcal{A}, \langle Op, c\rangle, q, \alpha, V, L$`)`
$\eta \leftarrow 1$
$N_{\alpha 1} \leftarrow |V_{\alpha 1}|$
$\mathcal{T} \leftarrow \mathcal{T}(\alpha 0)$
$L_r(\alpha, 0) \leftarrow L(\alpha 0)$
$Act \leftarrow InitialiseActive(\mathcal{T}, L_r(\alpha, 0))$
`EpsilonClosure(`$\mathcal{A}, \mathcal{T}, \langle Op, c\rangle, q, V, Act, \alpha, L_r(\alpha, .), 0$`)`

```
   Inactivate(Act, 𝒯)
   for j = 1 to N_α1 do
      if L_r(α, j − 1) = ∅ then
         L_r(α, N_α1) ← ∅
         return empty trie
      fi
      InitialiseBuckets(⟨Op, c⟩, η, j, L_r(α, .))
      FillBuckets(𝒜, 𝒯, ⟨Op, c⟩, q, V, Act, α, η, L_r(α, .), j)
      EpsilonClosure(𝒜, 𝒯, ⟨Op, c⟩, q, V, Act, α, L_r(α, .), j)
      Inactivate(Act, 𝒯)
   done
   return 𝒯

LeftExtension(𝒜, ⟨Op, c⟩, q, α, V, L)
   η ← 0
   N_α0 ← |V_α0|
   𝒯 ← 𝒯(α1)
   L_l(α, 0)) ← L(α1)
   Act ← InitialiseActive(𝒯, L_l(α, 0))
   EpsilonClosure(𝒜, 𝒯, ⟨Op, c⟩, q, V, Act, α, L_l(α, .), 0)
   Inactivate(Act, 𝒯)
   for j = 1 downto N_α0 do
      if L_l(α, j − 1) = ∅ then
         L_l(α, N_α0) ← ∅
         return empty trie
      fi
      InitialiseBuckets(⟨Op, c⟩, η, j, L_l(α, .))
      FillBuckets(𝒜, 𝒯, ⟨Op, c⟩, q, V, Act, α, η, L_l(α, .), j)
      EpsilonClosure(𝒜, 𝒯, ⟨Op, c⟩, q, V, Act, α, L_l(α, .), j)
      Inactivate(Act, 𝒯)
   done
   return 𝒯

InitialiseActive(𝒯, L)
   A ← ∅
   for B bucket in L do
      for ⟨u, c⟩ ∈ B do
         act(u) ← true
         cost(u) ← c
         A.Insert(u)
      done
   done
   return A

Inactivate(Act, 𝒯)
   for u ∈ A do
         act(u) ← false
```

$A.Remove(u)$
`done`

*Reverse and Union.*

As a result from the LeftExtension we have obtained instead the desired edit-distance list $L_l(\alpha, N_{\alpha 0})$ its reverse $L_l^{rev}(\alpha, N_{\alpha 0})$. However, according to Lemma 4.4.7 $L_l^{rev}(\alpha, j)$ $\mathcal{L}^{rev}$-represent the alignment set $\overleftarrow{\mathfrak{A}^j}^{rev}$. Thus, we need to reverse the edit-distance list with respect to Corollary 4.4.5 in order to obtain a $\mathcal{L}$-representation for $\overleftarrow{\mathfrak{A}^j}$. This can be easily computed by the means of the procedure ReverseDistanceList:

```
ReverseDistanceList(A, T, L)
   T^rev ← new empty trie with root r
   s ←initial state of A
   for B bucket in L do
      for ⟨u, c⟩ ∈ B do
         W ← label(u, T)
         st ← TraverseAutomaton(A, s, W)
         u^rev ← TraverseTrie(T^rev, r, W)
         st(u^rev) ← st
         u ← u^rev
      done
   done
   return T^rev

label(u, T)
   if u is the root of T
      return ε
   else
      p ← par(u, T) // parent of u in T
      σ ← ch(u, T) // i.e. δ_T(p, σ) = u
      return label(p, T) ∘ σ
```

**Lemma 5.3.4** *Given a representation of the edit-distance list $L_l^{rev}(\alpha, j)$ and a deterministic automaton $\mathcal{IA}_r$ for the language $Inf(\mathcal{L})$ we can compute a representation of $L_l(\alpha, j)$ in time $O(\sum_{U \in Dom(L_l(\alpha,j))}(|U| + 1))$.*

*Proof.* Indeed we can invoke the procedure ReverseDistanceList($\mathcal{IA}_r, \mathcal{T}_l(\alpha), L_l^{rev}(\alpha, j)$) where $\mathcal{T}_l(\alpha)$ is the trie from the representation of $L_l^{rev}(\alpha, j)$. Clearly, it processes each element of the original edit-distance list and reverses its label. Traversing the automaton $\mathcal{IA}_r$ with the reversed label yields the automaton-state with respect to the straight language, $Inf(\mathcal{L})$. The time complexity follows by the fact that we spend $O(1)$ time per element of the edit-distance list plus the additional time in the computation of the label, $label(u)$, in the procedure $label(\mathcal{T}, u)$. Thus we spend $O(|U|+1)$ time per element $U \in Dom(L_l(\alpha, j))$ which yields the total bound of $O(\sum_{U \in Dom(L_l(\alpha,j))}(|U| + 1))$. □

We take these remarks into account when computing the edit-distance list $L(\alpha)$, see Figure 5.3. Since it should $\mathcal{L}$-represent the alignment set:

$$\mathfrak{A}^{\leq b_\alpha}(V_\alpha) = \mathfrak{A}(V_{\alpha 0}, b_{\alpha 0} \rightarrow V_\alpha, b_\alpha) \cup \mathfrak{A}(V_\alpha, b_\alpha \leftarrow V_{\alpha 1}, b_{\alpha 1}),$$

we can apply Lemma 4.3.6 to the edit-distance lists $L_r(\alpha, N_{\alpha 1})$ and $L_l(\alpha, N_{\alpha 0})$ that $\mathcal{L}$-represent the sets $\mathfrak{A}(V_{\alpha 0}, b_{\alpha 0} \rightarrow V_\alpha, b_\alpha)$ and $\mathfrak{A}(V_\alpha, b_\alpha \leftarrow V_{\alpha 1}, b_{\alpha 1})$, respectively.

To this end, for a fixed $\alpha$ we initiate a new trie, $\mathcal{T}(\alpha)$, which is responsible for the list $L(\alpha)$ and we construct $L(\alpha)$ by the means of the following procedure, see also procedure UnionDistanceLists:

1. we determine the nonempty buckets for $L(\alpha)$ and allocate them. To this end we take advantage of a merge-sort procedure of the indices of the buckets in $L_r(\alpha, N_{\alpha 1})$ and $L_l(\alpha, N_{\alpha 0})$.

2. we start with the empty trie $\mathcal{T}(\alpha)$.

   (a) for every element $\langle u, c(u)\rangle \in L_r(\alpha, N_{\alpha 1})$ insert $label(u)$ to $\mathcal{T}(\alpha)$. Let $u'$ be the resulting node on the trie.
      i. $st(u') = st(u)$.
      ii. $act(u') = true$, insert $u'$ to $Act$.
      iii. $cost(u') = c(u')$.

   (b) for every element $\langle u, c(u)\rangle \in L_l(\alpha, N_{\alpha 0})$, insert $label(u)$ to $\mathcal{T}(\alpha)$. Let $u'$ be the resulting node on the trie.
      i. $st(u') = st(u)$.
      ii. if $act(u') = false$, set $act(u') = true$ and $cost(u') = c(u)$, add $u'$ to $Act$.
      iii. if $cost(u') > c(u)$, set $cost(u') = c(u)$.

   (c) for every node $u \in Act$, insert $\langle u, cost(u)\rangle$ to $B_\alpha[|label(u)|]$. Set $act(u) = false$.

   (d) Reset $Act = \emptyset$.

```
UnionDistanceLists(T_l, T_r, V, L_l, L_r, α)
   T ← empty trie with root r
   N_α0 ← |V_α0|
   N_α1 ← |V_α1|
   B_left ← L_l(α, N_α0).First()
   B_right ← L_r(α, N_α1).First()
   while B_left or B_right is defined do
      n_left ← index of B_left in L_l(α, N_α0) // i.e. B_left = B_α^{Nα0}[n_left]
      n_right ← index of B_right in L_r(α, N_α1) // i.e. B_right = B_α^{Nα1}[n_right]
      if n_left < n_right then
         B ← new empty block B[n_left]
         B_left ← L_l(α, N_α0).Next(B_left)
```

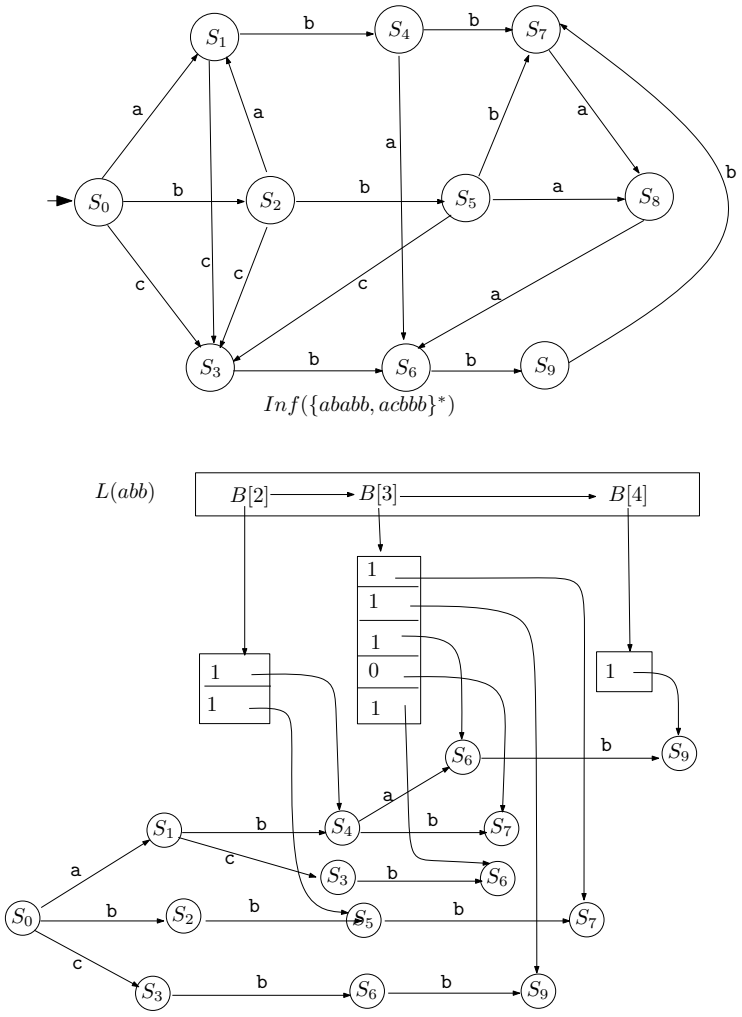



Figure 5.3: Merging left and right extension lists.

```
      fi
      if n_right < n_left then
         B ← new empty block B[n_right]
         B_right ← L_r(α, N_α1).Next(B_right)
      fi
      if n_right = n_left then
         B ← new empty block B[n_right]
         B_left ← L_l(α, N_α0).Next(B_left)
         B_right ← L_r(α, N_α1).Next(B_right)
      fi
      L(α).Append(B)
   done
   Act ← ∅
   for B ∈ L_r(α, N_α1) do
      for ⟨u, c⟩ ∈ B do
         W ← label(T_r, u)
         u' ← TraverseTrie(T, r, W)
         st(u') ← st(u)
         act(u') ← true
         cost(u') ← c
         Act.Insert(u')
      done
   done
   for B ∈ L_l(α, N_α0) do
      for ⟨u, c⟩ ∈ B do
         W ← label(T_l, u)
         u' ← TraverseTrie(T, r, W)
         st(u') ← st(u)
         if act(u') ← true then
            cost(u') ← min(cost(u'), c)
         else
            cost(u')⟨c
            Act.Insert(u')
         fi
      done
   done
   while Act ≠ ∅ do
      u ← Act.Remove()
      act(u) ← false
      B[|label(u)|].Insert(⟨u, cost(u)⟩)
      cost(u) ← −1
   done
   return T
```

**Lemma 5.3.5** *The above algorithm correctly computes the representation of* $L(\alpha)$*. Furthermore the time complexity if* $O(\sum_{\langle u,c(u)\rangle} |label(u)|)$ *where the sum-*

*mation is over all pairs in* $L(\alpha)$.

*Proof.* The algorithm processes each pair $\langle u, c(u)\rangle \in L_r(\alpha, N_{\alpha 1})$ and each pair $\langle u, c(u)\rangle \in L_l(\alpha, N_{\alpha 0})$ once only and inserts a node $u' \in \mathcal{T}(\alpha)$ with label $label(u')$ s.t.:

$$label(u') = \begin{cases} label(u) \text{ if } \langle u, c(u)\rangle \in L_r(\alpha, N_{\alpha 1}) \\ label(u) \text{ if } \langle u, c(u)\rangle \in L_l(\alpha, N_{\alpha 0}). \end{cases}$$

In either case $st(u')$ is the final state of the traversal of $label(u')$ in the infix automaton $\mathcal{AI}_r$ and each such step clearly requires $O(|label(u')|)$ time. Furthermore each such node $u'$ becomes active and the algorithm assures that $u'$ is inserted in $Act$. Therefore in the second pass along the elements of $Act$ we obtain that $label(u')$ takes each of the values in $Dom(L_l(\alpha, N_{\alpha 0})) \cup Dom(L_r(\alpha, N_{\alpha 1}))$ once only. Since $u'$ is then inserted to the bucket $L_\alpha$ we obtain that:

$$Dom(L(\alpha)) = Dom(L_l(\alpha, N_{\alpha 0})) \cup Dom(L_r(\alpha, N_{\alpha 1})).$$

In Steps (a).iii and (b).iii the algorithm asserts that $c(u')$ is the minimum of the two values $c(u')$ which might be computed by $L_r(\alpha, N_{\alpha 1}) \cup L_l(\alpha, N_{\alpha 0})$. Therefore according to Lemma 4.3.6 $L(\alpha)$ $\mathcal{L}$-represents the union of the alignment sets represented by $L_r(\alpha, N_{\alpha 1})$ and $L_l(\alpha, N_{\alpha 0})$ as required. The time complexity is evidently in $O(\sum_{\langle u, c(u)\rangle} |label(u)|)$ since for each such pair we spend $O(|label(u)|)$ in traversals of the infix automaton and the trie. □

Summing up, we obtain a procedure that given $L(\alpha 0)$ and $L(\alpha 1)$ computes the edit-distance list $L(\alpha)$, see ExtensionStep. It takes advantage of an infix automata, $\mathcal{IA}_r$ and $\mathcal{IA}_l$, for the language $Inf(\mathcal{L})$ and $Inf(\mathcal{L}^{rev})$, respectively, the word $V$ and the edit-distance, $(Op, c)$. First it extends to the right $L(\alpha 0)$, then it extends to the left (i.e. applies LeftExtension to) $L^{rev}(\alpha 1)$ to obtain $L_l^{rev}(\alpha, N_{\alpha 0})$ and reverses this output in order to obtain $L_l(\alpha, N_{\alpha 0})$. Finally, it merges the results $L_r(\alpha, N_{\alpha 1})$ and $L_l(\alpha, N_{\alpha 0})$.

`ExtensionStep`$(\mathcal{IA}_r, \mathcal{IA}_l, \mathcal{T}_l, L, V, \alpha, \langle Op, c\rangle, q)$
  $\mathcal{T}_r \leftarrow RightExtension(\mathcal{IA}_r, \langle Op, c\rangle, q, \alpha, V, L)$
  $\mathcal{T}(\alpha 1) \leftarrow ReverseDistanceList(\mathcal{T}(\alpha 1), L(\alpha 1))$
  $\mathcal{T}_l \leftarrow LeftExtension(\mathcal{IA}_l, \langle Op, c\rangle, q, \alpha, V, L)$
  $\mathcal{T}_l \leftarrow ReverseDistanceList(\mathcal{T}_l, L_l(\alpha, |V_{\alpha 0}|)))$
  `return UnionDistanceLists`$(\mathcal{T}_l, \mathcal{T}_r, V, L_l, L_r, \alpha)$

Given the ExtensionStep algorithm we can easily device a recursive procedure, see ApproximateSearchRecursive. In particular, to determine the edit-distance list $L(\alpha)$, we first check whether $\alpha$ is a leaf according to the definition of the search tree, $\mathcal{T}(V)$. If this is the case we apply 5.2.1. Otherwise we recursively compute $L(\alpha 0)$ and $L(\alpha 1)$ and then apply the ExtensionStep to obtain $L(\alpha)$.

```
ApproximateSearchRecursive(IA_r, IA_l, L, V, α, ⟨Op, c⟩, q)
   if qN_α < 1 then
      return T(α)
   else
      T_r ← ApproximateSearchRecursive(IA_r, IA_l, L, V, α0, ⟨Op, c⟩, q)
      T_l ← ApproximateSearchRecursive(IA_r, IA_l, L, V, α1, ⟨Op, c⟩, q)
      return ExtensionStep(IA_r, T_r, IA_l, T_l, L, V, α, ⟨Op, c⟩, q)
   fi
```

In order to answer the query, we compute the edit-distance list $L(\varepsilon)$ and afterwards select those words in the domain of the list that are also words in the language $\mathcal{L}$. To this end we additionally use a deterministic finite state automaton, $\mathcal{A}$, with language $\mathcal{L}$, see procedure ApproximateSearchSimple.

```
ApproximateSearchSimple(V, N, q, α, A, IA_r, IA_l, ⟨Op, c⟩)
   ApproximateSearchInitialise(V, N, q, ε)
   T ← ApproximateSearchRecursive(IA_r, IA_l, L, V, ε, ⟨Op, c⟩, q)
   for B ∈ L(ε) do
      for ⟨u, c⟩ ∈ B do
         W ← label(T, u)
         s ←initial state of A
         if TraverseAutomaton(A, s, W) is defined then
            report W
         fi
      done
   done
```

**Proposition 5.3.6** *Given a query word $V$ of length $|V| = N$, the extension steps of approximate search problem can be solved in time $O(T_1 + T_2)$ where:*

$$\begin{aligned} T_1 &= \sum_{\alpha \in \mathcal{T}_N} \left( \sum_{j=0}^{N_{\alpha 0}} |L_l(\alpha, j)| + \sum_{j=0}^{N_{\alpha 1}} |L_r(\alpha, j)| \right) \\ T_2 &= \sum_{\alpha \in \mathcal{T}_N} \sum_{U \in Dom(L(\alpha))} |U|. \end{aligned}$$

*Proof.* Applying the algorithm described in this Section we have that the search splits into two main interleaving phases. The first one is the computation of the intermediate edit-distance lists $L_l^{rev}(\alpha, j)$ and $L_r(\alpha, j)$. And the second is the computation of the edit-distance lists $L(\alpha)$. As argued by Lemma 6.1.3 and Lemma 4.2.4 we have that the time for the computation of the intermediate list $L_r(\alpha, j)$ is $O(T_r(\alpha, j))$ where:

$$T_r(\alpha, j) = \begin{cases} |L(\alpha)| + |L_r(\alpha, 0)| \text{ if } j = 0 \\ |L_r(\alpha, j-1)| + |L_r(\alpha, j)| \text{ if } j > 0. \end{cases}$$

Since $j$ is bounded by the length of right child, $\alpha 1$, of $\alpha$ we derive that the time spent in the computation of intermediate lists $L_r(\alpha, j)$ is $O(T_r)$:

$$\begin{aligned} T_r &= \sum_{\alpha \in \mathcal{T}(V)} \sum_{j=0}^{N_{\alpha 1}} T_r(\alpha, j) \\ &= \sum_{\alpha \in \mathcal{T}(V)} \left( \sum_{j=1}^{N_{\alpha 1}} (|L_r(\alpha, j-1)| + |L_r(\alpha, j)|) + |L_r(\alpha, 0)| + |L(\alpha)| \right) \\ &\leq 2 \sum_{\alpha \in \mathcal{T}(V)} \sum_{j=0}^{N_{\alpha 1}} |L_r(\alpha, j)| + \sum_{\alpha \in \mathcal{T}(V)} |L(\alpha)|. \end{aligned}$$

However $|L(\alpha)| \leq |L_r(\alpha, N_{\alpha 1})| + |L_l(\alpha, N_{\alpha_0})|$ and consequently we get that:

$$\begin{aligned} T_r &\leq 2 \sum_{\alpha \in \mathcal{T}(V)} \sum_{j=0}^{N_{\alpha 1}} |L_r(\alpha, j)| + \sum_{\alpha \in \mathcal{T}(V)} (|L_r(\alpha, N_{\alpha 1})| + |L_l(\alpha, N_{\alpha_0})|) \\ &\leq 3 \sum_{\alpha \in \mathcal{T}(V)} \sum_{j=0}^{|N_{\alpha 1}|} |L_r(\alpha, j)| + \sum_{\alpha \in \mathcal{T}(V)} \sum_{j=1}^{N_{\alpha 0}} |L_l(\alpha, j)| \leq 3T_1. \end{aligned}$$

Similar argument reveals that the time spent in the computation of the intermediate edit-distance lists $L_l(\alpha, j)$ does not exceed $O(T_1)$ and therefore the time for constructing the intermediate edit-distance lists is $O(T_1)$.

To complete the proof it remains to analyse the time required for the construction of the edit-distacnce lists $L(\alpha)$. According to Lemma 4.3.6 our algorithm spends $O(T(\alpha))$ time for the list $L(\alpha)$ where:

$$T(\alpha) = \sum_{U \in Dom(L_r(\alpha, N_{\alpha 1}))} |U| + \sum_{U \in Dom(L_l(\alpha, N_{\alpha 0}))} |U|.$$

Since each word $U$ which belongs to $Dom(L_r(\alpha, N_{\alpha 1})) \cup Dom(L_l(\alpha, N_{\alpha 0}))$ is inserted in the domain of the list $L(\alpha)$, we get that:

$$T(\alpha) \leq 2 \sum_{U \in L(\alpha)} |U|.$$

Therefore the total time required for the construction of the lists $L(\alpha)$ is bounded by a constant factor of:

$$\sum_{\alpha \in \mathcal{T}(V)} T(\alpha) \leq 2 \sum_{\alpha \in \mathcal{T}(V)} \sum_{U \in Dom(L(\alpha))} |U| = 2T_2.$$

Therefore the time-complexity of these steps is $O(T_2)$ and we derive a total bound for the time required for all the extension phases as $O(T_1 + T_2)$. □

### 5.3.2 Deg-lex Order of Edit-distance Lists. Faster Technique for Finite Languages

In case that the given regular language $\mathcal{L}$ is finite we can significantly simplify the algorithm presented in the previous section as well as to reduce its time complexity.

To this end we use a representation of the language $\mathcal{L}$ as a bidirectional CDWAG as described in Chapter 2. It is not important which of both structures, the structure of Blumer et al. or the one of Inenaga. The only properties of the structure we are going to use are listed below:

1. for each infix $U$ of $\mathcal{L}$ we have a unique constant size representation $st(U)$.
2. given a representation $st(U)$ and an arbitrary character $a \in \Sigma$ we have an $O(1)$ algorithm which:
   (a) determines whether $U \circ a$ ($a \circ U$) is an infix of the language $\mathcal{L}$
   (b) and in the positive case produces a representation $st(U \circ a)$ ($st(a \circ U)$).
3. there is an $O(1)$ algorithm which given a representation of an infix $U$, $st(U)$, provides an index $ind(U)$ in the word $W_{\mathcal{L}}$ (the concatenation of all the words in $\mathcal{L}$ where an occurrence of $U$ starts.)

In the sequel we describe how the above properties give rise to a more efficient and elegant algorithm which solves the approximate search problem in for languages. Since both the Inenaga's and the Blumers' constructions obey these properties, see Chapter 2, this implies that this algorithm is applicable and can be implemented at the cost of linear space with respect to the size of the finite language $\mathcal{L}$.

**Definition 5.3.7** The deg-lex order on words $U, V \in \Sigma^*$ is defined as:

$$U \preceq_{deg-lex} V \iff |U| < |V| \text{ or } (|U| = |V| \text{ and } U \preceq_{lex} V).$$

It should be clear $(\Sigma^*, \preceq_{deg-lex})$ is a linear ordering.

The idea is to keep the edit-distance lists $L_l(\alpha, j)$, $L_r(\alpha, j)$ and $L(\alpha)$ ordered with respect to $\preceq_{deg-lex}$. Each pair of the edit-distance list $L$, say $\langle U, c_U \rangle$ is represented as $\langle st(U), len(U), c_U \rangle$ where $st(U)$ is the constant size representation of $U$ w.r.t. the data-structure we are using and $len(U) = |U|$ is the length of the word. Finally, we preprocess the long word $W = W_{\mathcal{L}}$ in order to compute the lexicographic order of the suffixes of $W$, $w_i \circ w_{i+1} \cdots \circ w_{|W|}$. More specifically, we maintain an array $f[1..|W|]$ such that for each index $i$ we get the position $f[i]$ where the suffix of $W$ starting at position $i$ would be placed if all the (nonempty) suffixes of $W$ were lexicographically sorted.

`Join(`$\mathbf{L}, SA, f, k$`)`
`//`$\mathbf{L}[1..k]$ `is an array of lists. Each list contains representations,`
`//`$\langle st(U), len(U), c(U) \rangle$ `of words` $U$ `sorted in deg-lex increasing order.`

```
//f compares
   Heap ←empty heap
   for i = 1 to k do
      ⟨st, l, c⟩ ← L[i].First()
      if ⟨st, l, c⟩ is defined then
         Heap.InsertJoinHeap(⟨st, l, c, i⟩, f, SA)
      fi
   done
   LOrd ← ∅
   last ← ⟨⊥, −1⟩
   while Heap ≠ ∅ do
      ⟨st, l, c, i⟩ ← ExtractJoinMin(Heap, f, SA)
      if ⟨st, l⟩ ≠ last then
         LOrd.Append(⟨st, l, c⟩)
         last ← ⟨st, l⟩
      fi
      ⟨st, l, c⟩ ← L[i].Next(⟨st, l, c⟩)
      if ⟨st, l, c⟩ is defined then
         InsertJoinHeap(Heap, ⟨st, l, c, i⟩, f, SA)
      fi
   done
   return LOrd

CompareJoinHeap(Heap,m,n,f,SA)
   ⟨st′, l′, c′, i′⟩ ← Heap[m]
   ⟨st″, l″, c″, i″⟩ ← Heap[n]
   if l′ < l″ then return true
   if l′ > l″ then return false
   if SA[f[st′]] < SA[f[st″]] then return true
   if SA[f[st″]] < SA[f[st′]] then return false
   if c′ < c″ then return true
   if c′ > c″ then return false
   return true //the order w.r.t. to the specific list-index is not important.

InsertJoinHeap(Heap, ⟨st, l, c, i⟩, f, SA)
   n ← Heap.size + 1
   Heap[n] ← ⟨st, l, c, i⟩
   m ← ⌊(n+1)/2⌋
   while m > 0 and CompareJoinHeap(Heap, m, n, f, SA) = false do
      Swap(Heap, m, n)
      n ← m
      m ← ⌊n/2⌋
   done

ExtractJoinMin(Heap, f, SA)
   result ← Heap[1]
   n ← Heap.size
```

```
Heap[1] ← Heap[n]
Heap.size ← n − 1
m ← 1
while 2m < n do
   if 2m+1 < n and CompareJoinHeap(Heap, 2m+1, 2m, f, SA) then
      min ← 2m + 1
   else
      min ← 2m
   if CompareJoinHeap(Heap, min, m, f, SA) then
      Swap(Heap, min, m)
      m ← min
   else
      m ← n //i.e. break the loop.
done
return result
```

The basic observation which gives rise to our modified algorithm is that the union (join) of deg-lex sorted edit-distance lists can be efficiently computed by the means of an merge-sort-like algorithm:

**Lemma 5.3.8** *Let $L_1, L_2, \ldots, L_k$, be deg-lex sorted edit-distance lists. Then we can compute a deg-lex sorted edit-distance list for their join, $L = \bigvee_{j=1}^{k} L_j$, in time $O(\sum_{j=1}^{k} |L_j| \log k)$ where the join $L = \bigvee_{j=1}^{k} L_j$ is defined as:*

$$\begin{aligned} Dom(L) &= \cup_{j=1}^{k} Dom(L_j) \text{ and} \\ L(U) &= \min_j L_j(U) \text{ for } U \in \cup_{j=1}^{k} Dom(L_j). \end{aligned}$$

*Proof.* The key property we use is that any two words $U', U'' \in Inf(\mathcal{L})$ can be compared in $O(1)$ time using only their representation $\langle st(U'), len(U'), c_{U'} \rangle$ and $\langle st(U''), len(U''), c_{U''} \rangle$. Indeed first we compare the lengths $len(U')$ and $len(U'')$. If they are distinct, then the word of smaller length is also smaller with respect to the deg-lex order. Consider the negative case when $len(U') = len(U'')$. Then we look at the representation $st(U')$ and $st(U'')$. If $st(U') = st(U'')$ then obviously $U' = U''$. Finally, if $st(U') \neq st(U'')$ then by the third property of our data-structure we can find indices $ind'$ and $ind''$ in constant time, such that $ind'$ is a starting position of $U'$ in $W_{\mathcal{L}}$ and $ind''$ is a starting position of $U''$ in $W_{\mathcal{L}}$.

It should be clear that $ind' \neq ind''$. Otherwise since $len(U') = len(U'')$ we get that $U' = U''$ which implies that $st(U') = st(U'')$. Since $ind' \neq ind''$ we have that $f[ind'] \neq f[ind'']$ and therefore $f[ind'] < f[ind'']$ or $f[ind'] > f[ind'']$. In the former case we conclude that the suffixes of $W_{\mathcal{L}}$ starting at position $ind'$ is lexicographically less than that starting at position $ind''$. The first suffix starts with $U'$ and the second with $U''$. Since $|U'| = |U''|$ but $U' \neq U''$ we deduce that $U'$ and $U''$ differ at some position. Let $j$ be the first position where $U'$ and $U''$ differ. Then this is also the first position where the suffix starting at position $ind'$ does not match the corresponding position of the suffix

starting at position $ind''$. Since $f[ind'] < f[ind'']$ this implies that $U' \prec_{lex} U''$ and therefore $U' \prec_{deg-lex} U''$. The case where $f[ind'] > f[ind'']$ is considered analogously.

Since $st(U')$ and $st(U'')$ have constant representation, we can check their equality in $O(1)$ time. The constant execution of the rest of the steps is also clear.

Now given the deg-lex sorted edit-distance lists we can easily adopt a merge sort algorithm which renders their join. The only thing we need to be conscious of is that when two elements $\langle st(U'), len(U'), c' \rangle$ and $\langle st(U''), len(U''), c'' \rangle$ have the property that $st(U') = st(U'')$ (and $len(U') = len(U'')$), i.e. when they represent the same word, we always store the minimum of $\min(c', c'')$ of the values $c'$ and $c''$.

With these remarks it should be clear that merging the lists $L_1, \ldots L_k$ using a binary heap we achieve both a deg-lex ordered representation for their join and the claimed running time $O(\sum_{j=1}^{k} |L_j| \log k)$.

The algorithm described in the proof is illustrated in procedure Join. □

Based on this result we can easily modify the algorithm which constructs the intermediate edit-distance lists $L_r(\alpha, j)$ and $L_l(\alpha, j)$ as follows. We describe only the case with $L_r(\alpha, j)$ the second being dual:

1. Construct a deg-lex sorted list $L'_r(\alpha, j)$ using the deg-lex sorted list $L_r(\alpha, j-1)$.
2. Construct a deg-lex sorted list $L_r(\alpha, j)$ using the list $L'_r(\alpha, j)$.

We highlight the details on each of these two steps next.

*Construction of the deg-lex sorted $L'_r(\alpha, j)$.*

The idea is simple. We extend the already available deg-lex sorted edit-distance list $L_r(\alpha, j-1)$ with each compatible operation $op$. As a result we obtain $L_r(\alpha, j-1; op)$. Each of these lists is again deg-lex sorted (as we shall see) and their total number is $O(|Op|)$. Thus, we can apply the algorithm of the previous lemma in order to determine their join, $L'_r(\alpha, j)$.

The details are as follow, see also procedure ExtensionJoinStepStimple:

1. for every operation $op \in Op$ with $|r(op)| = 1$ such that $r(op) = I_j^j(V_{\alpha 1})$, i.e. is a the $j$-th character of $V_{\alpha 1}$, construct an edit-distance list $L_r(\alpha, j-1; op)$. To this end:
   (a) consider the elements $\langle st(U'), len(U'), c' \rangle$ of $L_r(\alpha, j-1)$ in order.
   (b) for each such element determine whether $c' + c(op) \leq b_\alpha$. If not then proceed with the next element.
   (c) if $c' + c(op) \leq b_\alpha$, traverse the bidirectional data structure from $st(U')$ with the label $l(op)$. If this procedure fails at some point continue with the next element of the list. Otherwise call $st''$ the last constant size representation obtained.
   (d) attach to the back of $L_r(\alpha, j-1; op)$ the element $\langle st'', len(U') + |l(op)|, c' + c(op) \rangle$.

2. use the merge-sort algorithm to compute the join of all the resulting lists $L_r(\alpha, j-1; op)$. Thus we obtain $L'_r(\alpha, j)$.

```
ExtensionJoinStepSimple(A, SA, f, ⟨Op, c⟩, q, V, α, η, L, j)
   pos ← if η = 1 then j else |V_α0| − j + 1
   k ← 0
   for op ∈ Op with |r(op)| ≠ ε do
      k ← k + 1
      L(α, j; op) ← ∅
      for ⟨st, l, c⟩ ∈ L(α, j − 1) do
         if r(op) = V[pos] then
            if η = 1 then
               st' ← ExtendRight(A, st, l(op))
            else
               st' ← ExtendLeft(A, st, l(op))
            fi
            if st' is defined and c(op) + c ≤ q|V_α| then
               L(α, j; op).Insert(⟨st', l + |l(op)|, c(op) + c⟩)
            fi
         fi
      done
   done
   return Join(L(α, j; .), SA, f, k)
```

**Lemma 5.3.9** *The edit-distance list $L'_r(\alpha, j)$ computed by the above procedure $\mathcal{L}$-represents $\widetilde{\mathfrak{A}}^j$ and is deg-lex ordered. The computation of each such list requires $O(|L_r(\alpha, j-1)|)$ time and the total time spent for the computation of $L'_r(\alpha, j)$ is:*

$$O(|L_r(\alpha, j-1)|).$$

*Proof.* Using that $L_r(\alpha, j-1)$ $\mathcal{L}$-represents $\overrightarrow{\mathfrak{A}}^{j-1}$, Lemma 4.3.3 implies that $L_r(\alpha, j-1; op)$ represents $(\overrightarrow{\mathfrak{A}}^{j-1} \circ op)^{\leq b_\alpha}$. Further, we prove that $L_r(\alpha, j-1; op)$ is deg-lex sorted. To see this consider two elements $\langle st'_1, len'_1, c'_1\rangle$ and $\langle st'_2, len'_2, c'_2\rangle$ inserted in $L_r(\alpha, j-1; op)$. Assume that $\langle st'_i, len'_i, c'_i\rangle$ was inserted when considering the element $\langle st_i, len_i, c_i\rangle$ in the list $L_r(\alpha, j-1)$ for $i = 1, 2$ and let $\langle st_1, len_1, c_1\rangle$ be considered first. If $st_1$ represents the word $U_1$, then by the construction of $st'_1$ it follows that $st'_1$ represents $U_1 \circ l(op)$. Furthermore $len'_1 = len_1 + l(op) = |U_1 \circ l(op)|$. Similarly if $st_2$ represents $U_2$, then $st'_2$ represents $U_2 \circ l(op)$ and $len'_2 = len_2 + |l(op)| = |U_2 \circ l(op)|$. Now since $\langle st_1, len_1, c_1\rangle$ was considered first, it follows that $U_1 \prec_{deg-lex} U_2$ and therefore $U_1 \circ l(op) \prec_{deg-lex} U_2 \circ l(op)$. However when $\langle st'_2, len'_2, c'_2\rangle$ is inserted to $L_r(\alpha, j-1; op)$, the triple $\langle st'_1, len'_1, c'_1\rangle$ resides already there and therefore being inserted at the back, $\langle st'_2, len'_2, c'_2\rangle$ correctly preserves the deg-lex order of the list. Since each of the lists $L_r(\alpha, j-1; op)$ is deg-lex ordered and represents $(\overrightarrow{\mathfrak{A}}^{j-1} \circ op)^{\leq b_\alpha}$, respectively, their join is also deg-lex ordered and by

Lemma 4.3.6 it represents $\widetilde{\mathfrak{A}}^j$. Therefore the computed edit-distance list $L'_r(\alpha, j)$ has the desired properties.

It is also clear that the time spent per triple $\langle st, len, c\rangle \in L_r(\alpha, j-1)$ is proportional to $O(|l(op)| + |r(op)|)$. This follows from the properties about the data-structure representing the infixes of the language $\mathcal{L}$. Applying Property 1 subsequently we can compute the representation $st''$ from the representation $st$ in time $O(|l(op)|)$. In particular we spend $O(\sum_{op\in Op}(|l(op)|+|r(op)|))$ time per element $\langle st, len, c\rangle \in L_r(\alpha, j-1)$ and thus the time for the construction of the lists $L_r(\alpha, j-1; op)$ is bounded by $O(|L_r(\alpha, j-1)|$. Applying Lemma 5.3.8 we obtain the conclusion of the lemma. □

*Construction of the deg-lex sorted list $L_r(\alpha, j)$.*

We start this step by considering the elements of $L'_r(\alpha, j)$. The idea is again to use Lemma 4.2.4 in order to construct $L_r(\alpha, j)$. We pass through the elements of $L'_r(\alpha, j)$ and split it in buckets $B^j_\alpha[n]$ according to their length component, *len*. Specifically, the bucket $B^j_\alpha[n]$ occurs if and only if there are elements $\langle st, n, c\rangle \in L'_r(\alpha, j)$ and it contains all these elements in the same order as the order in which these elements occur in the list $L'_r(\alpha, j)$.

**Lemma 5.3.10** *The buckets $B^j_\alpha[n]$ can be computed in $O(|L'_r(\alpha, j)|)$ time and each of them is deg-lex ordered.*

*Proof.* Preserving the order of the elements of $L'_r(\alpha, j)$ also preserves their property to be sorted in the buckets. Furthermore since $L'_r(\alpha, j)$ is deg-lex ordered once we encounter an element $\langle st, n, c\rangle$ it follows that no element $\langle st', n', c'\rangle$ with $n' < n$ will occur afterwards. Thus filling in the buckets can be performed in a single pass through the elements of $L'_r(\alpha, j)$. □

Now we complete the buckets $B^j_\alpha[n]$ in increasing order of $n$. To this end the buckets are organised in a list in increasing order of $n$. During this procedure new buckets might be created and they also will be inserted and considered as follows, see also procedure EpsilonClosureJoinStepSimple:

1. consider the buckets $B^j_\alpha[n]$ in increasing order of $n$.
2. for each operation $op \in \Lambda$ construct a list $B^j_\alpha[n; op]$ as follows:
   (a) consider the elements $\langle st, len, c\rangle \in B^j_\alpha[n]$ in order ($len = n$ for each such element).
   (b) if $c + c(op) > b_\alpha$ proceed with the next element.
   (c) traverse the data-structure starting from $st$ to the right with the label $l(op)$. If all the representation on the way exist define $st'$ to be the last one, otherwise continue with the next element of the bucket.
   (d) insert $\langle st', len + l(op), c + c(op)\rangle$ to the back of the list $B^j_\alpha[n; op]$.
3. go along the list of buckets to find the bucket $B^j_\alpha[n + |l(op)|]$. If there is no such bucket create an empty bucket $B^j_\alpha[n + |l(op)|]$ and insert it as to preserve the general order of the buckets.

4. find the join $B^j_\alpha[n+|l(op)|] \vee B^j_\alpha[n;op]$ using the merge sort algorithm and set this as the new value of $B^j_\alpha[n+|l(op)|]$.

5. append the elements of $B^j_\alpha[n]$ in this order at the back of $L_r(\alpha, j)$.

```
EpsilonClosureJoinStepSimple(A, SA, f, ⟨Op, c⟩, V, α, η, q, L, j)
   LBuck ← SplitInBuckets(L(α, j)
   for B[n] ∈ LBuck in order do //this corresponds to increasing order of n
      for op ∈ Op with r(op) = ε do
         B[n; op] ← ∅
         for ⟨st, l, c⟩ ∈ B[n] in order do //assert that n = l
            if η = 1 then
               st' ← RightTraverse(A, st, l(op))
            else
               st' ← LeftTraverse(A, st, l(op))
            fi
            if st' is defined and c(op) + c ≤ q|V_α| then
               B[n; op].Insert(⟨st', l + |l(op)|, c + c(op)⟩)
            fi
         dobe
         B[m] ← L.Next(B[n])
         last ← n
         while B[m] is defined and m ≤ n + |l(op)| do
            last ← m
            B[m] ← L.Next(B[m])
         done
         if last = n + |l(op)| then
            B[last] ← Join({B[last], B[n; op]}, SA, f, 2
         else
            B[n + |l(op)|] ← B[n; op]
         LBuck.InsertAfter(B[last], B[n + |l(op)|])
         fi
      done
      return Concat(LBuck)

Concat(LBuck)
   LCat ← ∅
   for B[n] ∈ LBuck in order do
      for ⟨st, l, c⟩ ∈ B[n] in order do
         LCat.Append(⟨st, l, c⟩)
      done
   done
   return LCat

SplitInBuckets(L)
   LBuck ← ∅
   last ← ⊥
```

```
for ⟨st, l, c⟩ ∈ L in order do
   if last ≠ l then
      B[l] ← new bucket
      LBuck.Append(B[l])
      last ← l
   fi
   B[l].Append(⟨st, l, c⟩)
done
return LBuck
```

**Lemma 5.3.11** *During the algorithm all the buckets are deg-lex ordered. Furthermore every list $B^j_\alpha[n; op]$ is deg-lex ordered and represents $(\overrightarrow{\mathfrak{A}}^j[n] \circ op)^{\leq q|V_\alpha|}$ and the list $L_r(\alpha, j)$ is computed correctly.*

*Proof.* As in the proof of Lemma 5.3.2 the construction of the supplementary lists $B^j_\alpha[n; op]$ preserves the deg-lex order and $B^j_\alpha[n; op]$ represents exactly $B^j_\alpha[n; op]$. Afterwards, in Step (d), the algorithm also preserves the deg-lex order provided that the input lists are deg-lex ordered. Since at the beginning all the buckets are deg-lex ordered, we deduce that at each step of the algorithm this property is preserved.

Next an easy inductive argument shows that at the time when $B^j_\alpha[n]$ is considered it $\mathcal{L}$-represents the union:

$$\widetilde{\mathfrak{A}}^j[n] \cup \bigcup_{op \in \Lambda} (\overrightarrow{\mathfrak{A}}^j[n - l(op)] \circ op)^{\leq q|V_\alpha|} = \overrightarrow{\mathfrak{A}}^j[n]$$

and according to Lemma 4.2.4 we deduce that the union of the buckets $B^j_\alpha[n]$ upon the termination of the algorithm yield the desired result $L_r(\alpha, j)$. Furthermore if $n < n'$, then any element $\langle st, n, c\rangle \in B^j_\alpha[n]$ and any element $\langle st', n', c'\rangle \in B^j_\alpha[n']$ representing words $U$ and $U'$ respectively have the property that $|U| = n < n' = |U'|$, in particular $U \prec_{deg-lex} U'$. This implies that $L_r(\alpha, j)$ is correctly computed and is deg-lex sorted upon the termination of the algorithm. □

Now we can easily complete single extension step by combining the previous results. Essentially RightExtensionJoin and LeftExtensionJoin are analogues of RightExtension and LeftExtension, that carry out the extension to right and the reversed left extension. The main difference is that we do not encode the generated candidates explicitly in tries, rather they are implicitly stored in deg-lex ordered edit-distance lists. This circumstance imposes that we construct several deg-lex edit-distance lists and then join them together.

```
RightExtensionJoin(𝒜, SA, f, ⟨Op, c⟩, q, α, V, L)
   L_r(α, 0) ← L(α0)
   L_r(α, 0) ← EpsilonClosureJoinStepSimple(𝒜, SA, f, 𝒯, ⟨Op, c⟩, V, α, 1, q, L, 0)
   for j = 1 to |V_α1| do
      L_r(α, j) ← ExtensionJoinStepSimple(𝒜, SA, f, 𝒯, ⟨Op, c⟩, q, V, α, 1, L, j)
```

$L_r(\alpha, j) \leftarrow EpsilonClosureJoinStepSimple(\mathcal{A}, SA, f, \mathcal{T}, \langle Op, c\rangle, V, \alpha, 1, q, L, j)$
`if` $L_r(\alpha, j) = \emptyset$ `return` $\emptyset$
`done`
`return` $L_r(\alpha, |V_{\alpha 1}|)$

`LeftExtensionJoin`$(\mathcal{A}, SA, f, \langle Op, c\rangle, q, \alpha, V, L)$
$L_l(\alpha, 0) \leftarrow L(\alpha 1)$
$L_l(\alpha, 0) \leftarrow EpsilonClosureJoinStepSimple(\mathcal{A}, SA, f, \mathcal{T}, \langle Op, c\rangle, V, \alpha, 0, q, L, 0)$
`for` $j = 1$ `dowto` $|V_{\alpha 0}|$ `do`
$L_l(\alpha, j) \leftarrow ExtensionJoinStepSimple(\mathcal{A}, SA, f, \mathcal{T}, \langle Op, c\rangle, q, V, \alpha, 0, L, j)$
$L_l(\alpha, j) \leftarrow EpsilonClosureJoinStepSimple(\mathcal{A}, SA, f, \mathcal{T}, \langle Op, c\rangle, V, \alpha, 0, q, L, j)$
`if` $L_l(\alpha, j) = \emptyset$ `return` $\emptyset$
`done`
`return` $L_l(\alpha, |V_{\alpha 0}|)$

**Lemma 5.3.12** *The computation of $L_r(\alpha, j)$ requires $O(|L_r(\alpha, j)|)$ time.*

*Proof.* It should be clear that for each element $\langle st, len, c\rangle \in B^j_\alpha[n]$ we spend $O(1)$ time in Steps 2(a)-(d) and Step 3. Additionally we spend $O(1)$ time for each element of $B^j_\alpha[n]$ in steps of type 3 when the merge sort is invoked by some bucket list $B^j_\alpha[n - |l(op)|; op]$. However there can be at most $O(|\Lambda|)$ such operations for a particular $n$, and therefore the total number of steps an element $\langle st, len, c\rangle \in B^j_\alpha[n]$ is processed is $O(|\Lambda|)$ which is considered as constant. Therefore the algorithm runs in $O(\sum_n |B^j_\alpha|)$ time and since the buckets are pairwise disjoint and their union is $L_r(\alpha, j)$, the upper bound follows. □

**Remark 5.3.13** We stress that the described algorithm is applicable for the steps of left extensions as well. The only details are that we should substitute the edit-distance $(Op, c)$ with the reversed $(Op^{rev}, c^{rev})$ and traverse the bidirectional structure leftwards instead of rightwards. The correctness then follows by the properties of the reversed alignments, see Section 4.4.

Given $L(\alpha 0)$ and $L(\alpha 1)$ we can apply the subroutines RightExtensionJoin and LeftExtensionJoin to compute $L(\alpha)$. Since the representation of the infixes is uniform, we do not need any reverse operation as in the general case, see procedure ExtensionJoin.

`ExtensionJoin`$(\mathcal{A}, SA, f, \langle Op, c\rangle, q, \alpha, V, L)$
$LRight \leftarrow RightExtensionJoin(\mathcal{A}, SA, f, \langle Op, c\rangle, q, \alpha, V, L)$
$LLeft \leftarrow LeftExtensionJoin(\mathcal{A}, SA, f, \langle Op, c\rangle, q, \alpha, V, L)$
$L(\alpha) \leftarrow Join(\{LRight, LLeft\}, SA, f, 2)$

**Lemma 5.3.14** *Given deg-lex sorted edit-distance lists $L_r(\alpha, N_{\alpha 1})$ and $L_l(\alpha, N_{\alpha 0})$ we can compute a deg-lex sorted edit-distance list $L(\alpha)$ in $O(|L(\alpha)|)$.*

*Proof.* Since $L(\alpha) = L_r(\alpha, N_{\alpha 1}) \vee L_l(\alpha, N_{\alpha 0})$ it suffices to apply the merge-sort algorithm which renders $L(\alpha)$ in time proportional to the result. □

Now the recursive computation of $L(\alpha)$ is clear, see procedure RecursiveExtensionJoin. The only issue is in the initialisation where we need to encode the infix $V_\alpha$, appropriately, see InitialiseJoin.

```
RecursiveExtensionJoin(A, SA, f, ⟨Op, c⟩, q, α, V, L)
   if q|V_α| ≥ 1 then
      RecursiveExtensionJoin(A, SA, f, ⟨Op, c⟩, q, α0, V, L)
      RecursiveExtensionJoin(A, SA, f, ⟨Op, c⟩, q, α1, V, L)
      ExtensionJoin(A, SA, f, ⟨Op, c⟩, q, α, V, L)
   else
      InitialiseJoin(A, q, α, N, V, L)
   fi
```

```
InitialiseJoin(A, q, α, N, V, L)
   s ←initial state of A
   if qN_α < 1 then
      st ← TraverseRight(A, s, V_α)
      if st is defined then
         L(α) ← {⟨st, N_α, 0⟩}
      else
         L(α) ← ∅
   fi
```

Finally we can solve the query by simply computing $L(\varepsilon)$ and reporting those words that are represented in $L(\varepsilon)$ and in the same time are words $\mathcal{L}$, see ApproximateSearchJoin.

```
ApproximateSearchJoin(A, SA, f, ⟨Op, c⟩, q, V)
   N_ε ← |V|
   V_ε ← V
   InitialiseJoin(A, q, ε, N, V, L)
   RecursiveExtensionJoin(A, SA, f, ⟨Op, c⟩, q, ε, V, L)
   for ⟨st, l, c⟩ ∈ L(ε) do
      if st is final in A
         report the word U with st = st(U)
      fi
   done
```

**Proposition 5.3.15** *If $\mathcal{L}$ is finite, there is a data structure $O(||\mathcal{L}||)$ which enables the extension steps for every query word $V$ of length $N$ in time:*

$$O(T_1 + T_2')$$

*where:*

$$\begin{aligned} T_1 &= \sum_{\alpha \in \mathcal{T}(V)} \sum_{j=0}^{N_{\alpha 1}} |L_r(\alpha, j)| + \sum_{\alpha \in \mathcal{T}(V)} \sum_{j=0}^{N_{\alpha 0}} |L_l(\alpha, j)| \\ T_2' &= \sum_{\alpha \in \mathcal{T}(V)} |L(\alpha)|. \end{aligned}$$

*Proof.* We use the Blumer et Blumer's structure from Chapter 2 which obeys the Properties 1-3 and we apply the algorithms described above. Then the result follows as in Proposition 5.3.6.

□

## 5.4 Reporting the Answers

Once we have determined the edit-distance list $L(\varepsilon)$ we can easily filter it so that we obtain the answers of the original query:

`Given:` $\mathcal{L} \subseteq \Sigma^*$ `regular language,`
$d = (Op, c)$ `an edit-distance,`
$q \in (0; 1)$ `a threshold parameter`
`Input:` $V \in \Sigma^*$
`Output:` $\{U \in \mathcal{L} \mid d(U, V) \leq q|V|\}$.

The following lemma states that this can be done efficiently:

**Lemma 5.4.1** *Assume that for a given query word $V$ we have the edit-distance list $L(\varepsilon)$ determined by the root of search tree $\mathcal{T}(V)$. If in addition we dispose on a (deterministic) final state automaton for the language $\mathcal{L}$ we can answer the query:*

`Given:` $\mathcal{L} \subseteq \Sigma^*$ `regular language,`
$d = (Op, c)$ `an edit-distance,`
$q \in (0; 1)$ `a threshold parameter`
`Input:` $V \in \Sigma^*$
`Output:` $\{U \in \mathcal{L} \mid d(U, V) \leq q|V|\}$.

*in time* $O(\sum_{U \in Dom(L(\varepsilon)} |U|)$.

*Proof.* The procedure we use is a simple one. For every word $U$ in the domain of the edit-distance list $L_\varepsilon$ we use the deterministic automaton for the language $\mathcal{L}$ in order to check in $O(|U|)$ time whether $U \in \mathcal{L}$ or not. If it does belong to the language $\mathcal{L}$, we report it otherwise we proceed with the next word. Clearly, this algorithm runs in $O(\sum_{U \in Dom_\varepsilon} |U|)$ total time. Furthermore it provides the correct answer of the query. Indeed since $L_\varepsilon$ $\mathcal{L}$-represents the alignment set $(\mathfrak{A}(V))^{\leq b_\varepsilon}$ we deduce that $U \in Dom_\varepsilon$ if and only if:

1. $U \in Inf(\mathcal{L})$.
2. there is an alignment $\omega \in \mathfrak{A}(V_\varepsilon)$ with $l(\omega) = U$ and $c(\omega) \leq b_\varepsilon$.

Since $V_\varepsilon = V$ the second condition is equivalent to $d(U, V) \leq q|V|$. Now the correctness of the algorithm follows by the fact that $\mathcal{L} \subseteq Inf(\mathcal{L})$, i.e. each word of $\mathcal{L}$ is also an infix of $\mathcal{L}$. Therefore the set of answers of the original query is contained in the set of answers represented by $L(\varepsilon)$ and thus applying the automaton $\mathcal{A}$ for the language $\mathcal{L}$ we can correctly filter out the answer of the query. □

# Chapter 6

# Approximate Search in the General Case, $\rho(Op) \geq 1$

In the previous chapters we considered the approximate search problem in the special case when $\rho(Op) = 1$. In this case each operation has a right side of length at most 1. This was the reason for the validity of Lemma 4.1.1. In particular, if we consider the alignment, $\omega = (\mathtt{d},\mathtt{d})(\varepsilon,\mathtt{r})(\varepsilon,\mathtt{e})(\mathtt{a},\mathtt{a})(\mathtt{d},\mathtt{d})$, in terms of Levenshtein operation of `dad` and `dread` and we determine $V_0 = \mathtt{dre}$ and $V_1 = \mathtt{ad}$, we can decompose $\omega = \omega_0 \circ \omega_1$ such that $r(\omega_0) = V_0 = \mathtt{dre}$ and $r(\omega_1) = V_1 = \mathtt{ad}$. To this end it is enough to set $\omega_0 = (\mathtt{d},\mathtt{d})(\varepsilon,\mathtt{r})(\varepsilon,\mathtt{e})$ and $\omega_1 = (\mathtt{a},\mathtt{a})(\mathtt{d},\mathtt{d})$. However, this is no more the case if we allow the operation $(\mathtt{a},\mathtt{ea})$ along the standard Levenshtein operations. Then we can consider the alignment $\omega' = (\mathtt{d},\mathtt{d})(\varepsilon,\mathtt{r})(\mathtt{a},\mathtt{ea})(\mathtt{d},\mathtt{d})$. In this situation it is impossible to decompose $\omega'$ as $\omega' = \omega'_0 \circ \omega'_1$ such that $r(\omega') = \mathtt{dre}$ and $r(\omega'_1) = \mathtt{ad}$.

Actually, this is the main, if not the only problem we need to address on our way of generalisation of the approximate search algorithm to the case, $\rho(Op) \geq 1$. Furthermore, it is not difficult to see a rather simple and natural solution of this problem. In fact "solution" is not the right word. We are not going to force the alignment $\omega'$ to have a decomposition into $\omega'_0$ and $\omega'_1$ with $r(\omega') = \mathtt{dre}$ and $r(\omega'_1) = \mathtt{ad}$. This is impossible! Rather, we are going to get around the problem. This means that instead of looking for subalignments with predetermined right sides, we are going to determine the right sides so that they comply with the alignment and that are not much longer/shorter than the required one. The main observation is that only one operation can stay on our way to achieve the desired decomposition. In our case, this is $(\mathtt{a},\mathtt{ea})$ and it has a right side of length $2 \leq \rho(Op)$. We shall see, that this simple fact already enables a decomposition that s not too far away from the one we had in the simple case, $\rho(Op) = 1$.

In the sequel, we shall first adapt Lemma 4.1.1 to the case $\rho(Op) \geq 1$. This would naturally give us the corresponding variant of Corollary 4.1.2 that will indicate how to proceed with an appropriate variant of Lemma 4.2.7 and Lemma 4.2.9. Our experience with the algorithm from Chapter 5 suggests that

these preparations should lead to a generalisation of the approximate search algorithm to arbitrary $\rho(Op) \geq 1$. We shall formally defend this hypothesis in Section 6.2.

The main ideas from this chapter were essentially presented in [20].

## 6.1 Decomposition Techniques for $\rho(Op) \geq 1$

We start this section with a variant of Lemma 4.1.1 which applies for the case $\rho(Op) \geq 1$. Actually, it also reflects a tiny still important detail that we have skipped in the case $\rho(Op) = 1$. Namely, we can always assume that the left subalignment $\omega_0$ is always maximal in the sense that $\omega_0 \in Op^* \circ (Op \setminus \Lambda) \cup \{\varepsilon\}$. The details are formally presented in the next lemma:

**Lemma 6.1.1** *Let $V = V_0 \circ V_1$ be a word, and $\omega$ be an alignment with right side $r(\omega) = V$. Then there exist alignments $\omega_0$ and $\omega_1$, and $o \in Op \cup \{\varepsilon\}$ such that:*

$$\begin{array}{rcl} \omega & = & \omega_0 \circ o \circ \omega_1 \\ |r(\omega_0)| & \leq & |V_0| \leq |r(\omega_0)| + |r(o)| \\ |r(\omega_1)| & \leq & |V_1| \leq |r(\omega_1)| + |r(o)| \\ \omega_0 \circ o & \notin & Op^* \circ \Lambda. \end{array}$$

*Furthermore, if $o \in Op$, the inequalities are strict.*

*Proof.* Let $\omega = op_1 \circ op_2 \circ \cdots \circ op_N$. We define $\omega^{(m)} = op_m \circ op_{m+1} \circ \cdots \circ op_N$ for $1 \leq m \leq N+1$. Since $\omega^{(N+1)} = \varepsilon$ and $\omega^{(1)} = \omega$ we deduce that $0 = |r(\omega^{(N+1)})| \leq |V_1| \leq |V| = |r(\omega^{(1)})|$. Consequently, we can find a minimal integer number $m_0$ such that $|r(\omega^{(m_0)})| \leq |V_1|$. We set $\omega_1 = \omega^{(m_0)}$. There are two possible cases, (i) $r(\omega_1)$ is of length $|V_1|$ and (ii) the length of $r(\omega_1)$ is strictly smaller than $|V_1|$. We consider each of these two cases:

1. $|V_1| = |r(\omega_1)|$. Then we set $o = \varepsilon$ and $\omega_0 = op_1 \circ \cdots \circ op_{m_0-1}$. Clearly we have that $\omega = \omega_0 \circ \omega_1$. It is also clear that $|V| = |r(\omega)| = |r(\omega_0)| + |r(\omega_1)| = |r(\omega_0)| + |V_1|$. Since $|V| = |V_0| + |V_1|$, we deduce that $|r(\omega_0)| = |V_0|$. Thus the inequalities hold.

2. $|V_1| > |r(\omega_1)|$. Then it should be clear that $m_0 > 1$. We set $o = op_{m_0-1}$ and $\omega_0 = op_1 \circ \cdots \circ op_{m_0-2}$. Again, a straightforward computation shows that $\omega = \omega_0 \circ o \circ \omega_1$. Furthermore, according to the definition of $m_0$ we have that:
$$|V_1| < |r(op_{m_0-1} \circ \omega_1)| = |r(o)| + |r(\omega_1)|.$$
From our assumption in this case we also have that $|V_1| > |r(\omega_1)|$.
Next $|V| = |r(\omega_0)| + |r(o)| + |r(\omega_1)|$. Since $|V| = |V_0| + |V_1|$ the inequalities for $r(\omega_1)$ imply that:
$$|r(\omega_0)| = |V| - |r(\omega_1)| - |r(o)| < |V| - |V_1| = |V_0|$$

| | $\rho(Op) = 1$ | $\rho(Op) = \rho \geq 1$ |
|---|---|---|
| $\omega$ | $\omega_0 \circ \omega_1$ | $\omega_0 \circ o \circ \omega_1$, $o = \varepsilon$ or $1 \leq \lvert r(o)\rvert \leq \rho$ |
| $c(\omega) \leq b_0 + b_1$ | $c(\omega_0) \leq b_0$ or $c(\omega_1) \leq b_1$ | $c(\omega_0) \leq b_0$ or $c(\omega_1) \leq b_1$ |
| $\mathfrak{A}(V)$ (concatenation) | $V = V_1 \circ x$ where $x \in \Sigma$ $\cup_{op=(U,x)\in Op} \mathfrak{A}(V') \circ op \circ \Lambda^*$ | $V = V_1 \circ V'$ where $1 \leq \lvert V'\rvert \leq \rho$ $\cup_{V'} \cup_{op=(U,V')\in Op} \mathfrak{A}(V') \circ op \circ \Lambda^*$ |
| $\mathfrak{A}^{\leq b}(V)$ $(V = V_0 \circ V_1)$ $(\ b = b_0 + b_1)$ | $\mathfrak{A}(V_0, b_0 \to V, b)$ $\mathfrak{A}(V, b, \leftarrow V_1, b_1)$ $\mathfrak{A}(V_0, b_0 \to V, b) \cup \mathfrak{A}(V, b \leftarrow V_1, b_1)$ | $\mathfrak{A}(V_0, b_0 \xrightarrow{k} V, b)$ $\mathfrak{A}(V, b \xleftarrow{k} V_1, b_1)$ $\bigcup_k \left(\mathfrak{A}(V_0, b_0 \xrightarrow{k} V, b) \cup \mathfrak{A}(V, b \xleftarrow{k} V_1, b_1)\right)$ |

Table 6.1: Main differences between the case $\rho(Op) = 1$ and $\rho(Op) \geq 1$.

and

$$|r(\omega_0)| + |r(o)| = |V| - |r(\omega_1)| > |V| - |V_1| = |V_0|.$$

Finally, in either case we have $\omega_0 \circ o = op_1 \circ \cdots \circ op_{m_0-1}$. Assume that $m_0 > 1$, then since $|r(op_{m_0-1})| + |r(\omega_1)| > |V_1| \geq |r(\omega_1)|$, we obtain that $|r(op_{m_0-1})| > 0$ and therefore $op_{m_0-1} \notin \Lambda$. Consequently, $\omega_0 \circ o \notin Op^* \circ \Lambda$. If $m_0 = 1$, then $\omega_0 \circ o = \varepsilon$ is of length 0. □

What Lemma 6.1.1 tells us, is that instead of the desired decomposition $\omega = \omega_0 \circ \omega_1$ guaranteed by Lemma 4.1.1, we can achieve a decomposition $\omega = \omega'_0 \circ o \circ \omega'_1$ such that the difference between the lengths of $r(\omega_i)$ and $r(\omega'_i)$ is bounded in the interval $\{0, 1, \ldots, \rho(Op) - 1\}$. In the case when $\rho(Op) = 1$ this set is a singleton. In the general case we should account for $\rho(Op)$ different possibilities. This naturally suggests the generalisation approach, sketched in Table 6.1.

Next, we shall get into more details.

**Corollary 6.1.2** *Let $b = b_0 + b_1$ be nonnegative rational numbers and $V = V_0 \circ V_1$ be a word with $n_0 = |V_0|$ and $n_1 = |V_1|$ such that $n_i > 0$ for $i = 0, 1$. Then an alignment $\omega$ with $r(\omega) = V$ has cost $c(\omega) \leq b$ only if there exist an integer $0 \leq k < \rho$ and alignments $\omega'_0$ and $\omega'_1$ such that $\omega = \omega'_0 \circ \omega'_1$ and:*

1. *either $r(\omega'_0) = I_1^{n_0-k}(V_0)$ and $c(\omega_0) \leq b_0$ and for $k > 0$ the first operation of $\omega'_1$ has right side of length at least $k + 1$,*
2. *or $r(\omega'_1) = I_{k+1}^{n_1}(V_1)$ and $c(\omega_1) \leq b_1$ and the last operation of $\omega'_0$ has right side of length at least $k + 1$.*

*Proof.* Let $\omega = \omega_0 \circ o \circ \omega_1$ be the decomposition of $\omega$ provided by Lemma 6.1.1. Then $c(\omega) = c(\omega_0) + c(o) + c(\omega_1)$. We consider two cases:

1. $b_0 \geq c(\omega_0)$, then we set $\omega'_0 = \omega_0$, $\omega'_1 = o \circ \omega_1$ and $k = |V_0| - |r(\omega_0)| \geq 0$. Since $|r(o)| + |r(\omega_0)| \geq |V_0|$ with equality only if $o = \varepsilon$, we deduce that

$k < \rho$. Now, since $r(\omega_0)$ is of length $n_0 - k$ we obtain that $r(\omega_0) = I_1^{n_0-k}(V_0)$. If $k \neq 0$, then $o \neq \varepsilon$ and $o$ is the first operation of $\omega_1'$. Since $|r(\omega_0)| + |r(o)| > |V_0|$ we obtain that $|r(o)| \geq k+1$ as required.

2. $b_0 < c(\omega_0)$, then since $b_1 + b_0 = b \geq c(\omega_0) + c(o) + c(\omega_1)$, we obtain that $c(\omega_1) < b_1$. Now we set $\omega_1' = \omega_1$ and $\omega_0' = \omega_0 \circ o$ and $k = n_1 - |r(\omega_1)| \geq 0$. Again, using that $|r(\omega_1)| + |r(o)| - n_1 \geq 0$ with equality only if $o = \varepsilon$, we deduce that $k < \rho$. Now a straightforward computation shows that $r(\omega_1') = I_{k+1}^{n_1}(V_1)$. Hence, if $k > 0$, then $o \neq \varepsilon$ and therefore $|r(o)| \geq k+1$. Therefore the last operation of $\omega_0'$ is of length at least $k+1$. Finally if $o = \varepsilon$, then according to Lemma 6.1.1 $\omega_0' \notin Op^* \circ \Lambda$ and thus the last operation of $\omega_0'$ is of length at least 1. □

Next we consider the extension of alignments of shorter words to alignments of longer words. Next lemma generalises Lemma 4.2.6 in a straightforward way:

**Lemma 6.1.3** *Let $Op$ be a set of operations with $\rho = \rho(Op)$ and $V \in \Sigma^*$ be a word of length $|V| = n > 0$, then:*

$$\mathfrak{A}(V) = \left( \bigcup_{j=n-\rho}^{n-1} \bigcup_{op=(U, I_{j+1}^n(V)) \in Op} \mathfrak{A}(I_1^j(V)) \circ op \right) \circ \Lambda^*$$

*Proof.* As in the proof of Lemma 4.2.6 we first consider an alignment $\omega \in \mathfrak{A}(V)$. Since $V$ is nonempty, $r(\omega) = V \neq \varepsilon$ and it can be uniquely decomposed as $\omega = \omega' \circ op \circ \omega_\varepsilon$ where $\omega_\varepsilon$ has right side the empty word and $r(op) \neq \varepsilon$. Hence $r(op)$ is some nonempty suffix of $V$ and consequently it has the form $op = (U, I_{j+1}^n(V))$. Since $1 \leq |r(op)| \leq \rho$ we deduce that $1 \leq n - j \leq \rho$ and therefore $n - \rho \leq j \leq n - 1$. It follows that $r(\omega') = I_1^j(V)$ and consequently $\omega' \in \mathfrak{A}(I_1^j(V))$. It is also clear that the fact $r(\omega_\varepsilon) = \varepsilon$ implies that $\omega_\varepsilon \in \Lambda^*$. Thus we have proved that:

$$\omega = \omega' \circ op \circ \omega_\varepsilon \in (\mathfrak{A}(I_1^j(V)) \circ op) \circ \Lambda^*$$

with $n - \rho \leq j \leq n - 1$ and $r(op) = I_{j+1}^n(V)$. This shows the inclusion from left to right.

The inclusion from right to left is clear. Indeed, if $\omega' \in \mathfrak{A}(I_1^j(V))$ and $r(op) = I_{j+1}^n(V)$, and $\omega_\varepsilon \in \Lambda^*$, then:

$$r(\omega' \circ op \circ \omega_\varepsilon) = r(\omega') \circ r(op) \circ r(\omega_\varepsilon) = I_1^j(V) \circ I_{j+1}^n(V) \circ \varepsilon = I_1^n(V) = V.$$

This means that $\omega' \circ op \circ \omega_\varepsilon \in \mathfrak{A}(V)$.

□

We should note that $I_1^j(V)$ with $j < |V|$, is a proper prefix of $V$. Hence in view of Lemma 4.2.4, Lemma 6.1.3 characterises $\mathfrak{A}(V)$ recursively.

Before we generalise the rest of the results from Chapter 4, let us reconsider our running example, $\omega' = (\mathtt{d}, \mathtt{d})(\varepsilon, \mathtt{r}), (\mathtt{a}, \mathtt{ea})(\mathtt{d}, \mathtt{d})$. As we have already

realised the operation $(\mathtt{a}, \mathtt{ea})$ prevents us to apply Lemma 4.1.1 and we needed Lemma 6.1.1 to handle this case. This shows that we have to put some efforts in order to control the operations with longer right side. To achieve this we introduce the following notions:

**Definition 6.1.4** Given a word $V \in \Sigma^*$ and an integer $k \in \mathbb{N}$ we define:

$$\overrightarrow{\mathfrak{A}}_k(V) = \mathfrak{A}(V) \cap (\{op \in Op \,|\, |r(op)| > k\} \circ Op^* \cup \{\varepsilon\})$$
$$\overleftarrow{\mathfrak{A}}_k(V) = \mathfrak{A}(V) \cap (Op^* \circ \{op \in Op \,|\, |r(op)| > k\} \cup \{\varepsilon\}).$$

Now we can use Lemma 6.1.2 in order to obtain a divide-and-conquer description of alignment sets. In the sequel we explain how to generalise Lemma 4.2.7:

**Lemma 6.1.5** *Let $\rho = \rho(Op)$ and $V = V_0 \circ V_1$ for some words $V_0, V_1 \in \Sigma^*$ with lengths $n_0$ and $n_1$, respectively such that $n_i \geq \rho$ for $i = 0, 1$ and $n = n_0 + n_1$. Let $b = b_0 + b_1$ for some rational numbers $b_0, b_1 \in \mathbb{Q}^+$, and:*

$$\overrightarrow{\mathfrak{B}}_k = \mathfrak{A}(V_0, b_0 \xrightarrow{k} V, b) = \begin{cases} \left(\mathfrak{A}^{\leq b_0}(I_1^{n_0-k}(V_0)) \circ \mathfrak{A}(I^n_{n_0-k+1}(V_0 \circ V_1))\right)^{\leq b} & \text{if } k = 0 \\ \left(\mathfrak{A}^{\leq b_0}(I_1^{n_0-k}(V_0)) \circ \overrightarrow{\mathfrak{A}}_k(I^n_{n_0-k+1}(V_0 \circ V_1))\right)^{\leq b} & \text{, else} \end{cases}$$

$$\overleftarrow{\mathfrak{B}}_k = \mathfrak{A}(V, b \xleftarrow{k} V_1, b_1) = \left(\overleftarrow{\mathfrak{A}}_k(I_1^{n_0+k}(V_0 \circ V_1)) \circ \mathfrak{A}^{\leq b_1}(I^{n_1}_{k+1}(V_1))\right)^{\leq b}$$

*Then:*

$$\mathfrak{A}^{\leq b}(V) = \bigcup_{k=0}^{\rho-1} \overrightarrow{\mathfrak{B}}_k \cup \bigcup_{k=1}^{\rho-1} \overleftarrow{\mathfrak{B}}_k.$$

*Proof.* We first prove the inclusion from left to right. Let $\omega \in \mathfrak{A}^{\leq b}(V)$. Thus, by Corollary 6.1.2 there are alignments $\omega'_0$ and $\omega'_1$ such that $\omega = \omega'_0 \circ \omega'_1$ and an integer $0 \leq k < \rho$ such that:

1. either $r(\omega'_0) = I_1^{n_0-k}(V_0)$ and $c(\omega'_0) \leq b_0$ and for $k > 0$ the first operation of $\omega'_1$ is with right side of length at least $k + 1$
2. or $r(\omega'_1) = I^{n_1}_{k+1}(V_1)$ and $c(\omega'_1) \leq b_1$ and the last operation of $\omega'_0$ is with right side of length at least $k + 1$.

We consider each of these two cases separately. In the first case we have that $\omega'_0 \in \mathfrak{A}^{\leq b_0}(I_1^{n_0-k}(V_0))$. Next since $r(\omega'_0) \circ r(\omega'_1) = V_0 \circ V_1$ and $r(\omega'_0) = I_1^{n_0-k}(V_0)$ we deduce that $r(\omega'_1) = I^{n_0}_{n_0-k+1}(V_0) \circ V_1 = I^n_{n_0-k+1}(V_0 \circ V_1)$ because the total length of $V_0 \circ V_1$ is $n$. Finally, for $k > 0$ the first operation of $\omega'_1$ has right hand side of length at least $k + 1$ which implies that:

$$\omega'_1 \in \begin{cases} \mathfrak{A}_k(I^n_{n_0-k+1}(V_0 \circ V_1)) \text{ if } k = 0 \\ \overrightarrow{\mathfrak{A}_k}(I^n_{n_0-k+1}(V_0 \circ V_1)) \text{ if } k > 0 \end{cases}$$

Finally, using that $c(\omega) = c(\omega'_0) + c(\omega'_1) \leq b$ we obtain that:

$$\omega = \omega'_0 \circ \omega'_1 \in \mathfrak{A}(V_0, b_0 \xrightarrow{k} V, b) = \overrightarrow{\mathfrak{B}}_k.$$

The second case is considered dually. We easily see that $\omega'_1 \in \mathfrak{A}^{\leq b_1}(I_{k+1}^{n_1}(V_1))$ and that $r(\omega'_0) = V_0 \circ I_1^k(V_1) = I_1^{n_0+k}(V_0 \circ V_1)$. Finally since the last operation of $\omega'_0$ has right side of length at least $k+1$ we deduce that:

$$\omega'_0 \in \overleftarrow{\mathfrak{A}_k}(I_1^{n_0+k}(V_0 \circ V_1))$$

which implies that:

$$\omega = \omega'_0 \circ \omega'_1 \in \overleftarrow{\mathfrak{A}_k}(I_1^{n_0+k}(V_0 \circ V_1)) \circ \mathfrak{A}^{\leq b_1}(I_{k+1}^{n_1}(V_1)).$$

Since $c(\omega) \leq b$ we conclude that $\omega \in \mathfrak{A}(V, b \overset{k}{\leftarrow} V_1, b_1) = \overleftarrow{\mathfrak{B}}_k$. These completes the proof of the inclusion from left to right. The inclusion from right to left is rather straightforward. It follows by the immediate observation that $V_0 \circ V_1 = I_1^{n_0-k}(V_0) \circ I_{n_0-k+1}^{n}(V_0 \circ V_1)$ and $V_0 \circ V_1 = I_1^{n_0+k}(V_0 \circ V_1) \circ I_{k+1}^{n_1}(V_1)$.

□

Let us recall that the approximate search algorithm presented in Chapter 5 essentially computed the edit-distance lists for the alignment sets $\mathfrak{A}(V_0, b_0 \rightarrow V, b)$ and $\mathfrak{A}(V, b \leftarrow V_1, b_1)$. The analogue of these sets in case $\rho(Op) \geq 1$ are the unions:

$$\begin{aligned} \cup_{k=0}^{\rho-1} \overrightarrow{\mathfrak{B}}_k &= \cup_{k=0}^{\rho-1} \mathfrak{A}(V_0, b_0 \overset{k}{\rightarrow} V, b) \text{ and} \\ \cup_{k=0}^{\rho-1} \overleftarrow{\mathfrak{B}}_k &= \cup_{k=0}^{\rho-1} \mathfrak{A}(V, b \overset{k}{\leftarrow} V_1, b_1). \end{aligned}$$

Thus it is natural to determine how these sets can be expressed recursively. The approach extends in a natural way Lemma 4.2.9. We first show the case "$\rightarrow$". The case of left extensions will be given at the end of this section.

**Lemma 6.1.6** *Let $V = V_0 \circ V_1$ be words with lengths $|V_i| = n_i \geq \rho$ and let $b = b_0 + b_1$ be nonnegative rational numbers. For $k + j \geq 0$ we define the alignment sets $\overrightarrow{\mathfrak{B}}_k^j = \mathfrak{A}(V_0, b_0 \overset{k}{\rightarrow} I_1^{n_0+j}(V_0 \circ V_1), b)$ as:*

$$\overrightarrow{\mathfrak{B}}_k^j = \begin{cases} \left(\mathfrak{A}^{\leq b_0}(I_1^{n_0-k}(V_0)) \circ \overrightarrow{\mathfrak{A}}_k(I_{n_0-k+1}^{n_0+j}(V_0 \circ V_1))\right)^{\leq b} & \text{for } k > 0 \\ \left(\mathfrak{A}^{\leq b_0}(I_1^{n_0}(V_0)) \circ \mathfrak{A}(I_{n_0+1}^{n_0+j}(V_0 \circ V_1))\right)^{\leq b} & \text{for } k = 0. \end{cases}$$

*Then for each $j \geq 1$ and $k < \rho$ it holds :*

$$\overrightarrow{\mathfrak{B}}_k^j = \left( \left( \bigcup_{j'=j-\rho}^{j-1} \bigcup_{op=(U, I_{n_0+j'+1}^{n_0+j}(V_0 \circ V_1)) \in Op} (\overrightarrow{\mathfrak{B}}_k^{j'} \circ op)^{\leq b} \right) \circ \Lambda^* \right)^{\leq b}$$

The result in this lemma concerns the values $j \geq 1$. However, it is important that $j = 0$ and $k = 0$, we have:

$$\overrightarrow{\mathfrak{B}}_0^0 = \left(\mathfrak{A}^{\leq b_0}(I_1^{n_0}(V_0)) \circ \mathfrak{A}(\varepsilon)\right)^{\leq b} = \left(\mathfrak{A}^{\leq b_0}(I_1^{n_0}(V_0)) \circ \Lambda^*\right)^{\leq b},$$

whereas for $j = -k$ and $k > 0$, we have that:

$$\overrightarrow{\mathfrak{B}}_k^j = \left(\mathfrak{A}^{\leq b_0}(I_1^{n_0-k}(V_0)) \circ \overrightarrow{\mathfrak{A}}_k(\varepsilon)\right)^{\leq b} = \mathfrak{A}^{\leq b_0}(I_1^{n_0-k}(V_0)))^{\leq b} = \mathfrak{A}^{\leq b_0}(I_1^{n_0-k}(V_0)).$$

Here the second equality follows by the definition of $\overrightarrow{\mathfrak{A}}_k(\varepsilon)$ which implies that the unique alignment in this set is the empty alignment, $\varepsilon$.

*Proof.* (of Lemma 6.1.6) The case $k = 0$ concerns alignment sets of the form:

$$\overrightarrow{\mathfrak{B}}_0^j = \left(\mathfrak{A}^{\leq b_0}(I_1^{n_0}(V_0)) \circ \mathfrak{A}(I_{n_0+1}^{n_0+j}(V_0 \circ V_1))\right)^{\leq b}.$$

In this sense they are similar to the alignment sets $\mathfrak{A}(V_0, b_0 \rightarrow V, b)$ considered in Chapter 4. Thus they can be handled in the same way but instead of Lemma 4.2.6 we have to use Lemma 6.1.3 tailored for $\rho \geq 1$.

In the sequel we concentrate our attention on the case $j \geq 1$ and $0 < k < \rho$. Let $\omega$ be an alignment in $\overrightarrow{\mathfrak{B}}_k^j$. Then, $\omega$ can be uniquely decomposed as $\omega = \omega_0' \circ \omega_1 \circ op \circ \omega_\varepsilon$ such that:

$$\omega_0' \in \mathfrak{A}^{\leq b_0}(I_1^{n_0-k}(V_0)) \text{ and } \omega_\varepsilon \in \Lambda^* \text{ and } op \notin \Lambda.$$

Therefore $r(op) = I_{n_0+j'+1}^{n_0+j}(V_0 \circ V_1)$ for some $j - \rho \leq j' < j$. Now, there are two possible cases. Either $\omega_1 = \varepsilon$ or $\omega_1 \neq \varepsilon$. In the former case, we conclude that the first operation of $\omega_1$ is the same as the first operation of $\omega_1 \circ op \circ \omega_\varepsilon$. Therefore the facts: $\omega_1 \circ op \circ \omega_\varepsilon \in \overrightarrow{\mathfrak{A}}_k(I_{n_0-k+1}^{n_0+j}(V_0 \circ V_1))$, $r(op) = I_{n_0+j'+1}^{n_0+j}(V_0 \circ V_1)$ and $r(\omega_\varepsilon) = \varepsilon$ imply that $\omega_1 \in \overrightarrow{\mathfrak{A}}_k(I_{n_0-k+1}^{n_0+j'}(V_0 \circ V_1))$. Therefore we conclude that: $\omega_0' \circ \omega_1 \in \mathfrak{A}^{\leq b_0}(I_1^{n_0-k}(V_0)) \circ \overrightarrow{\mathfrak{A}}_k(I_{n_0-k+1}^{n_0+j'}(V_0 \circ V_1))$. Since $c(\omega_0' \circ \omega_1) \leq c(\omega) \leq b$ we obtain that $\omega_0' \circ \omega_1 \in \overrightarrow{\mathfrak{B}}_k^{j'}$ and consequently: $\omega \in \left((\overrightarrow{\mathfrak{A}}_k^{j'} \circ op)^{\leq b} \circ \Lambda^*\right)^{\leq b}$.

The case when $\omega_1 = \varepsilon$ is a particular one. In this case $\omega_1 \circ op = op$ and the assumption that $\omega_1 \circ op \circ \omega_\varepsilon \in \overrightarrow{\mathfrak{A}}_k(I_{n_0-k+1}^{n_0+j}(V_0 \circ V_1))$ implies that $|r(op)| \geq k+1$. As in the first case we also have that $r(op) = I_{n_0+j'+1}^{n_0+j}(V_0 \circ V_1)$ for some $j - \rho \leq j' < j$. Taking into account that $|r(op)| \geq k+1$ and $|r(op)| = j - j'$ we get that $|r(op)| + j' = j > 0$ and $|r(op)| \geq k+1$. Therefore we get that $\omega_0' \in \mathfrak{A}^{\leq b_0}(I_1^{n_0+j'}(V_0)) = \overrightarrow{\mathfrak{B}}_k^{j'}$ and we can conclude the proof as above.

The inclusion from right to left follows by an immediate computation.

□

If we now unite the sets $\overrightarrow{\mathfrak{B}}_k^j$ for $k < \rho$ we get the following corollary:

**Corollary 6.1.7** *In the notations of Lemma 6.1.6 we determine:*

$$\overrightarrow{\mathfrak{B}}^j = \cup_{k=0}^{\rho-1} \overrightarrow{\mathfrak{B}}_k^j.$$

*Then:*

$$\overrightarrow{\mathfrak{B}}^j = \left(\left(\bigcup_{j'=j-\rho}^{j-1} \bigcup_{op=(U, I_{n_0+j'}^{n_0+j}(V_0 \circ V_1)) \in Op} (\overrightarrow{\mathfrak{B}}^{j'} \circ op)^{\leq b}\right) \circ \Lambda^*\right)^{\leq b}$$

*for every integer number* $j \geq 1$.

□

Since the notion and facts derived for edit-distance lists and reversed alignments from Section 4.3 and 4.4, respectively, do not rely on the specificity of $\rho(Op) = 1$ they extend immediately to the general case, $\rho(Op) \geq 1$. It is only in Lemma 4.4.7 that we essentially use the assumption $\rho(Op) = 1$. This is due to the necessity to use the concatenation of an alignment set with some operations in order to express the terms $\mathfrak{A}(V)$. This imposes the use of Lemma 4.2.6. In case that $\rho(Op) \geq 1$ we should apply Lemma 6.1.3, instead. Thus we get the following recursive description of the sets $\cup_k \mathfrak{A}(V, b \overset{k}{\leftarrow} V_1, b_1)$.

**Lemma 6.1.8** *Let* $V = V_0 \circ V_1$ *be a word with lengths* $|V_i| = n_i \geq \rho$ *and let* $b = b_0 + b_1$ *be nonnegative rational numbers. Let* $\overleftarrow{\mathfrak{B}}_k^j$ *be the alignments:*

$$\overleftarrow{\mathfrak{B}}_k^j = \mathfrak{A}(I^n_{n_0-j+1}(V_0 \circ V_1), b \overset{k}{\leftarrow} V_1, b_1) = \left(\overleftarrow{\mathfrak{A}}_k(I^{n_0+k}_{n_0-j+1}(V_0 \circ V_1)) \circ \mathfrak{A}^{\leq b_1}(I^{n1}_{k+1}(V_1))\right)^{\leq b}$$

*Then for each* $j \geq 1$ *and* $k < \rho$ *it holds :*

$$\overleftarrow{\mathfrak{B}}_k^j = \left(\Lambda^* \circ \left(\bigcup_{j'=j-\rho}^{j-1} \bigcup_{op=(U, I^{n_0-j'}_{n_0-j+1}(V_0 \circ V_1)) \in Op} (op \circ \overleftarrow{\mathfrak{B}}_k^{j'})^{\leq b}\right)\right)^{\leq b}$$

*Proof.* Let $W_0 = V_1^{rev}$ and $W_1 = V_0^{rev}$ and $n = n_0 + n_1$. Therefore position $l$ in $W_0 \circ W_1$ is position $l$ in $(V_0 \circ V_1)^{rev}$ and thus it holds the same character as the character at position $n - l + 1$ in $V_0 \circ V_1$. With this remark it is rather straightforward that $(I^{n_0+k}_{n_0-j+1}(V_0 \circ V_1))^{rev} = I^{n_1+j}_{n_1-k+1}(W_0 \circ W_1)$ and $(I^{n_1}_{k+1}(V_1))^{rev} = I_1^{n_1-k}(W_0)$. Now using Lemma 4.4.6 we see that:

$$\begin{aligned} \mathfrak{A}^{rev}(I^{n1}_{k+1}(V_1)) &= \mathfrak{A}_{Op^{rev}}(I_1^{n_1-k}(W_0)) \\ \overleftarrow{\mathfrak{A}}_k^{rev}(I^{n_0+k}_{n_0-j+1}(V_0 \circ V_1)) &= \overrightarrow{\mathfrak{A}}_{Op^{rev},k}(I^{n_1+j}_{n_1-k+1}(W_0 \circ W_1)). \end{aligned}$$

Therefore for $k > 0$ we have that:

$$\overleftarrow{B}_k^{j,rev} = \mathfrak{A}_{Op^{rev}}(W_0, b_1 \overset{k}{\rightarrow} I_1^{n_1+j}(W_0 \circ W_1), b)$$

and applying Lemma 6.1.6 to the sets $\mathfrak{B}_k^{j,rev}$, and then the reverse operation we get the result. The case when $k = 0$ is similar.

□

As a by-product of the above prove we obtain:

**Corollary 6.1.9** *In the notions of Lemma 6.1.8 we set* $\overleftarrow{\mathfrak{B}}^j = \cup_{k=0}^{\rho-1} \overleftarrow{\mathfrak{B}}_k^j$. *Then it holds:*

$$\overleftarrow{\mathfrak{B}}^{j,rev} = \left(\left(\bigcup_{j'=j-\rho}^{j-1} \bigcup_{op=(U, I^{n_1+j}_{n_1+j'+1}(V_1^{rev} \circ V_0^{rev})) \in Op^{rev})} (\overleftarrow{\mathfrak{B}}^{j',rev} \circ op)^{\leq b}\right) \circ (\Lambda^{rev})^*\right)^{\leq b}.$$

□

| $\rho$ | $\rho(Op) = 1$ | $\rho(Op) \geq 1$ |
|---|---|---|
| nodes of $\mathcal{T}(V)$ | $\alpha \in \{0,1\}^*$, $N_\alpha \geq 0$ | $\alpha \in \{0,1\}^*$, $N_\alpha \geq 4(\rho - 1)$ |
| queries associated with a node $\alpha$ | single $(V_\alpha, qN_\alpha)$ | $\rho^2$ pairs $(k_0, k_1)$ $(V_\alpha[k_0; k_1], qN_\alpha)$ |
| initialisation of the leaves | $O(\|V\|)$ no bookkeeping | $O(\|V\| + \#answers)$ some additional bookkeeping |
| edit-distance lists | $L(\alpha)$, $L_r(\alpha, j)$, $L_l(\alpha, j)$ | $L(\alpha, k_0, k_1)$, $L_r(\alpha, k_0, j)$, $L_l(\alpha, k_1, j)$ $k_0, k_1 \leq \rho - 1$ |
| representation | buckets $B_\alpha^j$ and tries $\mathcal{T}_\alpha$ | buckets $B_\alpha^{k,j}$, $k \leq \rho - 1$ and tries $\mathcal{T}_\alpha$ |
| computation of extensions | based on Lemma 4.4.7 | based on Lemma 6.1.8 |
| complexity (asymptotical) | $\sum_{\alpha,j}(\|L_r(\alpha, j)\| + \|L_l(\alpha, j)\|)$ $+ \sum_{\alpha, U \in Dom(L(\alpha))} \|U\|)$ | $\sum_{k=0}^{\rho-1} \sum_{\alpha,j}(\|L_r(\alpha, k, j)\| + \|L_l(\alpha, k, j)\|)$ $+ \sum_{k_0,k_1=0}^{\rho-1} \sum_{\alpha, U \in Dom(L(\alpha,k_0,k_1))} \|U\|)$ |

Table 6.2: Comparison between the main characteristics of the basic and generalised algorithms.

## 6.2 Approximate Search Algorithm, $\rho(Op) \geq 1$

As Table 6.1 indicates the main difference between the case $\rho(Op) = 1$ and $\rho(Op) \geq 1$ is that in case $\rho(Op) = 1$ we have one (or two) object(s) that we have to compute, whereas in case $\rho(Op) \geq 1$ we have unions of $\rho(Op)$ or $2\rho(Op)$ to account for. Furthermore, Lemma 4.2.7 did not impose any constraints on the lengths of the words $V_0$ and $V_1$. However, the validity of its analogue, Lemma 6.1.5, requires that both $|V_0| \geq \rho(Op) - 1$ and $|V_1| \geq \rho(Op) - 1$.

It turns out that these constraints can be easily reflected in our original algorithm by some additional bookkeeping that increases the complexity of the algorithm with a factor $O(\rho^2)$. Considering $\rho = \rho(Op)$ as a global constant, this does not change the asymptotical complexity of the algorithm.

The differences between the basic and the generalised algorithm are summarised in Table 6.2 and we shall highlight the most essential ones in the sequel.

### 6.2.1 Organisation of the Query Tree $\mathcal{T}(V)$. Initialisation

In the general case when $Op$ may contain arbitrary operations $op$, i.e. $|r(op)|$ is not necessarily less than 2, we cannot use the Lemma 4.2.7 but rather Lemma 6.1.5. This means that the alignment sets $\mathfrak{A}(V_\alpha)^{\leq b_\alpha}$ defined in Chapter 5 cannot be expressed in general only by the means of the corresponding alignment sets for the nodes $\alpha 0$ and $\alpha 1$. What we need to do is to consider some more alignment sets which leave a 'slot' in $V_\alpha$ for another operation either at the end or at the beginning. In order to have enough space for a prefix and for a suffix slot, we need to guarantee that the word $V_\alpha$ is long enough. This is the first constraint which we reflect in the construction of the search tree, $\mathcal{T}(V)$.

We modify the construction from the previous chapter as follows:

1. $V_\varepsilon = V$ and $\varepsilon$ is the root of $\mathcal{T}(V)$.
2. if $V_\alpha$ and $\alpha$ are defined, $q|V_\alpha| \geq 1$ and $|V_\alpha| \geq 4(\rho - 1)$ we define $V_{\alpha 0}$ and $V_{\alpha 1}$ such that:

$$V_\alpha = V_{\alpha 0} \circ V_{\alpha 1}$$
$$0 \leq |V_{\alpha 0}| - |V_{\alpha 1}| \leq 1.$$

We set $\alpha 0$ to be the left child of $\alpha$ and $\alpha 1$ to be the right child of $\alpha$.

Again, we denote with $N_\alpha$ the length of $V_\alpha$ and we use $b_\alpha = qN_\alpha$ as the threshold corresponding to the query word $V_\alpha$. The only difference between the constructions in Section 5.2 and the one in this section is the constraint $N_\alpha \geq 4(\rho - 1)$ next to $b_\alpha \geq 1$. However, if $\rho = 1$, then $b_\alpha \geq 1$ already implies that $N_\alpha > 0$. Therefore the construction above generalises the construction of the search tree in the Section 5.2.

Before associating queries with the nodes $\alpha$ as we did in the simple case, let us establish an easy still useful fact about the leaves of the search trees $\mathcal{T}(V)$.

**Lemma 6.2.1** *Assume that $|V| = N \geq 2(\rho - 1)$ and let $\alpha$ be a leaf in $\mathcal{T}(V)$. Then $N_\alpha \geq 2(\rho - 1)$.*

*Proof.* First, if $\alpha = \varepsilon$, i.e. we have that the tree is a trivial one, the statement becomes apparent because $N \geq 2(\rho - 1)$. Thus, let us consider the case when $\alpha$ has a parent $\beta$ in $\mathcal{T}(V)$ and let $\eta \in \{0, 1\}$ be such that $\alpha = \beta \circ \eta$. Now, since $\beta$ is an inner node of $\mathcal{T}(V)$ we deduce that $N_\beta \geq 4(\rho - 1)$. Consequently, if we denote $\overline{\eta} = 1 - \eta$ we obtain:

$$N_{\beta\overline{\eta}} + N_\alpha = N_\beta \geq 4(\rho - 1) \text{ and } |N_{\beta\overline{\eta}} - N_\alpha| \leq 1.$$

Clearly, this implies that $N_\alpha \geq \frac{N_\beta - 1}{2} \geq 2(\rho - 1)$ in the case when $N_\beta > 4(\rho - 1)$. If, however, $N_\beta = 4(\rho - 1)$, then $N_\beta$ is even and therefore $N_{\beta\overline{\eta}}$ and $N_\alpha$ have the same parity which implies that $N_{\beta\overline{\eta}} = N_\alpha = 2(\rho - 1)$.

□

Unlike the situation when $\rho = 1$ where we defined a single query per node $\alpha$, in case $\rho > 1$ we define $\rho^2$ queries for every node $\alpha$. Specifically, for every two nonnegative integers $k_0, k_1 < \rho$ and a word $V_\alpha$ we introduce:

$$V_\alpha[k_0; k_1] = I_{k_0+1}^{N_\alpha - k_1}(V_\alpha)).$$

What this expression means, is that $V_\alpha[k_0; k_1]$ is the infix of $V_\alpha$ after removing the first $k_0$ and the last $k_1$ characters. Now the natural question is whether this is possible. And indeed we have not removed fantom-characters since by Lemma 6.2.1 we have that $|V_\alpha| \geq 2(\rho - 1) \geq k_0 + k_1$.

Now with each node $\alpha$ we can associate the queries $V_\alpha[k_0; k_1]$ specified as:

`Given:` $\mathcal{L} \subseteq \Sigma^*$ `regular language,`
$d = (Op, c)$ `an edit-distance,`
$q \in (0; 1)$ `a threshold parameter`
`Input:` $V_\alpha \in \Sigma^*, k_0, k_1 \in \{0, 1, \ldots, \rho - 1\}$
`Output:` $\{U \in Inf(\mathcal{L}) \mid d(U, V_\alpha[k_0; k_1]) \leq q|V_\alpha|\}$.

We stress that the threshold is still determined by $V_\alpha$ and it might be a little bit greater than the threshold $q|V_\alpha[k_0; k_1]|$ which the word $V_\alpha[k_0; k_1]$ would have determined. Nevertheless, we cannot do any better if we rely only on Lemma 6.1.5.

Again the objective for the node $\alpha$ is to resolve all the queries $V_\alpha[k_0; k_1]$ for $k_0, k_1 < \rho$ which are attributed to it. We specify this task as computing the edit-distance lists $L(\alpha, k_0, k_1) = L[V_\alpha[k_0; k_1]]$ which $\mathcal{L}$-represent the alignment sets:

$$(\mathfrak{A}(V_\alpha[k_0; k_1]))^{\leq b_\alpha}.$$

**Lemma 6.2.2** *Assume that we have an infix automaton $\mathcal{IA}$ for the language $Inf(\mathcal{L})$ and a precomputed index $\mathcal{I}$ which maps every triple $(W, k_0, k_1)$ where $W \in \Sigma^*$ is a word of length $|W| < 4(\rho - 1)$, and $k_0, k_1 < \rho$ are integers to $\mathcal{I}(W, k_0, k_1) = L[W[k_0; k_1]]$. Let $V \in \Sigma^*$ be arbitrary. Then we can compute the edit-distance lists $L(\alpha, k_0, k_1)$ where $\alpha$ ranges over the leaves of the search tree $\mathcal{T}(V)$ in time:*

$$O\left(\rho^2 N + \sum_{\alpha \text{ leaf of } \mathcal{T}(V)} \sum_{k_0, k_1 < \rho} |L(\alpha, k_0, k_1)|\right)$$

*Proof.* Consider a leaf $\alpha$ in $\mathcal{T}(V)$. There are two possibilities: (i) it satisfies $qN_\alpha < 1$ or (ii) it satisfies $N_\alpha < 4(\rho-1)$. In the former case using the automaton $\mathcal{IA}$, we can proceed as in Lemma 5.2.1. Thus the list $L(\alpha, k_0, k_1)$ is either a nowhere defined function or defined only for $V_\alpha[k_0; k_1]$ and attaining value 0 at this word. We can resolve which of these two cases applies in time $O(N_\alpha)$. In the latter case, when $N_\alpha < 4(\rho - 1)$, the triple $(V_\alpha, k_0, k_1)$ is in the domain of $\mathcal{I}$ and we can access the edit-distance list:

$$\mathcal{I}(V_\alpha, k_0, k_1) = L[V_\alpha[k_0; k_1]] = L(\alpha, k_0, k_1)$$

at the cost of a single traversal of $V_\alpha[k_0, k_1]$ and the size of the edit-distance list. Summing up we get that the total time required for the initialisation is within the bounds

$$O\left(\sum_{\alpha \text{ leaf of } \mathcal{T}(V)} \sum_{k_0, k_1 < \rho} (N_\alpha + |L(\alpha, k_0, k_1)|)\right)$$

and since $\sum_{\alpha \text{ leaf of } \mathcal{T}(V)} N_\alpha = N$ and there are $\rho^2$ pairs $(k_0, k_1)$ we deduce the initialisation step described above requires:

$$O\left(\rho^2 N + \sum_{\alpha \text{ leaf of } \mathcal{T}(V)} \sum_{k_0, k_1 < \rho} |L(\alpha, k_0, k_1)|\right)$$

for all the leaves $\alpha$ and $k_0, k_1 < \rho$. □

Procedures ApproximateSearchInitialiseGeneral executes the main initialisation step as described in the proof of the previous Lemma. The procedure PreComputeIndex uses the algorithm of Mihov and Schulz, [42], i.e. procedure Search2, in order to compute the required index.

```
ApproximateSearchInitialiseGeneral(IA_r, V, N, q, α, ρ, I)
//we need additional an inverted index, I, and the parameter ρ
   N_α ← N
   V_α ← V
   if qN < 1 or N < 4(ρ − 1) then
      if qN < 1 then
         for k_0 = 0 to ρ − 1 do
            for k_1 = 0 to ρ − 1 do
               ⟨T, L⟩ ← InitialiseWithTraverse(IA_r, V[k_0 + 1..N − k_1])
               ⟨T(α, k_0, k_1), L(α, k_0, k_1) ← ⟨T, L⟩
            done
         done
      else // qN ≥ 1 and N < 4(ρ − 1)
         for k_0 = 0 to ρ − 1 do
            for k_1 = 0 to ρ − 1 do
               ⟨T(α, k_0, k_1), L(α, k_0, k_1)⟩ ← RetrieveFromIndex(I, V[k_0+1..N−
k_1], ⌊qN⌋)
            done
         done
   else
      N_left ← ⌈N/2⌉
      N_right ← ⌊N/2⌋
      V_left ← V[1..N_left]
      V_right ← V[N_left + 1..N]
      ApproximateSearchInitialiseGeneral(IA, V_left, N_left, q, α0, ρ, I)
      ApproximateSearchInitialiseGeneral(IA, V_right, N_right, q, α1, ρ, I)

InitialiseWithTraverse(A, V)
   T ←empty trie with (new) root r
   s ←the initial state of A
   st ← TraverseAutomaton(A, s, V)
   L ← ∅
   if st is defined then
      u ← TraverseTrie(T, r, V)
      st(u) ← st
      B[N] ←new empty bucket
      B[N].Insert(⟨u, 0⟩
      L.Append(B[N])
   return ⟨T, L⟩
```

```
RetrieveFromIndex(I, V, b)
   ⟨IT[.], IT(.), IL(.)⟩ ← I
   s ←the initial state of IT[b]
   u ← TraverseAutomaton(IT[b], V)
   if u is defined then
      return ⟨IT(u), IL(u)⟩
   else
      return ⟨∅, ∅
   fi

PreComputeIndex(A_r, A_l, ⟨Op, c⟩, ρ, q)
   N ← 4(ρ − 1) − 1
   s ←the initial state of A_r
   if qN < 1 return ∅
   for V ∈ Σ^{≤N} do
      for b = 0 to ⌊qN⌋ do
         IT[b] ←empty trie with root r[b]
         N − hood ← Search2(A_r, A_l, V, b, ⟨Op, c⟩)
         if N − hood ≠ ∅
            u ← TraverseTrie(IT[b], r[b], V
            IT(u) ←empty trie with root r(u)
            IL(u) ← ∅
            for U ∈ N − hood in deg-lex order do
               u' ← TraverseTrie(IT(u), r(u), U)
               st(u) ← TraverseAutomaton(A_r, s, U
               c(u') ← Edit − Distance(U, V, ⟨Op, c⟩)
               if B[|U|] is not the last bucket in IL(u)
                  B[|U|] ←new empty bucket
                  IL(u).Append(B[|U|])
               fi
               B[|U|].Append(⟨u', st(u), c(u')⟩)
            done
         fi
      done
   done
   return ⟨IT[.], IT(.), IL(.)⟩
```

**Remark 6.2.3** The idea for an index $\mathcal{I}$ was used by Myers, [47], and Baeza-Yates and Navarro, [49]. However, their indices are tailored for finite languages (infixes of a long text) and their size depends on the length of the language. The assumptions in Lemma 6.2.2 are weaker. Specifically the minimal index $\mathcal{I}$ which one can select has a finite domain which is determined by $q$ and $\rho$ and involves no information for longer words of the language $\mathcal{L}$. Furthermore every query has a finite number of answers which might depend on the query word, though. Nevertheless we can rely that such a finite index $\mathcal{I}$ with the properties as described in the Lemma exists. In Section 6.2.4 and in the Appendix we are going to find an upper bound for its size in terms of $\rho$ and $q$.

### 6.2.2 Extension Steps

The necessity to compute more queries in a single node of the query tree, $\mathcal{T}(V)$, imposes that we compute more edit-distance lists and more intermediate edit-distance lists. Specifically for every pair $k_0, k_1 \leq \rho - 1$ and each node $\alpha$ of $\mathcal{T}(V)$ we need to compute a edit-distance list $L(\alpha, k_0, k_1)$ that $\mathcal{L}$-represents the alignment set:

$$\mathfrak{A}^{\leq b_\alpha}(V_\alpha[k_0; k_1]).$$

In order to achieve this for the non-leaf nodes, $\alpha$, we shall first compute the edit-distance lists $L_r(\alpha, k_0, j)$ that $\mathcal{L}$-represents the alignment sets:

$$\cup_{k=0}^{\rho-1} \overrightarrow{\mathfrak{A}}(I_{k_0+1}^{N_{\alpha_0}}(V_{\alpha_0}), b_{\alpha 0} \xrightarrow{k} I_{k_0+1}^{N_{\alpha_0}+j}(V_{\alpha_0} \circ V_{\alpha 1}), b_\alpha)$$

and similarly edit-distance lists $L_l(\alpha, k_1, j)$ that $\mathcal{L}$-represents the alignment sets:

$$\cup_{k=0}^{\rho-1} \overleftarrow{\mathfrak{A}}(I_{N_{\alpha 0}-j+1}^{N_\alpha - k_1}(V_{\alpha_0} \circ V_{\alpha 1}), b_\alpha \xleftarrow{k} I_{k+1}^{N_{\alpha_1}-k_1}(V_{\alpha 1}), b_{\alpha 1}).$$

As soon as we dispose on the edit-distance lists $L_r(\alpha, k_0, N_{\alpha 1} - k_1)$ and $L_l(\alpha, k_1, N_{\alpha 0} - k_0)$ we can find their join and obtain the edit-distance list $L(\alpha, k_0, k_1)$. In fact, due to the reverse manner we handle the left extension, we shall rather have $L_l^{rev}(\alpha, k_1, N_{\alpha 0} - k_0)$. This is exactly what our procedure for union of edit-distance lists from Chapter 5 is designed for.

We compute the edit-distance lists in increasing order of $j$. Namely, for fixed $\alpha$ and $k_0$ we compute the edit-distance lists $L_r(\alpha, k_0, j)$ for $j = 0, 1, \ldots, N_{\alpha 1}$ and similarly for fixed $\alpha$ and $k_1$ we compute the edit-distance lists $L_l^{rev}(\alpha, k_1, j)$ for $j = 1, 2, \ldots, N_{\alpha 0}$. The reason for the asymmetry of left and right extensions is that a left extension never starts with an operation $op \in \Lambda$ and hence we do not need to compute $L_l^{rev}(\alpha, k_1, 0)$ which would correspond to the edit-distance list $L_r(\alpha, k_0, 0)$.

As in the algorithm from Chapter 5, the computation of edit-distance lists $L_r(\alpha, k_0, j)$ and $L_l^{rev}(\alpha, k_1, j)$ is reduced to the two main steps: *extension* and $\varepsilon$-*closure*. These two steps are applied interchangingly as prescribed by Lemma 6.1.8. In particular given that the edit-distance lists $L_r(\alpha, k_0, j')$ represent the alignment sets:

$$\overrightarrow{\mathfrak{B}}^{j'} = \cup_{k=0}^{\rho-1} \mathfrak{A}(I_{k_0+1}^{N_{\alpha 0}}(V_{\alpha 0}), b_{\alpha 0} \xrightarrow{k} I_{k_0+1}^{N_{\alpha 0}+j'}(V_{\alpha 0} \circ V_{\alpha 1}), b)$$

for $j' = j-1, j-2, \ldots j-\rho$ we first compute the edit-distance list $L'_r(\alpha, k_0, j)$ that $\mathcal{L}$-represents the alignment set:

$$\widetilde{\mathfrak{B}}^j = \bigcup_{j'=j-\rho}^{j-1} \bigcup_{op=(U, I_{N_{\alpha 0}+j'+1}^{N_{\alpha 0}+j}(V_{\alpha 0} \circ V_{\alpha 1})) \in Op} (\overrightarrow{\mathfrak{B}}^j \circ op)^{\leq b_\alpha}.$$

In a second step we compute the edit-distance list $L_r(\alpha, k_0, j)$ that $\mathcal{L}$-represents the alignment set:

$$(\widetilde{\mathfrak{B}}^j \circ \Lambda^*)^{\leq b_\alpha}.$$

With respect to Lemma 6.1.6 we have that $(\widetilde{\mathfrak{B}}^j \circ \Lambda^*)^{\leq b_\alpha} = \overrightarrow{\mathfrak{B}}^j$ and thus $L_r(\alpha, k_0, j)$ will be correctly computed.

A subtle detail is the initialisation of the edit-distance lists $L_r(\alpha, k_0, j')$ for $j' \leq 0$. However, it is easy to see that $L_r(\alpha, k_0, -k)$ for $0 < k < \rho$ would be consistent with the recurrence if it $\mathcal{L}$-represents the alignment set:

$$\mathfrak{A}^{\leq b_{\alpha 0}}(V_{\alpha 0}[k_0; k]).$$

And $L_r(\alpha, k_0, 0)$ should $\mathcal{L}$-represent the alignment set:

$$(\mathfrak{A}^{\leq b_{\alpha 0}}(V_{\alpha 0}[k_0; 0]) \circ \Lambda^*)^{\leq b_\alpha}.$$

Thus, we can set $L_r(\alpha, k_0, -k) = L(\alpha 0, k_0, k)$ for $0 < k < \rho$ and we obtain $L_r(\alpha, k_0, 0)$ with a single $\varepsilon$-step applied on the edit-distance list $L(\alpha, k_0, 0)$.

*$\varepsilon$-closure steps.* At this point we should clarify the $\varepsilon$-closure step. Since it considers only operations $op \in \Lambda$ it is not concerned by the value $\rho(Op) \geq 1$. Thus, regardless which of the two representations of edit-distance lists (with tries and buckets, or with deg-lex ordered lists) we use, the $\varepsilon$-step can be performed in the same (respective) way as in the basic case $\rho(Op) = 1$. Thus, we have:

**Lemma 6.2.4** *Let $Size'(k_0, j)$ be the size of the representation of the edit-distance list $L'_r(\alpha, k_0, j)$ and $Size(k_0, j)$ be the size of $L_r(\alpha, k_0, j)$, then $L_r(\alpha, k_0, j)$ can be computed in time $O(Size(k_0, j) + Size'(k_0, j))$.*

□

*Extension steps.* The only difference between the basic case $\rho(Op) = 1$ and the case $\rho(Op) \geq 1$ is the presence of a parameter $j'$ that varies between $j - \rho$ and $j - 1$. Actually, if $\rho = 1$, then $j' = j - 1$ is the only integer in this range. Hence, our algorithm from Chapter 5 should be modified as to reflect this detail. Thus, if $B_\alpha^{k_0,j'}[n]$ are the buckets representing the edit-distance lists $L(\alpha, k_0, j')$, the allocation of new buckets, $B_\alpha^{k_0,j}[n]$ in the representation $L(\alpha, k_0, j)$, is carried out on the bases not only of $j' = j - 1$ but with respect to $j' = j - 1, j - 2, \ldots, j - \rho$. Recall, that $\mu = \max\{|l(op)| \,|\, op \in Op\}$. Thus, we first compute $\rho \times (\mu + 1)$ sets:

$$S(j', l) = \{n + l \,|\, B_\alpha^{k_0,j'}[n] \text{ is a (nonempty) bucket for } L(\alpha, k_0, j')\}.$$

Here $j' = j - 1, j - 2, \ldots, j - \rho$ and $l = 0, 1, \ldots, \mu$. Assuming that the buckets are originally increasingly sorted with respect to $n$, we get the sets $S(j', l)$ in increasing order as well. Then we can merge them in a single set $S = \{s_1 < s_2 < \cdots < s_{|S|}\}$ using a merge sort. Finally, we create the buckets $B_\alpha^{k_0,j}[s_m]$ for $m \leq |S|$ and we assign $l$-pointers for each bucket $B_\alpha^{k_0,j'}[n]$ with $j' < j$ and each $l \leq \mu$ which indicates the bucket $B_\alpha^{k_0,j}[n + l]$. Similarly to Lemma 5.3.1 we can prove that this procedure can be efficiently performed, see procedure InitialiseBucketsGeneral:

**Lemma 6.2.5** *Let $M_{j'}$ be the number of buckets in $L_r(\alpha, k_0, j')$, then the creation of the buckets $B_\alpha^{k_0,j}[n]$ can be carried out in time $O\left(|M_{j-1}|\rho(\mu + 1)\log(\rho(\mu + 1))\right)$. Furthermore, each bucket $B_\alpha^{k_0,j-1}[n_m]$ is assigned with an $l$-pointer to a valid bucket $B_\alpha^{k_0,j}[n_m + l]$.*

□

```
InitialiseBucketsGeneral(⟨Op, c⟩, η, j, L)
//L is an array of lists, e.g. L_r(α,k,.) or L_l(α,k,.)
    μ ← max{|l(op)| | op ∈ Op}
    ρ ← max{|r(op)| | op ∈ Op}
    for k = 1 to ρ do
       for l = 0 to μ do
          S[k,l] ← ∅
       for B[n] ∈ L[j − k] in (increasing) order do
          S[k,l].Insert(⟨n + l, n, k⟩)
       done
    done
    SOrd ← MergeSortGeneral(S, μ, ρ)
    last ← ⊥
    while SOrd ≠ ∅ do
       ⟨m, n, k⟩ ← SOrd.RemoveFirst()
       if m ≠ last then
          Bnew[m] ← new empty block
          L[j].Append(Bnew[m])
          last ← m
       fi
       l ← m − n
       B ← B[n] bucket from L[j − k]
       B.pointer[l] ← Bnew[m]
    done


MergeSortGeneral(S, μ, ρ)
    H ← ∅ //empty heap of pairs ordered w.r.t. the first component
    SOrd ← ∅
    for k = 1 to ρ do
       for l = 0 to μ do
          ⟨m, n⟩ ← S[k,l].First()
          if ⟨m, n⟩ is defined then
             H.InsertToHeap(⟨m, n, k⟩)
          fi
        done
    done
    while H ≠ ∅ do
       ⟨m, n, k⟩ ← H.ExtractMin()
       SOrd.Append(⟨m, n, k⟩)
       l ← m − n
       ⟨m_next, n_next, k_next⟩ ← S[k,l].Next(⟨m, n⟩)
       if ⟨m_next, n_next, k_next⟩ is defined then
          H.InsertToHeap(⟨m_next, n_next, k_next⟩)
```

```
    fi
  done
  return SOrd
```

Filling in the buckets $B_\alpha^{k_0,j}[n]$ can be carried out in essentially the same way as in Chapter 5. We only need to replace the lines:

1. $|r(op)| = 1$.
2. $r(op) = I_j^j(V_{\alpha 1})$, *i.e. the* $j$*-th character of* $V_{\alpha 1}$.

*For every such operation, op, we pass through the buckets* $B_\alpha^{j-1}[n_k]$ *and for each element* $\langle u', c(u')\rangle \in B_\alpha^{j-1}[n_k]$ *we proceed in the following way:. . . .*

with the lines:

1. $|r(op)| \geq 1$.
2. $r(op) = I_{j-|r(op)|+1}^j(V_{\alpha 0} \circ V_{\alpha 1})$, *i.e. the last* $|r(op)|$ *characters of* $V_{\alpha 0} \circ V_{\alpha 1}$.

*For every such operation, op, we pass through the buckets* $B_\alpha^{j-|r(op)|}[n_k]$ *and for each element* $\langle u', c(u')\rangle \in B_\alpha^{j-|r(op)|}[n_k]$ *we proceed in the following way:. . . .*

This means that we consider not only the last edit-distance list, but the last $\rho$ edit-distance lists. A pseudo-code reflecting these changes is presented in procedure FillBucketsGeneral. Thus, analogously to Lemma 5.3.2 we have:

**Lemma 6.2.6** *For* $j' < j$ *let* $Size(j')$ *be the size of the list* $L_r(\alpha, k_0, j')$, *i.e. the number of pairs in all of its buckets. Then the step of filling the buckets* $B_\alpha^{k_0,j}$ *performed by the modified algorithm requires* $O(\sum_{j'=j-\rho}^{j-1} Size(j'))$ *time. Furthermore upon termination of this step the following two properties hold:*

1. $u \in Act$ *if and only if* $act(u) = true$ *if and only if* $\langle u, -1\rangle \in B_\alpha^{k_0,j}[n]$ *with* $n = |label(u)|$.
2. $L'_r(\alpha, k_0, j)$ $\mathcal{L}$*-represents* $\widetilde{\mathfrak{B}}^j$.

□

```
FillBucketsGeneral(A, T, ⟨Op, c⟩, q, V, Act, α, η, L, j)
//L is an array of lists, e.g. L_r(α, k, .) or L_l(α, k, .)
  b_α ← q|V_α|
  pos ← if η = 1 then j else |V_α0| − j + 1
  for op ∈ Op with r(op) ≠ ε do
    if Equal(r(op), V, α, η, j) then // check if r(op) matches at position j in V_αη
      for B[n] ∈ L[j − |r(op)|] do
        for ⟨u', c'⟩ ∈ B[n] do
          if c' + c(op) ≤ b_α then
            st ← TraverseAutomaton(A, st(u'), l(op))
```

```
                        if st is defined then
                           u ← TraverseTrie(T, u', l(op))
                           if act(u) = false then
                              act(u) ← true
                              st(u) ← st
                              cost(u) ← c' + c(op)
                              Act.Insert(u)
                              B[n].pointer[|l(op)|].Insert(⟨u, −1⟩)
//B[n].pointer[|l(op)|] is actually the bucket B[n + |l(op)|] of L[j]
                           fi
                           if cost(u) > c' + c(op) then
                              cost(u) ← c' + c(op)
                           fi
                        fi
                     fi
                  fi
               done
            done
         fi
      done
      for B[n] ∈ L[j] do
         for ⟨u, c⟩ ∈ B[n] do
            c ← cost(u)
         done
      done

Equal(U, V, α, η, j)
   if η = 0 then
      sign ← 1
      end_pos ← j
   else
      sign ← −1
      end_pos ← j
   fi
   i ← |U|
   while i > 0 and U[i] = V_α[end_pos] do
      i ← i − 1
      end_pos ← end_pos − sign
   done
   return i = 0
```

**Corollary 6.2.7** *The size of the representation of $L'_r(\alpha, k_0, j)$ is $O(\sum_{j'=j-\rho}^{j-1} Size(j'))$.*

(The size of the trie or other automata are not accounted as a part of the representation of $L'_r(\alpha, k_0, j)$.

*Proof.* Clearly there are at most $\sum_{j'=j-\rho}^{j-1} Size(j')$ buckets allocated for the $L'_r(\alpha, k_0, j)$ before the procedure filling-in the buckets is invoked. Since its

execution requires $O(\sum_{j'=j-\rho}^{j-1} Size(j'))$ total time it cannot modify more than this amount of the structure of $L'_r(\alpha, k_0, j)$. Which gives the desired upper bound.

□

The important fact here is that the size of a specific edit-distance list $L_r(\alpha, k_0, j')$ is involved in the time complexity of its own construction $L_r(\alpha, k_0, j')$, and the construction of the next $\rho$ edit-distance lists $L_r(\alpha, k_0, j'+1)$,... $L_r(\alpha, k_0, j'+\rho)$. However, it is not used in any other computations. Therefore each edit-distance lists contributes at most $O(\rho)$ times to the total complexity of the algorithm.

```
RightExtension(A, ⟨Op, c⟩, q, α, V, ρ, L, T)
   η ← 1
   N_α1 ← |V_α1|
   T ← T(α0)
   for k_0 = 0 to ρ − 1 do
      T_r(α, k_0) ← RightInitialise(A, ⟨Op, c⟩, q, α, V, k_0, L_r)
      for j = 1 to N_α1 do
         if L_r(α, j') = ∅ for j' = j − ρ ... j − 1 then
            L_r(α, N_α1) ← ∅
            return empty trie
         else
            RightExtensionStep(A, T_r(α, k_0), ⟨Op, c⟩, q, V, Act, α, η, L_r, j)
         fi
      done
   done
   return T(α, k_0)

RightInitialise(A, ⟨Op, c⟩, q, T, α, V, k_0, L)
   T_r(α, k_0) ←new empty trie with root r
   for k_1 = 0 to ρ − 1 do
      L_r(α, k_0, −k_1) ← L(α0, k_0, k_1)
      for B[n] ∈ L_r(α, k_0, −k_1) do
         for ⟨u, c⟩ ∈ B[n] do
            U⟨label(T(α0, k_0, k_1), u)
            u' ← TraverseTrie(T_r(α, k_0), r, U)
            u ← u'
         done
      done
   done
   Act ← InitialiseActive(T(α, k_0), L_r(α, k_0, 0)
   EpsilonClosure(A, T(α, k_0), ⟨Op, c⟩, q, V, Act, α, 1, L_r(α, k_0, .), 0)
   Inactivate(Act, T(α, k_0))
   return T_r(α, k_0)

RightExtensionStep(A, T_r(α, k_0), ⟨Op, c⟩, q, V, Act, α, η, L_r, j, ρ)
      InitialiseBucketsGeneral(⟨Op, c⟩, η, j, L_r(α, k_0, .)
```

```
      FillBucketsGeneral(A, T(α, k0), ⟨Op, c⟩, q, V, Act, α, η, Lr(α, k0, .), j)
      EpsilonClosure(A, T(α, k0), ⟨Op, c⟩, q, V, Act, α, η, Lr(α, k0, .), j)
      Inactivate(Act, T(α, k0))
   done

LeftExtension(A, ⟨Op, c⟩, q, α, V, ρ, L, T)
   η ← 0
   Nα0 ← |Vα0|
   for k1 = 0 to ρ − 1 do
      Tl(α, k1) ← LeftInitialise(A, ⟨Op, c⟩, q, α, V, k1, Ll)
      for j = 1 to Nα0 do
         if Ll(α, k1, j′) = ∅ for j′ = j + ρ . . . j + 1 then
            Ll(α, k1, Nα0) ← ∅
            return empty trie
         else
            LeftExtensionStep(A, Tl(α, k1), ⟨Op, c⟩, q, V, Act, α, η, Ll, j, ρ)
         fi
      done
   done
   return T(α, k0)

LeftInitialise(A, ⟨Op, c⟩, q, T, α, V, k1, L)
   Tl(α, k1) ←new empty trie with root r
   Nα0 ← |Vα0|
   for k0 = 0 to ρ − 1 do
      Ll(α, k1, −k0) ← L(α1, k0, k1)
      for B[n] ∈ Lr(α, k1, Nα0 + k0) do
         for ⟨u, c⟩ ∈ B[n] do
            U⟨label(T(α1, k0, k1), u)
            u′ ← TraverseTrie(Tl(α, k1), r, U)
            u ← u′
         done
      done
   done
   return Tl(α, k1)

LeftExtensionStep(A, T, ⟨Op, c⟩, q, V, Act, α, η, L, k1, j, ρ)
      InitialiseBucketsGeneral(⟨Op, c⟩, η, j, L(α, k1, .))
      FillBucketsGeneral(A, T, ⟨Op, c⟩, q, V, Act, α, η, Ll(α, k1, .), j)
      EpsilonClosure(A, T, ⟨Op, c⟩, q, V, Act, α, η, L(α, k1, .), j)
      Inactivate(Act, T)
   done
```

*Union.* The procedures for union of edit-distance lists do not depend on the specific value $\rho(Op)$. Thus they do not require any modifications and Lemma 5.3.5 and Lemma 5.3.8 remain valid.

This analysis leads to similar complexity results as the Proposition 5.3.6 and Proposition 5.3.15 from Chapter 5.

**Proposition 6.2.8** *Given a query word $V$ of length $|V| = N$, the extension steps of approximate search problem can be solved in time $O(T_1 + T_2)$ where:*

$$\begin{aligned} T_1 &= \rho \sum_{k=0}^{\rho-1} \sum_{\alpha \in \mathcal{T}_N} \left( \sum_{j=1}^{N_{\alpha 0}} |L_l(\alpha, k, j)| + \sum_{j=0}^{N_{\alpha 1}} |L_r(\alpha, k, j)| \right) \\ T_2 &= \sum_{k_0, k_1 = 0}^{\rho-1} \sum_{\alpha \in \mathcal{T}_N} \sum_{U \in L(\alpha, k_0, k_1)} |U|. \end{aligned}$$

*Proof.* The reason for the additional factor of $\rho$ was explained above. The remaining details can be filled in analogously to Proposition 5.3.6. They are illustrated in the procedure ApproximateSearchGeneral which generalises the procedure ApproximateSearch form Chapter 5. □

**Proposition 6.2.9** *If $\mathcal{L}$ is finite there is a data structure of size $O(||\mathcal{L}||)$ which enables the execution of the extension steps for every query word $V$ of length $N$ in time:*

$$O(T_1 + T_2')$$

*where:*

$$\begin{aligned} T_1 &= \rho \sum_{k=0}^{\rho-1} \sum_{\alpha \in \mathcal{T}(V)} \sum_{j=0}^{|\alpha 1|} |L_r(\alpha, k, j)| + \sum_{\alpha \in \mathcal{T}(V)} \sum_{j=1}^{|\alpha 0|} |L_l(\alpha, k, j)| \\ T_2' &= \sum_{k_0, k_1 = 0}^{\rho-1} \sum_{\alpha \in \mathcal{T}_N} |L(\alpha, k_0, k_1)|. \end{aligned}$$

*Proof.* The algorithm from Chapter 5 can be generalised in essentially the same way as the algorithm for the infinite regular language. The details are reflected in the procedure ExtensionJoinStepGeneral and the main procedure that solves the approximate search problem in the finite language case is ApproximateSearchJoinGeneral. The analysis of this algorithm can be thus performed as in Proposition 6.2.8 combined with Proposition 6.2.8. □

```
ExtensionStepGeneral(IA_r, IA_l, T_l, L, V, α, ⟨Op, c⟩, q, ρ)
  for k_0 = 0 to ρ − 1 do
    T_r(α, k_0) ← RightExtension(IA_r, T, ⟨Op, c⟩, q, α, L, V, k_0, ρ)
  done
  for k_0 = 0 to ρ − 1 do
    for k_1 = 0 to ρ − 1 do
      T(α1, k_0, k_1) ← ReverseDistanceList(T(α1, k_0, k_1), L(α1, k_0, k_1))
    done
  done
  for k_1 = 0 to ρ − 1 do
    T_l(α, k_1) ← LeftExtension(IA_r, T, ⟨Op, c⟩, q, α, L, V, k_0, ρ)
```

```
   done
   for k_0 = 0 to ρ − 1 do
      for k_1 = 0 to ρ − 1 do
         T_l(α1, k_0, k_1) ← ReverseDistanceList(T_l(α1, k_1), L_l(α1, k_1, |V_α0|−k_0))
      done
   done
   for k_0 = 0 to ρ − 1 do
      for k_1 = 0 to ρ − 1 do
         T(α, k_0, k_1) ← UnionDistanceLists(T_l(α, k_1), T_r(α, k_0), V, L_l(α, k_1, |V_α0|−
k_0), L_r(α, k_0, |V_α1| − k_1), α)
      done
   done

ApproximateSearchRecursiveGeneral(IA_r, IA_l, L, V, α, ⟨Op, c⟩, q, ρ)
   if q|V_α|_α < 1 or ρ(4|V_α|) − 1 then
      return
   else
      T_r ← ApproximateSearchRecursive(IA_r, IA_l, L, V, α0, ⟨Op, c⟩, q, ρ)
      T_l ← ApproximateSearchRecursive(IA_r, IA_l, L, V, α1, ⟨Op, c⟩, q, ρ)
      return ExtensionStepGeneral(IA_r, T_r, IA_l, T_l, L, V, α, ⟨Op, c⟩, q, ρ)
   fi

ApproximateSearchGeneral(V, N, q, α, I, A, IA_r, IA_l, ⟨Op, c⟩)
   ρ ← ρ(Op)
   ApproximateSearchInitialiseGeneral(A, V, N, q, ε, ρ, I)
   ApproximateSearchRecursiveGeneral(IA_r, IA_l, L, V, ε, ⟨Op, c⟩, q, ρ)
   for B ∈ L(ε, 0, 0) do
      for ⟨u, c⟩ ∈ B do
         W ← label(T(ε, 0, 0), u)
         s ←initial state of A
         if TraverseAutomaton(A, s, W) is defined then
            report W
         fi
      done
   done

ExtensionJoinStepGeneral(A, SA, f, ⟨Op, c⟩, q, V, α, η, L, j)
   pos ← if η = 1 then j else |V_α0| − j + 1
   k ← 0
   for op ∈ Op with |r(op)| ≠ ε do
      k ← k + 1
      L(α, j; op) ← ∅
      for ⟨st, l, c⟩ ∈ L(α, j − |r(op)|) do
         if Equal(r(op), V, α, η, j) then
            if η = 1 then
               st' ← ExtendRight(A, st, l(op))
            else
```

```
            st' ← ExtendLeft(A, st, l(op))
          fi
          if st' is defined and c(op) + c ≤ q|V_α| then
            L(α, j; op).Insert(⟨st', l + |l(op)|, c(op) + c⟩)
          fi
        fi
      done
    done
    return Join(L(α, j; .), SA, f, k)

RightExtensionJoinGeneral(A, SA, f, ⟨Op, c⟩, q, α, V, L, k_0, ρ)
    for k = 0 to ρ − 1 do
      L_r(α, k_0, −k) ← L(α0, k_0, k)
    done
    L_r(α, k_0, 0) ← EpsilonClosureJoinStepSimple(A, SA, f, T, ⟨Op, c⟩, V, α, 1, q, L_r(α, k_0, .), 0)
    for j = 1 to |V_α1| do
      L_r(α, k_0, j) ← ExtensionJoinStepSimple(A, SA, f, T, ⟨Op, c⟩, q, V, α, 1, L_r(α, k_0, .), j)
      L_r(α, j) ← EpsilonClosureJoinStepSimple(A, SA, f, T, ⟨Op, c⟩, V, α, 1, q, L_r(α, k_0, .), j)
      if L_r(α, k_0, j') = ∅ for j' ∈ [j − ρ + 1; j] return ∅
    done
    return L_r(α, |V_α1|)

LeftExtensionJoinGeneral(A, SA, f, ⟨Op, c⟩, q, α, V, L, k_1, ρ)
    for k = 0 to ρ − 1 do
      L_l(α, k_1, −k) ← L(α1, k, , k_1)
    done
    for j = 1 to |V_α0| do
      L_l(α, k_1, j) ← ExtensionJoinStepSimple(A, SA, f, T, ⟨Op, c⟩, q, V, α, 0, L_l(α, k_1, .), j, )
      L_l(α, k_1, j) ← EpsilonClosureJoinStepSimple(A, SA, f, T, ⟨Op, c⟩, V, α, 0, q, L_l(α, k_1, .), j)
      if L_l(α, j) = ∅ for j' ∈ [j; j + ρ − 1] return ∅
    done
    return L_l(α, k_1, 1)

ExtensionJoinGeneral(A, SA, f, ⟨Op, c⟩, q, α, V, L, k_0, k_1, ρ)
    LRight ← RightExtensionJoinGeneral(A, SA, f, ⟨Op, c⟩, q, α, V, L, k_0, ρ)
    LLeft ← LeftExtensionJoinGeneral(A, SA, f, ⟨Op, c⟩, q, α, V, L, k_1, ρ)
    L(α) ← Join({LRight, LLeft}, SA, f, 2)

InitialiseJoinGeneral(I, A, q, α, N, V, L, ρ)
    s ←initial state of A
    if qN < 1 then
      for k_0 = 0 to ρ − 1 do
        for k_1 = 0 to ρ − 1 do
          st ← TraverseRight(A, s, V_α
          if st is defined then
          L(α, k_0, k_1) ← {⟨st, N_α, 0⟩}
        done
```

```
      done
   else // qN ≥ 1 and N < 4(ρ − 1)
      for k_0 = 0 to ρ − 1 do
         for k_1 = 0 to ρ − 1 do
            L(α, k_0, k_1) ← RetrieveFromJoinIndex(I, V[k_0+1..N−k_1], ⌊qN⌋)
         done
      done
   fi

RecursiveExtensionJoinGeneral(I, A, SA, f, ⟨Op, c⟩, q, α, V, L, ρ)
   if q|V_α| ≥ 1 and N ≥ 4(ρ − 1) then
      RecursiveExtensionJoinGeneral(A, SA, f, ⟨Op, c⟩, q, α0, V, L)
      RecursiveExtensionJoinGeneral(A, SA, f, ⟨Op, c⟩, q, α1, V, L)
      for k_0 = 0 to ρ − 1 do
         for k_1 = 0 to ρ − 1 do
            ExtensionJoinGeneral(A, SA, f, ⟨Op, c⟩, q, α, V, L, k_0, k_1, ρ)
         done
      done
   else
      InitialiseJoinGeneral(I, A, q, α, N, V, L, ρ)
   fi

ApproximateSearchJoinGeneral(I, A, SA, f, ⟨Op, c⟩, q, V)
   N_ε ← |V|
   V_ε ← V
   RecursiveExtensionJoinGeneral(I, A, SA, f, ⟨Op, c⟩, q, ε, V, L)
   for ⟨st, l, c⟩ ∈ L(ε, 0, 0) do
      if st is final in A
         report the word U with st = st(U)
      fi
   done

RetrieveFromIndex(I, V, b)
   ⟨IT[.], IT(.), IL(.)⟩ ← I
   s ←the initial state of IT[b]
   u ← TraverseAutomaton(IT[b], V)
   if u is defined then
      return IL(u)
   else
      return ⟨∅, ∅
   fi

PreComputeIndex(A, A_l, ⟨Op, c⟩, ρ, q)
   N ← 4(ρ − 1) − 1
   s ←the initial state of A_r
   if qN < 1 return ∅
   for V ∈ Σ^{≤N} do
```

```
for b = 0 to ⌊qN⌋ do
   IT[b] ←empty trie with root r[b]
   N − hood ← Search2(A_r, A_l, V, b, ⟨Op, c⟩)
   if N − hood ≠ ∅
      u ← TraverseTrie(IT[b], r[b], V
      IL(u) ← ∅
      for U ∈ N − hood in deg-lex order do
         st(u) ← RightTraverse(A, s, U
         c(u) ← Edit − Distance(U, V, ⟨Op, c⟩)
         IL(u).Append(⟨st(u), c(u)⟩)
      done
   fi
done
done
return ⟨IT[.], IT(.), IL(.)⟩
```

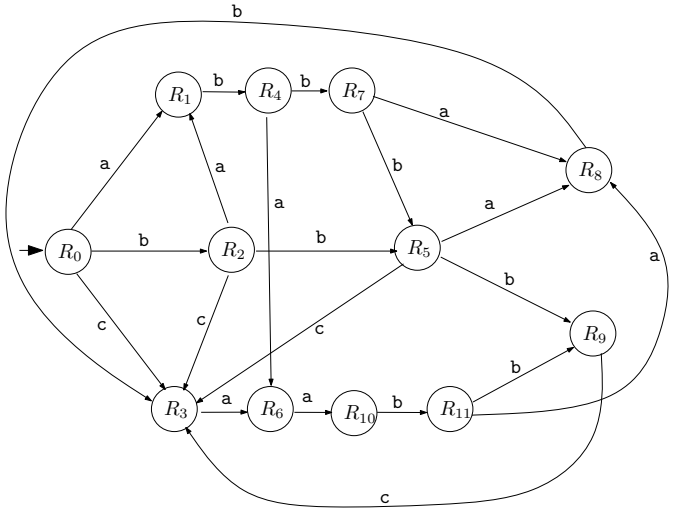
A subtle detail that is on a level of semantics and not on level of representation is the following. The left extensions always start with an operation that is not an insertion while this is not a constraint for the right extensions. Thus provided that `abb` has a left child corresponding to `ab` and a right child corresponding to `b`, the right extension of `ab` will generate first `aba` as a candidate for `ab` at edit-distance 1. On the other hand considering the left extension of the exact match `b` must start either with a deletion or with a substitution on the side of `ba`. Consequently the candidate `abab` for `abb` which corresponds to a left extension of `b` with `aba` is not generated, see Figure 6.1. The reason is that the only way to align `abab` with `abb` at cost not exceeding 1 is by deleting the second `a` on the side of `abab` and this is inadmissible. Still `abab` is generated as a right extension of `ab`, Figure 5.2. This asymmetry is reflected also in the shift of the summing index $j$ when it concerns the left extensions. This detail was not presented in the basic case algorithm in Chapter 5 for the sake of simplicity.

### 6.2.3 Reporting the Answers

At the end of the recursive procedure presented in the previous paragraph, we dispose on several edit-distance lists $L(\varepsilon, k_0, k_1)$. Each of them answers the query for $V[k_0; k_1] = I_{k_0+1}^{N-k_1}(V)$. Clearly we need only the answers for the word $V = V[0; 0]$. Thus we can discard all but the edit-distance list $L(\varepsilon, 0, 0)$ and handle it in the same way as we did with the edit-distance list $L(\varepsilon)$ in Chapter 5. Analogously to Lemma 6.2.10 we obtain the following result:

**Lemma 6.2.10** *Assume that for a given query word $V$ we have the edit-distance list $L(\varepsilon, 0, 0)$ determined by the root of search tree $\mathcal{T}(V)$. If in addition we dispose on a deterministic finite state automaton for the language $\mathcal{L}$ we can answer the query:*

```
Given: L ⊆ Σ* regular language,
   d = (Op, c) an edit-distance,
```




$Inf((\{ababb, acbbb\}^*)^{rev})$

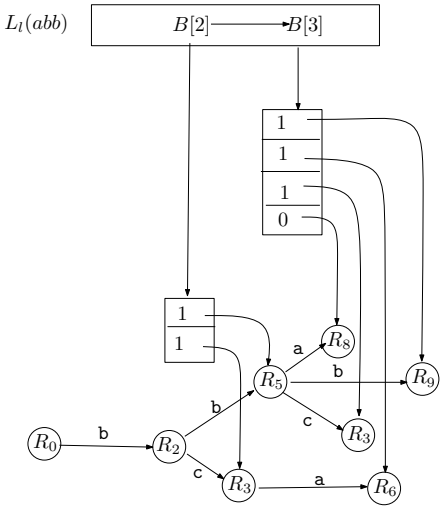



Figure 6.1: The representation of edit-distance lists.

$q \in (0;1)$ `a threshold parameter`
`Input:` $V \in \Sigma^*$
`Output:` $\{U \in \mathcal{L} \mid d(U,V) \leq q|V|\}$.

*in time* $O(\sum_{U \in Dom(L(\varepsilon,0,0))} |U|)$.

*Proof.* We traverse each word $U \in Dom(L(\varepsilon, 0, 0)$ with the deterministic finite state automaton for $\mathcal{L}$. This is done $O(|U|)$ time. If the automaton recognises $U$, then we report it as an answer of the query, otherwise we proceed with the next word in the list. The claimed complexity is then evident.

□

### 6.2.4 Memory Bookkeeping

In the construction of the query tree, $\mathcal{T}(V)$, in Subsection 6.2.1 we imposed the requirement that each inner node $\alpha$ of the tree must satisfy $|V_\alpha| \geq 4(\rho - 1)$. This is the minimal possible value that guarantees that the infixes considered in all the subqueries will be well defined. As a result we needed an index, $\mathcal{I}$, in Lemma 6.2.2 that stores the answers for queries of length less than $4(\rho - 1)$.

It should be clear that we can substitute the lower bound $4(\rho - 1)$ with a bigger value, say $2n_0 + 4(\rho - 1)$. This will lead to the fact that for $\alpha \neq \varepsilon$ the lengths $|V_\alpha| \geq n_0 + 2(\rho - 1)$. The algorithm described in the previous subsections needs only a subtle adjustment in order to run correctly in this case as well. Specifically, instead of an index for the words of length less than $4(\rho - 1)$ we will need an index for the words of length less than $2n_0 + 4(\rho - 1)$.

The possibility to adjust the parameter $n_0$ has the following effect. In the next chapter we shall derive an upper bound for the efficiency of the algorithm described above. Under certain constraints it will turn out that the algorithm runs fast on average and the bigger the $n_0$ the better the performance. However, the maintenance of an appropriate index, $\mathcal{I}$, requires space. In order to estimate the size of $\mathcal{I}$, one can proceed as follows.

Given two positive integers, $N$ and $M$, we consider all the alignments $\omega \in \mathcal{A}_{N,M}$ with the properties: (i) $|r(\omega)| = N$ and (ii) $c(\omega) \leq qM$. Intuitively, each such alignment, $\omega$, with cost, $c(\omega) \leq qM$ does not contain more than $qM$ nonidentity operations because each nonidentity operation costs at least one unit. Therefore, if we additionally know that $r(\omega) = V$ where $V$ is a fixed word of length $N$, $\omega$ can be regarded in the following way. First we choose the exact cost $c(\omega) \in \{0, 1, \ldots, [qM]\}$ of the alignment, then we select the positions in $\{0, 1 \ldots, N + c(\omega)\}$ where a nonidentity operation in $\omega$ occurs and finally we select the operation itself. A careful analysis of this procedure shows that the number of alignments with fixed right side $V$ of length $N$ and cost not exceeding $qM$ does not exceed:

$$\sum_{k=0}^{[qM]} \binom{N+k}{k} |Op|^k.$$

Formalising the above idea and taking into account that there are $|\Sigma|^N$ words of length $N$, in the Appendix we shall formally prove that:

$$|\mathcal{A}_{N,M}| \in \exp(O(N + qM)).$$

Clearly, $|\mathcal{A}_{N,M}|$ is also an upper bound for the number of pairs of words $(U, V)$ with $|V| = N$ and $d(U, V) \leq qM$. Now we can prove the following lemma:

**Lemma 6.2.11** *There is an algorithm, that given an automaton $\mathcal{A}$ with language $\mathcal{L}$ and an integer $n_0$, computes an index $\mathcal{I}$ with the following properties:*

1. *for each $n < 2n_0 + 4(\rho - 1)$ and $k_0, k_1 < \rho$ and each word $V_\alpha \in \Sigma^n$, $\mathcal{I}$ an provides the edit-distance list:*

$$L(\alpha, k_0, k_1) = \{\langle U, c\rangle \,|\, U \in Inf(\mathcal{L}) \text{ and } d(U, V_\alpha[k_0; k_1]) = c \leq q|V_\alpha|\}$$

   *in time:*

$$O(n + ||L(\alpha, k_0, k_1)||)$$

2. *$\mathcal{I}$ requires $\exp(O(2n_0 + 4\rho_0))$ storage space.*

*Proof.* The index $\mathcal{I}$ can be constructed as follows, see also procedure PreComputeIndex. For each $n < 2n_0 + 4(\rho - 1)$ we generate the words $V \in \Sigma^n$. We define the bound $b_n = q \min(2n_0 + 4(\rho - 1), n + 2(\rho - 1))$ and determine the set:

$$\{\langle U, c\rangle \,|\, d(U, V[k_0; k_1]) = c \leq b_n\}$$

by the means of the algorithm of Mihov and Schulz, [42], for example. We represent the words $V[k_0; k_1]$ in a trie $\mathcal{T}(\mathcal{I})$. Each node of this trie stores an additional link to the answers of the above query that are ordered in lists in increasing order of $c$.

With this index, we can easily handle the first requirement. Indeed, given a word $V = V_\alpha$ of length $|V| < 2n_0 + 4(\rho - 1)$ and $k_0, k_1 < \rho$ we traverse the trie $\mathcal{T}(\mathcal{I})$ with the word $V[k_0; k_1]$. Clearly, the length, $n$, of $V[k_0; k_1]$ satisfies that $|V| \leq n + 2(\rho - 1)$ because each of $k_0$ and $k_1$ is smaller than $\rho$. We also have that $n \leq |V| < 2n_0 + 4(\rho - 1)$. This implies that $q|V| \leq b_n$ and therefore the edit-distance list $L(\alpha, k_0, k_1)$ can be easily computed on the bases of the lists attached to the node of $\mathcal{T}(\mathcal{I})$ corresponding to $V[k_1; k_2]$. Indeed, we scan these lists in order of $c$ until $c \leq q|V|$, see procedure RetrieveFromIndex.

Next we argue the space requirements for $\mathcal{I}$. Clearly, the number of pairs $(U, V)$ with $|V| \leq 2n_0 + 4(\rho - 1) - 1$ and $d(U, V) \leq q \min(|V| + 2(\rho - 1), 4(\rho - 1))$ is bounded by the number of alignments $\omega \in \mathcal{A}_{N,M}$ where $N < 2n_0 + 4(\rho - 1)$ and $M = \min(N + 2(\rho - 1), 2n_0 + 4(\rho - 1))$. Since $|\mathcal{A}_{N,M}| \in exp(O(N + qM))$ we get:

$$\begin{aligned}
\sum_{N < 2n_0 + 4(\rho-1)} |\mathcal{A}_{N,2n_0+4(\rho-1)}| &\in \sum_{N<4(\rho-1)} \exp(O(4q(\rho - 1) + N)) \\
&= \exp(O(q(4(\rho - 1) + 2n_0) + 2n_0 + 4(\rho - 1))) \\
&= \exp(O((1 + q)(2n_0 + 4(\rho - 1))).
\end{aligned}$$

For each such pair we need additional $O(n_0 + \rho)$ space two represent the words $V$ and $U$. This concludes the proof. □

**Remark 6.2.12** In the particular case when $n_0 = 0$, the size of the index, $\mathcal{I}$, described in the Lemma 6.2.11 has the magnitude of $\exp((1 + q)4(\rho - 1))$ and thus is of constant size.

# Chapter 7

# Running Time of the Generalised Myers' Algorithm

Now, that we have generalised the Myers' algorithm as to solve the approximate search problem for arbitrary regular sets, we address its complexity. As in [47] the main benefit of the algorithm is that it will generate small number of alignments on average. This implies that the portion of those query words that require more time is neglectful. Myers shows this for the Levenshtein edit-distance and a special case of finite languages. In order to extend this result for arbitrary edit-distance and regular languages, we apply an approach based on generating functions. It allows us to encode the essential part of the information of the edit-distance and the regular language in the terms of power series, where the variables represent the distribution of the individual characters in the alphabet.

In Chapter 5 we proved the linear time for the initialisation required by our algorithm and we also provided an estimate of the complexity of the entire algorithm in terms of the sizes of the generated edit-distance lists. Similar bounds were derived for the general case in Chapter 6. In this Chapter we relate the expectation for this value with the characteristics of the given regular language, $\mathcal{L}$, the generalised edit-distance, $(Op, c)$, and the threshold, $q$.

The basic results from this chapter were described in [19].

## 7.1 Average Number of Generated Candidates during the Extension Steps

We cannot uniformly bound the time required for this step in a nontrivial way. The reason is that some specific query words, $V$, may generate exponential number of different words $U \in \mathcal{L}$ with $d(U, V) \leq q|V|$.

In order to obtain an upper bound in the average-case of our algorithm, we shall consider some distribution on the query words $V \in \Sigma^*$. We assume that this distribution is induced by the independently distributed characters $\sigma_1, \dots \sigma_{|\Sigma|}$ of the input alphabet.

In these settings, it is natural to consider the distribution as a $|\Sigma|$-dimensional vector, where the coordinates correspond to the distinct characters. In a significant part of the outline of this subsection, we would not use the specificity of this vector and it will come into play when the final computation has to be done.

Hence, we naturally come to the use of generating functions, which should be considered as multivariable polynomials whose terms $\boldsymbol{t}^{\boldsymbol{s}}$ correspond either explicitly or implicitly to the type of word $||U||$, i.e. often we shall count some features of the words of type $s$ in the coefficients of the terms $\boldsymbol{t}^{\boldsymbol{s}}$. (recall that $\boldsymbol{t}^{\boldsymbol{s}}$ is a short hand for $\prod_{i=1}^{|\Sigma|} t_i^{s_i}$)

With this remarks in mind, we proceed as follows. Firstly, we shall bound uniformly alignments the number of alignments $\omega$ with a fixed left side $l(\omega) = U$ – an infix of $\mathcal{L}$ – and such that $c(\omega) \leq q|r(\omega)|$. Secondly, we shall turn our viewpoint and changing the summation order we shall obtain an upper bound for the alignments $\omega$ with $c(\omega) \leq q|r(\omega)|$, $|r(\omega)|$ is fixed (but not $r(\omega)$ itself) and $l(\omega)$ is an infix of the language $\mathcal{L}$. In order to model these quantitative properties of the language we use generating function $g_{\mathcal{A}}$ where $\mathcal{A}$ is a finite automaton with $\mathcal{L}(\mathcal{A}) = \mathcal{L}$ is some $\varepsilon$-free automaton recognising the language $\mathcal{L}$. Thus, we obtain an upper bound for the average running time of the algorithms from Chapter 5 and Chapter 6 in terms of the language $\mathcal{L}$, the edit-distance induced by $(Op, c)$ and the threshold $q \in (0; 1)$.

Finally, under convergence assumptions, we prove $O(N)$ time on average for the algorithms from Chapter 5 and Chapter 6 for query words, $V$, of length $N$. In Section 7.3, we shall give sufficient conditions that guarantee convergence and thus make the result consistent.

Recall Definition 1.8.1 where we defined $||U||_i$ to be the number of occurrences of the character $\sigma_i$ in $U$ and we set:

$$||U|| = (||U||_1, ||U||_2, \dots, ||U||_{|\Sigma|}).$$

Next definition plays an important role in our counting technique.

**Definition 7.1.1** Let $\Sigma$ be an alphabet, $\Sigma = \{\sigma_1, \dots, \sigma_{|\Sigma|}\}$, and $(Op, c)$ be a generalised edit-distance over $\Sigma$. We define $Op_\varepsilon = Op \cap (\{\varepsilon\} \times \Sigma^*)$ and $Op_i = Op \cap (\sigma_i \Sigma^* \times \Sigma^*)$. The generating functions $f_\varepsilon(t_1, \dots, t_{|\Sigma|}, z)$ and $f_i(t_1, \dots, t_{|\Sigma|}, z)$ with respect to the set of operations $Op$ are introduced as:

$$f_\varepsilon(t_1, \dots, t_{|\Sigma|}, z) = \sum_{(\varepsilon, V) \in Op_\varepsilon} \boldsymbol{t}^{||V||} z^{c(op)} \text{ and}$$

$$f_i(t_1, \dots t_{|\Sigma|}, z) = \sum_{(U, V) \in Op_i} \boldsymbol{t}^{||V|| - ||U||} z^{c(op)}.$$

We determine the generating function $f_\Sigma : \mathbb{R}^{|\Sigma|} \times \mathbb{R} \to \mathbb{R}^{|\Sigma|}$ as:

$$f_\Sigma(\boldsymbol{t}; z) = (f_1(\boldsymbol{t}; z), f_2(\boldsymbol{t}; z), \dots f_{|\Sigma|}(\boldsymbol{t}; z)).$$

Intuitively, each of the terms $\boldsymbol{t}^{||r(op)||-||l(op)||} z^{c(op)}$ encodes the following two features of the operation *op*. Firstly, the power of $z$ determines the cost of *op*. Secondly, the power (positive or negative) of $t_j$ indicates how the application of *op* modifies the number of characters $\sigma_j$. Since we shall interpret $t_j$ as probabilities for the occurrence of the character $\sigma_j$, $\boldsymbol{t}^{||r(op)||-||l(op)||}$ for specific $\boldsymbol{t}$ would encode the the probability of $r(op)$ divided by the probability for $l(op)$.

We illustrate Definition 7.1.1 on the special case of Levenshtein edit-distance (see Remark 1.3.3):

$$Op_L = \Sigma \times \Sigma \cup \{\varepsilon\} \times \Sigma \cup \Sigma \times \{\varepsilon\}$$
$$c_L(op) = \begin{cases} 0, \text{ if } op = (\sigma_i, \sigma_i) \text{ for some } i \leq |\Sigma| \\ 1, \text{ otherwise.} \end{cases}$$

According to the definition, the sets $Op_i$ are given by:

$$Op_i = \{\sigma_i\} \times \Sigma \cup \{\sigma_i\} \times \{\varepsilon\} = \{\sigma_i\} \times (\Sigma \cup \{\varepsilon\})$$

because each operation *op* whose left side $l(op)$ starts with $\sigma_i$ is actually of the form $l(op) = \sigma_i$. Next, the set $Op_\varepsilon$ is specified by:

$$Op_\varepsilon = \{\varepsilon\} \times \Sigma.$$

In order to to better understand the function $f_i(t; z)$ we simplify the expression:

$$f_i(\boldsymbol{t}; z) = \sum_{op \in Op_i} \boldsymbol{t}^{||r(op)||-||l(op)||} z^{c(op)}.$$

By the discussion above, the operations $op \in Op_i$ is either of the form $op = (\sigma_i, \sigma_j)$ or $op = (\sigma_i, \varepsilon)$ where $\sigma_j$ varies in $\Sigma$. Hence we obtain:

$$f_i(\boldsymbol{t}; z) = \sum_{j=1}^{|\Sigma|} \boldsymbol{t}^{||\sigma_j||-||\sigma_i||} z^{c_L((\sigma_i,\sigma_j))} + \boldsymbol{t}^{||\sigma_i||-||\varepsilon||} z^{c_L((\sigma_i,\varepsilon))}.$$

Now, recall that $||V||$ is a $|\Sigma|$-dimensional vector where the $k$-th position indicates the number of characters $\sigma_k$ in the word $V$. Therefore:

$$||\sigma_i|| = (0, \dots, \underset{\substack{\uparrow \\ i}}{1}, \dots, 0) \text{ and } ||\varepsilon|| = (0, \dots, 0).$$

Thus, the definition of $\boldsymbol{t}^{\boldsymbol{s}} = \prod_{j=1}^{|\Sigma|} t_j^{s_j}$ implies that:

$$\boldsymbol{t}^{||\sigma_j||} = t_j, \quad \boldsymbol{t}^{||\sigma_i||} = t_i \text{ and } \boldsymbol{t}^{||\varepsilon||} = 1.$$

Plugging this into the expression for $f_i$, we get:

$$\begin{aligned} f_i(\boldsymbol{t};z) &= \sum_{j=1}^{|\Sigma|} t_j t_i^{-1} z^{c_L((\sigma_i,\sigma_j))} + t_i^{-1} z^{c_L((\sigma_i,\varepsilon))} \\ &= t_i t_i^{-1} z^{c_L((\sigma_i,\sigma_i))} + t_i^{-1} \sum_{j\neq i} t_j z^{c_L((\sigma_i,\sigma_j))} + t_i^{-1} z^{c_L((\sigma_i,\varepsilon))}. \end{aligned}$$

Now, we use the definition of $c_L$. Since $c_L((\sigma_i,\sigma_i)) = 0$ and $c_L((\sigma_i,\sigma_j)) = 1$ for $i \neq j$ and $c_L((\sigma_i,\varepsilon)) = 1$, we derive:

$$f_i(\boldsymbol{t};z) = 1 + t_i^{-1} \sum_{j\neq i} t_j z + t_i^{-1} z = 1 + t_i^{-1} z \left(1 + \sum_{j\neq i} t_j\right).$$

We observe that the term 1 will always appear in the expressions for $f_i$ (considered as polynomials of $z$). It is due to the identity operation $(\sigma_i,\sigma_i)$ which is of cost 0 and according to the definition of the operation set, $Op$, is the only identity operation starting with $\sigma_i$. The rest of the terms of the polynomial $f_i(\boldsymbol{t};z)$ are edit-distance-specific and they correspond to the nonidentity operations involving $\sigma_i$ as an initial character of the left side of the operations. (But since the nonidentity operations are of positive cost they can contribute only to the positive powers of $z$ in $f_i(t;z)$)

Next, let us consider the function $f_\varepsilon$ which was defined as:

$$f_\varepsilon(\boldsymbol{t};z) = \sum_{op\in Op_\varepsilon} \boldsymbol{t}^{||r(op)||} z^{c_L(op)}.$$

Note that $||\varepsilon|| = (0,0,\ldots,0)$ is the $|\Sigma|$-dimensional zero vector, and therefore we can represent $f_\varepsilon$ similarly to $f_i$, i.e.:

$$f_\varepsilon(\boldsymbol{t};z) = \sum_{op\in Op_\varepsilon} \boldsymbol{t}^{||r(op)||-||l(op)||} z^{c_L(op)}.$$

Since $Op_\varepsilon = \{\varepsilon\} \times \Sigma$ we obtain:

$$f_\varepsilon(\boldsymbol{t};z) = \sum_{j=1}^{|\Sigma|} \boldsymbol{t}^{||\sigma_j||-||\varepsilon||} z^{c_L((\sigma_j,\varepsilon))}.$$

As we already discussed, $\boldsymbol{t}^{||\sigma_j||} = t_j$ and $\boldsymbol{t}^{||\varepsilon||} = 1$. By the definition of the Levenshtein edit-distance $c_L((\sigma_j,\varepsilon)) = 1$ and therefore:

$$f_\varepsilon(\boldsymbol{t};z) = \sum_{j=1}^{|\Sigma|} t_j z = z \sum_{j=1}^{|\Sigma|} t_j.$$

Although this was only an example, it facilitates us to draw some general conclusions about the form of the functions $f_i$ and $f_\varepsilon$. Firstly, since $c(op) \geq 0$ is

an integer number for all operations $op$, the functions $f_i$ and $f_\varepsilon$ can be considered as polynomials of $z$ (with coefficients $\alpha(t)$ which are rational functions of $t$). Secondly, since $c(op) = 0$ if and only if $op$ is an identity operation and the only identity operations are of the form $op = (\sigma_i, \sigma_i)$ we deduce that:

$$f_i(\boldsymbol{t}; z) = 1 + zh_i(\boldsymbol{t}; z) \text{ and } f_\varepsilon(\boldsymbol{t}; z) = zh_\varepsilon(\boldsymbol{t}; z)$$

where $h_\varepsilon$ and $h_i$ are polynomials of $z$ with parameters depending on $\boldsymbol{t}$. This is due to the fact that only $(\sigma_i, \sigma_i)$ is an identity operation starting with $\sigma_i$ and that there are no identity operations $(\varepsilon, V)$.

Finally, in the framework where $t_i$ are probabilities of independently distributed random variables $\sigma_i$ we have that $\sum_{i=1}^{|\Sigma|} t_i = 1$ and $t_i \in (0; 1)$. Under these assumptions $h_i(\boldsymbol{t}; z)$ and $h_\varepsilon(\boldsymbol{t}; z)$ are uniformly continuous considered as functions of $z$ and therefore they are bounded when $z$ varies in any finite interval. Later, we shall benefit from this observation in the special case $z \in (0; 1)$.

**Definition 7.1.2** Let $P(\boldsymbol{t})$ and $Q(\boldsymbol{t})$ be $n$-dimensional generating functions of the form:

$$P(\boldsymbol{t}) = \sum_{\boldsymbol{s} \in \mathbb{Z}^n} a(\boldsymbol{s})\boldsymbol{t}^{\boldsymbol{s}} \text{ and } Q(\boldsymbol{t}) = \sum_{\boldsymbol{s} \in \mathbb{Z}^n} b(\boldsymbol{s})\boldsymbol{t}^{\boldsymbol{s}}$$

where the coefficients $a(\boldsymbol{s})$ and $b(\boldsymbol{s})$ are real numbers for all $n$-tuples $\boldsymbol{s}$. We say that $P \preceq Q$ if and only if:

$$\forall \boldsymbol{s} \in \mathbb{Z}^n (a(\boldsymbol{s}) \leq b(\boldsymbol{s})).$$

It should be clear that $\preceq$ defines a partial ordering on power series. Next, assume that $P \preceq Q$ are as above and the coefficients $a(\boldsymbol{s})$ are nonnegative, then for a nonnegative real-valued vector $\boldsymbol{t} \in \mathbb{R}_+^n$ such that $Q(\boldsymbol{t})$ converges, $P(\boldsymbol{t})$ also converges and:

$$P(\boldsymbol{t}) \leq Q(\boldsymbol{t}).$$

Furthermore, if $P$ is with nonnegative coefficient, $P \preceq Q$, and $R(\boldsymbol{t}) = \sum_{\boldsymbol{s}} r(\boldsymbol{s})\boldsymbol{t}^{\boldsymbol{s}}$ is a power series with nonnegative coefficients, then $P(\boldsymbol{t})R(\boldsymbol{t}) \preceq Q(\boldsymbol{t})R(\boldsymbol{t})$.

In order to derive an upper bound for the efficiency of our algorithm, we proceed in a somewhat implicit way. That is, instead of counting how many alignments a certain input would invoke, we count in how many different inputs a fixed infix $U$ may be generated. This viewpoint motivates the following definition:

**Definition 7.1.3** Let $U \in \Sigma^*$ be a word, $q \in (0; 1)$ be rational and $s \in \mathbb{N}^{|\Sigma|}$, with $\mathcal{A}(U, q; \boldsymbol{s})$ we denote the set of all alignments $\omega \in Op^*$, s.t.:

$$l(\omega) = U, \quad ||r(\omega)|| = \boldsymbol{s} \text{ and } c(\omega) \leq q|r(\omega)|.$$

We set $a(U, q; \boldsymbol{s}) = |\mathcal{A}(U, q; \boldsymbol{s})|$ and with $\mathcal{A}(U, q)$ we denote the union of all sets $\mathcal{A}(U, q; \boldsymbol{s})$ when $\boldsymbol{s}$ varies in $\mathbb{N}^{|\Sigma|}$. Finally, we set $Q(U, q; \boldsymbol{t})$ to be the series:

$$Q(U, q; \boldsymbol{t}) = \sum_{\boldsymbol{s}} a(U, q; \boldsymbol{s})\boldsymbol{t}^{\boldsymbol{s}},$$

where $\boldsymbol{s} = (s_1, \ldots, s_{|\Sigma|})$ varies in $\mathbb{N}^{|\Sigma|}$.

Next, we show how to uniformly bound the parameters $a(U, q; \boldsymbol{s})$ in terms of the generating functions $f_i$ and $f_\varepsilon$ and the parameter $q$ which depend only on the set of operations an the threshold parameter $q$.

**Lemma 7.1.4** *Let $U \in \Sigma^*$ and $q \in (0; 1)$ be fixed, then there exists a function $b_{U,q} : \mathbb{Z}^{|\Sigma|} \times \mathbb{R} \to \mathbb{R}$ with the following properties:*

1. *for each $\boldsymbol{s} \in \mathbb{Z}^{|\Sigma|}$ and each $z \in (0; 1)$:*

$$a(U, q; \boldsymbol{s}) \leq b_{U,q}(\boldsymbol{s}, z).$$

2. *for all real vectors $\boldsymbol{t} \in \mathbb{R}^{|\Sigma|}$ and $z \in \mathbb{R}$ such that $f_\varepsilon(z^{-q}\boldsymbol{t}; z) < 1$, it holds:*

$$\sum_{\boldsymbol{s} \in \mathbb{Z}^{|\Sigma|}} b_{U,q}(\boldsymbol{s}, z)\boldsymbol{t}^{\boldsymbol{s}} = \frac{(z^{-q}\boldsymbol{t})^{||U||}}{1 - f_\varepsilon(z^{-q}\boldsymbol{t}; z)} \left( \frac{f_\Sigma(z^{-q}\boldsymbol{t}; z)}{1 - f_\varepsilon(z^{-q}\boldsymbol{t}; z)} \right)^{||U||}.$$

Before stepping to the proof, it is worth mentioning that for $q \in (0; 1)$ and $\boldsymbol{t} \in \mathbb{R}_+^{|\Sigma|}$, (specifically, no zero coordinates are allowed) the function $f_\varepsilon(z^{-q}\boldsymbol{t}; z) = zh_\varepsilon(z^{-q}\boldsymbol{t}; z) = O(z^{-\lambda}z) = O(z^{1-\lambda})$ where $\lambda$ is defined as:

$$\lambda = \max\{-c(op) + 1 + q|r(op)| \,|\, op \in Op_\varepsilon\}.$$

In particular if $\lambda < 1$ we can always find a $z \in (0; 1)$ such that $f_\varepsilon(z^{-q}t; z) < 1$. Hence both conditions 1 and 2 will be satisfied. On the other hand, if there is an operation $op \in Op_\varepsilon$ with $-c(op) + 1 + q|r(op)| \geq 1$, all the words $V \in \{r(op)\}^*$ will have the property:

$$V = (r(op))^n \text{ and } d(\varepsilon, V) \leq nc(op) \leq nq|r(op)| = q|V|.$$

Thus $\varepsilon$ would be *close* to an infinite set of (longer and longer) words $V$, which from certain philosophical point of view seems to be irrelevant.

*Proof.* (of Lemma 7.1.4) Let $U = u_1u_2 \dots u_n$ be a word of length $n$. Each alignment $\omega = op_1 \circ op_2 \dots op_N$ with $l(\omega) = U$ can be considered in the following way:

1. a subalignment $\omega_\varepsilon = op'_{j_1} \circ op'_{j_2} \circ \cdots \circ op'_{j_k}$ of all operations of $\omega$ such that $op'_{j_i} \in Op_\varepsilon$. In particular, the number of these operations is $|\omega_\varepsilon| = k$ within $\omega$. (Note, that subalignment means that $j_1 < j_2 < \cdots < j_k$.)

2. a subalignment $\omega_\Sigma = op''_{j_1} \circ op''_{j_2} \cdots \circ op''_{j_{N-k}}$ of $\omega$ which consists of all operations $op''_{j_i} \notin Op_\varepsilon$.

3. a sequence of $\beta_\omega \in \{0, 1\}^N$ which contains $k$ zeroes and $N - k$ ones such that:

$$\beta_\omega(j) = 0 \iff op_j \in Op_\varepsilon.$$

Clearly, given the data $\langle \omega_\varepsilon, \omega_\Sigma, \beta_\omega \rangle$ we can uniquely reconstitute the alignment $\omega$. To this end it suffices to shuffle the sequences $\omega_\varepsilon$ and $\omega_\Sigma$ as encoded in $\beta_\omega$.

In order to satisfy the property $a(U; \boldsymbol{s}) \leq b_{U,q}(\boldsymbol{s}; z)$ for all $z \in (0; 1)$ we are searching for a power series $P(\boldsymbol{t}; z)$ such that:

$$Q(U, q; \boldsymbol{t}) = \sum_{\boldsymbol{s} \in \mathbb{Z}} a(U, q; \boldsymbol{s}) \boldsymbol{t}^{\boldsymbol{s}} \preceq P(\boldsymbol{t}; z)$$

for all $z \in (0; 1)$. If we manage to determine such a power series $P(\boldsymbol{t}; z)$ we can simply set $b_{U,q}(\boldsymbol{s}; z) = [\boldsymbol{t}^{\boldsymbol{s}}]P(\boldsymbol{t}; z)$.

We approach the problem in the sequel by replacing the sum over $\boldsymbol{s} \in \mathbb{Z}$ with a sum over all alignments in $\mathcal{A}(U) = \mathcal{A}(U, q)$. This can be easily achieved as follows:

$$\begin{aligned} \sum_{\boldsymbol{s} \in \mathbb{Z}} a(U, q; \boldsymbol{s}) \boldsymbol{t}^{\boldsymbol{s}} &= \sum_{\boldsymbol{s} \in \mathbb{Z}} \sum_{\omega \in \mathcal{A}(U, q; \boldsymbol{s})} \boldsymbol{t}^{||r(\omega)||} \\ &= \sum_{\omega \in \mathcal{A}(U)} \boldsymbol{t}^{||r(\omega)||}. \end{aligned}$$

As we already explained we can think of $\omega = \langle \omega_\varepsilon, \omega_\Sigma, \beta_\omega \rangle$ where $\omega_\varepsilon \in (Op_\varepsilon)^*$, $\omega_\Sigma \in Op^* \setminus (Op_\varepsilon)^*$ and $\beta_\omega \in \{0, 1\}^*$ with $|\omega_\varepsilon|$ zeroes and $|\omega_\Sigma|$ ones. Each such alignment $\omega$ belongs to $\mathcal{A}(U)$ if and only if:

$$l(\omega) = U \text{ and } c(\omega) = c(\omega_\Sigma) + c(\omega_\varepsilon) \leq q|r(\omega)|.$$

Observe also that $||r(\omega)|| = ||r(\omega_\varepsilon)|| + ||r(\omega_\Sigma)||$. We use the notion $||\beta||_0$ for the number of zeroes and $||\beta||_1$ for the number of ones for a word $\beta \in \{0, 1\}^*$ and $Op_\Sigma = Op \setminus Op_\varepsilon$. Hence we obtain:

$$\begin{aligned} \sum_{\boldsymbol{s} \in \mathbb{Z}} a(U, q; \boldsymbol{s}) \boldsymbol{t}^{\boldsymbol{s}} &= \sum_{\omega \in \mathcal{A}(U)} \boldsymbol{t}^{||r(\omega)||} \\ &= \sum_{\omega_\varepsilon \in (Op_\varepsilon)^*} \sum_{\substack{\omega_\Sigma \in (Op_\Sigma)^*, l(\omega_\Sigma) = U, \\ c(\omega_\varepsilon) + c(\omega_\Sigma) \leq q|r(\omega_\varepsilon)| + q|r(\omega_\Sigma)|}} \sum_{\substack{\beta \in \{0,1\}^* \\ ||\beta||_0 = |\omega_\varepsilon| \\ ||\beta||_1 = |\omega_\Sigma|}} \boldsymbol{t}^{||r(\omega_\varepsilon||+||r(\omega_\Sigma||} \\ &= \sum_{\omega_\varepsilon \in (Op_\varepsilon)^*} \sum_{\substack{\omega_\Sigma \in (Op_\Sigma)^*, l(\omega_\Sigma) = U, \\ c(\omega_\varepsilon) + c(\omega_\Sigma) \leq q|r(\omega_\varepsilon)| + q|r(\omega_\Sigma)|}} \binom{|\omega_\varepsilon| + |\omega_\Sigma|}{|\omega_\varepsilon|} \boldsymbol{t}^{||r(\omega_\varepsilon||+||r(\omega_\Sigma||}. \end{aligned}$$

We use the following two simple observations which allow us to overcome the technical problems in this summation and still obtain an appropriate upper bound. Firstly, $|\omega_\Sigma| \leq n$ since each operation $op''_{j_i} \in Op_\Sigma$ implies that $|l(op''_{j_i})|| \geq 1$ and $l(\omega_\varepsilon) = \varepsilon$. Therefore:

$$n = |l(\omega_\Sigma| = \sum_{i=1}^{|\omega_\Sigma|} |l(op''_{j_i})| \geq |\omega_\Sigma|.$$

This implies that $\binom{n+|\omega_\varepsilon|}{|\omega_\varepsilon|} \geq \binom{|\omega_\Sigma|+|\omega_\varepsilon|}{|\omega_\varepsilon|}$. Subsequently we obtain:

$$\begin{aligned}\sum_{\boldsymbol{s}\in\mathbb{Z}} a(U,q;\boldsymbol{s})\boldsymbol{t}^{\boldsymbol{s}} &= \sum_{\omega_\varepsilon\in(Op_\varepsilon)^*} \sum_{\substack{\omega_\Sigma\in(Op_\Sigma)^*, l(\omega_\Sigma)=U,\\ c(\omega_\varepsilon)+c(\omega_\Sigma)\leq q|r(\omega_\varepsilon)|+q|r(\omega_\Sigma)|}} \binom{|\omega_\varepsilon|+|\omega_\Sigma|}{|\omega_\varepsilon|}\boldsymbol{t}^{||r(\omega_\varepsilon||+||r(\omega_\Sigma||}\\ &\preceq \sum_{\omega_\varepsilon\in(Op_\varepsilon)^*} \sum_{\substack{\omega_\Sigma\in(Op_\Sigma)^*, l(\omega_\Sigma)=U,\\ c(\omega_\varepsilon)+c(\omega_\Sigma)\leq q|r(\omega_\varepsilon)|+q|r(\omega_\Sigma)|}} \binom{|\omega_\varepsilon|+n}{|\omega_\varepsilon|}\boldsymbol{t}^{||r(\omega_\varepsilon||+||r(\omega_\Sigma||}.\end{aligned}$$

The second observation aims to remove the condition $c(\omega_\varepsilon)+c(\omega_\Sigma) \leq q|r(\omega_\varepsilon)|+q|r(\omega_\Sigma)|$ from the summing and still preserve the sum from blowing up. We achieve this by introducing a fresh variable $z \in (0;1)$. It is rather straightforward that:

$$\begin{aligned}&\forall\omega \in Op^*[z^{c(\omega)-q|r(\omega)|} \geq 0] \text{ and}\\ z^{c(\omega)-q|r(\omega)|} \geq 1 \iff c(\omega)-q|r(\omega)| \leq 0 &\iff c(\omega) \leq q|r(\omega)|.\end{aligned}$$

Finally, $z^{c(\omega)-q|r(\omega)|} = z^{c(\omega_\varepsilon)-q|r(\omega_\varepsilon)|}z^{c(\omega_\Sigma)-q|r(\omega_\Sigma)|}$. Therefore for each $z \in (0;1)$ we obtain:

$$\begin{aligned}\sum_{\boldsymbol{s}\in\mathbb{Z}} a(U,q;\boldsymbol{s})\boldsymbol{t}^{\boldsymbol{s}} &\preceq \sum_{\omega_\varepsilon\in(Op_\varepsilon)^*} \sum_{\substack{\omega_\Sigma\in(Op_\Sigma)^*, l(\omega_\Sigma)=U,\\ c(\omega_\varepsilon+c(\omega_\Sigma)\leq q|r(\omega_\varepsilon)|+q|r(\omega_\Sigma)|}} \binom{|\omega_\varepsilon|+n}{|\omega_\varepsilon|}\boldsymbol{t}^{||r(\omega_\varepsilon||+||r(\omega_\Sigma||}\\ &\preceq \sum_{\omega_\varepsilon\in(Op_\varepsilon)^*} \sum_{\substack{\omega_\Sigma\in(Op_\Sigma)^*\\ l(\omega_\Sigma)=U}} \binom{|\omega_\varepsilon|+n}{|\omega_\varepsilon|}\boldsymbol{t}^{||r(\omega_\varepsilon||+||r(\omega_\Sigma||}z^{c(\omega_\varepsilon)-q|r(\omega_\varepsilon|}z^{c(\omega_\Sigma)-q|r(\omega_\Sigma)|}.\end{aligned}$$

Now we can single out the sums over $\omega_\varepsilon$ and $\omega_\Sigma$ and obtain:

$$\begin{aligned}\sum_{\boldsymbol{s}\in\mathbb{Z}} a(U,q;\boldsymbol{s})\boldsymbol{t}^{\boldsymbol{s}} &\preceq \sum_{\omega_\varepsilon\in(Op_\varepsilon)^*} \binom{|\omega_\varepsilon|+n}{|\omega_\varepsilon|}\boldsymbol{t}^{||r(\omega_\varepsilon||}z^{c(\omega_\varepsilon)-q|r(\omega_\varepsilon|} \sum_{\substack{\omega_\Sigma\in(Op_\Sigma)^*\\ l(\omega_\Sigma)=U}} \boldsymbol{t}^{||r(\omega_\Sigma||}z^{c(\omega_\Sigma)-q|r(\omega_\Sigma|}\\ &= \sum_{\omega_\varepsilon\in(Op_\varepsilon)^*} \binom{|\omega_\varepsilon|+n}{|\omega_\varepsilon|}(z^{-q}\boldsymbol{t})^{||r(\omega_\varepsilon||}z^{c(\omega_\varepsilon)} \sum_{\substack{\omega_\Sigma\in(Op_\Sigma)^*\\ l(\omega_\Sigma)=U}} (z^{-q}\boldsymbol{t})^{||r(\omega_\Sigma)||}z^{c(\omega_\Sigma)}.\end{aligned}$$

We bound uniformly each of the two sums. First we deal with the easier

one:

$$\begin{aligned}
\sum_{\omega_\varepsilon\in(Op_\varepsilon)^*}\binom{|\omega_\varepsilon|+n}{|\omega_\varepsilon|}(z^{-q}\boldsymbol{t})^{||r(\omega_\varepsilon||}z^{c(\omega_\varepsilon)} &= \sum_{k=0}^{\infty}\sum_{\omega_\varepsilon\in(Op_\varepsilon)^k}\binom{|\omega_\varepsilon|+n}{|\omega_\varepsilon|}(z^{-q}\boldsymbol{t})^{||r(\omega_\varepsilon||}z^{c(\omega_\varepsilon)}\\
&= \sum_{k=0}^{\infty}\sum_{\substack{\omega_\varepsilon=op_1\circ op_2\cdots\circ op_k\\ op_j\in Op_\varepsilon}}\binom{k+n}{k}\prod_{j=1}^{k}(z^{-q}\boldsymbol{t})^{||r(\omega_\varepsilon||}z^{c(\omega_\varepsilon)}\\
&= \sum_{k=0}^{\infty}\binom{k+n}{k}\sum_{\substack{op_1\circ op_2\cdots\circ op_k\\ op_j\in Op_\varepsilon}}(z^{-q}\boldsymbol{t})^{\sum_{j=1}^{k}||r(op_j||}z^{\sum_{j=1}^{k}c(op_j)}\\
&= \sum_{k=0}^{\infty}\binom{k+n}{k}\sum_{\substack{op_1\circ op_2\cdots\circ op_k\\ op_j\in Op_\varepsilon}}\prod_{j=1}^{k}(z^{-q}\boldsymbol{t})^{||r(op_j)||}z^{c(op_j)}\\
&= \sum_{k=0}^{\infty}\binom{k+n}{k}f_\varepsilon^k(z^{-q}\boldsymbol{t};z).
\end{aligned}$$

Finally, we step to the sum:

$$R(\boldsymbol{t};z)=\sum_{\omega_\Sigma\in(Op_\Sigma)^*,l(\omega_\Sigma)=U}(z^{-q}\boldsymbol{t})^{||r(\omega_\Sigma)||}z^{c(\omega_\Sigma)}$$

Our aim is to show that:

$$R(\boldsymbol{t};z)\preceq(z^{-q}\boldsymbol{t})^{||U||}f_\Sigma^{||U||}(z^{-q}\boldsymbol{t};z).$$

First, note that $||r(\omega_\Sigma)||=||l(\omega_\Sigma||+(||r(\omega_\Sigma||-||l(\omega_\Sigma)||)$. Since $l(\omega_\Sigma)=U$, we get:

$$\begin{aligned}
R(\boldsymbol{t};z) &= \sum_{\omega_\Sigma\in(Op_\Sigma)^*,l(\omega_\Sigma)=U}(z^{-q}\boldsymbol{t})^{||l(\omega_\Sigma||}(z^{-q}\boldsymbol{t})^{||r(\omega_\Sigma)||-||l(\omega_\Sigma||}z^{c(\omega_\Sigma)}\\
&= (z^{-q}\boldsymbol{t})^{||U||}\sum_{\omega_\Sigma\in(Op_\Sigma)^*,l(\omega_\Sigma)=U}(z^{-q}\boldsymbol{t})^{||r(\omega_\Sigma)||-||l(\omega_\Sigma||}z^{c(\omega_\Sigma)}.
\end{aligned}$$

Next we manipulate the sum $R_1(\boldsymbol{t};z)=\sum_{\omega_\Sigma\in(Op_\Sigma)^*,l(\omega_\Sigma)=U}(z^{-q}\boldsymbol{t})^{||r(\omega_\Sigma)||-||l(\omega_\Sigma||}z^{c(\omega_\Sigma)}$ by unfolding each alignment operation by operation:

$$\begin{aligned}
R_1(\boldsymbol{t};z) &= \sum_{\omega_\Sigma\in(Op_\Sigma)^*,l(\omega_\Sigma)=U}(z^{-q}\boldsymbol{t})^{||r(\omega_\Sigma)||-||l(\omega_\Sigma)||}z^{c(\omega_\Sigma)}\\
&= \sum_{\substack{op_1\circ op_2\cdots\circ op_m\in(Op_\Sigma)^*\\ l(op_1)\circ l(op_2)\ldots l(op_m)=U}}(z^{-q}\boldsymbol{t})^{\sum_{j=1}^{m}||r(op_j)||-||l(op_j)||}z^{\sum_{j=1}^{k}c(op_j)}\\
&= \sum_{\substack{op_1\circ op_2\cdots\circ op_m\in(Op_\Sigma)^*\\ l(op_1)\circ l(op_2)\ldots l(op_m)=U}}\prod_{j=1}^{k}(z^{-q}\boldsymbol{t})^{||r(op_j)||-||l(op_j)||}z^{c(op_j)}
\end{aligned}$$

Let $u_j = \sigma_{i_j}$ for $j = 1 \dots n$. We weaken the constraint $l(op_1) \circ l(op_2) \dots l(op_m) = U$ by replacing it with a requirement only on the first character of $l(op_i)$. Specifically, the first character of $l(op_i)$ has to match the corresponding character of $U$, but the next characters are not forced to this constraint. Formally we require:

$$l(op_1) \in Op_{i_1} \text{ and } l(op_j) \in Op_{i_{\sum_{j'<j} |l(op_{j'})|+1}}.$$

Clearly, since the coefficient before each term $t^s$ is some power of $z$ we get a non-smaller power series:

$$\begin{aligned} R_1(\boldsymbol{t}; z) &= \sum_{\substack{op_1 \circ op_2 \cdots \circ op_m \in (Op_\Sigma)^* \\ l(op_1) \circ l(op_2) \dots l(op_m) = U}} \prod_{j=1}^{k} (z^{-q}\boldsymbol{t})^{||r(op_j)|| - ||l(op_j)||} z^{c(op_j)} \\ &\preceq \sum_{\substack{op_1 \circ op_2 \cdots \circ op_m \\ l(op_1) \in Op_{i_1} \text{ and } l(op_j) \in Op_{i_{\sum_{j'<j} |l(op_{j'})|+1}}}} \prod_{j=1}^{k} (z^{-q}\boldsymbol{t})^{||r(op_j)|| - ||l(op_j)||} z^{c(op_j)} \end{aligned}$$

Finally we can uniquely map each such term to a term with the same value by adding to the set of operations the identity operations $(\sigma_{j_i}, \sigma_{j_i}) \in Op_{j_i}$ for al $j_i$ such that:

$$\exists k \leq m [\sum_{j' \leq k} |l(op_{j'})| < j_i < \sum_{j' \leq k+1} |l(op_{j'})|],$$

that is in the case that $u_j$ is covered by the operation $op_{k+1}$ but $u_j$ is not the starting position of $op_{k+1}$ we map it to the identity operation which has the characteristics $||r((\sigma_j, \sigma_j))|| - ||l((\sigma_j, \sigma_j)|| = 0$ and $c((\sigma_j, \sigma_j)) = 0$. Clearly this mapping is injective since it is uniquely determined by the set of operations $op_j$. Therefore:

$$\begin{aligned} R_1(\boldsymbol{t}; z) &\preceq \sum_{\substack{op_1 \circ op_2 \cdots \circ op_m \\ l(op_1) \in Op_{i_1} \text{ and } l(op_j) \in Op_{i_{\sum_{j'<j} |l(op_{j'})|+1}}}} \prod_{j=1}^{k} (z^{-q}\boldsymbol{t})^{||r(op_j)|| - ||l(op_j)||} z^{c(op_j)} \\ &\preceq \sum_{\substack{op_1 \circ op_2 \cdots \circ op_n \\ l(op_j) \in Op_{j_i}}} \prod_{j=1}^{k} (z^{-q}\boldsymbol{t})^{||r(op_j)|| - ||l(op_j)||} z^{c(op_j)} \\ &= \prod_{i=1}^{|\Sigma|} f_i^{||U||_i}(z^{-q}\boldsymbol{t}; z) = f_\Sigma^{||U||}(z^{-q}\boldsymbol{t}; z). \end{aligned}$$

Therefore we obtain that:

$$R(\boldsymbol{t}; z) = (z^{-q}\boldsymbol{t})^{||U||} R_1(\boldsymbol{t}; z) \preceq (z^{-q}\boldsymbol{t})^{||U||} f_\Sigma^{||U||}(z^{-q}\boldsymbol{t}; z)$$

This already proves that:

$$\begin{aligned}\sum_{\boldsymbol{s}\in\mathbb{Z}} a(U,q;\boldsymbol{s})\boldsymbol{t}^{\boldsymbol{s}} &\preceq \sum_{k=0}^{\infty}\binom{n+k}{k} f_{\varepsilon}^{k}(z^{-q}\boldsymbol{t};z)R(\boldsymbol{t};z)\\ &\preceq (z^{-q}\boldsymbol{t})^{||U||} f_{\Sigma}^{||U||}(z^{-q}\boldsymbol{t};z)\sum_{k=0}^{\infty}\binom{n+k}{k} f_{\varepsilon}^{k}(z^{-q}\boldsymbol{t};z).\end{aligned}$$

Now setting:

$$b_{U,q}(\boldsymbol{s},z) = [\boldsymbol{t}^{\boldsymbol{s}}](z^{-q}\boldsymbol{t})^{||U||} f_{\Sigma}^{||U||}(z^{-q}\boldsymbol{t};z)\sum_{k=0}^{\infty}\binom{n+k}{k} f_{\varepsilon}^{k}(z^{-q}\boldsymbol{t};z)$$

we get the result for the first part of the lemma and in the particular case that $0 \leq f_{\varepsilon}(z^{-q}\boldsymbol{t};z) < 1$ we obtain:

$$\begin{aligned}\sum_{\boldsymbol{s}\in\mathbb{Z}^{|\Sigma|}} b_{U,q}(\boldsymbol{s},z)\boldsymbol{t}^{\boldsymbol{s}} &= (z^{-q}\boldsymbol{t})^{||U||} f_{\Sigma}^{||U||}(z^{-q}\boldsymbol{t};z)\sum_{k=0}^{\infty}\binom{n+k}{k} f_{\varepsilon}^{k}(z^{-q}\boldsymbol{t};z)\\ &= (z^{-q}\boldsymbol{t})^{||U||} f_{\Sigma}^{||U||}(z^{-q}\boldsymbol{t};z)\frac{1}{(1-f_{\varepsilon}(z^{-q}\boldsymbol{t};z))^{|U|+1}}\\ &= \frac{(z^{-q}\boldsymbol{t})^{||U||}}{1-f_{\varepsilon}(z^{-q}\boldsymbol{t};z)}\left(\frac{f_{\Sigma}(z^{-q}\boldsymbol{t};z)}{1-f_{\varepsilon}(z^{-q}\boldsymbol{t};z)}\right)^{||U||}.\end{aligned}$$

□

For the main result in this Section we shall use the Definition 1.8.8 from Chapter 1. Formally, to each automaton $\mathcal{A}$ with no $\varepsilon$-transitions it assigns the power series:

$$g_{\mathcal{A}}(\boldsymbol{t}) = \sum_{\pi\in\Pi(\mathcal{A})} \boldsymbol{t}^{||\lambda(\pi)||}.$$

Finally, we define the measure we are actually interested in – the number of infixes of a given language which will be considered by the algorithms in Chapter 5 and Chapter 6 by an input $V$.

**Definition 7.1.5** For a word $V \in \Sigma^*$, language $\mathcal{L}$ and a rational number $q \in (0;1)$ we denote with $Gen_{\mathcal{L}}(V,q)$ the set of all alignments $\omega \in Op^*$ such that $l(\omega) \in Inf(\mathcal{L})$, $r(\omega) = V$ and $c(\omega) \leq q|V|$. Formally:

$$Gen_{\mathcal{L}}(V,q) = \{\omega \in Op^* \,|\, l(\omega) \in Inf(\mathcal{L}) \,\&\, r(\omega) = V \,\&\, c(\omega) \leq q|V|\}.$$

We set $gen_{\mathcal{L}}(V,q) = |Gen_{\mathcal{L}}(V,q)|$.

Based on Lemma 7.1.4 we are able to uniformly bound the values $gen_{\mathcal{L}}(V)$:

**Lemma 7.1.6** *Let $\mathcal{A} = \langle \Sigma, Q, I, \Delta, T\rangle$ be an automaton and $q \in (0;1)$ be a rational number. Define the functions $v_i : \mathbb{R}^{|\Sigma|} \times \mathbb{R} \to \mathbb{R}$ and $\boldsymbol{v} : \mathbb{R}^{|\Sigma|} \times \mathbb{R} \to \mathbb{R}^{|\Sigma|}$ as:*

$$v_i(t, z) = z^{-q} t_i \frac{f_i(z^{-q}\boldsymbol{t}; z)}{1 - f_\varepsilon(z^{-q}\boldsymbol{t}; z)}$$
$$\boldsymbol{v}(\boldsymbol{t}; z) = (v_1(\boldsymbol{t}; z), \ldots, v_m(\boldsymbol{t}; z)).$$

*Then for each positive real vector $t \in \mathbb{R}_+^{|\Sigma|}$ and a real number $z \in (0;1)$ with the properties $f_\varepsilon(z^{-q}\boldsymbol{t}; z) < 1$, it holds:*

$$\textstyle\sum_{V\in\Sigma^*} gen_{\mathcal{L}(\mathcal{A})}(V, q)\boldsymbol{t}^{||V||} \le \frac{g_{\mathcal{A}}(\boldsymbol{v}(\boldsymbol{t},z))}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)}$$
$$\textstyle\sum_{V\in\Sigma^*}\sum_{\omega\in Gen_{\mathcal{L}(\mathcal{A})}(V,q)} |l(\omega)|\boldsymbol{t}^{||V||} \le \sum_{i=1}^{|\Sigma|} \frac{v_i(\boldsymbol{t};z)}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)} \frac{\partial g_{\mathcal{A}}}{\partial v_i}(\boldsymbol{v}(\boldsymbol{t};z)).$$

Note that $g_{\mathcal{A}}(\boldsymbol{v}(\boldsymbol{t}; z))$ may be $\infty$ if the power series diverge. In this case the the inequalities hold for trivial reasons. The interesting case is when $g_{\mathcal{A}}(\boldsymbol{v}(\boldsymbol{t}; z))$ converges. In Section 7.3 we shall consider some sufficient properties of the automaton $\mathcal{A}$ and the edit-distance which guarantee convergence.

*Proof.* (of Lemma 7.1.6) Note that $\omega \in Gen_{\mathcal{L}(\mathcal{A})}(V, q)$ implies that there $c(\omega) \le q|r(\omega)|$ and therefore $V = r(\omega) \in \mathcal{A}(l(\omega), q)$. Conversely if an alignment $\omega$ belongs to the set $\mathcal{A}(U, q)$ for some infix $U \in Inf(\mathcal{L})$, then clearly $\omega \in Gen_{\mathcal{L}(\mathcal{A})}(r(\omega), q)$. Thus we obtain:

$$\begin{aligned}
\sum_{V\in\Sigma^*} gen_{\mathcal{L}(\mathcal{A})}(V)\boldsymbol{t}^{||V||} &= \sum_{V\in\Sigma^*}\sum_{\omega\in Gen_{\mathcal{L}(\mathcal{A})}(V,q)} \boldsymbol{t}^{||V||} \\
&= \sum_{U\in Inf(\mathcal{L}(\mathcal{A}))}\sum_{\omega\in\mathcal{A}(U,q)} \boldsymbol{t}^{||r(\omega)||} \\
&= \sum_{U\in Inf(\mathcal{L}(\mathcal{A}))}\sum_{\boldsymbol{s}\in\mathbb{Z}^{|\Sigma|}} a(U, q; \boldsymbol{s})\boldsymbol{t}^{\boldsymbol{s}}.
\end{aligned}$$

Now, for a fixed infix $U$ and $f_\varepsilon(z^{-q}\boldsymbol{t}; z) < 1$, Lemma 7.1.4 states that:

$$\sum_{\boldsymbol{s}\in\mathbb{Z}^{|\Sigma|}} a(U; \boldsymbol{s})\boldsymbol{t}^{\boldsymbol{s}} \preceq \sum_{\boldsymbol{s}\in\mathbb{Z}^{|\Sigma|}} b_{U,q}(\boldsymbol{s}, z)\boldsymbol{t}^{\boldsymbol{s}} \text{ and}$$
$$\sum_{\boldsymbol{s}\in\mathbb{Z}^{|\Sigma|}} b_{U,q}(\boldsymbol{s}, z)\boldsymbol{t}^{\boldsymbol{s}} = \frac{(z^{-q}\boldsymbol{t})^{||U||}}{1 - f_\varepsilon(z^{-q}\boldsymbol{t}; z)} \left(\frac{f_\Sigma(z^{-q}\boldsymbol{t}; z)}{1 - f_\varepsilon(z^{-q}\boldsymbol{t}; z)}\right)^{||U||}$$

However, the right hand side of this equality is precisely $\frac{\boldsymbol{v}^{||U||}(\boldsymbol{t},z)}{1-f_\varepsilon(z^{-q}t;z)}$. Thus, since $t$ is a positive real vector we obtain:

$$\sum_{\boldsymbol{s}\in\mathbb{Z}^{|\Sigma|}} a(U, q; \boldsymbol{s})\boldsymbol{t}^{\boldsymbol{s}} \le \frac{1}{1 - f_\varepsilon(z^{-q}\boldsymbol{t}; z)} \boldsymbol{v}(\boldsymbol{t}, z)^{||U||}.$$

Now, it is obvious that:

$$\begin{aligned}\sum_{V\in\Sigma^*} gen_{\mathcal{L}(\mathcal{A})}(V,q)\boldsymbol{t}^{||V||} &\leq \frac{1}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)}\sum_{U\in Inf(\mathcal{L}(\mathcal{A}))}\boldsymbol{v}(\boldsymbol{t};z)^{||U||}\\ &\leq \frac{1}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)}g_{\mathcal{A}}(\boldsymbol{v}(\boldsymbol{t};z)).\end{aligned}$$

The last inequality follows by the fact that each infix of $\mathcal{L}(\mathcal{A})$ corresponds to at least one path in $\mathcal{A}$ and the observation that $v(t,z)$ is a nonnegative vector. For the second part of the Lemma note that:

$$\begin{aligned}\sum_{V\in\Sigma^*}\sum_{\omega\in Gen_{\mathcal{L}(\mathcal{A})}(V,q)}|l(\omega)|\boldsymbol{t}^{||V||} &= \sum_{U\in Inf(\mathcal{L}(\mathcal{A}))}\sum_{\omega\in\mathcal{A}(U,q)}|l(\omega)|\boldsymbol{t}^{||r(\omega)||}\\ &= \sum_{U\in Inf(\mathcal{L}(\mathcal{A}))}\sum_{\omega\in\mathcal{A}(U,q)}|U|\boldsymbol{t}^{||r(\omega)||}\\ &\preceq \frac{1}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)}\sum_{U\in Inf(\mathcal{L}(\mathcal{A}))}|U|\boldsymbol{v}^{||U||}(\boldsymbol{t};z)\\ &= \frac{1}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)}\sum_{U\in Inf(\mathcal{L}(\mathcal{A}))}\sum_{i=1}^{|\Sigma|}v_i(\boldsymbol{t};z)\frac{\partial\boldsymbol{v}^{||U||}}{\partial v_i}(\boldsymbol{t};z)\\ &= \frac{1}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)}\sum_{i=1}^{|\Sigma|}v_i(\boldsymbol{t};z)\frac{\partial\sum_{U\in Inf(\mathcal{L}(\mathcal{A}))}\boldsymbol{v}^{||U||}}{\partial v_i}(\boldsymbol{t};z)\\ &\preceq \frac{1}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)}\sum_{i=1}^{|\Sigma|}v_i(\boldsymbol{t};z)\frac{\partial g_{\mathcal{A}}}{\partial v_i}(\boldsymbol{v}(\boldsymbol{t};z))\\ &= \sum_{i=1}^{|\Sigma|}\frac{v_i(\boldsymbol{t};z)}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)}\frac{\partial g_{\mathcal{A}}}{\partial v_i}(\boldsymbol{v}(\boldsymbol{t};z))\end{aligned}$$

□

The result from Lemma 7.1.6 refers to all the words $V\in\Sigma^*$. If we consider more carefully the proof, we will realise that it can be strengthened if we are interested only of long enough words $V\in\Sigma^*$. Indeed, if $N$ is a fixed integer and $V\in\Sigma^*$ is of length $|V|\geq N$, then all the alignments $\omega\in Gen_{\mathcal{L}}(V,q)$ have the property:

$$c(\omega)\leq q|V|.$$

Now, for every operation $op\in Op$ we have that $|r(op)|-|l(op)|\leq\rho$ and furthermore $|r(op)|-|l(op)|>0$ implies that $c(op)\geq 1$. Therefore:

$$q|V|\geq c(\omega)\geq\frac{1}{\rho}(|r(\omega)|-|l(\omega)|).$$

Since $r(\omega)=V$, we deduce that $|l(\omega)|\geq(\frac{1}{\rho}-q)|V|\geq(\frac{1}{\rho}-q)N$. Therefore words $U\in Inf(\mathcal{L})$ of length less than $(\frac{1}{\rho}-q)N$ will never belong to some set

$Gen_{\mathcal{L}}(V,q)$ provided that $|V| \geq N$. Thus, if we are interested only of the words generated by words $V$ of length $|V| \geq N$ we can replace the generating function $g_{\mathcal{A}}$ that reflects all the infixes in $\mathcal{L}(\mathcal{A})$ with the generating function $g_{\mathcal{A},n}$ that reflects the infixes in $\mathcal{L}(\mathcal{A})$ of length at least $n = (\frac{1}{\rho} - q)N$. Formally, we obtain the following result:

**Corollary 7.1.7** *In the notation of Lemma 7.1.6 if $n_0$ is positive integer and $\frac{1}{\rho} > q$, and $n_1 = (\frac{1}{\rho} - q)n_0$, then for each positive real vector $\boldsymbol{t} \in \mathbb{R}_+^{|\Sigma|}$ and a real number $z \in (0;1)$ with the properties $f_\varepsilon(z^{-q}\boldsymbol{t}, z) < 1$, it holds:*

$$\sum_{V\in\Sigma^*} gen_{\mathcal{L}(\mathcal{A})}(V,q)\boldsymbol{t}^{||V||} \leq \frac{g_{\mathcal{A},n_1}(\boldsymbol{v}(\boldsymbol{t},z))}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)}$$
$$\sum_{V\in\Sigma^*}\sum_{\omega\in Gen_{\mathcal{L}(\mathcal{A})}(V,q)} |l(\omega)|\boldsymbol{t}^{||V||} \leq \sum_{i=1}^{|\Sigma|} \frac{v_i(\boldsymbol{t};z)}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)} \frac{\partial g_{\mathcal{A},n_1}}{\partial v_i}(\boldsymbol{v}(\boldsymbol{t};z)).$$

□

## 7.2 Average Time Complexity of the Extension Steps

The technique developed in the previous section allows us to estimate the running time of the algorithms described in Chapter 5 and Chapter 6. The analysis is carried out on average under the assumption of independent distribution of the characters $\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}$ in the alphabet $\Sigma$.

We start with the following folklore which is however useful for the complete understanding of the result:

**Lemma 7.2.1** *Let $pr : \Sigma \to (0;1)$ be a probability on $\Sigma$. Assume that the characters of $\Sigma$ are independently distributed in $\Sigma^*$ with distribution $pr$. Let $t_i = pr(\sigma_i)$ for $i = 1 \ldots |\Sigma|$ and $j, n, N \in \mathbb{N}$ be such that $j + n \leq N$. If $W \in \Sigma^n$, then:*

$$\sum_{I_{j+1}^{j+n}(V)]=W} pr(V|V \in \Sigma^N) = t^{||W||}.$$

*Proof.* Indeed since the characters in $\Sigma$ are independently distributed we have that:

$$pr(V|V \in \Sigma^N) = t^{||V||} = t^{||I_1^j(V)||}t^{||I_{j+1}^{j+n}(V)||}t^{||I_{j+n+1}^N(V)||}.$$

Now using once again the independence of the characters' distribution we obtain:

$$\begin{aligned}
\sum_{I_{j+1}^{j+n}(V)=W} pr(V|V\in\Sigma^N) &= \sum_{\substack{V\in\Sigma^N\\ I_{j+1}^{j+n}(V)=W}} pr(V)\\
&= \sum_{\substack{V\in\Sigma^N\\ I_{j+1}^{j+n}(V)=W}} t^{||I_1^j(V)||}t^{||I_{j+1}^{j+n}(V)||}t^{||I_{j+n+1}^N(V)||}\\
&= \sum_{\substack{V\in\Sigma^N\\ I_{j+1}^{j+n}(V)=W}} t^{||I_1^j(V)||}t^{||W||}t^{||I_{j+n+1}^N(V)||}\\
&= t^{||W||}\sum_{V_1\in\Sigma^j} t^{||V_1||}\sum_{V_2\in\Sigma^{N-j-n}} t^{||V_2||}\\
&= t^{||W||}.
\end{aligned}$$

□

**Proposition 7.2.2** *Let $pr:\Sigma\to(0;1)$ be a probability and $t_i=pr(\sigma_i)$ for $\sigma_i\in\Sigma$. Let $q\in(0;1)$ and $z\in(0;1)$ be such that $f_\varepsilon(z^{-4q}t;z)<1$ and $f_\varepsilon(z^{-2q}t;z)<1$. Define the functions $v_i(t;z)$ and $v_i^{(2)}(t;z)$, and $v(t;z)$ and $v^{(2)}(t;z)$ as:*

$$\begin{aligned}
v_i(t;z) &= \tfrac{z^{-2q}t_if_i(z^{-2q}t;z)}{1-f_\varepsilon(z^{-2q}t;z)}\\
v(t;z) &= (v_1(t;z),v_2(t;z),\dots,v_{|\Sigma|}(t;z)\\
v_i^{(2)} &= \tfrac{z^{-4q}t_if_i(z^{-4q}t;z)}{1-f_\varepsilon(z^{-4q}t;z)}\\
v^{(2)}(t;z) &= (v_1^{(2)}(t;z),v_2^{(2)}(t;z),\dots,v_{|\Sigma|}^{(2)}(t;z).
\end{aligned}$$

*If $\mathcal{A}$ is a finite state automaton with language $\mathcal{L}=\mathcal{L}(\mathcal{A})$ and generating function $g_\mathcal{A}$ and $N\in\mathbb{N}$ then the expected running time $\mathbb{E}_{V\in\Sigma^N}T(V)$ of the algorithm in Chapter 6, is bounded by:*

$$\mathbb{E}_{V\in\Sigma^N}T(V)\le 2c_0N\left(\frac{g_\mathcal{A}(v^{(2)}(t;z))}{1-f_\varepsilon(z^{-4q}t;z)}+\sum_{i=1}^{|\Sigma|}\frac{v_i(t;z)}{1-f_\varepsilon(z^{-2q}t;z)}\frac{\partial g_\mathcal{A}}{\partial v_i}(v(t;z))\right),$$

*where $c_0=c_0(Op)$ is some global constant.*

*Proof.* For a fixed a word $V\in\Sigma^N$ and $\rho=\rho(Op)$, let:

$$\begin{aligned}
T_0(V) &= \textstyle\sum_{\alpha\in\mathcal{T}(V)}\sum_{k=0}^{\rho-1}\sum_{j=0}^{|V_{\alpha1}|}|L_r(\alpha,k,j)|\\
T_1(V) &= \textstyle\sum_{\alpha\in\mathcal{T}(V)}\sum_{k=0}^{\rho-1}\sum_{j=1}^{|V_{\alpha0}|}|L_l(\alpha,k,j)|\\
T_2(V) &= \textstyle\sum_{\alpha\in\mathcal{T}(V)}\sum_{k_0=0}^{\rho-1}\sum_{k_1=0}^{\rho-1}\sum_{U\in L(\alpha,k_0,k_1)}|U|.
\end{aligned}$$

Then by Proposition 6.2.8 there exists a constant $c_0 = c_0(Op)$ such that the time spent for answering the query $V$ is $T(V) \leq c_0(T_0(V) + T_1(V) + T_2(V))$. Therefore:

$$\mathbb{E}_{V\in\Sigma^N} T(V) \leq c_0(\mathbb{E}_{V\in\Sigma^N} T_0(V) + \mathbb{E}_{V\in\Sigma^N} T_1(V) + \mathbb{E}_{V\in\Sigma^N} T_2(V)).$$

Next we have that $|L_r(\alpha,k,j)| \leq gen_{\mathcal{L}}(W,4q)$ where $W = I_{k+1}^{N_{\alpha_0}+j}(V_\alpha)$. This follows by the following argument. Each pair $\langle U, c_U\rangle \in L_r(\alpha,k,j)$ has the property that $d(U,W) \leq q|V_\alpha|$. Now taking into account that $k \leq \rho - 1$ and $|V_{\alpha 0}| \geq \frac{|V_\alpha|}{2}$ we deduce that:

$$|V_\alpha| \leq 2|V_{\alpha 0}| \leq 2|I_{k+1}^{N_{\alpha 0}}(V_{\alpha 0})| + 2k \leq 4|I_{k+1}^{N_{\alpha 0}}(V_{\alpha 0})| \leq 4|W|.$$

Similar argument shows that $|L_l(\alpha,k,j)| \leq gen_{\mathcal{L}}(W;4q)$ where $W = I_{N_{\alpha 0}-j}^{N_\alpha - k}(V_\alpha)$. However, here we use that $j \geq 0$ and $|V_{\alpha 1}| \geq \frac{|V_\alpha|-1}{2}$ which yields the same result.

We consider each of the terms $\mathbb{E}_{V\in\Sigma^N} T_0(V)$, $\mathbb{E}_{V\in\Sigma^N} T_1(V)$ and $\mathbb{E}_{V\in\Sigma^N} T_2(V)$ separately. However the arguments for the first two are very similar. We start with $\mathbb{E}_{V\in\Sigma^N} T_0(V)$. Note that the structure of the tree $\mathcal{T}(V)$ depends only on the length $N = |V|$ of $V$. Thus, we can consider $\mathcal{T}(V) = \mathcal{T}(N)$ and we set $N_\alpha = |V_\alpha|$. Hence:

$$\begin{aligned}
\mathbb{E}_{V\in\Sigma^N} T_0(V) &= \sum_{V\in\Sigma^N} pr(V) \sum_{\alpha\in\mathcal{T}(V)} \sum_{k=0}^{\rho-1} \sum_{j=0}^{|N_{\alpha 1}|} gen_{\mathcal{L}}(I_{k+1}^{N_{\alpha 0}+j}(V_\alpha), 4q) \\
&= \sum_{\alpha\in\mathcal{T}(N)} \sum_{k=0}^{\rho-1} \sum_{j=0}^{|N_{\alpha 1}|} \sum_{V\in\Sigma^N} gen_{\mathcal{L}}(I_{k+1}^{N_{\alpha 0}+j}(V_\alpha), 4q) pr(V) \\
&= \sum_{\alpha\in\mathcal{T}(N)} \sum_{k=0}^{\rho-1} \sum_{j=0}^{|N_{\alpha 1}|} \sum_{W\in\Sigma^{N_{\alpha 0}-k+j}} \sum_{\substack{V\in\Sigma^N \\ I_{k+1}^{N_{\alpha 0}+j}(V_\alpha)=W}} gen_{\mathcal{L}}(W,4q) pr(V) \\
&= \sum_{\alpha\in\mathcal{T}(N)} \sum_{k=0}^{\rho-1} \sum_{j=0}^{|V_{\alpha 1}|} \sum_{W\in\Sigma^{N_{\alpha 0}-k+j}} gen_{\mathcal{L}}(W,4q) \sum_{\substack{V\in\Sigma^N \\ k^{-1}V_{\alpha 0}\circ(j]V_{\alpha 1}=W}} pr(V) \\
&= \sum_{\alpha\in\mathcal{T}(N)} \sum_{k=0}^{\rho-1} \sum_{j=0}^{|V_{\alpha 1}|} \sum_{W\in\Sigma^{N_{\alpha 0}-k+j}} gen_{\mathcal{L}}(W,4q) pr(W) \\
&= \sum_{\alpha\in\mathcal{T}(N)} \sum_{k=0}^{\rho-1} \sum_{j=0}^{|V_{\alpha 1}|} \sum_{W\in\Sigma^{N_{\alpha 0}-k+j}} gen_{\mathcal{L}}(W,4q) \boldsymbol{t}^{||W||}
\end{aligned}$$

where the last two equalities follow by Lemma 7.2.1. Now suppose that the triples $\langle\alpha,k_1,j_1\rangle$ and $\langle\beta,k_2,j_2\rangle$ specify the same subintervals in $[1;N]$. Since $N_\alpha \geq 4(\rho-1)$ and $k_1 < \rho$ and similarly $N_\beta \geq 4(\rho-1)$ and $k_2 \leq \rho-1$, we get that $\beta$ is an ancestor of $\alpha$ or vice versa and $k_1 = k_2$. W.l.o.g. assume that $\beta$ is an

ancestor of $\alpha$. Hence $\beta$ has as descendants $\alpha 0$ and $\alpha 1$. Now $I_{k_2+1}^{N_{\beta 0}}(V_\beta)$ properly contains $I_{k_1+1}^{N_\alpha}(V_\alpha)$ unless $\beta 0 = \alpha$. In the former case $I_{k_1+1}^{N_\alpha}(V_\alpha) = I_{k_1+1}^{N_{\beta 0}+j_2}(V_\beta)$ if and only if $j_1 = N_{\alpha 1}$ and $j_2 = 0$. Hence each subinterval of $[1; N]$ is described by at most two triples $\langle \alpha, k, j \rangle$. Since there are at most $O(N)$ subintervals of $[1; N]$ of certain length $|W|$ we obtain:

$$\begin{aligned}
\mathbb{E}_{V \in \Sigma^N} T_0(V) &= \sum_{\alpha \in \mathcal{T}(N)} \sum_{k=0}^{\rho_0} \sum_{j=0}^{|V_{\alpha 1}|} \sum_{W \in \Sigma^{|V_{\alpha 0}|-k+j}} gen_{\mathcal{L}}(W, 4q) \boldsymbol{t}^{||W||} \\
&\leq 2N \sum_{n=0}^{N} \sum_{W \in \Sigma^n} gen_{\mathcal{L}}(W, 4q) \boldsymbol{t}^{||W||} \\
&\leq 2N \sum_{n=0}^{\infty} \sum_{W \in \Sigma^n} gen_{\mathcal{L}}(W, 4q) \boldsymbol{t}^{||W||} \\
&\leq 2N \frac{g_{\mathcal{A}}(\boldsymbol{v}^{(2)}(\boldsymbol{t}; z))}{1 - f_\varepsilon(z^{-4q}\boldsymbol{t}; z)}.
\end{aligned}$$

The last inequality follows by the first part of Lemma 7.1.6.

Analogously, one can show that:

$$\mathbb{E}_{V \in \Sigma^N} T_1(V) \leq 2 \frac{g_{\mathcal{A}}(\boldsymbol{v}^{(2)}(\boldsymbol{t}; z))}{1 - f_\varepsilon(z^{-4q}\boldsymbol{t}; z)}.$$

Finally, consider the expectation $\mathbb{E}_{V \in \Sigma^N} T_2(V)$. Here, we use that a pair $\langle U, c_U \rangle \in L(\alpha, k_0, k_1)$ will be generated if and only if $d(U, W) = c_U \leq q|V_\alpha|$ where $W = V_\alpha[k_0; k_1]$. However, since $k_0, k_1 \leq \rho - 1$ and $N_\alpha = |V_\alpha| \geq 4(\rho - 1)$ for every inner node $\alpha$, we obtain that:

$$|W| \geq |V_\alpha| - 2(\rho - 1) \geq |V_\alpha| - \frac{|V_\alpha|}{2} = \frac{|V_\alpha|}{2}.$$

This implies that for every pair $\langle U, c_U \rangle \in L(\alpha, k_0, k_1)$, there is an alignment

$\omega \in Gen(W, 2q)$ with $l(\omega) = U$. Therefore we can write:

$$\begin{aligned}
\mathbb{E}_{V\in\Sigma^N} T_2(V) &= \sum_{V\in\Sigma^N} T_2(V) pr(V) \\
&\leq \sum_{V\in\Sigma^N} \sum_{k_0,k_1=0}^{\rho-1,\rho-1} \sum_{P_\alpha\in\mathcal{T}(N)} \sum_{\omega\in Gen(V_\alpha[k_0;k_1],2q)} |l(\omega)| pr(V) \\
&= \sum_{k_0,k_1=0}^{\rho-1,\rho-1} \sum_{\alpha\in\mathcal{T}(N)} \sum_{W\in\Sigma^{N_\alpha-k_0-k_1}} \sum_{\substack{V\in\Sigma^N \\ V_\alpha[k_0;k_1]=W}} \sum_{\omega\in Gen(W,2q)} |l(\omega)| pr(V) \\
&= \sum_{k_0,k_1=0}^{\rho-1,\rho-1} \sum_{\alpha\in\mathcal{T}(N)} \sum_{W\in\Sigma^{N_\alpha-k_0-k_1}} \sum_{\omega\in Gen(W,2q)} |l(\omega)| \sum_{\substack{V\in\Sigma^N \\ V_\alpha[k_0;k_1]=W}} pr(V) \\
&= \sum_{k_0,k_1=0}^{\rho-1,\rho-1} \sum_{\alpha\in\mathcal{T}(N)} \sum_{W\in\Sigma^{N_\alpha-k_0-k_1}} \sum_{\omega\in Gen(W,2q)} |l(\omega)| pr(W) \\
&= \sum_{k_0,k_1=0}^{\rho-1,\rho-1} \sum_{\alpha\in\mathcal{T}(N)} \sum_{W\in\Sigma^{N_\alpha-k_0-k_1}} \sum_{\omega\in Gen(W,2q)} |l(\omega)| \boldsymbol{t}^{||W||}.
\end{aligned}$$

Clearly, for different triples $\langle \alpha, k_0, k_1\rangle$ the subintervals $V_\alpha[k_0; k_1]$ define different subintervals of $[1; N]$. Therefore:

$$\begin{aligned}
\mathbb{E}_{V\in\Sigma^N} T_2(V) &\leq \sum_{k_0,k_1=0}^{\rho-1,\rho-1} \sum_{\alpha\in\mathcal{T}(N)} \sum_{W\in\Sigma^{N_\alpha-k_0-k_1}} \sum_{\omega\in Gen(W,2q)} |l(\omega)| \boldsymbol{t}^{||W||} \\
&\leq N \sum_{n=0}^{N} \sum_{W\in\Sigma^n} \sum_{\omega\in Gen(W,2q)} |l(\omega)| \boldsymbol{t}^{||W||} \\
&\leq N \sum_{n=0}^{\infty} \sum_{W\in\Sigma^n} \sum_{\omega\in Gen(W,2q)} |l(\omega)| \boldsymbol{t}^{||W||} \\
&\leq N \sum_{i=1}^{|\Sigma|} \frac{v_i(\boldsymbol{t}; z)}{1 - f_\varepsilon(z^{-2q}\boldsymbol{t}; z)} \frac{\partial g_{\mathcal{A}}}{\partial v_i}(\boldsymbol{v}(\boldsymbol{t}; z))
\end{aligned}$$

The last inequality follows by the second part of Lemma 7.1.6. Summing up we get that the expected time of the algorithm in Chapter 6 is:

$$\mathbb{E}_{V\in\Sigma^N} T(V) \leq c_0(\mathbb{E}_{V\in\Sigma^N} T_0(V) + \mathbb{E}_{V\in\Sigma^N} T_1(V) + \mathbb{E}_{V\in\Sigma^N} T_2(V)).$$

Under the assumptions that $g_{\mathcal{A}}(\boldsymbol{v}^{(2)}(\boldsymbol{t}; z))$ and $\frac{\partial g_{\mathcal{A}}}{\partial v_i}(\boldsymbol{v}(\boldsymbol{t}; z))$ converge we get that $\mathbb{E}_{V\in\Sigma^N} T(V) \in O(N)$ since the set of operations $Op$ is of fixed.

□

Taking into account Corollary 7.1.7 and the discussion from Chapter 6, Subsection 6.2.4, we get that:

**Corollary 7.2.3** *In the notation of Proposition 7.2.2, assume that we further dispose on an index providing the answers for all the queries of length less than* $2n_0 + 4(\rho - 1)$ *for some integer number* $n_0$ *in the sense of Lemma 6.2.11. Let* $n_1 = (\frac{1}{\rho} - 2q)n_0$ *and* $n_2 = (\frac{1}{\rho} - 4q)n_0$*, then:*

$$\mathbb{E}_{V\in\Sigma^N} T(V) \leq 2c_0' N \left( \frac{g_{\mathcal{A},n_2}(\boldsymbol{v}^{(2)}(\boldsymbol{t};z))}{1 - f_\varepsilon(z^{-4q}t;z)} + \sum_{i=1}^{|\Sigma|} \frac{v_i(\boldsymbol{t};z)}{1 - f_\varepsilon(z^{-2q}\boldsymbol{t};z)} \frac{\partial g_{\mathcal{A},n_1}}{\partial v_i}(\boldsymbol{v}(\boldsymbol{t};z)) \right),$$

*where* $c_0' = c_0'(Op)$ *is some absolute constant.*

**Remark 7.2.4** In the proof of Proposition 7.2.2 we substituted the natural parameter $q$ with $2q$ and $4q$, respectively. The reason is that we needed a uniform bound of the ratio of the lengths of the words $I_{k_0+1}^{N_\alpha - k_1}(V_\alpha)$ and $V_\alpha$ on the one hand, and the lengths of words $I_{k_0+1}^{N_{\alpha 0}+j}(V_\alpha)$ and $V_\alpha$, on the other. Since $k_0, k_1$ are globally bounded by $\rho$ and the length of $V_\alpha$ is always at least $4(\rho-1)$, we easily got an estimate of $\frac{1}{2}$ for the first ratio and $\frac{1}{4}$ for the second ratio.

**Remark 7.2.5** In view of Remark 7.2.4 it follows that with the increase of the lengths of words $V_\alpha$ the first ratio will tend to 1 and the second ratio will tend to $\frac{1}{2}$. Thus, modifying the initialisation step allowing longer lengths for the leaves of the tree $\mathcal{T}(V)$ we will increase the time efficiency of the algorithm. However, the space requirements will also increase exponentially with the longest admissible length of the tree.

**Remark 7.2.6** Actually, if $\rho(Op) = 1$, then the only possible values for $k_0$ and $k_1$ is $k_0 = k_1 = 0$. Hence, $I_{k_0+1}^{N_\alpha - k_1}(V_\alpha) = V_\alpha$ and thus first ratio of lengths is 1. Similarly, the ratio of the lengths of $I_{k_0+1}^{N_{\alpha 0}+j}(V_\alpha) = I_1^{N_{\alpha 0}+j}(V_\alpha)$ and $V_\alpha$ is at least $\frac{1}{2}$.

In view of Remark 7.2.6 we can strengthen the statement from Proposition 7.2.2 in the special case $\rho(Op) = 1$:

**Proposition 7.2.7** *Let* $pr : \Sigma \to (0;1)$ *be a probability and* $t_i = pr(\sigma_i)$ *for* $\sigma_i \in \Sigma$ *and* $\rho(Op) = 1$*. Let* $q \in (0;1)$ *and* $z \in (0;1)$ *be such that* $f_\varepsilon(z^{-2q}\boldsymbol{t};z) < 1$ *and* $f_\varepsilon(z^{-q}\boldsymbol{t};z) < 1$*. Define the functions* $v_i(\boldsymbol{t};z)$ *and* $v_i^{(2)}(\boldsymbol{t};z)$*, and* $\boldsymbol{v}(\boldsymbol{t};z)$ *and* $\boldsymbol{v}^{(2)}(\boldsymbol{t};z)$ *as:*

$$v_i(\boldsymbol{t};z) = \tfrac{z^{-q}t_i f_i(z^{-q}\boldsymbol{t};z)}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)}$$
$$\boldsymbol{v}(\boldsymbol{t};z) = (v_1(\boldsymbol{t};z), v_2(\boldsymbol{t};z), \ldots, v_{|\Sigma|}(\boldsymbol{t};z)$$
$$v_i^{(2)} = \tfrac{z^{-2q}t_i f_i(z^{-2q}\boldsymbol{t};z)}{1-f_\varepsilon(z^{-2q}\boldsymbol{t};z)}$$
$$\boldsymbol{v}^{(2)}(\boldsymbol{t};z) = (v_1^{(2)}(\boldsymbol{t};z), v_2^{(2)}(\boldsymbol{t};z), \ldots, v_{|\Sigma|}^{(2)}(\boldsymbol{t};z).$$

*If* $\mathcal{A}$ *is a finite state automaton with language* $\mathcal{L} = \mathcal{L}(\mathcal{A})$ *and generating function* $g_\mathcal{A}$ *and* $N \in \mathbb{N}$ *then the expected running time* $\mathbb{E}_{V\in\Sigma^N} T(V)$ *of the algorithm in*

*Chapter 6, is bounded by:*

$$\mathbb{E}_{V\in\Sigma^N}T(V) \leq 2c_1N\left(\frac{g_{\mathcal{A}}(\boldsymbol{v}^{(2)}(\boldsymbol{t};z))}{1-f_\varepsilon(z^{-2q}\boldsymbol{t};z)}+\sum_{i=1}^{|\Sigma|}\frac{v_i(\boldsymbol{t};z)}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)}\frac{\partial g_{\mathcal{A}}}{\partial v_i}(\boldsymbol{v}(\boldsymbol{t};z))\right),$$

*where $c_1 = c_1(Op)$ is some global constant.*

*Proof.* Follows from the proof of Proposition 7.2.2, Remark 7.2.4 and Remark 7.2.6.

□

Taking into account Remark 7.2.6 and Corollary 7.2.3 in the special case when $\rho = 1$ we obtain:

**Corollary 7.2.8** *In the notation of Proposition 7.2.7, let $\rho(Op) = 1$ and assume that we further dispose on an index providing the answers for all the queries of length less than $2n_0$ for some integer number $n_0$. Let $n_1 = (1-q)n_0$ and $n_2 = (1-2q)n_0$, then:*

$$\mathbb{E}_{V\in\Sigma^N}T(V) \leq 2c_1'N\left(\frac{g_{\mathcal{A},n_2}(\boldsymbol{v}^{(2)}(\boldsymbol{t};z))}{1-f_\varepsilon(z^{-2q}\boldsymbol{t};z)}+\sum_{i=1}^{|\Sigma|}\frac{v_i(\boldsymbol{t};z)}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)}\frac{\partial g_{\mathcal{A},n_1}}{\partial v_i}(\boldsymbol{v}(\boldsymbol{t};z))\right),$$

*where $c_1' = c_1'(Op)$ is some absolute constant and:*

□

## 7.3 Sufficient Convergency Conditions

Next we show some sufficient conditions which guarantee that the upper bounds given in terms of $g_{\mathcal{A}}(\boldsymbol{v}(\boldsymbol{t};z))$ exist. Here:

$$\begin{aligned} v_i(\boldsymbol{t};z) &= z^{-q}t_i\frac{f_i(z^{-q}\boldsymbol{t};z)}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)} \\ \boldsymbol{v}(\boldsymbol{t};z) &= (v_1(\boldsymbol{t};z),\ldots,v_{|\Sigma|}(\boldsymbol{t};z)). \end{aligned}$$

and $g_{\mathcal{A}}$ is the generating function for the automaton $\mathcal{A}$. These results are based on simple algebraic facts which successfully apply to finite state automata theory.

For the understanding this section the reader should recall the relationship between *automata and matrices* and the *norm* of a matrix, see Chapter 1, Section 1.7 and Section 1.8. As in Section 1.8 we shall assume that the set of states, $Q$, of some finite state automaton, $\mathcal{A}$, is identified with the first $|Q|$ positive integers, i.e. $\{1,2,\ldots,|Q|\}$.

In Section 1.8 we associated with each finite state automaton $\mathcal{A} = \langle Q,\Sigma,I,\Delta,T\rangle$ without $\varepsilon$-transitions the matrix $M_{\mathcal{A}}(\boldsymbol{t})$ with entries $a_{j,k}(\boldsymbol{t})$ for $1 \leq j,k \leq |Q|$, as:

$$a_{j,k}(\boldsymbol{t}) = \sum_{i:(j,\sigma_i,k)\in\Delta} t_i.$$

Then, using Definition 1.8.8 we also saw that:

$$g_{\mathcal{A}}(\boldsymbol{t}) = \sum_{\pi \in \Pi(\mathcal{A})} \boldsymbol{t}^{||\lambda(\pi)||} = \sum_{N=0}^{\infty} \sum_{\pi \in \Pi(\mathcal{A}):|\pi|=N} \boldsymbol{t}^{||\lambda(\pi)||} = \sum_{j=1}^{n} \sum_{k=1}^{n} a^*_{j,k}(\boldsymbol{t})$$

where $a^*_{j,k}(t)$ are the entries of the matrix:

$$M^*_{\mathcal{A}}(\boldsymbol{t}) = \sum_{N=0}^{\infty} M^N_{\mathcal{A}}(\boldsymbol{t})$$

defined in Chapter 1, Section 1.8.

One way to prove convergence of $g_{\mathcal{A}}(\boldsymbol{t})$ is to show that the norm of $||M^*_{\mathcal{A}}(t)||$ is less than $\infty$. To this end it suffices to show that:

$$||M_{\mathcal{A}}(\boldsymbol{t}))|| < 1$$

as we shall see.

Next lemma is a standard application of calculus which however favourably serves our purposes.

**Lemma 7.3.1** *Let $||.||$ be a norm on matrices and let $\boldsymbol{t} \in \mathbb{R}_+^{|\Sigma|}$. For a matrix $A(\boldsymbol{t}) = \{a_{i,j}(\boldsymbol{t})\}_{i,j=1}^n$ let $\frac{\partial A}{\partial t_k}(\boldsymbol{t})$ be the matrix with entries $a'_{i,j}(\boldsymbol{t}) = \frac{\partial a_{i,j}}{\partial t_k}(\boldsymbol{t})$ for every $i, j$ and $k \leq |\Sigma|$.*

1. *If $A(\boldsymbol{t})$ and $B(\boldsymbol{t})$ are arbitrary matrices and $C(\boldsymbol{t}) = A(t\boldsymbol{t}B(\boldsymbol{t})$, then.*

$$\frac{\partial C}{\partial t_k}(\boldsymbol{t}) = \frac{\partial A}{\partial t_k}(\boldsymbol{t})B(\boldsymbol{t}) + A(\boldsymbol{t})\frac{\partial B}{\partial t_k}(\boldsymbol{t}).$$

2. *For arbitrary $N \in \mathbb{N}$ and $k \leq |\Sigma|$, it holds:*

$$\frac{\partial A^N}{\partial t_k}(\boldsymbol{t}) = \sum_{m=0}^{N-1} A^m(\boldsymbol{t})\frac{\partial A}{\partial t_k}(\boldsymbol{t})A^{N-1-m}(\boldsymbol{t}).$$

*Proof.* The proof of the first part of the lemma follows by straightforward computation. Indeed since $C(\boldsymbol{t}) = A(\boldsymbol{t})B(\boldsymbol{t})$ we have:

$$c_{i,j}(\boldsymbol{t}) = \sum_{s=1}^{n} a_{i,s}(\boldsymbol{t})b_{s,j}(\boldsymbol{t}).$$

Therefore:

$$\frac{\partial c_{i,j}}{\partial t_k}(\boldsymbol{t}) = \sum_{s=1}^{n} \frac{\partial a_{i,s}}{\partial t_k}(\boldsymbol{t})b_{s,j}(\boldsymbol{t}) + \sum_{s=1}^{n} a_{i,s}(\boldsymbol{t})\frac{\partial b_{s,j}}{\partial t_k}(\boldsymbol{t}).$$

But $\frac{\partial a_{i,s}}{\partial t_k}(\boldsymbol{t})$ is exactly the $(i,s)$-entry of $\frac{\partial A}{\partial t_k}(\boldsymbol{t})$ whereas $\frac{\partial b_{s,j}}{\partial t_k}(\boldsymbol{t})$ is the $(s,j)$-entry of the matrix $\frac{\partial B}{\partial t_k}(\boldsymbol{t})$. Hence we obtain that:

$$\frac{\partial C}{\partial t_k}(\boldsymbol{t}) = \frac{\partial A}{\partial t_k}(\boldsymbol{t})B(\boldsymbol{t}) + A(\boldsymbol{t})\frac{\partial B}{\partial t_k}(\boldsymbol{t}).$$

The second part of the lemma now follows immediately by the first part by a straightforward induction argument on $N$. □

**Lemma 7.3.2** *Let $||.||$ be a norm on matrices and let $t \in \mathbb{R}_+^{|\Sigma|}$ and $\mathcal{A}$ be an $\varepsilon$-free finite state automaton with adjacency matrix $M_{\mathcal{A}}(\boldsymbol{t})$. If*

$$||M_{\mathcal{A}}(\boldsymbol{t})|| < 1.$$

*Then:*

1. $||M^*_{\mathcal{A}}(\boldsymbol{t})|| < \infty$.

2. $||\frac{\partial M^*_{\mathcal{A}}(\boldsymbol{t})}{\partial t_j}|| < \infty$ *for each* $j = 1 \dots |\Sigma|$.

*Proof.* The proof of the first part follows immediately by the triangle inequality and $||AB|| \leq ||A||||B||$ property. Specifically:

$$||M^*_{\mathcal{A}}(\boldsymbol{t})|| = ||\sum_{N=0}^{\infty} M^N_{\mathcal{A}}(\boldsymbol{t})|| \leq \sum_{N=0}^{\infty} ||M_{\mathcal{A}}(\boldsymbol{t})||^N = \frac{1}{1 - ||M_{\mathcal{A}}(\boldsymbol{t})||}.$$

The last equality follows by $||M_{\mathcal{A}}(\boldsymbol{t})|| < 1$.

For the second part we have:

$$\frac{\partial M^*_{\mathcal{A}}}{\partial t_j}(\boldsymbol{t}) = \sum_{N=0}^{\infty} \frac{\partial M^N_{\mathcal{A}}}{\partial t_j}(\boldsymbol{t}).$$

By the second part of Lemma 7.3.1 we deduce:

$$\frac{\partial M^*_{\mathcal{A}}}{\partial t_j}(\boldsymbol{t}) = \sum_{N=0}^{\infty} \sum_{m=0}^{N-1} M^m_{\mathcal{A}}(\boldsymbol{t}) \frac{\partial M_{\mathcal{A}}}{\partial t_j}(\boldsymbol{t}) M^{N-m-1}_{\mathcal{A}}(\boldsymbol{t}).$$

Now we apply the triangle inequality and the multiplication inequality with

respect to the matrix norm to get:

$$\begin{aligned}
\left|\left|\frac{\partial M^*_{\mathcal{A}}}{\partial t_j}(\boldsymbol{t})\right|\right| &= \left|\left|\sum_{N=0}^{\infty}\sum_{m=0}^{N-1} M^m_{\mathcal{A}}(\boldsymbol{t})\frac{\partial M_{\mathcal{A}}}{\partial t_j}(\boldsymbol{t})M^{N-n-1}_{\mathcal{A}}(\boldsymbol{t})\right|\right| \\
&\leq \sum_{N=0}^{\infty}\left|\left|\sum_{m=0}^{N-1} M^m_{\mathcal{A}}(\boldsymbol{t})\frac{\partial M_{\mathcal{A}}}{\partial t_j}(t)M^{N-m-1}_{\mathcal{A}}(\boldsymbol{t})\right|\right| \\
&\leq \sum_{N=0}^{\infty}\sum_{m=0}^{N-1} ||M_{\mathcal{A}}(\boldsymbol{t})||^m \left|\left|\frac{\partial M_{\mathcal{A}}}{\partial t_j}(\boldsymbol{t})\right|\right| ||M_{\mathcal{A}}(\boldsymbol{t})||^{N-m-1} \\
&= \sum_{N=0}^{\infty} N \left|\left|\frac{\partial M_{\mathcal{A}}}{\partial t_j}(\boldsymbol{t})\right|\right| ||M_{\mathcal{A}}(\boldsymbol{t})||^{N-1} \\
&= \left|\left|\frac{\partial M_{\mathcal{A}}}{\partial t_j}(\boldsymbol{t})\right|\right| \sum_{N=0}^{\infty} N||M_{\mathcal{A}}(\boldsymbol{t})||^{N-1} = \left|\left|\frac{\partial M_{\mathcal{A}}}{\partial t_j}(\boldsymbol{t})\right|\right| \frac{1}{\left(1-||M_{\mathcal{A}}(\boldsymbol{t})||\right)^2}
\end{aligned}$$

□

As a corollary we obtain.

**Lemma 7.3.3** *Let $\mathcal{A}$ be a finite state automaton, $z \in (0;1)$ and $\boldsymbol{t} \in \mathbb{R}_+^{|\Sigma|}$ satisfy the following properties:*

1. *$f_\varepsilon(z^{-q}\boldsymbol{t}; z) < 1$.*
2. *for each state $k \in Q$ it holds:*

$$\sum_{\langle k,\sigma_i,j\rangle\in\Delta} z^{-q}t_i \frac{f_i(z^{-q}\boldsymbol{t};z)}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)} < 1,$$

*then $g_{\mathcal{A}}(\boldsymbol{v}(\boldsymbol{t};z))$ and $\sum_{j=1}^{|\Sigma|} v_j(\boldsymbol{t};z)\frac{\partial g_{\mathcal{A}}}{\partial v_j}(\boldsymbol{v}(\boldsymbol{t};z))$ converge.*

*Proof.* For the proof we consider the norm $||.||_\infty$, see Section **??**:

$$||A||_\infty = \max_i \sum_{j=1}^{n} |a_{i,j}|.$$

Let $M_{\mathcal{A}}(\boldsymbol{t})$ be the adjacency matrix of the automaton $\mathcal{A}$. Hence, the norm of $||M_{\mathcal{A}}(\boldsymbol{v}(\boldsymbol{t};z))||_\infty$ is given as:

$$||M_{\mathcal{A}}(\boldsymbol{v}(\boldsymbol{t};z))||_\infty = \max_k \sum_{j=1}^{n} |M_{\mathcal{A}}(k,j;\boldsymbol{v}(\boldsymbol{t};z))|.$$

However, the functions $M_{\mathcal{A}}(k,j;\boldsymbol{v}(\boldsymbol{t};z)))$ are nothing else but:

$$M_{\mathcal{A}}(k,j;\boldsymbol{v}(\boldsymbol{t};z)) = \sum_{i=1}^{|\Sigma|}\sum_{\langle k,\sigma_i,j\rangle\in\Delta} v_i(t,z) = \sum_{i=1}^{|\Sigma|}\sum_{\langle k,\sigma_i,j\rangle\in\Delta} z^{-q}t_i\frac{f_i(z^{-q}\boldsymbol{t};z)}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)}$$

which are positive numbers under the assumptions of the lemma. Hence we get:

$$||M_{\mathcal{A}}(\boldsymbol{v}(\boldsymbol{t},z))||_\infty = \max_k \sum_{j=1}^{n} \sum_{\langle k,\sigma_i,j\rangle\in\Delta} z^{-q}t_i \frac{f_i(z^{-q}\boldsymbol{t};z)}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)}$$
$$= \max_k \sum_{\langle k,\sigma_i,j\rangle\in\Delta} z^{-q}t_i \frac{f_i(z^{-q}\boldsymbol{t};z)}{1-f_\varepsilon(z^{-q}\boldsymbol{t};z)} < 1$$

where the last inequality is fulfilled according to the assumptions of the lemma. Now using that $||M_{\mathcal{A}}(\boldsymbol{v}(\boldsymbol{t};z))||_\infty < 1$ we obtain the result by applying Lemma 7.3.2. □

**Corollary 7.3.4** *Let $(Op,c)$ be the Levenshtein edit-distance and $q \leq \frac{1}{2}$. If $|\Sigma| \geq 9$, $pr : \Sigma \to (0;1)$ be a probability distribution and $t_i = pr(\sigma_i)$. If the deterministic automaton $\mathcal{A}$ has the property that for each state $k$:*

$$\frac{4}{3}|\Sigma| \sum_{i:!\delta(k,\sigma_i)} t_i + 2|\{i :!\delta(p,\sigma_i)\}| < \sqrt{|\Sigma|},$$

*then there exists $z \in (0,1)$ such that $g_{\mathcal{A}}(v(t;z))$ is finite.*

*Proof.* The idea of the proof is to find such a $z$ that the preconditions of the Lemma 7.3.3 hold. To this end, first note that:

$$\begin{aligned} f_\varepsilon(z^{-q}\boldsymbol{t};z) &= z\sum_{i=1}^{|\Sigma|} z^{-q}t_i = z^{1-q} \text{ and} \\ f_i(z^{-q}\boldsymbol{t};z) &= 1 + zt_i^{-1}z^q \sum_{j\neq i} t_j z^{-q} + zz^q t_i^{-1} \\ &= 1 + zt_i^{-1} - z + z^{1+q}t_i^{-1}. \end{aligned}$$

Therefore we have that:

$$\begin{aligned} v_i(\boldsymbol{t};z) &= z^{-q}t_i \frac{1 + zt_i^{-1} - z + z^{1+q}t_i^{-1}}{1-z^{1-q}} \\ &= z^{-q}t_i \frac{1-z}{1-z^{1-q}} + \frac{z^{1-q}+z}{1-z^{1-q}}. \end{aligned}$$

Now since $q \leq \frac{1}{2}$ and we assume $z \in (0;1)$ we obtain that:

$$\frac{z^{1-q}+z}{1-z^{1-q}} \leq \frac{\sqrt{z}+z}{1-\sqrt{z}} = \sqrt{z}\frac{1+\sqrt{z}}{1-\sqrt{z}}.$$

Now for $\sqrt{z} \leq \frac{1}{3}$ or equivalently for $z \leq \frac{1}{9}$ we get that $\frac{1+\sqrt{z}}{1-\sqrt{z}} \leq \frac{1+\frac{1}{3}}{1-\frac{1}{3}} = 2$ and therefore:

$$\frac{z^{1-q}+z}{1-z^{1-q}} \leq 2\sqrt{z}.$$

On the other hand, for $q \leq \frac{1}{2}$ we have:

$$z^{-q} t_i \frac{1-z}{1-z^{1-q}} \leq z^{-q} t_i \frac{1-z}{1-\sqrt{z}} = z^{-q} t_i (1+\sqrt{z}) \leq \frac{4}{3} z^{-q} t_i$$

whenever $z \leq \frac{1}{3}$. Finally, again since $q \leq \frac{1}{2}$, $z^{-q} \leq z^{-\frac{1}{2}}$, which implies:

$$z^{-q} t_i \frac{1-z}{1-z^{1-q}} \leq \frac{4}{3} \frac{t_i}{\sqrt{z}}.$$

Summing up for $z \leq \frac{1}{9}$ we have:

$$v_i(t; z) \leq 2\sqrt{z} + \frac{4}{3} \frac{t_i}{\sqrt{z}}.$$

Thus setting $z = \frac{1}{|\Sigma|} \leq \frac{1}{9}$ for $|\Sigma| \geq 9$, we obtain:

$$v_i\left(t; \frac{1}{|\Sigma|}\right) \leq 2\frac{1}{\sqrt{|\Sigma|}} + \frac{4}{3} t_i \sqrt{|\Sigma|}$$

and therefore for each state $k \in Q$ we have:

$$\begin{aligned}
\sum_{\langle k, \sigma_i, j \rangle \in \Delta} v_i\left(\boldsymbol{t}; \frac{1}{|\Sigma|}\right) &= \sum_{i : !\delta(k,\sigma_i)} v_i\left(t; \frac{1}{|\Sigma|}\right) \\
&\leq 2\frac{1}{\sqrt{|\Sigma|}} |\{i : !\delta(k, \sigma_i)\}| + \frac{4}{3}\sqrt{|\Sigma|} \sum_{i : !\delta(k,\sigma_i)} t_i \\
&\leq \frac{1}{\sqrt{|\Sigma|}} \left( 2|\{i : !\delta(k, \sigma_i)\}| + \frac{4}{3}|\Sigma| \sum_{i : !\delta(k,\sigma_i)} t_i \right) \\
&< \frac{1}{\sqrt{|\Sigma|}} \sqrt{|\Sigma|} = 1.
\end{aligned}$$

The last inequality follows by the assumptions of the Corollary. Now the result follows by Lemma 7.3.3.

**Corollary 7.3.5** *In the notation of Corollary 7.3.4 if the distribution of the characters is uniform, i.e. $t_i = \frac{1}{|\Sigma|}$ and the for each state $k$ of the automaton it holds:*

$$|\{i \,|\, !\delta(k, \sigma_i)\}| < \frac{3}{10}\sqrt{|\Sigma|},$$

*then there exists $z \in (0; 1)$, such that $g_{\mathcal{A}}(v(t; z))$ converges.*

*Proof.* First, if $|\Sigma| \leq 9$, then the automaton has no transitions and the claim is obvious. Thus, $|\Sigma| > 9$. Next, in the case of uniform distribution we have that:

$$|\Sigma| \sum_{i : !\delta(k,\sigma_i)} t_i = |i : !\delta(k, \sigma_i)|$$

because each $t_i = \frac{1}{|\Sigma|}$. Now the condition:

$$\frac{4}{3}|\Sigma| \sum_{i:!\delta(k,\sigma_i)} t_i + 2|i :!\delta(k,\sigma_i)| < \sqrt{|\Sigma|}$$

is equivalent to:

$$\frac{10}{3}|i :!\delta(k,\sigma_i)| < \sqrt{|\Sigma|}$$

which trivially holds by the assumptions of the claim.

□

The meaning of Corollary 7.3.5 is the following. It gives a local constraint on the structure of the automaton, $\mathcal{A}$, that guarantees that the algorithm from Chapter 5 will perform only linear time work for most of the query words. Indeed, if the portion of those words that require more than linear time was essential, they would contribute to the average running time and make it exceed any linear function. However, according to Corollary 7.3.5 this is not the case. Certainly, this result depends also on the magnitude of the threshold, $q$, which should be $q < \frac{1}{4}$ in order to apply Corollary 7.3.5 in the framework of Proposition 7.2.7.

Furthermore if $g_{\mathcal{A}}(v(t;z))$ converges under the assumptions from Lemma 7.3.2 or Lemma 7.3.3, then the functions $g_{\mathcal{A},n_0}(v(t;z))$ in Corollary 7.2.3 and $g_{\mathcal{A},n_0}(v(t;z))$ tend to zero when $n_0$ tends to infinity. This shows that by an appropriate choice of $n_0$ we can the constant $g_{\mathcal{A},n_0}(v(t;z))$ arbitrary small.

# Chapter 8

# Learning the Edit-Distance

In the previous chapters we developed a divide and conquer algorithm for the approximate search problem in arbitrary regular sets and described a general framework in which we argued its efficiency.

In this chapter we shall address the more challenging problem of reconstruction the original word on the basis of the observed noisy, i.e. query, word. To this end we shall assume that we have a finite set of instances, $\mathcal{I} = \{(U_i, V_i)\}_i$. A pair, $(U_i, V_i)$, in this set tells us that the correct original word for the noisy word $V_i$ is $U_i$. In a sense, the set of instances reflects the nature of noise but does not determine it in an explicit way as the generalised edit-distance does. Along with the set $\mathcal{I}$ we also assume a dictionary, $\mathcal{D}$, for the language of original words. It contains the words $U_i$, but it is in general much larger than $\mathcal{I}$.

Given these data, the problem is to determine the original words, $U$, for arbitrary noisy (query) words, $V$.

In some previous works, [54, 41, 52], this problem is addressed by solving the approximate search problem at a preliminary stage. Using a Levenshtein edit-distance, [54, 41], or generalised edit-distance whose operations are based on expert-knowledge, [52], these algorithms first retrieve a set of candidates. Based on the set of instances, $\mathcal{I}$, one can further develop techniques for ranking these candidates.

Alternative approaches, [53, 14, 59, 48], explore a statistical framework. They fix a set of operations and afterwards "learn" their probabilities (i.e. minus logarithm of costs) by using a Log- Linear Method, [53, 14, 59], or Bayesian Networks and Hidden Markov Models, [48]. In this way they achieve essentially a generalised edit-distance and the problem of retrieval the original word is reduced to the determination of the closest word to the given noisy (query) word.

Our approach differs from the above named methods. It does not use any predefined operation sets and does not constrain their lengths in any way. It explores the structure of the instances and the structure of the dictionary in order to compute the operations and their probabilities that take into account the context in which the noise has corrupted the original word.

The essential part of this Chapter was published in [22]. In the sequel we describe our idea in more details. In Section 8.1 we present the extraction of the operations and the computation of their probabilities. In Section 8.2 we consider an efficient technique for extracting an ordered list of candidates for a given noisy word. In Section 8.3 we finally argue the adequateness of our approach empirically.

## 8.1 Extraction of Operations

Given a finite (multi)set of instances, $\mathcal{I} = \{(U_i, V_i)\}_i$, which implicitly reveals that due noise the original word $U_i$ from a dictionary $\mathcal{D}$, was transformed to a noisy word $V_i$, we denote with $\mathcal{N} = \{V_i\}_i$ the set of noisy words in the set of instances.

The concept of our approach is the following. The operations exhibited in the set of instances, $\mathcal{I}$, transform distinctive infixes in the noisy words into distinctive infixes in the dictionary words, $\mathcal{D}$. This is the background of our approach which also obeys some basic principles. Firstly, most of the characters in the noisy word and its original dictionary word are the same. Otherwise, we would not be able to recognise it. Secondly, the general character order is preserved, i.e. although local transpositions may occur, suffixes and prefixes of words are not interchanged on a regular basis. Finally, the length of the noisy word does not deviate too much from the length of its dictionary original(s).

### 8.1.1 Canonical and Candidate Trees of a Word

An infix which is distinctive for a set of words, $\mathcal{W}$, would be either explicitly pointed out in this set as a word, or it would be implicitly encoded in this set. In the latter case there should be natural markers that indicate its significance. We consider as such markers the different contexts in which infixes occur. This is the motivation to formally define a *distinctive infix* in a set of words, $\mathcal{W}$, as either (i) a prefix in $\mathcal{W}$, or (ii) an infix in $\mathcal{W}$ which occurs in at least two:

**Definition 8.1.1** Given a finite set of words, $\mathcal{W}$, a distinctive infix (distinx), is an infix $V \in Inf(\mathcal{W})$ that has one of the following two properties:

1. $V \in Pref(\mathcal{W})$,
2. or there exist distinct characters $a, b \in \Sigma$ such that $aV, bV \in Inf(\mathcal{W})$.

The set of all distinxes in $\mathcal{W}$ is denoted with $\mathcal{S}_{\mathcal{W}}$.

From Lemma 2.2.1 ditinxes are just another name for the representatives in the set $\mathcal{W}$ and $\mathcal{S}_{\mathcal{W}}$ is just the structure of Blumer et al,[12].

Let us now assume that we have an instance, say $(knows, knoweth) \in \mathcal{I}$, saying that *knows* is the original word for the noisy word *knoweth*. The idea is to propagate this knowledge to shorter distinxes that compose both words. This will allow us to deduce properties of shorter distinxes that occur more often

and thus are responsible for the properties of more words. In the same time we would like to preserve the information for the longer distinxes, that determine the words in a more unambiguous way.

To achieve this goal, we consider a hierarchical decomposition of the words $N \in \mathcal{N}$, see Figure 8.1. Intuitively, we intend to split $N = knoweth$ into a prefix and a suffix, say $k$ and $noweth$, respectively. Still, they cannot be arbitrary, but $\mathcal{N}$-distinxes. This is why the suffix $noweth$ might be excluded in our case. As a splitting criterion we choose the maximal proper suffix of $knoweth$, which is an $\mathcal{N}$-distinx. In our case this is $oweth$. Formally, we define:

**Definition 8.1.2** Given a finite set of words, $\mathcal{W}$, and a distinx $V \in \mathcal{S}_{\mathcal{W}}$, the longest proper suffix of $V$, $lps_{\mathcal{W}}(V) = lps(V)$ is the longest infix $V_1 \neq V$ s.t.:

$$V_1 \in Suf(V) \text{ and } V_1 \in \mathcal{S}_{\mathcal{W}}.$$

It might be the case that $lps(V)$ is not defined for some words, e.g. if $V = \varepsilon$.

Now, we have that $lps(knoweth) = oweth$. Therefore, the complementary proper prefix of $lps(knoweth)$ to the word $knoweth$ is $kn$. This is the motivation for the next definition:

**Definition 8.1.3** Given a finite set of words, $\mathcal{W}$, and a distinx $V \in \mathcal{S}_{\mathcal{W}}$ the complementary proper prefix of $V$, $cpp_{\mathcal{W}}(V) = cpp(V)$, is defined as follows:

1. if $lps(V)$ is defined and $lps(V) \neq \varepsilon$, then $cpp(V) = V_0$ s.t. $V = V_0 \circ lps(V)$.
2. if $lps(V) = \varepsilon$, then $cpp(V)$ is the prefix of $V$ of length $|V| - 1$.
3. if $lps(V)$ is not defined, then $cpp(V)$ is not defined either.

As an immediate corollary from the definition of the distinxes we have:

**Lemma 8.1.4** *If $\mathcal{W}$ is a finite set of words and $V$ is a distinx in $\mathcal{W}$, such that $cpp(V)$ is defined, then $cpp(V)$ is also a distinx.*

*Proof.* Indeed, if $V$ is a prefix in $\mathcal{W}$, then $cpp(V)$ is also a prefix of $\mathcal{W}$, since it is a prefix of $V$. In the alternative case, we have that there exist two distinct characters $a, b \in \Sigma$ such that $aV, bV \in Inf(\mathcal{W})$. Therefore $a$ and $b$ also precede the prefix $cpp(V)$ of $V$ and therefore $cpp(V) \in \mathcal{S}_{\mathcal{W}}$. □

Now, we repeat the decomposition of longest proper suffix and a complementary proper prefix recursively and thus, we get the notion of the canonical tree:

**Definition 8.1.5** Let $\mathcal{W}$ be a finite set of words and $V \in \mathcal{S}_{\mathcal{W}}$ be a distinx in $\mathcal{W}$. The canonical tree $\mathcal{T}_{\mathcal{W}}(V)$ is defined recursively as follows:

1. if $lps(V)$ is not defined, then $\mathcal{T}_{\mathcal{W}}(V)$ is a trivial tree with root $V$.
2. if $lps(V)$ is defined, then $\mathcal{T}_{\mathcal{W}}(V)$ is a tree with root $V$, left subtree:

$$\mathcal{T}^{(l)}_{\mathcal{W}}(V) = \mathcal{T}_{\mathcal{W}}(cpp(V))$$

and right subtree:

$$\mathcal{T}^{(r)}_{\mathcal{W}}(V) = \mathcal{T}_{\mathcal{W}}(lps(V))$$

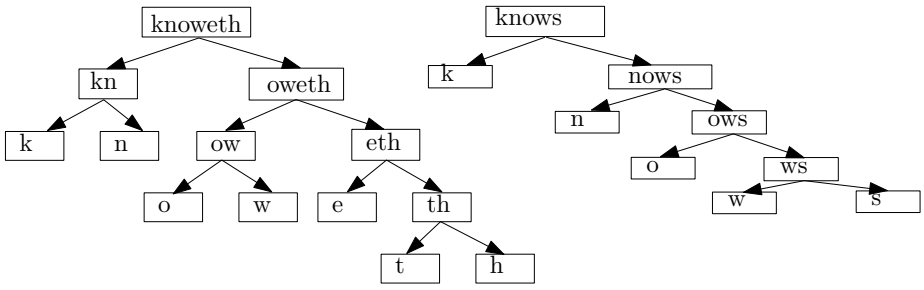



Figure 8.1: On the left is the canonical tree, $\mathcal{T}_{\mathcal{N}}(V)$, for the noisy word $V =$ *knoweth* w.r.t. the set of noisy words, $\mathcal{N}$. On the right is the canonical tree $\mathcal{T}_{\mathcal{D}}(U)$ of its dictionary original word $U =$ *knows*.

The result of applying Definition 8.1.5 to our example $V =$ *knoweth* is illustrated on Figure 8.1. Next Lemma shows that the size of the canonical trees is proportional to the length of the word they represent:

**Lemma 8.1.6** *For every distinx $V \in \mathcal{S}_{\mathcal{W}}$ with $V \neq \varepsilon$ the number of nodes in the canonical tree $\mathcal{T}_{\mathcal{W}}(V)$ is at most $4|V| - 1$.*

*Proof.* The proof proceeds by induction on the length of $V$. For $|V| = 1$, $\mathcal{T}_{\mathcal{W}}(V)$ has a single root and at most two other nodes, thus summing up to at most $3 = 4.1 - 1$. This proves the basis of the induction. For the induction step we have to consider two cases. The general case is when $lps(V) \neq \varepsilon$. Then we have that $cpp(V) \circ lps(V) = V$ and therefore $|cpp(V)| + |lps(V)| = |V|$. It should be also clear that since $lps(V)$ is a proper suffix of $V$, $cpp(V) \neq \varepsilon$. Therefore we can apply the induction hypothesis for both $lps(V)$ and $cpp(V)$. Thus, we conclude that:

$$\begin{aligned} |\mathcal{T}_{\mathcal{W}}(V)| &= |\mathcal{T}_{\mathcal{W}}(lps(V))| + |\mathcal{T}_{\mathcal{W}}(cpp(V))| + 1 \\ &\leq 4|lps(V)| - 1 + 4|cpp(V)| - 1 + 1 = 4|V| - 1. \end{aligned}$$

Therefore, the claim follows in this case. It remains to consider the case when $lps(V) = \varepsilon$. In this case $|cpp(V)| = |V| - 1$. Furthermore, since $|V| \neq 1$, we have that $cpp(V) \neq \varepsilon$. Hence, we can apply the induction hypothesis to $cpp(V)$ and as a result we obtain:

$$\begin{aligned} |\mathcal{T}_{\mathcal{W}}(V)| &= |\mathcal{T}_{\mathcal{W}}(cpp(V))| + 2 \\ &\leq 4(|V| - 1) + 2 = 4|V| - 2 < 4|V| - 1 \end{aligned}$$

which proves the claim in this case, either.

□

Next, recall that the dictionary $\mathcal{D}$ represents the dictionary words. Thus, we can apply the canonical tree procedure for $U =$ *knows* w.r.t. $\mathcal{D}$. It is tentative to match both canonical trees, the one for *knoweth* w.r.t. $\mathcal{N}$ and the one for

*knows* w.r.t. $\mathcal{D}$, see Figure 8.1. The problem is that $\mathcal{D}$ is in general much larger than the number of different noisy words in the set $\mathcal{N}$. Thus, we would observe some $\mathcal{D}$-distinxes in $\mathcal{D}$, e.g. *nows*, whose noisy variants have not occurred as $\mathcal{N}$-distinxes. To handle this situation we give more freedom for an hierarchical representation of the dictionary words, see Figure 8.2. Namely, we consider not only the split of *knows* as a prefix, *k*, and a suffix, *nows*, but also the splits, *kn* and *ows*, and *know* and *s*, etc. provided that both the prefix and the suffix are $\mathcal{D}$-distinxes. Formally, we define:

**Definition 8.1.7** Given a finite set of words, $\mathcal{W}$, and a distinx $V \in \mathcal{S}_{\mathcal{W}}$, the proper suffix of level $k$ of $V$, $ps^{(k)}(V)$, is defined as:

$$ps^{(k)}(V) = \begin{cases} lps(V) \text{ if } k = 1 \\ ps^{(k-1)}(lps(V)), \text{ else} \end{cases}$$

Analogously, to the complementary proper prefixes we can define the proper prefixes of level $k$. The relation between the proper prefixes of level $k$ and the proper suffixes of level $k$ is the same as the relation between the complementary proper prefixes and longest proper suffixes:

**Definition 8.1.8** Given a finite set of words, $\mathcal{W}$, and a distinx $V \in \mathcal{S}_{\mathcal{W}}$, the proper prefix of level $k$ of $V$, $pp^{(k)}(V)$ is defined as:

1. if $ps^{(k)}(V)$ is defined and $ps^{(k)}(V) \neq \varepsilon$, then $pp^{(k)}(V) = V_0$ s.t. $V = V_0 \circ ps^{(k)}(V)$.
2. if $ps^{(k)}(V) = \varepsilon$, then $pp^{(k)}(V)$ is the prefix of $V$ of length $|V| - 1$.
3. if $ps^{(k)}(V)$ is not defined, then $pp^{(k)}(V)$ is not defined, either.

The notion of proper suffixes of level $k$ and proper prefixes of level $k$ allow us to make a *guess* how the word, say *knows*, should be decompose, for instance in *kn* and *ows*. Applying a sequence of guesses recursively, we arrive at the definition of a candidate tree, see Figure 8.2:

**Definition 8.1.9** Given a finite set of words, $\mathcal{W}$, and a distinx $V \in \mathcal{S}_{\mathcal{W}}$, a candidate tree for $V$, is defined recursively:

1. the trivial tree with root $V$ is a candidate tree for $V$.
2. if $k \geq 1$ is such that $pp^{(k)}(V)$ and $ps^{(k)}(V)$ are defined and $T_1$ is a candidate tree for $pp^{(k)}(V)$ and $T_2$ is a candidate tree for $ps^{(k)}(V)$, then the tree with root $V$, left subtree $T_1$ and right subtree $T_2$ is a candidate tree for $V$.

It should be clear, that unlike the canonical tree, that is uniquely determined by the finite set of words and a distinx from this set, many candidate trees might be assigned to a single distinx. Actually, this number may be exponential in the length of the particular distinx. However, the number of different distinxes that can occur in some of the candidate trees for a particular distinx is polynomially bounded by the length of the distinx. This is what the next lemma claims:

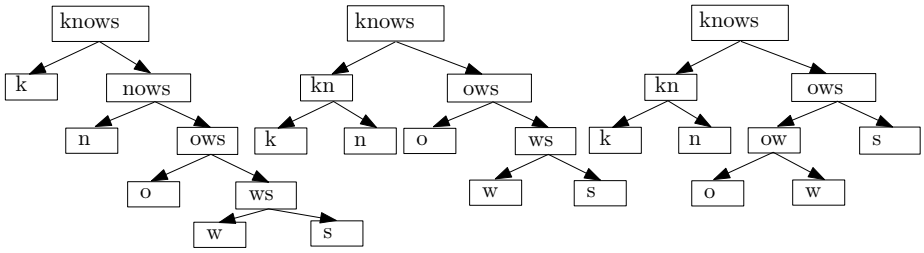



Figure 8.2: Different candidate trees for $R = knows$. One of them is $\mathcal{T}_{\mathcal{D}}(U)$.

**Lemma 8.1.10** *Given a set of words, $\mathcal{W}$, and a distinx $V \in \mathcal{S}_{\mathcal{W}}$, there are at most $\binom{|V|}{2}+1$ different distinxes that can occur in some candidate tree, $\mathcal{CT}_{\mathcal{W}}(V)$.*

*Proof.* Each node of a candidate tree $\mathcal{CT}_{\mathcal{W}}(V)$ is assigned with an infix of $V$. The result now follows since there are at most $\binom{|V|}{2}$ infixes of $V$ that are not the empty word and $\varepsilon$ is also an infix of $V$.

□

### 8.1.2 Retrieval of Operations and their Probabilities

Now, we expect that one or more of the candidate trees for *knows* should match the structure of the canonical tree for *knoweth*, see Figure 8.3. Knowing which this candidate tree is and how it matches the canonical tree for *knoweth* we would be able to propagate the knowledge that *knoweth* is a noisy variant of *knows* to shorter distinxes of both structures.

To achieve this objective, we are going to introduce an edit-distance between trees. The main idea is to reflect that typically we have identities that should be stimulated whereas the length discrepancies between distinxes that are supposed to form an operation, are less likely and thus, have to be penalised on general. There are many ways to model these two properties. Next definition presents one specific formal way to define similarity between candidate and canonical trees. It is used only to determine the 'real' operations and to assign them with appropriate probabilities. It is not used in the searching that we describe in the next section.

**Definition 8.1.11** Let $\mathcal{T}_{\mathcal{N}}(V)$ be a canonical tree for the distinx $V$ in the set of noisy words, $\mathcal{N}$. Let $\mathcal{CT}_{\mathcal{D}}(U)$ be a candidate tree for the distinx $U$ in the dictionary, $\mathcal{D}$, then the edit-distance between these two trees is defined as:

$$d_T(\mathcal{CT}_{\mathcal{D}}(U), \mathcal{T}_{\mathcal{N}}(V)) = \min \begin{cases} \max(|U|, |V|) - id \\ d_T(\mathcal{CT}^{(l)}_{\mathcal{D}}(U), \mathcal{T}^{(l)}_{\mathcal{N}}(V)) + d_T(\mathcal{CT}^{(r)}_{\mathcal{D}}(U), \mathcal{T}^{(r)}_{\mathcal{N}}(V)) \end{cases}$$

where $id = 1$ if $V$ and $U$ have the same last character and $id = 0$, otherwise. The superscripts $(l)$ and $(r)$ denote left and right subtree, respectively.

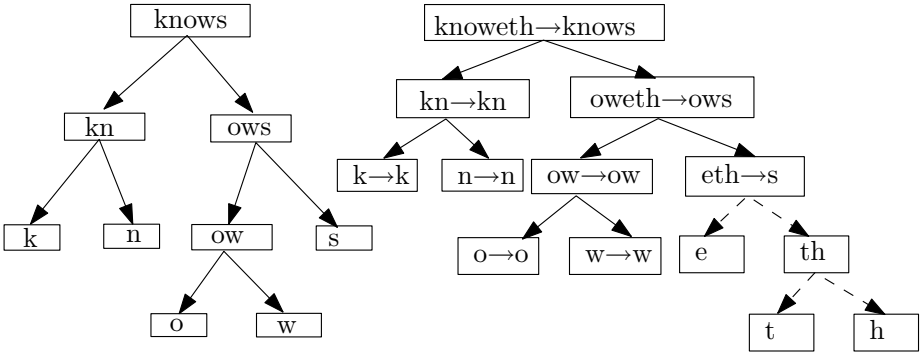



Figure 8.3: On the left, the candidate tree of $U = knows$ which is best ranked w.r.t. $V = knoweth$. On the right, the noisy variant $knoweth \rightarrow knows$ is propagated to the subtrees to obtain new (shorter) operations.

Actually, we are not interested in the tree mapping. We are rather interested of the mapping between distinxes from one structure into the other. The similarity notion between trees is only a strong motivation how to achieve this. Thus, we give the following definition:

**Definition 8.1.12** Let $U$ be a distinx in a finite set $\mathcal{D}$ and $V$ be a distinx in a finite set $\mathcal{N}$. Then we define:

$$d_W(U,V) = \min_{CT\text{: a candidate tree for } U} d_T(CT, \mathcal{T}_{\mathcal{N}}(V)).$$

**Lemma 8.1.13** *Given two distinxes $U \in \mathcal{S}_{\mathcal{D}}$ and $V \in \mathcal{S}_{\mathcal{N}}$ the values $d_W(U',V')$ where $U' \in \mathcal{S}_{\mathcal{D}}$ is an infix of $U$ and $V' \in \mathcal{T}_{\mathcal{N}}(V)$ can be determined in $O(|U|^3|V|)$ total time.*

*Proof.* From the definition of $d_W$ and $d_T$ it is easy to see that:

1. if $lps(V)$ is not defined, then $d_W(U,V) = \max(|U|,|V|) - id$,
2. if $lps(V)$ is defined, then:

$$d_W(U,V) = \min \begin{cases} \max(|U|,|V|) - id \\ \min_k(d_W(pp^{(k)}(U), cpp(V)) + d_W(ps^{(k)}(U), lps(V)). \end{cases}$$

Thus, the computation of the values $d_W(U',V')$ required by the lemma can be computed by a standard dynamic programming scheme using the above recurrence, see procedure TreeDistance. In particular, we consider the problem of finding $d_W(U',V')$ as a problem of filling in the entries of a table with rows assigned to the distinxes the $U'$ and columns assigned to the distinxes $V'$. Hence, our table has $O(|U|^2)$ rows and $O(|V|)$ columns. Each entry $(U',V')$ of this table

is associated with the value $d_W(U', V')$ which is computed once only. To compute $d_W(U', V')$ we use the above recurrence. If case 1 applies, the computation clearly needs $O(1)$ time. In the alternative case we consider all possible $k$ such that $ps^{(k)}(U')$ is defined. Since for distinct values of $k$, $ps^{(k)}(U')$ are different suffixes of $U'$, there are at most $|U'| \leq |U|$ possible values for the parameter $k$. If some of the values $d_W(pp^{(k)}(U'), pp(V'))$ or $d_W(ps^{(k)}(U'), lps(V'))$ are not computed yet, the computational efforts to retrieve these values are assigned to the corresponding entry of the table, $d_W(pp^{(k)}(U'), pp(V'))$ or $d_W(ps^{(k)}(U'), lps(V'))$, respectively and not to the computation of $d_W(U', V')$. Thus, we assign at most $|U|$ units of time for the computation of a particular value $d_W(U', V')$. Since there are $O(|U|^2|V|)$ entries in the table we arrive at the claimed upper bound of $O(|U|^3|V|)$ total time.. □

```
TreeDistance(CandNodes, CanNodes, i, j, Matrix)
   if Matrix[i][j] is defined then
      return Matrix[i][j]
   else
      if last_character(CandNodes[i]) = last_character(CanNodes[j]) then
         id ← 1
      else
         id ← 0
      fi
      Matrix[i][j] ← max(|CandNodes[i]|, |CanNodes[j]|) − id
      if lps(CanNodes[j] is not defined
         return Matrix[i][j]
      right ← index_of(CanNodes[j], CanNodes)
      left ← index_of(CanNodes[j], CanNodes)
      k ← i
      while k is defined do
         right1 ← index_of(CandNodes[k], CandNodes))
         left1 ← index_of(CandNodes[k]), CandNodes)
         if right1 is defined then
            TreeDistance(CandNodes, CanNodes, left1, left, Matrix)
            TreeDistance(CandNodes, CanNodes, right1, right, Matrix)
            if Matrix[left1][left]+Matrix[right1][right] < Matrix[i][j] then
               Matrix[i][j] ← Matrix[left1][left] + Matrix[right1][right]
            fi
         fi
         k ← right1
      done
   fi
```

Once we have the values $d_W(U', V')$ that measure how far away the distinxes $U'$ and $V'$ are, we propagate the knowledge that $V$ is a noisy variant of $U$ to all those pairs $(U', V')$ that contribute to the value $d_W(U, V)$. We will define these pairs $(U', V')$ as well as the pair $(U, V)$ as operations. In order to properly

define the pairs $(U', V')$ that contribute to $d_W(U, V)$ and also the magnitude of contribution we: (i) determine those pairs of distinxes that witness for the optimal edit distance, $d_W(U, V)$, between $U$ and $V$ and (ii) stimulate those of these pairs that occur in different contexts.

Whereas, the first property is more or less intuitive and is widely used, the second objective is a particular one for our case. The point is that the structure $\mathcal{S}_{\mathcal{W}}$ induces an hierarchical decomposition of words. Thus, we do not need to account for pairs of shorter distinxes that always entail the same pair of longer distinxes. However, if a shorter pair of distinxes is often characteristic for different longer extension pairs, then it is probable that also new, unobserved pairs of distinxes rely on such a decomposition. For this reason we boost the importance of such pairs of distinxes.

Following, this concept we first define the pairs $(U', V')$ that have property (i):

**Definition 8.1.14** Let $\mathcal{D}$ and $\mathcal{N}$ be finite sets, $U \in \mathcal{D}$ and $V \in \mathcal{N}$, then:

1. $(U, V)$ contributes to the value $d_W(U, V)$.
2. if $(U', V')$ contributes to $d_W(U, V)$ and $k$ is an integer such that:

$$d_W(U', V') = d_W(pp^{(k)}(U'), cpp(V)) + d_W(ps^{(k)}(U'), lps(V)),$$

then $(pp^{(k)}(U'), cpp(V))$ and $(ps^{(k)}(U'), lps(V))$ also contribute to the value $d_W(U, V)$.

In order to model the second property, i.e. pairs that contribute to $d_W(U, V)$ in different contexts, we use a naive counting principle:

**Definition 8.1.15** Let $\mathcal{D}$ and $\mathcal{N}$ be finite sets of words $U \in \mathcal{D}$ and $V \in \mathcal{N}$. For distinxes $U' \in \mathcal{S}_{\mathcal{D}}$ and $V' \in \mathcal{S}_{\mathcal{N}}$ we define $a_{U,V}(U', V')$ as:

$$\begin{aligned} a_{U,V}(U', V') \quad = \quad & |\{(U_l, V_l) \,|\, (U_l \circ U', V_l \circ V') \text{ contributes to } d_W(U, V)\}| + \\ & |\{(U_r, V_r) \,|\, (U' \circ U_r, V' \circ V_r) \text{ contributes to } d_W(U, V)\}| \end{aligned}$$

when $(U, V) \neq (U', V')$ and we set $a_{U,V}(U, V) = 1$.

The algorithmic structure of the above definitions allows us to efficiently compute the values $a_{U,V}(U', V')$, as the next Lemma states:

**Lemma 8.1.16** *Given an instance pair $(U, V) \in \mathcal{I}$ the (nonzero) values $a_{U,V}(U', V')$ can be computed in time $O(|U|^3|V|)$.*

*Proof.* We first compute the values $d_W(U', V')$ over all $U'$ that are distinxes in $\mathcal{D}$ and are infixes of $U$ and all the distinxes $V'$ in the canonical tree $\mathcal{T}_{\mathcal{N}}(V)$. Then, essentially the same dynamic programming scheme as in the computation of the values $d_W(U', V')$, see procedure Propagate, can be used to determine those pairs $(U', V')$ that contribute to $d_W(U, V)$. Finally, the values $a_{U,V}(U', V')$ are increased only when we arrive in $(U', V')$ either from a new pair, or from the

same pair but this $(U', V')$ play the role of suffixes and not of prefixes. Since the dynamic programming scheme is the same as in Lemma 8.1.13 we obtain the same running time, $O(|U|^3|V|)$.

□

```
Propagate(CandNodes, CanNodes, i, j, Matrix, a)
  U ← CandNodes[i]
  V ← CanNodes[j]
  if a[U, V] is not defined then
    a[U, V] ← 0
    right ← index_of(CanNodes[j], CanNodes)
    left ← index_of(CanNodes[j], CanNodes)
    if right is defined then
      k ← i
      while k is defined do
        right1 ← index_of(CandNodes[k], CandNodes)
        left1 ← index_of(CandNodes[k], CandNodes)
        if right1 is defined and
        Matrix[left1][left] + Matrix[right1][right] = Matrix[i][j] then
          Propagate(CandNodes, CanNodes, left1, left, Matrix, a)
          Propagate(CandNodes, CanNodes, right1, right, Matrix, a)
        fi
        k ← right1
      done
    fi
  fi
  a[U, V] ← a[U, V] + 1
```

Now, the entire training process can be regarded as an iteration over all the pairs of instance, $(U_i, V_i) \in \mathcal{I}$. For each such pair, $(U_i, V_i)$, we first determine the edit-distance $d_W(U_i, V_i)$ and then the scores, $a_{U_i,V_i}(U', V')$. For a fixed pair $(U', V')$ of distinxes we accumulate all the scores $a_{U_i,V_i}(U', V')$ in a single amount $a(U', V')$ which reflects the global number of contributions of the pair $(U', V')$. Formally, we define:

$$a(U', V') = \sum_{(U,V)\in\mathcal{I}} a_{U,V}(U', V').$$

**Lemma 8.1.17** *Given a (multi)set of instances, $\mathcal{I} = \{(U_i, V_i)\}_i$, and a dictionary, $\mathcal{D}$, let $\mathcal{N} = \{V \mid \exists U((U, V) \in \mathcal{I})\}$. Then, the values $a(U', V')$ can be computed in total time:*

$$O(\sum_{(U_i,V_i)\in\mathcal{I}} |U_i|^3|V_i|).$$

*Proof.* Follows from Lemma 8.1.16 and the definition of the values $a(U', V')$, see procedure Training.

□

```
Training(I = {(U_i, V_i)}_i, S_D, S_N, a)
   for (U, V) ∈ I do
      CandNodes ← array of Inf(U) ∩ S_D
      CanNodes ← array of the distinxes in T_N(V)
      Matrix ← matrix of dimension |CandNodes| × |CanNodes|
      i ← index_of(U, CandNodes)
      j ← index_of(V, CanNodes)
      TreeDistance(CandNodes, CanNodes, i, j, Matrix)
      b ←matrix of dimension |CandNodes| × |CanNodes|
      Propagate(CandNodes, CanNodes, i, j, Matrix, b)
      for U' ∈ CandNodes and V' ∈ CanNodes s.t. b[U', V'] ≠ 0 do
         if a[U', V'] is not defined then
            a[U', V'] ← 0
         fi
         a[U', V'] ← a[U', V'] + b[U', V']
      done
   done
```

Finally, we determine the set of operations, $Op$, as those pairs of distinxes $(U', V')$ that have contributed to at least one optimal edit-distance $d_W(U, V)$. Specifically, we set:

$$Op = \{(U', V') \mid a(U', V') > 0\}.$$

To each operation $(U', V')$ we attribute an empirical conditional probability $p(U'|V')$ that accounts how likely it is according to the scores $a(U'', V')$ that $V'$ is modified to $U'$. Formally we set:

$$p(U'|V') = \frac{a(U', V')}{\sum_{(U'', V') \in Op} a(U'', V')}$$

## 8.2 Searching Dictionary Candidates

Given a noisy word, $V$, we want to reconstruct the most likely original words, $U$, that correspond to $V$. To this end we proceed in two stages. In the first stage we recognise the $\mathcal{N}$-distinxes exhibited by $V$. Thus, we essentially determine the natural $\mathcal{N}$-structure of the noisy word $V$. In the second stage, we organise an exhaustive search which generates and ranks dictionary candidates. Using a variant of an $A$-star algorithm, [24, 25], similar to Mohri, [45], and Eppstein, [18], we generate the candidates in order.

In order to retrieve correction candidates $U \in \mathcal{D}$, for a query word, $V$, we rely on the operations, $Op$, their conditional probabilities, $p$, and the structures $\mathcal{S}_\mathcal{D}$ and $\mathcal{S}_\mathcal{N}$. It is important to stress that $\mathcal{N}$ is the set of the noisy words observed during the training, i.e. in the set of instances $\mathcal{I}$.

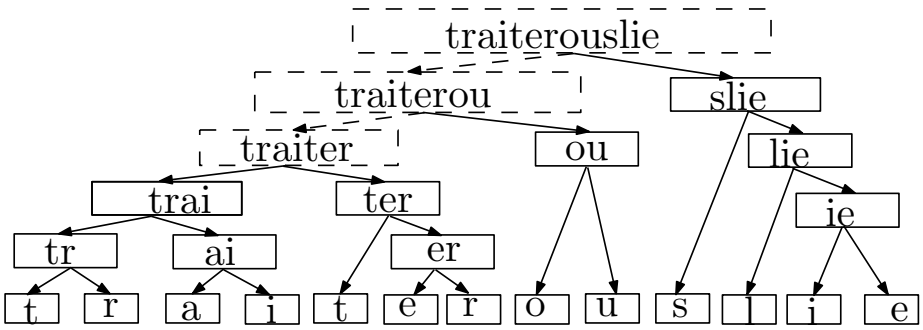



Figure 8.4: The tree $\widetilde{\mathcal{T}}_{\mathcal{N}}(N)$ constructed for the query word $N = traiterouslie$. The dashed nodes are infixes which are not $\mathcal{N}$-distinxes, the solid nodes are infixes which are $\mathcal{N}$-distinxes.

### 8.2.1 Approximate Canonical Trees

Ideally, if the query word $V$ was observed during the learning stage, we would have no problems to retrieve its canonical tree, $\mathcal{T}_{\mathcal{N}}(V)$. Unfortunately, this is not the case when $V \notin \mathcal{N}$, say $V = traiterouslie$ might be such a word. Still, $V$ would share some common $\mathcal{N}$-distinxes with the noisy words in $\mathcal{N}$. The asymmetric structure of the $\mathcal{N}$-distinxes which appears to give more credit to suffixes than to prefixes suggests a natural approach how to define an approximation, $\widetilde{T}_{\mathcal{N}}(V)$, of the canonical tree, see Figure 8.4:

**Definition 8.2.1** Let $V$ be a word and $\mathcal{W}$ be a finite set of words. An approximate canonical tree $\widetilde{\mathcal{T}}_{\mathcal{W}}(V)$ is defined recursively as follows:

1. if $V \in \mathcal{S}_{\mathcal{W}}$, then $\widetilde{\mathcal{T}}_{\mathcal{W}}(V) = \mathcal{T}_{\mathcal{W}}(V)$.

2. if $V \in \mathcal{S}_{\mathcal{W}}$, then $\widetilde{\mathcal{T}}_{\mathcal{W}}(V)$ is a tree with root $V$, right subtree $\mathcal{T}_{\mathcal{W}}(V_2)$ and left subtree $\widetilde{\mathcal{T}}_{\mathcal{W}}(V_1)$ where:

$$V_2 = lps_{\mathcal{W}}(V) \text{ and } V = V_1 \circ V_2.$$

We illustrate Definition 8.2.1 on a the word, $V = traiterouslie$, see Figure 8.4. Clearly, $V \neq \varepsilon$ and it turns out that the longest proper suffix of $V$ that is a distinx in $\mathcal{N}$ is $V_2 = slie$. This uniquely determines the prefix $V_1 = traiterou$. Now, since $V_2$ is a distinx in $\mathcal{N}$, it determines a canonical tree $\mathcal{T}_{\mathcal{N}}(slie)$. This is the right subtree of $\mathcal{T}_{\mathcal{N}}(traiterouslie)$. To compute the left subtree of the desired approximate canonical tree, Definition 8.2.1 refers recursively to the approximate canonical tree of $V_1 = traiterou$. Again, we determine the longest $\mathcal{N}$-distinx, $V_{12}$ that is a suffix of $V_1$. It turns out that $V_{12} = ou$. Hence, the prefix $V_{11}$ is $traiter$. Now, we can retrieve the canonical tree $\mathcal{T}_{\mathcal{N}}(ou)$ from the structure $\mathcal{N}$. This will be the right subtree of $\widetilde{\mathcal{T}}_{\mathcal{N}}(traiterou)$. To determine the left subtree of $\widetilde{\mathcal{T}}_{\mathcal{N}}(traiterou)$ we proceed with $V_{11} = traiter$ in the same fashion.

We note that in the special case when the query word $V$ belongs to the training set $\mathcal{N}$, we have $\widetilde{\mathcal{T}}_{\mathcal{N}}(V) = \mathcal{T}_{\mathcal{N}}(V)$.

In order to determine the approximate canonical tree, $\widetilde{\mathcal{T}}_{\mathcal{N}}(V)$, it is clear that the longest suffixes of any prefix, $I_1^i(V)$, of $V$ that occur in $Inf(\mathcal{N})$ will play an important role. We denote these suffixes with $s_i(V)$, for $i = 1, \ldots, |V|$, respectively. Thus, in a first step, given the query word $V$ and the structure $\mathcal{S}_{\mathcal{N}}$ we are going to find the representations of the suffixes $s_i(V)$. Given the $s_i(V)$ it will be then easy to apply the Definition 8.2.1 in order to compute $\widetilde{\mathcal{T}}_{\mathcal{N}}(V)$. Next Lemma takes after a previous result of Aho and Corasick, [7], and shows that representations of all the suffixes $ls_i(V)$ can be computed in linear time.

**Lemma 8.2.2** *Given a query word $V$ of length $N$, the representations of the longest suffixes $s_i(V)$ of $I_1^i(V)$ that is also an infix in $\mathcal{N}$ can be computed in $O(|V|)$ total time.*

*Proof.* For the proof we use an adapted version of the Aho-Corasick, [7], algorithm, see procedure ComputeLongestSuffixes. We assume that $\mathcal{N}$ is nonempty. Let, $V = v_1 \circ v_2 \cdots \circ v_n$. Then, $s_0(V) = \varepsilon$. Now, the step from $s_i(V)$ to $s_{i+1}(V)$ is conducted as follows. Knowing the representation of $s_i(V)$ in the structure $\mathcal{N}$, we check in time $O(1)$ whether $s_i(V)$ can be extended with the $v_{i+1}$. If this is the case, we also compute the representation of $s_{i+1}(V) = s_i(V) \circ v_{i+1}$, see Chapter 2. If this is not the case, we follow the longest proper suffix link of $s_i(V)$, $s_{i+1}^{(1)} = lps(s_i(V))$ and check whether $s_{i+1}^{(1)}$ can be extended with $v_{i+1}$. If it fails, we proceed with $s_{i+1}^{(2)} = lps(s_{i+1}^{(1)})$ and so on. Two cases may occur. We reach a step $k$ where $s_{i+1}^{(k)}$ can be extended with $v_{i+1}$. Alternatively, we end up with the situation where $lps(s_{i+1}^{(k)})$ is not defined for a certain $k$. In the latter case, we deduce that the character $v_{i+1}$ does not occur in the set $\mathcal{N}$ and thus $s_{i+1}(V)$ is not defined. In the former case, similarly to the algorithm of Aho and Corasick, $s_{i+1}(V) = s_{i+1}^{(k)} \circ v_{i+1}$. Hence, we can compute the representation of $s_{i+1}(V)$ in time $O(k+1)$ where $k$ is the number of unsuccessful attempts to extend $s_{i+1}^{(j)}$ with $v_{i+1}$. Since, at each such step $s_{i+1}^{(j+1)} = lps(s_{i+1}^{(j)})$ diminishes with at least one, we deduce that the number of such steps, $j$, is $k \leq |s_i(V)| - |s_{i+1}^{(k)}| = |s_i(V)| - |s_{i+1}(V)| + 1$.

A subtle point is the situation where $s_{i+1}(V)$ is not defined. In this case, we assume that $|s_{i+1}(V)| = -1$ and at the next step we have to restart our search not from $s_{i+1}$ as it is undefined, but from $\varepsilon$.

After this remark we can easily compute that the number of atomic steps performed by the algorithm is bounded by a factor of:

$$\sum_{i=0}^{n-1}(|s_i(V)| - |s_{i+1}(V)| + 1) = |s_0(V)| - |s_n(V)| + n = n - |s_n(V)| \leq n.$$

Therefore, the time spent for the computation of the representation of the infixes $s_i(V)$ is carried out in time $O(|V|)$.

□

```
ComputeLongestSuffixes(V, S_N, s)
//δ is the transition function of S_N
//V = v_1 ∘ v_2 ··· ∘ v_n
   n ← |V|
   s_ε ← the initial state of S_N
   s[0] ← (s_ε, 0)
   for i = 1 to n do
      (st, j) ← s[i − 1]
      if st is not defined
         (st, j) ← (s_ε, 0)
      fi
      while st is defined and δ(st, v_i) is not defined do
         st ← lps(st)
         j ← len(st)
      done
      if st is defined
         s[i] ← (δ(st, v_i), j + 1)
      else
         s[i] ← ⊥
      fi
   done
```

Once we dispose on the infixes $s_i(V)$ it is almost straightforward to compute the approximate canonical tree for $V$.

**Lemma 8.2.3** *Given a query word $V$ and the representations of $s_i(V)$ with respect to the structure $\mathcal{S}_\mathcal{W}$ for each $i = 1, 2, \ldots, |V|$, the approximate canonical tree $\widetilde{\mathcal{T}}_\mathcal{W}(V)$ can be computed in time $O(|V|)$.*

*Proof.* From the definition of the approximate canonical tree follows that it is comprised of right subtrees that are canonical trees and left subtrees that are approximate canonical trees.

First, storing $lps(U)$ and $cpp(U)$ for each distinx $U \in \mathcal{S}_\mathcal{W}$, we can compute the every canonical tree $\mathcal{T}_\mathcal{W}(U')$ for a distinx $U'$ in $\mathcal{W}$ in time proportional to the time of the tree itself. Thus, according to Lemma 8.1.6, the canonical tree $\mathcal{T}_\mathcal{W}(U')$ can be computed in time $O(|U'|)$.

This remark resolves the case when $V \in \mathcal{S}_\mathcal{W}$. Next, let us assume that $V \notin \mathcal{S}_\mathcal{W}$ and the length of $V$ is $n$, see procedure ApproximateCanonicalTree. Therefore $|s_n(V)| < |V|$. Now, we consider the representation $s_n(V)$ that is in the form $(V', j)$ where $V' \in \mathcal{S}_\mathcal{W}$ and $j = |s_n(V)|$. Two cases may arise:

1. $j = |V'|$ and thus $s_n(V) = V'$,
2. or $j < |V'|$, and therefore $s_n(V)$ is a proper suffix of $V'$.

In the first case, it is clear that $V'$ is the longest suffix of $V$ that is a distinx in $\mathcal{S}_\mathcal{W}$. Therefore $lps_\mathcal{W}(V) = V'$. In the second case, it is also easy to see that the

longest suffix of $V$ that is a distinx is the longest proper suffix of $V'$ that is a distinx. Therefore $lps_{\mathcal{W}}(V) = lps_{\mathcal{W}}(V')$.

Hence, we can determine $lps_{\mathcal{W}}(V)$ in constant time, $O(1)$. As a result we can compute the right subtree of $\widetilde{\mathcal{T}}_{\mathcal{W}}(V)$ in time $O(|lps_{\mathcal{W}}(V)|)$ because $lps_{\mathcal{W}}(V) \in \mathcal{S}_{\mathcal{W}}$. As for the complementary prefix of $V$, it is $V_1$ such that $V = V_1 \circ lps_{\mathcal{W}}(V)$. In particular its length is $n_1 = n - |lps_{\mathcal{W}}(V)|$. Since $V_1$ is a prefix of $V$, the infixes $s_j(V) = s_j(V_1)$ for $j \leq n_1$ and thus we can repeat the same procedure for $V_1$.

The time complexity of the described algorithm is:

$$Time(V) = Time(V_1) + O(|lps_{\mathcal{W}}(V)|).$$

Since $|lps_{\mathcal{W}}(V)| = |V| - |V_1|$ we get that $Time(V) = Time(V_1) + O(|V| - |V_1|)$ and an easy induction argument shows that $Time(V) = O(|V|)$.

□

```
ApproximateCanonicalTree(S_W, n, s)
   (V', j) ← s[n])
   if j = |V'|
      T_right ← T_W(V')
      n_1 ← n − j then
   else
      T_right ← T_W(lps(V'))
      n_1 ← n − |lps(V')|
   fi
   if n_1 = 0 then
      return T_right
   else
      T_left ← ApproximateCanonicalTree(S_W, n_1, s)
      T ←tree with a new root, left subtree T_left and right subtree T_right
      return T
   fi
```

### 8.2.2 Alignment Graphs. Searching of Candidates

Given a query word $V = v_1 \circ v_2 \circ \cdots \circ v_n$ and its approximate canonical tree $\widetilde{\mathcal{T}}_{\mathcal{N}}(V)$, we describe how to generate a list ranked candidates for $V$. To this end it is useful to consider an *alignment graph* associated with the word $V$, see Figure 8.5. This will allow us to define the search problem in terms of graphs and apply graph algorithms to efficiently solve it. The alignment graph for the word $V$ is a directed graph with vertices numbered from 0 to $n$ and arcs, $(i, j)$, correspond in one-to-one fashion to the nodes of $\widetilde{\mathcal{T}}_{\mathcal{N}}(V)$. To make this intuition precise we need the following definition that assigns with each node of the approximate canonical tree a corresponding segment from the word $V$:

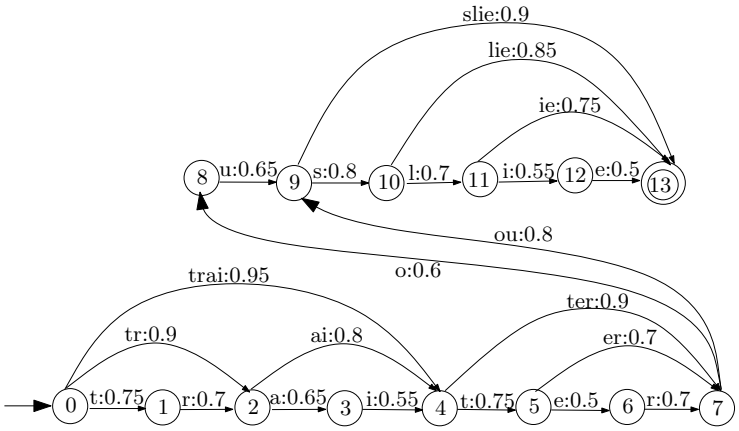



Figure 8.5: The alignment graph $G_{\mathcal{N}}(V)$ for $V = \textit{traiterouslie}$.

**Definition 8.2.4** Given an approximate canonical tree, $\mathcal{T}_{\mathcal{W}}(V)$, for a word $V = v_1 \circ v_2 \circ \cdots \circ v_n$ we associate with each node of the tree an interval in the following way:

1. the interval associated with the root $V$ is $[0; n]$.
2. if $V'$ is a node in the tree with associated interval $int(V') = [i; j]$ and $V'$ has a left child $V'_1$ and right child $V'_2$, then:

$$int(V'_1) = [i; i + |V'_1|] \text{ and } int(V'_2) = [j - |V'_2|; j].$$

It should be clear, that the length of the interval $int(V')$ is exactly $|V'|$. Furthermore, unless $V'_1 = \varepsilon$ or $V'_2 = \varepsilon$, we have that $int(V') = int(V'_1) \cup int(V'_2)$ and the two smaller intervals share only an endpoint. Now, we can define the alignment graph for a word $V$ formally:

**Definition 8.2.5** Given a set of words, $\mathcal{W}$, and a query word, $V = v_1 \circ v_2 \circ \cdots \circ v_n$ of length $n$, an alignment graph for $V$ with respect to the structure, $\mathcal{S}_{\mathcal{W}}$ is a directed graph $G_{\mathcal{W}}(V)$ with nodes $\{0, 1, \ldots, n\}$ and arcs:

$$E_{\mathcal{W}}(V) = \{(i, j) \mid [i; j] = int(V') \text{ for some nonempty word } V' \in \mathcal{T}_{\mathcal{W}}(V) \cap \mathcal{S}_{\mathcal{W}}\}$$

With each arc $(i, j)$ in the graph $G_{\mathcal{W}}(V)$ we associate a label $\lambda(i, j) = I^j_{i+1}(V)$.

It is straightforward that given an approximate canonical tree, we can construct the corresponding alignment graph in linear time:

**Lemma 8.2.6** *Given the approximate canonical tree $\mathcal{T}_{\mathcal{W}}(V)$ for a word $V$, the alignment graph $G_{\mathcal{W}}(V)$ can be constructed in time $O(|V|)$.*

*Proof.* First, we compute the intervals associated with each tree node, see procedure ComputeIntervals. Second, we follow the definition of an alignment

graph, see procedure ComputeAlignmentGraph. The first step requires $O(1)$ time per tree node summing up to $O(|\mathcal{T}_{\mathcal{W}}(V)|)$ time for all the nodes. The second step, see procedure ComputeAlignmentGraph, also requires $O(|V| + |\mathcal{T}_{\mathcal{W}}(V)|)$ time. Since by Lemma 8.2.3 $||\mathcal{T}_{\mathcal{W}}(V)| \in O(|V|)$, the complexity of this step and also of the entire algorithm is $O(|V|)$.

□

```
ComputeIntervals(T, V', i, j)
   int(V') ← [i; j]
   V'_1 ← the left child of V' in T
   V'_2 ← the right child of V' in T
   if V_1 is defined then
      ComputeIntervals(T, V'_1, i, i + |V'_1|)
      ComputeIntervals(T, V'_2, j − |V'_2|, j)
   fi

ComputeAlignmentGraph(S_W, T, V)
   n ← |V|
   ComputeIntervals(T, V, 0, n)
   G ← graph with vertices {0, 1, ..., n} and edges E = ∅
   for V' ∈ T do
      if V' ∈ S_W and V' ≠ ε then
         [i; j] ← int(V')
         E ← E ∪ {(i, j)}
         λ(i, j) ← I^j_{i+1}(V)
      fi
   done
   return G
```

Now, we model how likely it is that a dictionary word $U \in \mathcal{D}$ is the original for the noisy word $V$. To this end we account for: (i) the $\mathcal{N}$-distinxes shared by $V$; (ii) the operations $Op$ and their conditional probabilities $p$ derived during the training stage; (iii) the correct assemblage of operations w.r.t. $U \in \mathcal{D}$ and the query word $V$. We combine this constraints in an optimisation problem as it is commonly done in similar problems, e.g. Moore, [14, 59]:

$$\begin{aligned} \ell(V \to U) \quad &= \quad \max \prod_{j=1}^{k} p(U_j | \lambda(i_{j-1}, i_j)) \\ \text{subject to:} \quad & (i_0, i_1, \ldots, i_k) \text{ a } (0, n)\text{-path in } G_{\mathcal{N}}(V) \\ & (U_j, \lambda(i_{j-1}, i_j)) \in Op \text{ for } j \leq k \\ & U = U_1 \circ U_2 \circ \cdots \circ U_k. \end{aligned}$$

Since, the concatenation of the labels along each path from 0 to $n$ results to $V$, the second two lines essentially say that the path of interest in the graph $G_{\mathcal{N}}(V)$ must correspond to an alignment between $U$ and $V$ with respect to the set of operations, $Op$.

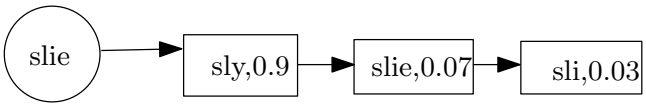



Figure 8.6: The precomputed lists $L(\nu)$ for $\nu = slie$.

**Remark 8.2.7** The reason to deviate from the initial definition of edit-distance that we considered in the previous Chapters is the following. We have modelled not the cost of the operations but rather their frequency, that is how often they are likely to occur. In this context, it is also natural to ask how likely it is that the original of $V$ is $U$, and not what the total cost of modifying $U$ to $V$ would be. In this terms, the original of $V$ should be the most probable amongst the possible words $U$. Technically, one can reduce the multiplication and maximisation to summation and minimisation, respectively, by taking minus logarithm of the probabilities. Yet, the problem remains different from the approximate search problem considered in the previous chapters.

Now, having the notion of likeliness, we state the ranked searching problem for a query word $V$ as:

$$\text{Find all } U^{(i)} \in \mathcal{D} \text{ in decreasing order of } \ell(V \to U^{(i)}) > 0.$$

In order to efficiently solve this problem, we organise the set of all operations, $(U', V') \in Op$, in sorted lists $L(V')$ w.r.t. $p(U'|V')$, see Figure 8.6. This enables a constant access to the most probable operation with right side $V'$ and a successive access to the less probable ones in decreasing order. Thus, we can easily compute an upper bound for the probability of a dictionary suffix which matches the noisy suffix, $v_{i+1} \dots v_n$.

To this end, it suffices to supply each arc $(i, j)$ with a cost $c(i, j) = p(U^{(0)}|\lambda(i, j))$ determined by the first element $U^{(0)}$ in the list $L(\lambda(i, j))$, see Figure 8.5. Then we compute the best cost of an $(i, n)$-path:

$$d(i) = \max \prod_{j=1}^{k} c(i_{j-1}, i_j).$$

Using that $G_{\mathcal{N}}(V)$ is acyclic this can be done in a simple and efficient way. It is rather straightforward that:

$$\begin{aligned} d(n) &= 1 \\ d(i) &= \max_{j:(i,j) \text{ is an arc } in G_{\mathcal{N}}(V)} c(i, j)d(j). \end{aligned}$$

From this observation we can derive the following lemma:

**Lemma 8.2.8** *Given an alignment graph $G_{\mathcal{N}}(V)$ and the values $c(i, j)$ for every arc $(i, j)$ in the graph, the upper bounds $d(i)$ for all $i$ can be computed in total time $O(|V|)$.*

*Proof.* For the proof it suffices to consider the vertices in decreasing order and compute the values $d(i)$ according to the above recurrence, see procedure ComputeUpperBounds. Since each arc $(i, j)$ in the graph $G_{\mathcal{N}}(V)$ has the property $i < j$, the values required on the right hand side in this recurrence will be already computed, by the time of considering the node $i$. In this way each arc in the graph will be considered once only. Thus, the total time of this procedure will be then proportional to the total number of arcs in the graph. However the number of arcs in the graph does not exceed the number of nodes in the approximate canonical tree $\widetilde{\mathcal{T}}_{\mathcal{N}}(V)$. Finally, since $\widetilde{\mathcal{T}}_{\mathcal{N}}(V)$ can be computed in time $O(|V|)$ according to Lemma 8.2.3 it has also at most $O(|V|)$ nodes, thus the number of arcs in $G_{\mathcal{N}}(V)$ is also $O(|V|)$.

□

```
ComputeUpperBounds(G_N(V), c, d)
   n ← |V|
   d[n] ← 1
   for i = n − 1 downto 1 do
      d[i] ← 0
      for each arc (i, j) ∈ G_N(V)
         if d[i] < c(i, j)d[j] then
            d[i] ← c(i, j)d[j]
         fi
      done
   done
```

Disposing on the upper bounds, $d(i)$, and the lists of operations, $L(V')$, we initiate an exhaustive search, see procedure SearchHypotheses. It consists of two main stages. In the first stage, we initialise a set of hypotheses with some hypotheses corresponding to the trivial path from 0 to 0. In the second stage, we *explore* an already defined hypothesis and *generate* new hypotheses.

Each *hypothesis* is a quintuple, $(P_j, U_j, s_j, t_j, p_j)$. It encodes a prefix $P_j$ of the dictionary $\mathcal{D}$ which aligns the prefix $I_1^{s_j}(V)$ according to the alignment graph $G_{\mathcal{N}}(V)$ with probability $p_j$. Furthermore, $(s_j, t_j)$ is an arc in the alignment graph and $(U_j, \lambda(s_j, t_j))$ is an operation. At each stage of the algorithm, we explore the most *promising* quintuple where we account for its current probability and the upper bound, $d(t_j)$, for the best (hypothetical) suffix it could be combined with. In particular, all the generated hypotheses are considered in decreasing order of the characteristic:

$$p_j \times p(U_j|\lambda(s_j, t_j)) \times d(t_j).$$

Due to the extreme properties of the values $d(t_j)$ the above value is the (hypothetically) best possible extension of the prefix $P_j$ followed by the assumption for $U_j$.

SearchHypotheses($\mathcal{S}_{\mathcal{D}}, \boldsymbol{L}, G, \boldsymbol{d}, n, Hypotheses$)

```
Marked ← ∅
InitialiseHypotheses(L, G, d, n, Hypotheses)
while Hypotheses ≠ ∅ do
   h ← Hypotheses.ExtractMin()
   (st(P), U, s, t, p) ← h
   if t ≠ n then
      ExtendHypothesis(h, S_D, L, Gd, n, Hypotheses)
   else
      if st(P) is final state in S_D then
         if st(P) not marked then
            report (P, p)
            mark st(P)
            Marked.Insert(st(P))
         fi
      fi
   fi
   GenerateNextHypothesis(h, L, d, Hypotheses)
done
for st ∈ Mark do
   st unmark
done
```

In the sequel we shall describe each of the two stages in details:

*Initialisation.* We consider the alignment graph $G_{\mathcal{N}}(V)$ and generate all the quintuples of the form $(\varepsilon, U_j^{(0)}, 0, j, 1)$, such that:

$$(0, j) \in E_{\mathcal{N}}(V) \text{ and } U_j^{(0)} \text{ is the first element in } L(\lambda(0, j)).$$

This corresponds to initialise the trivial path 0 and make all possible best assumptions for its immediate extensions, see procedure InitialiseHypotheses.

```
InitialiseHypotheses(S_D, L, G, d, n, Hypotheses)
   Hypotheses ← ∅
   for (0, j) an arc in G do
      s_ε ← initial state of S_D
      U ← L(λ(0, j)).FirstElement()
      h ← (s_ε, U, 0, j, 1)
      Hypotheses.Insert(h, d)
   done
```

*Extension.* Among all the currently generated quintuples, $(P_j, U_j, s_j, t_j, p_j)$, we select the one with highest score, see procedure SearchHypotheses:

$$p_j \times p(U_j | \lambda(s_j, t_j)) \times d(t_j).$$

Thus, we choose the one that seems the most promising to extend to a candidate of maximal probability. Next, we check this hypothesis by extending the prefix $P_j$ with the $U_j$ in the structure $\mathcal{S}_{\mathcal{D}}$. If our hypothesis is true and $P_j \circ U_j$ is a prefix of in $\mathcal{S}_{\mathcal{D}}$, then there are two possible cases:

1. $t_j = n$. In this case, we have reached the end of a path from 0 to $n$, and therefore $P_j \circ U_j$ aligns the word $V$ with probability $p_j \times p((U_j|\lambda(s_j, t_j))$. Thus, if $P_j \circ U_j \in \mathcal{D}$, we report this word as the next generated candidate with probability:
$$p_j \times p((U_j|\lambda(s_j, t_j)).$$

2. $t_j \neq n$. In this case $t_j < n$ and we have to traverse further edges before constructing a path reaching $n$, see procedure ExtendHypothesis. Each of these paths should follow one of the edges emerging from $t_j$ in the graph $G_{\mathcal{N}}(V)$. Therefore we determine the quintuples:
$$(P_j \circ U_j, U', t_j, t', p_j \times p((U_j|\lambda(s_j, t_j))),$$
where $(t_j, t')$ is an edge in $G_{\mathcal{N}}(V)$ and $U'$ is the first element of the list $L(\lambda(t_j, t'))$. Thus, the choice of $U'$ determines the most promising extension of the current hypothesis $P_j \circ U_j$ if we follow the edge $(t_j, t')$.

```
ExtendHypothesis(h, S_D, L, G, d, n, Hypotheses)
   h ← (st(P), U, s, t, p)
   st_next ← TraverseInfixStructure(S_D, st(P), U)
   if st_next is defined and corresponds to a prefix in D then
      p_next ← p × p(U|λ(s, t))
      for (t, t_next) an arc in G do
         U_next ← L(λ(t, t')).FirstElement()
         h_next ← (st_next, U_next, t, t_next, p_next)
         Hypotheses.Insert(h, d)
      done
   fi
```

Thus, we have verified the hypothesis $(P_j, U_j, s_j, t_j, p_j)$ and we extract it from the list of the generated hypotheses. Before proceeding with the next one, we need to account for the other possible extension assigned to the edge $(s_j, t_j)$, see procedure SearchHypotheses. The most promising among them is the one associated with the next element after $U_j$ in the list $L(\lambda(s_j, t_j))$. Thus, provided that $U_j$ is not the last element of $L(\lambda(s_j, t_j))$ and $U'_j$ is the next element in $L(\lambda(s_j, t_j))$, we substitute the hypothesis $(P_j, U_j, s_j, t_j, p_j)$ with the hypothesis $(P_j, U'_j, s_j, t_j, p_j)$. Otherwise, we simply remove $(P_j, U_j, s_j, t_j, p_j)$. This completes the description of the search and generation procedure of correction candidates.

In order to efficiently maintain the access to the most probable hypothesis, its removal and insertion of new hypotheses, we use a heap organised with respect to the values:
$$p_j \times p(U_j|\lambda(s_j, t_j)) \times d(t_j).$$
The method described above is an adapted version of an A-star approach, [24, 25], to solve the shortest path problem. A similar algorithm on acyclic graphs was proposed by Eppstein, [18], and also used by Mohri et at., [45]. The applicability of this approach in our case follows by the fact that the values $d(j)$

do provide an upper bound for any possible suffix in $\mathcal{D}$ that could be generated along a path from $j$ to $n$ in the alignment graph.

In summary, the search algorithm presented in this Section comprises of three main steps, see procedure SearchCorrectionCandidates:

1. computation of the longest suffixes $s_i(V)$ and the approximate canonical tree $\mathcal{T}_{\mathcal{N}}(V)$,
2. computation of the alignment graph $G_{\mathcal{N}}(V)$ and the upper bounds $d(j)$,
3. generation and exploration of hypotheses.

```
SearchCorrectionCandidates(S_N, S_D, L, V)
   ComputeLongestSuffixes(V, S_N, s)
   T ← ApproximateCanonicalTree(S_N, |V|, s)
   G ← ComputeAlignmentGraph(S_N, T, V)
   for (i, j) an arc in G do
      U' ← L(λ(i, j))
      c(i, j) ← p(U'|λ(i, j))
   done
   ComputeUpperBounds(G_N(V), c, d)
   SearchHypotheses(S_D, L, G, d, n, Hypotheses)
```

**Proposition 8.2.9** *For a given query word $V$, the running time of the SearchCorrectionCandidates is:*

$$O(|V| + \sum_{h \in Hypotheses} (|U(h)| + \log |Hypotheses|))$$

*where $Hypotheses$ is the set of all generated hypotheses by the procedure $SearchHypotheses$ and $U(h)$ is the extension infix $U$ for the hypothesis $h$.*

*Proof.* From Lemmata 8.2.2, 8.2.3, 8.2.6, 8.2.8 follows that first two steps of the algorithm are computed in time $O(|V|)$. To estimate the running time of the search procedure, we first observe that each hypothesis is inserted, considered and extracted once only. Hence, using a binary heap to maintain the set of all hypotheses, the cost for an insertion and deletion of a hypothesis is $\log |Hypotheses|$. When extending a hypothesis we spend additional $|U(h)|$ time to follow the transitions of the structure $\mathcal{S}_{\mathcal{D}}$. This shows that we spend $O(|U(h)| + \log |Hypotheses|)$ per hypothesis $h$, thus proving the desired upper bound.

□

**Remark 8.2.10** It is worth mentioning that for each hypothesis $h$ we spend at most $O(|U(h)| + |V| \log |Hypotheses|)$ time since we may add at most $|V| + 1$ new hypotheses. Thus, the running time of the algorithm is determined not by the size of all the hypotheses but by the size of all the generated hypotheses.

| Corpus | size | training set size | test set size | Language | Dictionary |
|---|---|---|---|---|---|
| TCD 1641 [1] | 499 texts 204 Kwords | 437 texts | 62 texts | Early Mod. English | Carnegie Mellon,[2] |
| IMPACT BG, [3] | 37 Kwords | 5 Kwords | 32 Kwords | 19th Cent. Bulgarian | BAS Bulg.,[35] |
| ICAMET,[62] | 469 texts 182 Kwords | 300 texts | 148 texts | Early Mod. English | Carnegie Mellon,[2] |
| TREC-5,[4] | 55 Ktexts 20 Gwords | 27 Ktexts 10 Gwords | 27 Ktexts 10 Gwords | OCR-ed English | Carnegie Mellon,[2] |

Table 8.1: General characteristics of the evaluation resources.

**Remark 8.2.11** We note that our search algorithm can be easily adapted to retrieve the "n-best" candidates or the candidates with probability above a fixed threshold. Furthermore, we can constrain the number of hypotheses it should consider. From the previous Remark, this will speed it up. Finally, we can control the threshold of the candidates' probability that we are interested in. All these features can be easily implemented by introducing an appropriate additional counter or threshold in the while-loop in procedure SearchHypotheses. The time to maintain it appropriately could be easily estimated at $O(1)$.

**Remark 8.2.12** To handle queries which may require non-dictionary candidates we can omit the constraints imposed by the structure $\mathcal{S}_{\mathcal{D}}$. Thus, our algorithm would provide non-dictionary candidates, $U_0^{\mathcal{ND}}, U_1^{\mathcal{ND}}, \ldots$ along with the dictionary candidates $U_0^{\mathcal{D}}, U_1^{\mathcal{D}}, \ldots$. One can rescore the former ones by a factor of $1 - \widetilde{p}(U_0^{\mathcal{D}}|V)$ in order to suppress their influence in case that the dictionary candidate is really a probable one.

## 8.3 Evaluation

We evaluated our approach, REBELS, on two different tasks – normalisation of historical texts and OCR-postcorrection, two different languages – English and Bulgarina, on four different data sets. Table 8.1 summarises the general information for the resources used for the tests.

### 8.3.1 TCD 1641

The TCD 1641 corpus consists of 8000 depositions from the 1641 Irish rebellion, digitised at the Trinity College Dublin [1]. These depositions are written in Early Modern English and the task was to rewrite the words in Modern English.

A representative part of the corpus was manually aligned on word level and we evaluated the performance of REBELS on different types of queries on it.

| Coverage | Moses TT | REBELS |
|---|---|---|
| all | 90.8 | 97.1 |
| nocc | 3.4 | 69.2 |
| occ | 99.5 | 99.8 |
| nid | 78.1 | 90.0 |
| id | 94.4 | 99.1 |

| Errors | Moses TT | REBELS |
|---|---|---|
| all | 9.4 | 5.0 |
| nocc | 96.9 | 47.5 |
| occ | 0.8 | 0.6 |
| nid | 22.7 | 16.5 |
| id | 5.9 | 1.5 |

Table 8.2: The coverage (on the left) and the errors (on the right) in percent of Moses TT and our new system, REBELS. Here *nocc=not occurred*, *occ=occurred*, *nid=nonidentity*, *id=identity*.

We measured the errors[1] and coverage[2] of the first ten candidates generated by the system. We compared the results of REBELS with the results produced by the Moses translation table (Moses TT), [5], after training on the same set of documents. Each query, $V$, was assigned as $nocc(occ)$ if $V \notin \mathcal{N}$ ($V \in \mathcal{N}$) and to $nid(id)$ if $V \neq U (V = U)$ where $U$ is the correct candidate according to the corpus. Tables 8.2 summarise the overall results and the results for these four categories of queries. They reveal an evident advantage for REBELS in unobserved query words and much better coverage on queries with $V \neq U$. In the same time it is almost impeccable on words observed during the training and words which should not be changed.

### 8.3.2 IMPACT BG

The IMPACT BG corpus for Bulgarian consists of 37640 words from late 19th century Bulgarian newspaper articles, digitised at the Bulgarian Academy of Sciences in the framework of the IMPACT Project [3]. The task here was again to find the correct modern word that corresponds to a historic word. Unlike the TCD corpus, the IMPACT BG corpus is unambiguous. This means, that the corpus itself is a set of pairs and not a multiset. However, the same historic words may be assigned to different modern words and vice versa. Another interesting feature of this corpus is that consists of "pure" historic words, i.e. there are no pairs $(U, V)$ with $U = V$.

On this corpus we studied the saturation of REBELS. To this end the words in the corpus were sorted in decreasing order with respect to their frequency in the historical language. Afterwards we set six experiments as follows. For experiment number $k \leq 6$ we used the first $5000k$ pairs for training and we evaluated the performance of our system on the last $37640 - 5000k$ historical words. Table 8.3 gives the coverage and errors from these six experiments.

[1] That is the ratio of the cases where the first suggested candidate is not the correct one to the number of all queries.

[2] The coverage is the ratio of the cases where one of the suggested candidates is the correct one to the number of all queries.

| k | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| coverage | 86.2 | 85.0 | 92.5 | 95.7 | 95.9 | 97.9 |
| errors | 18.5 | 20.0 | 13.4 | 8.9 | 8.0 | 4.5 |

Table 8.3: Saturation of REBELS on Historical Bulgarian dictionary. Coverage and errors in percent.

| | VARD2,[10] | Moses,[5] | REBELS |
|---|---|---|---|
| WER | 35.0 | 11.3 | 9.1 |

Table 8.4: WER in percent for ICAMET of VARD2, Moses and REBELS.

### 8.3.3 ICAMET

The ICAMET Corpus of Letters contains 469 complete letters (182 000 words), from different sources, written between 1386 and 1698 digitised at Innsbruck University [62]. On ICAMET we evaluated the word error rate[3] (WER) on the normalisation result. We trained Moses on training set and used the phrase pairs of length less than four from the Moses translation table. When translating a document we called our system for single words or for two consecutive words. We choose the sequence of modern words with best total score. We compared our results with the results obtained by Moses when the language model is switched off. Table 8.4 reports the results achieved on ICAMET by a previous system VARD2[4], [10], by Moses and by REBELS. Applying Moses with a language model trained on the Gutenberg corpus, [6], improved Moses' results to 9.7% WER, thus could not achieve the performance of REBELS without using any language model.

### 8.3.4 TREC-5

The TREC-5 Confusion Track Corpus is compiled from the 1994 edition of the Federal Register of the United States Government Printing Office. It consists of about 55 600 original texts and their OCR-ed variants that contain about 5% of errors on character level. We used 50% of the documents for training and evaluated the error rate and the coverage, both on word level, on the rest 50% of the documents. Our coverage ratio, see Table 8.5, improve previous results on this data set, reported in [54] and [41]. The error rate of REBELS is 4.31%.

Although we do not have a formal framework that would allow us to argue the superiority of our approach to previous techniques to defining similarity between words, the experiments presented in this section reveal that it is adequate, language independent and applicable in different areas.

[3]This is the Levenshtein edit-distance of modernisation and the corrected text divided by the length of the corrected text.

[4]In [10] the authors report precision of 65%.

| | REBELS | Levenshtein Automata with training, [41] | Levenshtein Automata, [54] |
|---|---|---|---|
| coverage | 98.37 | 96.82 | 91.21 |

Table 8.5: Coverage in percent for TREC-5 of REBELS and automata techniques described in [41] and [54].

# Appendix A

# Bookkeeping

In the sequel we describe the technical details involved in the memory bookkeeping required in Chapter 6. We formally prove an upper bound for the space requirement for an inverted index which provides the answers of the approximate search problem for an arbitrary word, $V$, of length not exceeding $N$. Setting $N = 4(\rho - 1)$ where $\rho = \rho(Op)$ is the length of the longest right side of an operation, results to a solution of our particular problem. The idea is a simplified version of the one outlined in Chapter 7.

**Definition A.1** The generating functions $\phi_\varepsilon(z)$ and $\phi_\Sigma(z)$ with respect to a generalised edit-distance $(Op, c)$ are introduced as:

$$\phi_\varepsilon(z) = \sum_{op \in \Lambda} z^{c(op)} \text{ and}$$
$$\phi_\Sigma(z) = \sum_{op \in Op \setminus \Lambda} z^{c(op)}.$$

**Lemma A.2** *Let $N, M \in \mathbb{N}$ be arbitrary and $q \in (0; 1)$. If*

$$\mathcal{A}_{N,M} = \{\omega \in Op^* \mid |r(\omega)| = N \,\&\, c(\omega) \leq qM\}$$

*then the size of $\mathcal{A}_{N,M}$ is bounded by:*

$$|\mathcal{A}_{N,M}| \leq z^{-qM} \sum_{k=0}^{qM} \binom{N+k}{k} \phi_\varepsilon^k(z) \phi_\Sigma^N(z)$$

*Proof.* For each alignment $\omega = op_1 \circ op_2 \circ \cdots \circ op_n$. we can view $\omega$ as a triple $\omega = \langle \omega_\varepsilon, \omega_\Sigma, \beta_\omega \rangle$ where:

1. $\omega_\varepsilon = op_{j'_1} \circ op_{j'_2} \circ \cdots \circ op_{j'_k}$ where $j'_1 < j'_2 < \cdots < j'_k \leq n$ and $op_{j'_i} \in \Lambda$.
2. $\omega_\Sigma = op_{j''_1} \circ op_{j''_2} \circ \cdots \circ op_{j''_m}$ where $j''_1 < j''_2 < \cdots < j''_m \leq n$ and $op_{j''_i} \in Op \setminus \Lambda$.
3. $\beta_\omega \in \{0, 1\}^n$ with $\beta_\omega(j) = 1$ if and only if $op_j \in Op_\varepsilon^{(l)}$

Clearly the cost of the alignment $\omega$ is $c(\omega) = c(\omega_\varepsilon) + c(\omega_\Sigma)$ and the length $|r(\omega)| = |r(\omega_\Sigma)|$ since $r(\omega_\varepsilon) = \varepsilon$. Thus $\omega \in \mathcal{A}_{N,M}$ if and only if:

$$\begin{aligned} |r(\omega_\Sigma)| &= N \\ c(\omega_\Sigma) + c(\omega_\varepsilon) &\leq qM. \end{aligned}$$

Now consider a variable $z \in (0;1)$. It is straightforward to see that:

$$\begin{aligned} z^{c(\omega_\Sigma)+c(\omega_\varepsilon)-qM} &\geq 0 \text{ and} \\ z^{c(\omega_\Sigma)+c(\omega_\varepsilon)-qM} \geq 1 &\iff c(\omega_\varepsilon) + c(\omega_\Sigma) \leq qM. \end{aligned}$$

Furthermore, since each operation of the set $\Lambda$ is of positive cost, there are no more than $qM$ operations in $\omega_\varepsilon$. Consequently, we obtain:

$$\begin{aligned} |\mathcal{A}_{N,M}| &= \sum_{\omega \in \mathcal{A}_{N,M}} 1 \\ &\leq \sum_{\substack{\omega_\varepsilon \in \Lambda^* \\ |\omega_\varepsilon| \leq qM}} \sum_{\substack{\omega_\Sigma \in (Op \setminus \Lambda)^* \\ |r(\omega)| = N}} \sum_{\substack{\beta \in \{0,1\}^* \\ ||\beta||_0 = |\omega_\Sigma| \\ ||\beta||_1 = \omega_\varepsilon}} z^{c(\omega_\varepsilon)+c(\omega_\Sigma)-qM} \\ &= \sum_{\substack{\omega_\varepsilon \in \Lambda^* \\ |\omega_\varepsilon| \leq qM}} \sum_{\substack{\omega_\Sigma \in (Op \setminus \Lambda)^* \\ |r(\omega)| = N}} \binom{|\omega_\varepsilon| + |\omega_\Sigma|}{|\omega_\varepsilon|} z^{c(\omega_\varepsilon)} z^{c(\omega_\Sigma)} z^{-qM} \\ &= z^{-qM} \sum_{\substack{\varepsilon \in \Lambda^* \\ |\omega_\varepsilon| \leq qM}} z^{c(\omega_\varepsilon)} \sum_{\substack{\omega_\Sigma \in (Op \setminus \Lambda)^* \\ |r(\omega)| = N}} \binom{|\omega_\varepsilon| + |\omega_\Sigma|}{|\omega_\varepsilon|} z^{c(\omega_\Sigma)}. \end{aligned}$$

Note, that for each $\omega_\Sigma$ we have that $|\omega_\Sigma| \leq |r(\omega_\Sigma)|$ because each of the operations which constitute $\omega_\Sigma$ is of non-zero length right hand side whereas $r(\omega_\varepsilon) = \varepsilon$ is of length 0. Therefore, $\binom{|\omega_\Sigma|+|\omega_\varepsilon|}{|\omega_\varepsilon|} \leq \binom{N+|\omega_\varepsilon|}{|\omega_\varepsilon|}$ whenever $|r(\omega_\Sigma)| = N$. Hence we obtain:

$$\begin{aligned} |\mathcal{A}_{N,M}| &\leq z^{-qM} \sum_{\substack{\varepsilon \in \Lambda^* \\ |\omega_\varepsilon| \leq qM}} z^{c(\omega_\varepsilon)} \sum_{\substack{\omega_\Sigma \in (Op \setminus \Lambda)^* \\ |r(\omega)| = N}} \binom{|\omega_\varepsilon| + |\omega_\Sigma|}{|\omega_\varepsilon|} z^{c(\omega_\Sigma)} \\ &\leq z^{-qM} \sum_{\substack{\omega_\varepsilon \in \Lambda^* \\ |\omega_\varepsilon| \leq qM}} \binom{N + |\omega_\varepsilon|}{|\omega_\varepsilon|} z^{c(\omega_\varepsilon)} \sum_{\substack{\omega_\Sigma \in (Op \setminus \Lambda)^* \\ |r(\omega)| = N}} z^{c(\omega_\Sigma)}. \end{aligned}$$

Now we compute the sum:

$$\begin{aligned}
\sum_{\substack{\varepsilon\in\Lambda^*\\ |\omega_\varepsilon|\le qM}} \binom{N+|\omega_\varepsilon|}{|\omega_\varepsilon|} z^{c(\omega_\varepsilon)} &= \sum_{k=0}^{\le qM} \sum_{\substack{\omega_\varepsilon\in\Lambda^*\\ |\omega_\varepsilon|=k}} \binom{N+k}{k} z^{c(\omega_\varepsilon)} \\
&= \sum_{k=0}^{\le qM} \binom{N+k}{k} \sum_{\substack{op_1\circ op_2\cdots\circ op_k\\ op_j\in\Lambda}} z^{\sum_{j=1}^k c(op_j)} \\
&= \sum_{k=0}^{\le qM} \binom{N+k}{k} \sum_{\substack{op_1\circ op_2\cdots\circ op_k\\ op_j\in\Lambda}} \prod_{j=1}^{k} z^{c(op_j)} \\
&= \sum_{k=0}^{\le qM} \binom{N+k}{k} \prod_{j=1}^{k} \sum_{op_j\in\Lambda} z^{c(op_j)} \\
&= \sum_{k=0}^{\le qM} \binom{N+k}{k} \prod_{j=1}^{k} \phi_\varepsilon(z) = \sum_{k=0}^{\le qM} \binom{N+k}{k} \phi_\varepsilon^k(z).
\end{aligned}$$

Next consider the sum:

$$\begin{aligned}
\sum_{\substack{\omega_\Sigma\in(Op\setminus\Lambda)^*\\ |r(\omega)|=N}} z^{c(\omega_\Sigma)} &= \sum_{\substack{op_1\circ op_2\ldots op_n\\ op_j\in Op\setminus\Lambda\\ \sum_j |r(op_j)|=N}} z^{\sum_{j=1}^n c(op_j)} \\
&= \sum_{\substack{op_1\circ op_2\ldots op_n\\ op_j\in Op\setminus\Lambda\\ \sum_j |r(op_j)|=N}} \prod_{j=1}^{n} z^{c(op_j)} \\
&= \sum_{\substack{op_1\circ op_2\ldots op_n\\ op_j\in Op\setminus\Lambda\\ \sum_j |r(op_j)|=N}} \prod_{j=1}^{n} \left[ z^{c(op_j)} \prod_{m=2}^{|r(op_j)|} z^0 \right]
\end{aligned}$$

Now we can interpret the term $z^0$ in the product $\prod_{k=2}^{|r(op_j)|} z^0$ as $z^{c((u_k,u_k))}$ where $u_k$ is the $k$-th character of $r(op_j)$. In this way we map injectively the sequence of the operations $op_1, op_2, \ldots op_n$ to a sequence of operations $op'_1, op'_2, op'_3, \ldots op'_N$ with:

$$\prod_{j=1}^{n} z^{c(op_j)} = \prod_{j=1}^{N} z^{c(op'_j)}.$$

Since $z > 0$ we obtain:

$$\begin{aligned}
\sum_{\substack{\omega_\Sigma \in (Op\setminus\Lambda)^* \\ |r(\omega)|=N}} z^{c(\omega_\Sigma)} &= \sum_{\substack{op_1 \circ op_2 \ldots op_n \\ op_j \in Op\setminus \\ \sum_j |r(op_j)|=N}} \prod_{j=1}^{n} \left[ z^{c(op_j)} \prod_{m=2}^{|r(op_j)|} z^0 \right] \\
&\leq \sum_{\substack{op_1 \circ op_2 \ldots op_N \\ op_j \in Op\setminus\Lambda}} \prod_{j=1}^{N} z^{c(op_j)} \\
&= \prod_{j=1}^{N} \sum_{op_j \in Op\setminus\Lambda} z^{c(op_j)} \\
&= \prod_{j=1}^{N} \phi_\Sigma(z) = \phi_\Sigma^N(z)
\end{aligned}$$

Summing up we get:

$$|\mathcal{A}_{N,M}| \sum_{\omega \in \mathcal{A}_{N,M}} 1 \leq z^{-qM} \sum_{k=0}^{qM} \binom{N+k}{k} \phi_\varepsilon^k(z) \phi_\Sigma^N(z)$$

as stated.

□

As a corollary we obtain:

**Corollary A.3** *Let $N, M \in \mathbb{N}$ be fixed and $q \in (0;1)$ and $z \in (0;1)$. Then:*

$$|\mathcal{A}_{N,M}| \leq \exp\left((N+qM)\phi_\varepsilon(z) - qM \log z + N \log \phi_\Sigma(z)\right)$$

*Proof.* Indeed for $k \leq qM$, we have that $\binom{N+k}{k} \leq \frac{(N+k)^k}{k!} \leq \frac{(N+qM)^k}{k!}$. Now according to Lemma A.2 we obtain:

$$\begin{aligned}
|\mathcal{A}_{N,M}| &\leq z^{-qM} \sum_{k=0}^{qM} \binom{N+k}{k} \phi_\varepsilon^k(z) \phi_\Sigma^N(z) \\
&\leq z^{-qM} \phi_\Sigma^N(z) \sum_{k=0}^{qM} \frac{(N+qM)^k \phi_\varepsilon^k(z)}{k!} \\
&\leq z^{-qM} \phi_\Sigma^N(z) \sum_{k=0}^{\infty} \frac{(N+qM)^k \phi_\varepsilon^k(z)}{k!} \\
&= z^{-qM} \phi_\Sigma^N(z) \exp((N+qM)\phi_\varepsilon(z)) \\
&= \exp\left((N+qM)\phi_\varepsilon(z) - qM \log z + N \log \phi_\Sigma(z)\right).
\end{aligned}$$

□

Using Corollary A.3 it is not difficult to give a rough upper bound for the number of alignments $\mathcal{A}_{N,M}$ which does not depend on the specific form of the operations $Op$ but only on the size of $|\Lambda|$, $|Op|$ and the size of the alphabet $\Sigma$. To this end we note that each function $\phi_\Sigma(z)$ is of the form:

$$\phi_\Sigma(z) = \sum_{op \in Op \setminus \Lambda} z^{c(op)} = \sum_{op \in Id} z^{c(op)} + \sum_{op \in Op \setminus (\Lambda \cup Id)} z^{c(op)}.$$

However, if $op \in Id$ we get that $c(op) = 0$ and otherwise $c(op) \geq 1$. Therefore for $z \in (0;1)$ we obtain:

$$\phi_\Sigma(z) \leq |Id| + z(|Op| - |\Lambda| - |Id|).$$

Analogously, using that $\Lambda \cap Id = \emptyset$ we can bound $\phi_\varepsilon(z)$ for $z \in (0;1)$ by:

$$\phi_\varepsilon(z) \leq |\Lambda| z.$$

Taking into account Corollary A.3 and $|Id| = |\Sigma|$ we get the following upper bound:

**Corollary A.4** *For integers number $N, M \in \mathbb{N}$ and $q, z \in (0;1)$ it holds:*

$$|\mathcal{A}_{N,M}| \leq \exp\left((N + qM)z|\Lambda| - qM \log z + N \log(|\Sigma| + z(|Op| - |\Lambda| - |\Sigma|)\right).$$

*In particular for $z = \frac{qM}{N+qM}|\Lambda|^{-1}$ this implies:*

$$|\mathcal{A}_{N,M}| \leq \exp\left(qM\left(\log\frac{N+qM}{qM} + 1 + \log|\Lambda|\right) + N\log\left(|\Sigma| + \frac{qM}{N+qM}\frac{|Op| - |\Lambda| - |\Sigma|}{|\Lambda|}\right)\right).$$

□

Actually, $z = \frac{qM}{N+qM}|\Lambda|^{-1}$ is the extreme point for the function:

$$(N + qM)z|\Lambda| - qM \log z.$$

Clearly a more sharp upper bound is provided by $z'$ which minimises the function:

$$(N + qM)z|\Lambda| - qM \log z + N \log(|\Sigma| + z(|Op| - |\Lambda| - |\Sigma|).$$

Corollary A.4 shows the following asymptotic upper bound for the size of $\mathcal{A}_{N,M}$:

**Lemma A.5** *For every two positive integers $N$ and $M$, $|\mathcal{A}_{N,M}| \in \exp O(qM + N)$.*

□

# Conclusion

In the current work we carried out a theoretical study on the approximate search problem in a regular set of words. However, one should keep in mind that for practical purposes, e.g. OCR-postcorrection, genome analysis, normalisation of historical texts, the approximate search problem arises only as a subtask. Essentially, solving the approximate search problem, yields a **list** of correction candidates. Choosing the *correct* candidate from this list is considered as a separate problem . In particular, it can be done by manual inspection or another automatic procedure that ranks the correction candidates. Therefore, if one follows this direction, the time efficiency of the approximate search stage becomes an important issue.

An alternative approach that extracts the relevant operations, their scores and provides a search procedure that can be directly applied to solve the above tasks, was proposed and developed in Chapter 8 .

In this context the main contributions of the current work are:

1. **General framework for study of alignments and efficient generation of correction candidates**. The approach considered in Chapter 4 provides a general framework to consider alignments and study their properties. The notion of edit-distance lists then allows to efficiently implement these observations in practice when we are interested only in the generation of correction candidates.

2. **New algorithm for approximate search in arbitrary regular sets of words.** In Chapter 5 we described a divide and conquer algorithm for the approximate search problem in regular sets. In this way we generalised the algorithm of Myers, [47], where a special case of a finite set of words is considered, algorithm of Baeza-Yates and Navarro, [49], and Mihov and Schulz [42]. Applying the technique of edit-distance lists we further avoid the redundancies encountered in all these algorithms during the generation of candidates.

3. **New algorithm for approximate search w.r.t. arbitrary edit-distance.** In Chapter 6 we showed that our algorithm can be easily extended to capture generalised edit-distances thus extending it beyond the standard Levenshtein edit-distance. To achieve this we made an observation on the properties of the alignments of interest and reflected it in terms

of edit-distance lists – the general framework developed in Chapter 4.

4. **Efficiency of the proposed algorithm, practically.** Due to the use of edit-distance lists, the proposed divide and conquer algorithm generates each correction candidate once only, Chapters 4 and 6. Proposition 5.3.6, Proposition 5.3.15 argue this result mathematically in a special case for the edit-distance. Proposition 6.2.8 and Proposition 6.2.9 lead to the same conclusion in the general case. It also preserves the basic property of the Myers' algorithm, [47]. Hence, at the beginning it starts with an exact match and the error-tolerance increases with the length of the currently generated candidate. In this way the error-tolerance is suppressed for short infixes and it comes close to the desired error-tolerance only for infixes that are long and thus informative with respect to the given language.

5. **Efficiency of the proposed algorithm, theoretically.** In Chapter 7 we proposed a general framework which enables to assess the quality of our algorithm. In Proposition 7.2.2 and Proposition 7.2.7 show that on average the running time of our algorithm is linear where the constant depends on type edit-distance, the threshold and the structure of the regular language.

   Although the proposed framework is different from the one used in [47] and [49], this result can be regarded in the following way. Given the distribution of the input alphabet and the regular language, how can we choose the threshold parameter and the edit-distance so that we achieve a desired time efficiency. This is motivated for application purposes where we desire to get a manageable list of correction candidates and definitely not the entire set of words (in the finite case).

6. **New approach to defining similarity between words.** In Chapter 8 we described a new approach for extraction of operations and definition of similarity between words. Based on the structure of the language and the structure of a given training set of instances, it takes into account context of various size that is implicitly contained in the data. Hence, it can be flexibly applied for different languages and different sources of noise. The designed algorithm is shown to be practical and knowing the parameters of its efficiency and the specific application of the notion of similarity, one can easily control its efficiency derived in Proposition 8.2.9.

The main concept that stays behind Results 1–5 extends previous ideas of Myers, [47], Baeza-Yates and Navarro, [49], and Mihov and Schulz, [42]. Its realisation relies on classical results in the theory of finite state automata and partially uses previous results due to Blumer et al., [11, 12], Ko and Aluru, [34]. To achieve Result 5 the author adapted an algebraic approach proposed of Eilenberg, [17], classical results from linear algebra, graph theory, theory of the finite state automata. Result 6 is a product of an original mathematical interpretation of the structure of Blumer et al., [12]. For its algorithmic implementation it applies ideas due to Aho and Corasick, [7], and also generic idea suggested by Hart et al., [24, 25], that nicely fits in the framework proposed by the author.

The Results 1 to 6 were reflected in one own paper and in several joint papers with Stoyan Mihov, Petar Mitankin, Klaus Schulz, Vladislav Nenchev, as follows:

1. **Some algebraic properties of alignments of words, S. Gerdjikov, Comptes rendu de l'Academie bulgare des Sciences, 65(10):1311–1319, 2012**,

   This is a paper with one author. In this paper are reflected the basic steps leading to Results 1 and 5. In particular, it describes the basic properties of the edit-distance lists and alignments sets that are in the background of achieving Propositions 5.3.6 and 5.3.15. It also communicates Lemma 7.1.6 that is the essential ingredient for the proof of Propositions 7.2.2 and 7.2.7.

2. **WallBreaker - overcoming the wall effect in similarity search, S. Gerdjikov, S. Mihov, P. Mitankin, and K. U. Schulz, ACM Proceedings of the 2013 Joint EDBT/ICDT Workshops, 2013,**

   and its full version:

   **Good parts first - a new algorithm for approximate search in lexica and string databases, S. Gerdjikov, S. Mihov, P. Mitankin, and K. U. Schulz, ArXiv, 2013 e-prints:http://adsabs.harvard.edu/abs/2013arXiv1301.0722G.**

   These two papers are joint papers with Petar Mitankin, Stoyan Mihov and Klaus Schulz. The algorithm for approximate search described in these papers, is a result obtained in the seminar led by Stoyan Mihov. During the regular sessions of this seminar both Petar Mitnakin and the author worked and presented their progress on this problem. Thus, they reached to two different yet equivalent solutions in the case of Levenshtein edit-distance. The differences were in the specific linear data structure used for the representation of the set of words.

   Studying in more details the references Petar Mitankin came across to an even more compact data structure proposed in [12, 29].

   The author extended the algorithm to the case of arbitrary edit-distance.

   Results 2, 3 and 4 build on the ideas presented in these two papers, [20] and [21]. Furthermore, the experimental results in [20] and the independent evaluation carried out during Workshop S4, see above, empirically confirm the statement expressed by Result 4.

3. **Extraction of spelling variations for noisy text correction. S. Gerdjikov, S. Mihov, and V. Nenchev, In Proceedings of 12th International Conference on Documents Analysis and Recognition 2013, 2013, p.324–328.**

   This is a joint paper with Stoyan Mihov and Vladislav Nenchev. In this paper the author contributed the approach, its development and imple-

mentation. The paper presents the key ideas of the approach along with its empirical evaluation. Thus, essentially it summarises Result 6.

Some of the results were also presented on international forums as follows:

- **WallBreaker - overcoming the wall effect in similarity search, S. Gerdjikov, S. Mihov, P. Mitankin, and K. U. Schulz, on the EDBT/ICDT Workshop for Scalable String Similarity Search/Join, Genoa, Italy, 2013.** (oral presentation S. Gerdjikov)

  In this talk the author presented the essential ideas from papers [21] and [20] with a special focus on Result 4 – the practical efficiency of the proposed approximate search algorithm.

- **Extraction of spelling variations for noisy text correction. S. Gerdjikov, S. Mihov, and V. Nenchev on the 12th International Conference on Documents Analysis and Recognition, Washington, DC, USA, 2013.** (poster presentation S. Gerdjikov)

  With this poster the author illustrated the results from [22]. In discussions with scientists from different areas, interested in processing historical texts or OCR-postcorrection, the author presented Result 6 from different viewpoints and motivated its capacity for various tasks.

- **On Modernisation of Historical Texts. S. Gerdjikov, Computability in Europe 2012, Cambridge, UK, 2012.** (informal talk)

  In this presentation the author presented Result 6 in the context of the normalisation of Early Modern English words.

Essential parts of Result 5 and Result 6 were also presented on the Spring Session of the Faculty for Mathematics and Informatics, Sofia University:

- **In what extent are the spelling variations in the 19th century Bulgarian language regular? S. Gerdjikov, Spring Session of the Faculty for Mathematics and Informatics, Sofia University, 2012** (oral presentation at the Chair of Logic and its Applications),

  In this talk the author presented Result 6 on the particular task of normalisation of historical Bulgarian language.

- **Combinatorial etude: "Alignments of words" S. Gerdjikov, Spring Session of the Faculty for Mathematics and Informatics, Sofia University, 2011** (oral presentation at the Chair of Logic and its Applications),

  In this talk the author presented the combinatorial results, Lemma 7.1.6, related to Result 5.

With regard to the previous paragraphs, the author declares that the current dissertation is an original scientific work. The use of previous results is fairly reflected with appropriate references, and complies with the copyrights of their authors and/or publishers and/or copyright holders of the specific results.

# Bibliography


[1] http://1641.tcd.ie.

[2] http://www.speech.cs.cmu.edu/cgi-bin/cmudict.

[3] http://www.impact-project.eu.

[4] http://trec.nist.gov/pubs/trec5/t5_proceedings.html.

[5] http://www.statmt.org/moses/.

[6] http://www.gutenberg.org.

[7] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18:333–340, June 1975.

[8] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. A basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[9] R. A. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[10] A. Baron and P. Rayson. Automatic standardisation of texts containing spelling variations how much training data do you need? In *Proceedings of the Corpus Linguistics Conference, CL2009*, 2009.

[11] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas. The smallest automation recognizing the subwords of a text. *Theoretical Computer Science*, 40:31 – 55, 1985. Eleventh International Colloquium on Automata, Languages and Programming.

[12] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the Association for Computing Machinery*, 34(3):578–595, 1987.

[13] L. Boytsov. Super-linear indices for approximate dictionary super-linear indices for approximate dictionary super-linear indices for approximate dictionary searching. In *Proceedings of the 5th International Conference on Similarity Search and Applications*, 2012.

[14] E. Brill and R. C. Moore. An improved error model for noisy channel spelling correction. In *Proc. 38th ACL*, pages 286–293, 2000.

[15] H.-L. Chan, T.-W. Lam, W.-K. Sung, S.-L. Tam, and S.-S. Wong. Compressed indexes for approximate string matching. In *ESA'06: Proceedings of the 14th conference on Annual European Symposium*, pages 208–219, London, UK, 2006. Springer-Verlag.

[16] R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 91–100, New York, NY, USA, 2004. ACM.

[17] S. Eilenberg. *Automata, Languages and Machines*, volume A. Academic Press, Inc. Orlando, Fl, USA, 1974.

[18] D. Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, pages 652–673, 1998.

[19] S. Gerdjikov. Some algebraic properties of alignments of words. *Comptes rendu de l'Academie bulgare des Sciences*, 65(10):1311–1319, 2012.

[20] S. Gerdjikov, S. Mihov, P. Mitankin, and K. Schulz. Good parts first - a new algorithm for approximate search in lexica and string databases. ArXiv e-prints:http://adsabs.harvard.edu/abs/2013arXiv1301.0722G, 2013.

[21] S. Gerdjikov, S. Mihov, P. Mitankin, and K. Schulz. Wallbreaker - overcoming the wall effect in similarity search. In *ACM Proceedings of the 2013 Joint EDBT/ICDT Workshops*, 2013.

[22] S. Gerdjikov, S. Mihov, and V. Nenchev. Extraction of spelling variations for noisy text correction. In *Proceedings of 12th International Conference on Documents Analysis and Recognition*, 2013.

[23] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[24] P. Hart, N. Y. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC4*, 4(2):100–107, 1968.

[25] P. Hart, N. Y. Nilsson, and B. Raphael. Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Newsletter*, 37:28–29, 1971.

[26] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24:664–75, 1977.

[27] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley Publishing Company, 1979.

[28] R. Horn and C. Johnson. *Norms for Vectors and Matrices*. Cambridge University Press, 1990.

[29] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa. On-line construction of symmetric compact directed acyclic word graphs. In *Proc. of 8th International Symposium on String Processing and Information Retrieval (SPIRE'01)*, pages 96–110. IEEE Computer Society, 2001.

[30] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi. On-line construction of compact directed acyclic word graphs. *Word Journal Of The International Linguistic Association*, 146(2):1–12, 2005.

[31] J. Kärkkäinen and P. Snaders. Simple linear work suffix array construction. In *In proceedings of the 30th International Colloquium Automata, Languages and Programming.*, pages 81–88, Cancun, Mexico, 2003. IEEE Computer Society.

[32] S. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, (34):3–41, 1956.

[33] J. Klovstad and L. Mondshein. The caspers linguistic analysis system. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 23(1):118–123, 1975.

[34] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, (3):143–156, 2005.

[35] S. Koeva. Grammar dictionary of the bulgarian language. description of the concept for organization of the linguistic data. *Bulgarian language*, 6:49–58, 1998.

[36] K. Kukich. Techniques for automatically correcting words in texts. *ACM Computing Surveys*, pages 377–439, 1992.

[37] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 1966.

[38] D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227:1435–41, 1985.

[39] M. G. Maaß. Linear bidirectional on-line construction of affix trees. In *Proc. of 11th Ann. Symp. on Combinatorial Pattern Matching (LNCS1848)*, pages 320–334. Springer-Verlag, 2000.

[40] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the Association for Computing Machinery*, 23(2):262–272, 1976.

[41] S. Mihov, P. Mitankin, A. Gotscharek, U. Reffle, and C. Schulz, K. U. Ringlstetter. Using automated error profiling of texts for improved selection of correction candidates for garbled tokens. In *AI 2007: Advances in Artificial Intelligence*, pages 456–465. Springer Berlin Heidelberg, 2007.

[42] S. Mihov and K. U. Schulz. Fast approximate search in large dictionaries. *Computational Linguistics*, 30(4):451–477, 2004.

[43] P. Mitankin, S. Mihov, and K. U. Schulz. Deciding word neighborhood with universal neighborhood automata. *Theoretical Computer Science*, 412(22):2340 – 2355, 2011.

[44] M. Mohri, P. Moreno, and E. Weinstein. General suffix automaton construction algorithm and space bounds. *Theoretical Computer Science*, 410(37):3553–3562, 2009.

[45] M. Mohri and M. Riley. An efficient algorithm for the n-best-strings problem. In *Proceedings ICSLP*, 2002.

[46] M. Mor and A. S. Fraenkel. A hash code method for detecting and correcting spelling errors. *Commun. ACM*, 25(12):935–938, 1982.

[47] E. W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12:345–374, 1994.

[48] P. Nabende. *Applying Dynamic Bayesian Networks in Transliteration Detection and Generation*. PhD thesis, University of Groningen, 2011.

[49] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, 1(1):205–239, 2000.

[50] K. Oflazer. Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics*, 22(1):73–89, 1996.

[51] S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2), 2007.

[52] U. Reffle. Efficiently generating correction suggestions for garbled tockens of historical language. *Natural Language Engineering*, 17(2):265–282, 2011.

[53] E. S. Rista and P. N. Yianilos. Learning string-edit distance. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 20(5):522–532, 1998.

[54] K. Schulz, S. Mihov, and P. Mitankin. Fast selection of small and precise candidate sets from dictionaies for text correction tasks. In *ICDAR '07: Proceedings of the Ninth International Conference on Document Analysis and Recognition*, pages 471–75, Washington, DC, USA, 2007. IEEE Computer Society.

[55] K. U. Schulz and S. Mihov. Fast string correction with Levenshtein automata. *IJDAR*, 5(1):67–85, 2002.

[56] F. J. Sellers. Bit loss and gain correction code. *Information Theory, IRE Transactions on*, 8(1):35–38, 1962.

[57] J. Stoye. Affixbäume. Master's thesis, Universität Bielefeld, May 1995.

[58] J. Stoye. Affix trees. Technical Report 2000-04, Universität Bielefeld, Technische Fakultät, 2000.

[59] K. Toutaniva and R. C. Moore. Pronunciation modeling of improved spelling correction. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 144–151, 2002.

[60] E. Ukkonen. Algorithms for approximate string matching. *Information Control*, 64:100–18, 1985.

[61] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14(3):249–260, 1995.

[62] http://www.uibk.ac.at/anglistik/projects/icamet.

[63] M. S. Watermann. *Computational Biology: Maps, Sequences, Genomes.* Chapman and Hall, London, England, 1995.

[64] M. S. Watermann, M. Eggert, and E. Lander. A new algorithm for best subsequence alignments with application to trna-rrna comparison. *J. Mol. Biol.*, 197:723–728, 1987.

[65] M. S. Watermann and M. Vingron. Rapid and accurate estimates of statistical significance for sequence data base searches. In *Proc. Natl. Academy Sciences*, volume 81, pages 4625–4628, 1994.

[66] P. Weiner. Linear pattern matching algorithms. In *Proceedings of 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[67] S. Wu and U. Manber. Fast text searching: allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.