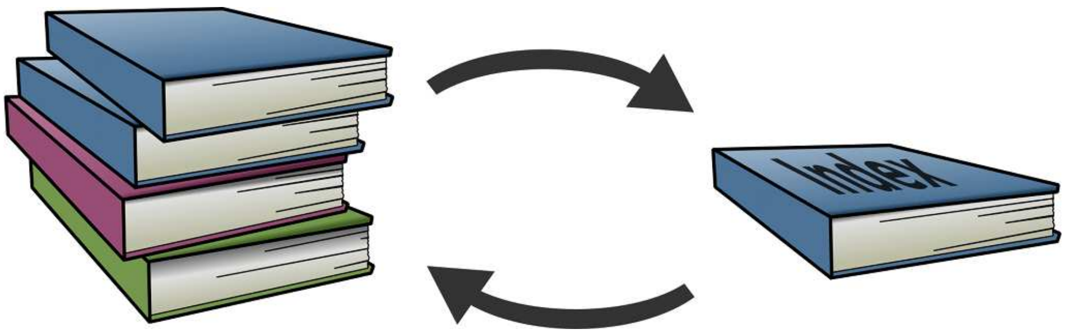
university of groningen
faculty of mathematics and natural sciences

Indexing Compressed Text

Wouter Lueks



Bachelor Thesis in Mathematics and Computing Science

August 2008

Indexing Compressed Text

Summary

We study a method by Ferragina and Manzini for creating an index of a text. This index allows us to find any string in the original text. What is so special about this index is that it is smaller than the original text, while still allowing quick searching and recovery of the original text.
In order to understand the performance bounds given by Ferragina and Manzini we first examine the concept of information density, the entropy. Next we examine the details of the method suggested by Ferragina and Manzini. Finally we design an extention to their method. Using this method we are not only able to search for any specific string in the text, but also for some more generalized descriptions of pieces of text. More precisely we can find all matches for a given regular expression. Using this we are able to find answers to the question like 'give all quoted piece of text'.

Bachelor Thesis in Mathematics and Computing Science
Author: Wouter Lueks
Supervisors: prof.dr. M. Aiello, prof.dr. W.H. Hesselink and prof.dr. J. Top
Date: August 2008

Institute of Mathematics and Computing Science
P.O. Box 407
9700 AK Groningen
The Netherlands

Contents

Chapter 1

Introduction

There are various methods of searching for a substring in a given text. Indexing is one of them. The indices we will consider here are reminiscent of the indices at the end of many textbooks. Given a certain keyword or subject we can use those paper-based indices to locate the relevant pages on which it occurs. With the full text indices we consider here, we improve upon these well established indices in the following two ways. First of all, searching is possible for any given substring, not just those predefined words that can be found in the index of a book. Secondly this method yields the exact position in the text, as opposed to just a page number. To make searching fast we would like the time the procedure takes to be independent of the length of the text.

The only way to achieve this independence on the length of the text is by precomputing some kind of data structure that facilitates fast searching for arbitrary substrings. This data structure is what we will call a *full-text index*. Computing this data structure may or may not be very efficient. We will, however, not concern ourselfs in this thesis with the efficiency of actually constructing these data structures.

Ferragina and Manzini manage to achieve the goals described in the first paragraph even though they added an additional requirement: the index should contain the entire text and this text should be compressed in order to keep the index, i.e. the set of data structures, small. They describe their results in their 2005 paper [8], which has been the basis for this thesis.

Not every text is easily compressible. For example, a set of completely random texts is likely to resist compression completely, while English prose is probably highly compressible. This would defeat the purpose of an upper bound on the space requirements since the worst case is always a noncompressible string. To have more realistic space bounds Ferragina and Manzini have opted to include a measure of compressibility in their space bounds. This measure, the empirical entropy, is discussed in Chapter 2.

Before doing the actual compression the Burrows-Wheeler transform is applied to facilitate the compression. We discuss this transformation in detail in Section 3.1. The Burrows-Wheeler transform compresses the text in such a way that we can easily count the number of occurences and even locate them within the original text. The general idea of these methods is outlined in Sections 3.3 and 3.5. The resulting algorithm for locating occurrences is, although promising, not yet optimal with respect to time complexity. By using a Lempel-Ziv parsing, Ferragina and Manzini are able to obtain the desired speedup in locating the occurrences. A detailed exposition about the Lempel-Ziv parsing and a sketch of the resulting algorithms can be found in Sections 3.6, 3.7 and 3.8.

Searching for an exact pattern is a powerfull tool, but it is not always the most useful solution. Regular expressions are often better suited to describe more complex queries. Usually regular expression engines will process the entire text linearly. In Chapter 4 we discuss two alternative implementations that depend on the full-text index we built earlier. This can cause a rather significant speedup in searching for matches of the regular expression.

1.1 Notation

Before we can continue we need to introduces some notation. Much of this is borrowed from Ferragina and Manzini. Throughout this thesis T will be the text we want to compress and create an index of. To be more precise we may write $T[1, n]$ instead, using a range notation, to emphasize the fact that the length of T is n. Furthermore we will always use one-based indexing, as implied by the notation. The ith character of T then is $T[i]$. In the following we often need to denote prefixes and suffixes of a string. We again use the range notation, $T[1, i]$ is the prefix of T of length i, while $T[i, n]$ denotes the suffix of T of length $n - i + 1$. We assume a constant alphabet Σ, so $T \in \Sigma^*$. We write $|A|$ for the number of elements in the set A, and $|w|$ for the length of the string w.

We use this notation to describe the problem of full-text indexing a bit more precise. Given the text T we would like to find all occurrences of a pattern $P[1, p]$ for arbitrary patterns P (rather than a limited set of index items). To do this we use an additional data structure for T, the index, which is precomputed.

Chapter 2

Entropy

Some pieces of data are more equal than others. Compare for example an essay written by a six year old on his favourite pet and a piece of writing by Shakespeare of equal length. The latter is most likely more complex in both the choice of vocabulary and sentence construction. This difference in complexity makes it very unlikely that we will ever be able to compress the piece by Shakespeare as good as the one by the six year old.

We would like to have a measure of the compressibility of a given piece of data. Shannon introduced the concept of *entropy* [15] that we can adapt to measure this compressibility in some useful sense, thereby allowing us to determine how 'compressible' pieces of data are.

Shannon deals primarily with data from a probabilistic point of view. A data source is considered to produce data according to some probabilistic rules, specific to that data source. He then derives the entropy as a measure for the 'amount' of information the source produces. Note the huge difference between information and data in this context. While a source producing just zeros will produce quite some data, the amount of information it produces will be negligable. Ferragina and Manzini use a notion of *empirical entropy* instead. This entropy is not defined for a data source that can produce infinitely many messages while following its rules, but for a specific piece of text. We can say it measures, in some sense, the information density of the text. We first consider the entropy in the context of Shannons paper. After deriving a formula for the entropy we will transform it to obtain the empirical entropy.

2.1 Uncertainty

We want to measure the amount of information produced by a data source. It is insightful to think of the amount of information in relation to how uncertain we are of the next symbol. Intuitively, if the uncertainty is high, the amount of information transmitted is high, while if it is low, the amount of information transmitted is low.

We will derive a formula for this uncertainty following roughly the same method as Shannon used. Suppose our data source can produce l different characters, with probabilities $p_1, \dots, p_l$. Of course the sum of these should be one. Let $H(p_1, \dots, p_l)$ be the measure of the uncertainty for this data source. In order to derive an actual formula for H we require the following properties:

1. The measure of uncertainty should be a mathematically well behaved function, so H should be continuous in all of the p_i's.
2. When we increase the number of sides on a fair dice, we are more uncertain about the outcome. The uncertainty, H, should mimic this behaviour. So if we put $p_i = 1/n$ for all i and increase n, then H should increase as well.
3. Not every decision has to be made in one magic step. Suppose we throw a fair coin. If we get heads the result is A, if we get tails we are not yet done. We throw the coin once more. If it is heads the result is B and otherwise it is C. So we have probabilities $\frac{1}{2}$, $\frac{1}{4}$ and $\frac{1}{4}$ for the events A, B and C respectively. The uncertainty can just be written as $H(\frac{1}{2}, \frac{1}{4}, \frac{1}{4})$, however there is a second plausible method of calculating the uncertainty. Looking at the uncertainties produced by throwing the coin we see at the first throw introduced some uncertainty, while the second throw adds more uncertainty in half of the cases. The uncertainty introduced by throwing a coin is $H(\frac{1}{2}, \frac{1}{2})$. The second uncertainty is only added half of the time, so the total uncertainty should equal $H(\frac{1}{2}, \frac{1}{2}) + \frac{1}{2}H(\frac{1}{2}, \frac{1}{2})$ as well. Since H should behave nicely these two expressions should be the same. If a choice can be decomposed into successive choices, then the entropy should be equal to the weighted sum of the individual choices.

Using just these requirements Shannon managed to prove the following theorem.

Theorem 2.1. *The only function H that satisfies the properties is of the form:*

$$H = -K \sum_{i=1}^{l} p_i \log p_i,$$

where K is a positive constant. H is called the entropy.

Note that K only determines the unit of measure.

Proof. This proof is almost identical to the one presented by Shannon in [15]. Introduce the function $A(n) = H(\frac{1}{n}, \dots, \frac{1}{n})$, with n a natural number, so we have n choices with equal probability. We will show $A(n)$ must be of the form $K \log n$. Take $n = st$ with s, t natural numbers. So we choose between n equally likely options. It is possible to split this choice in two steps. First choose between s equally likely options. Then choose again between t equally likely options. This gives a choice between $st = n$ equally likely options, as required. By property 3 the following equality holds:

$$A(st) = A(s) + A(t).$$

From this we get for any natural number p that

$$A(s^p) = pA(s). \tag{2.1}$$

Suppose s and t are fixed. For any natural number q there exists a natural number p such that

$$s^p \leq t^q \leq s^{p+1}. \tag{2.2}$$

Taking logarithms and dividing by $q \log s$ we get

$$\frac{p}{q} \leq \frac{\log t}{\log s} \leq \frac{p}{q} + \frac{1}{q}.$$

This can be made to hold for any q, so

$$\left| \frac{p}{q} - \frac{\log t}{\log s} \right| \leq \frac{1}{q}. \tag{2.3}$$

for $1/q$ arbitrarily small. Note that p/q is not constant in this expression. Its value may vary when an other bound is required. We will remove this term shortly.

Property 2 requires $A(n)$ to be monotonic in n, so it follows directly from equation (2.2) that $A(s^p) \leq A(t^q) \leq A(s^{p+1})$. Applying equality (2.1) three times gives

$$pA(s) \leq qA(t) \leq (p+1)A(s).$$

Dividing by $qA(s)$ gives

$$\frac{p}{q} \leq \frac{A(t)}{A(s)} \leq \frac{p}{q} + \frac{1}{q},$$

so we see

$$\left| \frac{p}{q} - \frac{A(t)}{A(s)} \right| \leq \frac{1}{q} \tag{2.4}$$

for $1/q$ arbitrarily small. Combining (2.3) and (2.4), and using the triangular inequality gives

$$\left| \frac{A(t)}{A(s)} - \frac{\log t}{\log s} \right| \leq \frac{2}{q}$$

for all s and t and $q \in \mathbb{N}$. Therefore $A(t)/\log t = A(s)/\log s$, the right-hand side is independent of t and thus is a constant, say K. Rewriting gives $A(n) = K \log n$. Note that K should be positive in order to satisfy property 2.

Consider now the case where the probabilities are not all equal. Assume that $p_1, \ldots, p_l$ are rational numbers. Then there exists natural numbers $m_1, \ldots, m_l$ and m such that $p_i = m_i/m$ for all $1 \leq i \leq l$ and $m = \sum m_i$. We will now use a trick to derive the entropy $H(p_1, \ldots, p_l)$. Consider exactly m equally likely choices. These m choices can be decomposed by choosing between l possibilities with probabilities $p_1, \ldots, p_l$ and then choosing between $m_1, \ldots, m_l$ equally likely choices. This gives

$$K \log m = H(p_1, \ldots, p_l) + K \sum_{i=1}^{l} p_i \log m_i.$$

α_i	a	b	c	d	e	f	g	h	i
p_i	.0575	.0128	.0263	.0285	.0913	.0173	.0133	.0313	0.0599
α_i	j	k	l	m	n	o	p	q	r
p_i	.0006	.0084	.0335	.0235	.0596	.0689	.0192	.008	.0508
α_i	s	t	u	v	w	x	y	z	-
p_i	.0567	.0706	.0334	.0069	.0119	.0073	.0164	.0007	.1928

Table 2.1: A table with the probabilities of the characters a through z and the space in an English text.

So

$$\begin{aligned} H(p_1,\ldots,p_l) &= K\Big(\log m - \sum_i p_i \log m_i\Big) \\ &= K\Big(\sum_i p_i \log m - \sum_i p_i \log m_i\Big) \\ &= K\sum_i p_i \log \frac{m}{m_i} = -K\sum_i p_i \log p_i. \end{aligned}$$

This almost concludes the proof for the general case. When not all the p_i's are rational numbers, they can be approximated arbitrarily well by rational numbers, since $\mathbb{Q}$ is dense in $\mathbb{R}$. Because H is continuous in the p_i this does still yield the same result. $\square$

Once again, note that the constant K only determines a choice of the unit of measure. Assume, for the remainder of this chapter, that the logarithm has base two and that $K = 1$, i.e. the unit of measure is in bits.

The entropy as defined above has a second, much more interesting, interpretation. Suppose we have a data source which emits characters from $\Sigma = \{\alpha_1, \ldots, \alpha_l\}$ with probabilities $p_1, \ldots, p_l$. Then the entropy can be interpreted as the minimal average number of bits per character, provided we always use the same code for a given character.

Example 2.2. The frequencies of the characters a,b,…,z and the space in an English text are displayed in Table 2.1, see [12]. Calculating the entropy gives a value of about 4.1 bits per character on average. Note that this value is a bit lower than it usually will be since we did not take punctuation and capital letters into account.

Since Shannon a more contemporary notation has been devised. Let (X, A_X, P_X) be an ensemble, that is A_X is a finite set of possible values for a random variable X and P_X contains the corresponding probabilities. We say that $H(X)$ is the entropy of this ensemble.

2.2 Conditional Entropy

Up till now we have thought of data sources as devices that produce each subsequent character independent of the previous ones. Remember our intent was to say something about the compressibility of text. Almost always there exists some relation between a sequence of previous characters and the next one. For example when reading ‘entrop’ it is highly likely that the next character will be a ‘y’. Another slightly less likely possibility is an ‘i’, if the word actually were to be ‘entropies’. In English texts the ‘e’ is very common, it has a high chance of appearing at a random location in the text. Yet, the chance of an ‘e’ following ‘entrop’ is negligible. So including information about previous characters tends to reduce the number of possibilities. In a sense this seems to reduce the entropy at that specific point as well, since the number of plausible options is a lot smaller. Unfortunately entropy is defined on a data source, not for a specific point in its output. We will solve this by introducing the conditional entropy, once again basing our exposition on Shannon’s work.

The previous paragraph showed that it is plausible that adding information reduces the entropy. As before let (X, A_X, P_X) be an ensemble of which we want to measure the entropy. Now let us suppose we know another ensemble (Y, A_Y, P_Y), that is for each event we know the value of the random variable Y. How does knowing this change the entropy of X. We saw in the previous paragraph that having additional information, in this case the value of Y, might lower the uncertainty of X. It is easy to give the uncertainty of X for each specific value of Y, but this is not nice. Now the uncertainty for X depends on the value of Y. This does not help since we only know the distribution of Y, not its values. A natural solution would be to define the entropy of X given knowledge about Y as the weighted average of the entropy of X given that $Y = y$, $H(X|Y = y)$, over all values y of Y. So the conditional entropy of X given Y,

$$\begin{aligned} H(X,Y) &= -\sum_{y\in A_Y} P(y) \sum_{x\in A_X} P(x|y)\log P(x|y) \\ &= -\sum_{x\in A_X, y\in A_Y} P(x,y)\log P(x|y). \end{aligned} \tag{2.5}$$

Intuitively the entropy of X should not increase when taking into account the knowledge about Y, that is $H(X|Y) \leq H(X)$. We will prove that this is indeed the case. First we need to introduce the concept of *joint entropy*. Consider again the ensembles X and Y as above. Consider now a joint event (x, y), that is $X = x$ and $Y = y$ simultaneously. The joint entropy of X and Y is

$$H(X,Y) = -\sum_{x\in A_X, y\in A_Y} P(x,y)\log P(x,y). \tag{2.6}$$

In order to prove that $H(X|Y) \leq H(X)$ we need some additional equations. The following lemma helps us to derive the first of them, see for example [12].

Lemma 2.3 (Jensen’s Inequality). *Let f be a concave, continuous function over (a, b). That is for all $x_1, x_2 \in (a, b)$ and $0 \leq \lambda \leq 1$,*

$$f(\lambda x_1 + (1-\lambda)x_2) \geq \lambda f(x_1) + (1-\lambda) f(x_2). \tag{2.7}$$

Let $\alpha_i > 0$ for $1 \leq i \leq n$ and $\sum_{i=1}^{n} \alpha_i = 1$. Then

$$\sum_{i=1}^{n} \alpha_i f(x_i) \leq f(\sum_{i=1}^{n} \alpha_i x_i), \tag{2.8}$$

provided that $\sum_{i=1}^{n} \beta_i x_i \in (a,b)$ *for all* $0 \le \beta_i \le \alpha_i$.

Proof. We will use a limited induction argument. Let $A_k = \sum_{i=k}^{n} \alpha_i$, so as a result $A_k - \alpha_k = A_{k+1}$. We will show that

$$f(\sum_{i=1}^{n} \alpha_i x_i) \ge \sum_{i=1}^{n} \alpha_i f(x_i). \tag{2.9}$$

Simple expansion of the left-hand side gives, using $A_1 = 1$, that

$$f(\sum_{i=1}^{n} \alpha_i x_i) = A_1 f(\frac{\alpha_1}{A_1} x_1 + \frac{A_2}{A_1} \sum_{i=2}^{n} \frac{\alpha_i}{A_2} x_i). \tag{2.10}$$

A slightly generalized version of the right-hand side yields, using the concavity of f, that for $k < n$

$$\begin{aligned} & A_k f\left(\frac{\alpha_k}{A_k} x_k + \frac{A_{k+1}}{A_k} \sum_{i=k+1}^{n} \frac{\alpha_i}{A_{k+1}} x_i\right) \ge \\ & \alpha_k f(x_k) + A_{k+1} f\left(\sum_{i=k+1}^{n} \frac{\alpha_i}{A_{k+1}} x_i\right) = \\ & \alpha_k f(x_k) + A_{k+1} f\left(\frac{\alpha_{k+1}}{A_{k+1}} x_{k+1} + \frac{A_{k+2}}{A_{k+1}} \sum_{i=k+2}^{n} \frac{\alpha_i}{A_{k+2}} x_i\right). \end{aligned} \tag{2.11}$$

Applying this $n-1$ times proves the theorem:

$$\begin{aligned} f\left(\sum_{i=1}^{n} \alpha_i x_i\right) &\ge \alpha_i f(x_1) + \cdots + \alpha_{n-1} f(x_{n-1}) + A_n f(\frac{\alpha_n}{A_n} x_n + 0) \\ &= \alpha_1 f(x_1) + \cdots + \alpha_n f(x_n) = \sum_{i=1}^{n} \alpha_i f(x_i). \end{aligned}$$ □

Notice how we require all the α_i to be positive. In a few moments we will apply Jensen's Inequality to the entropy. Then $\alpha_i = p_i$ and $x_i = p_i$ and $f(x) = \log(x)$. In this case however some of the α_i can be zero. A first question to ask is whether $0 \log 0$ is a well-defined expression. To see that it is actually equal to zero apply l'Hospital to $\lim_{x \to 0} x \log x = \lim_{x \to 0} \log x/(1/x)$. As a result terms with $\alpha_i = 0$ do neither contribute to the left-hand side nor to the right-hand side, so we can safely apply Jensen's Inequality in case of the entropy.

We will show that the joint entropy $H(X,Y)$ is never greater than the entropy of X plus the entropy of Y, that is $H(X,Y) \le H(X) + H(Y)$. In other words, the uncertainty of the joint event (x,y) is never greater than the sum of the uncertainty of its parts. So we want

$$\begin{aligned} H(X,Y) = -\sum_{x,y} P(x,y) \log P(x,y) &\le H(X) + H(Y) \\ &= -\sum_{x} P(x) \log P(x) - \sum_{y} P(y) \log P(y) \\ &= -\sum_{x,y} P(x,y) \Big[\log \sum_{y'} P(x,y') + \log \sum_{x'} P(x',y) \Big] \end{aligned} \tag{2.12}$$

or in other words

$$0 \le -\sum_{x,y} P(x,y)\Big[\log \sum_{y'} P(x,y') + \log \sum_{x'} P(x',y) - \log P(x,y)\Big]. \tag{2.13}$$

Simple manipulation shows that the right-hand side of (2.13) is equal to

$$-\sum_{x,y} P(x,y) \frac{\sum_{x',y'} P(x,y')P(x',y)}{P(x,y)}. \tag{2.14}$$

We may apply Jensen's Inequality, since the logarithm is a concave function on $[0,\infty)$, and all other requirements are met. Doing so gives:

$$-\sum_{x,y} P(x,y) \frac{\sum_{x',y'} P(x,y')P(x',y)}{P(x,y)} \ge -\log \sum_{x,y} \sum_{x',y'} P(x,y')P(x',y) \ge 0$$

where the latter inequality follows from the fact that the argument of the logarithm sums to 1. This proves that

$$H(X,Y) \ge H(X) + H(Y). \tag{2.15}$$

Let us now return to the conditional probability $H(X|Y)$. Expanding it gives

$$\begin{aligned} H(X|Y) &= -\sum_{x,y} P(x,y) \log P(x|y) \\ &= -\sum_{x,y} P(x,y) \log \frac{P(x,y)}{P(y)} \\ &= -\sum_{x,y} P(x,y) \Big[\log P(x,y) - \log P(y)\Big] \\ &= H(X,Y) - H(Y) \end{aligned} \tag{2.16}$$

Notice how this gives $H(X,Y) = H(X|Y) + H(Y)$, in other words the joint entropy of X and Y equals the conditional entropy of X given Y and the entropy of Y. Combining this with (2.15) proves that

$$H(X|Y) \le H(X). \tag{2.17}$$

So indeed the given knowledge about Y does not increase the entropy of X.

2.3 Empirical Entropy

Until now we have only considered the entropy in a probabilistic context. Remember once more that our original goal was to give a measure of the compressibility of a given text, and not the compressibility of a probabilistic source producing these texts. This change of perspective yields a slightly different concept: the *empirical entropy.* As implied by the name the empirical entropy uses empirical measurements as opposed to probabilistic assumptions. These empirical measures will replace the probabilities.

	bits/character	*bits*
original	8.0	1188328.0
H_0	4.512741	670326.91
H_1	3.501531	520117.47
H_2	2.510313	372879.33
H_3	1.794791	266594.67
H_4	1.320564	196152.63
H_5	0.977257	145157.81
H_6	0.718981	106793.87
H_7	0.511979	76046.29
H_8	0.357624	53118.95
H_9	0.251111	37297.97
H_{10}	0.177618	26381.72

Table 2.2: Some entropies for 'Alice's Adventures in Wonderland' by Lewis Carrol.

Create for a given text T an ensemble (X_T, Σ, P_T), where with a slight abuse of notation the possible values of X_T consist of the alphabet $\Sigma = \{\alpha_1, \ldots, \alpha_l\}$. The probabilities P_T are the corresponding probabilities of choosing α_i aselect from the text. Let n_i be the number of occurrences of α_i in T. Then $P_T = \{n_1/n, n_2/n, \ldots, n_l/n\}$. The result is a valid ensemble corresponding to the given text. We thus define the empirical entropy as

$$H_0(T) = H(X_T) = -\sum_{i=1}^{l} \frac{n_i}{n} \log \frac{n_i}{n}. \tag{2.18}$$

The significance of the subscripted zero will become apparent shortly. Also in the empirical case we want to reap the benefit of a conditional entropy. In this case the conditional part is a short string of characters that occurred before the current one. Let $n_{w\alpha_i}$ be the number of occurrences of $w\alpha_i$ in T, where $w \in \Sigma^*$ and $\alpha_i \in \Sigma$. That is the number of occurrences of the string w followed by the character α_i. Then the total number of occurrences of w that precede another character is $n_w = \sum_{i=1}^{l} n_{w\alpha_i}$. Suppose we include a history of k characters, then the corresponding conditional entropy is called the k-th order entropy and defined by

$$H_k(T) = -\frac{1}{n} \sum_{w \in \Sigma^k} n_w \left[\sum_{i=1}^{l} \frac{n_{w\alpha_i}}{n_w} \log \frac{n_{w\alpha_i}}{n_w} \right]. \tag{2.19}$$

As a result we can interpret the conditional entropy $H_k(T)$ as the average number of bits required to encode the next character, given the previous k characters. It would seem that the information can be encoded in only $nH_k(T)$ bits, for any k. This is however only partially true since we somehow need to tell the receiver how to decode the message. Describing these codewords takes an additional $\Omega(|\Sigma|^k)$ bits. It is however an established practise in Information Theory to ignore these additional costs [8].

Just as for the conditional entropy we have that adding more information does not increase the entropy, that is $H_{k+1} \leq H_k$.

Example 2.4. Table 2.2 shows the k-th order entropy for 'Alice's Adventures in Wonderland' by Lewis Carroll. In addition the total size is shown. Notice how including a larger history lowers the overall entropy. Remember though that we did not take into account the space required for the code words.

Chapter 3

Compressed Text Indices

This chapter is concerned with how we can create a full text index for a text while at the same time compressing it. The Burrows-Wheeler transform is used to make the text better compressible, see Section 3.1. The transformed text is then compressed in three steps, see Section 3.2.

To prevent this chapter from flooding the reader with information, the discussion of searching for a substring has been split up, just like Ferragina and Manzini did. First we will see how the number of occurrences can be counted, see Section 3.3. Then we optimize this solution by means of the four Russians trick in Section 3.4. Next we explain in Section 3.5 how the former solution for counting can be used to actually locate a given substring. This solution is not yet optimal. A Lempel-Ziv parsing is used to improve the time bounds. The theory behind this parsing is discussed in Section 3.6. How this parsing can be used to improve the time bounds will be explained in Sections 3.7 and 3.8.

3.1 Burrows-Wheeler Transform

The compression algorithm described in Ferragina's and Manzini's paper [8] heavily depends on the Burrows-Wheeler transform (from now on BWT). This transformation was first described in a paper by Burrows and Wheeler in 1994 [5]. It has two very important properties: firstly it makes compression easier and secondly it is easily reversible. Ferragina and Manzini have made some slight modifications to the transformation originally described by Burrows and Wheeler. It is this version that we will now examine more closely.

The BWT can be described by the following three steps. We begin by appending a special character hash (#) to T. This character is alphabetically smaller than any other character in the alphabet. Secondly we generate all cyclic permutations of $T\#$, see matrix M in Figure 3.1. Finally we sort these permutations in lexicographic order, see matrix M_T in Figure 3.1. We will often think of the result as a matrix M_T where the rows are made up by these sorted permutations. Note that every column of M_T is just a permutation of $T\#$. The result L of the BWT is the last column of matrix M_T.

1	`engineering#`		`#engineering`
2	`ngineering#e`		`eering#engin`
3	`gineering#en`		`engineering#`
4	`ineering#eng`		`ering#engine`
5	`neering#engi`		`g#engineerin`
6	`eering#engin`	$\longrightarrow$	`gineering#en`
7	`ering#engine`		`ineering#eng`
8	`ring#enginee`		`ing#engineer`
9	`ing#engineer`		`neering#engi`
10	`ng#engineeri`		`ng#engineeri`
11	`g#engineerin`		`ngineering#e`
12	`#engineering`		`ring#enginee`
	M		M_T

Figure 3.1: The Burrows-Wheeler transform of $T =$ `engineering` is obtained by creating all cyclic permutations of $T\#$ and then sorting them in lexicographic order. The result is $L =$ `gn#enngriiee`.

To understand why L is more easily compressed we first need to look at the effect of sorting in the presence of the special character $\#$. We will see that the BWT actually sorts the suffixes of T. Note that due to the cyclic permutations we have exactly one hash in every column of M_T. In addition the hash is lexicographically smaller than all other characters. As a result, everything to the right of a hash has no influence on the order of the rows in M_T. The characters to the left of a hash precisely form a suffix of T. So we have indeed sorted all suffixes of T.

Every last character in the ith row immediately precedes the first character of this row in the original text T. For example see Figure 3.1, the fifth row ends with a 'n', while it begins with 'g'. Indeed the 'n' immediately precedes the 'g' in text T, since 'engineering' ends with 'ng'. Let us now consider an example given by Burrows and Wheeler: suppose we apply the BWT to this thesis and imagine we look at the rows of M_T, that start with 'he␣' (we have to imagine this since the BWT only yields L, not M_T). Remember these will be consecutive rows, due to sorting. It will be very likely that these rows end with a 't', since the word 'the' occurs very often in this text, other possibilities include 'T', 'c', 's' and 'S'. Other characters in the last column are highly unlikely. So locally in L we only encounter very few characters with high probability. This makes the transformed text very easy to compress. One of the ways of seeing this is to consider the conditional entropy. Given a couple of previous characters in L, very few possible following characters remain. So the entropy will be very low. Therefore the text is highly compressible.

Now that we have clarified the first important property of the BWT it is time to spend some effort on the second property: reversibility. We will show that the BWT L allows us to 'step back' along T. Given the position of $T[k]$ in L we can find the preceding character in T, $T[k-1]$, provided it exists, i.e. $k > 1$. By construction we know that the last character of T is the first character of L. This allows us to reconstruct T by backstepping, starting with the first character of L.

We step back along T in the following way. Denote the first column of M_T by F. We will determine a formula to find, for a given character in L, the exact same character in F. Here the exact same character means not only that it is the same character, but also that it is at the same position in T. Suppose now that $L[i]$ is at position j in F, then we know that $L[j]$ precedes $L[i]$ in T, since $L[j]$ precedes $F[j]$ in T. So knowing where a character in L is in F allows us to step back along T. Suppose $L[i] = c$, then we would like to know which of the c's in F corresponds to this one. This is easy if we can show that the order of the rows in M_T starting with a c is the same as the order of the rows of M_T ending with the corresponding c's. Since then the pth c in L will just correspond with the pth c in F.

We will show this is true by examining a new imaginary matrix $\hat{M_T}$. This proof is based on the proof of Burrows and Wheeler [5]. We obtain this matrix by cyclic shifting every row of M_T once to the right, i.e. remove L from M_T and add it again on the left. Note that $\hat{M_T}$ is still sorted when we ignore the first column. Let us now return to the objects of our interest: the rows of M_T ending with a c. In $\hat{M_T}$ these will be the rows starting with a c. Now comes the crux: all these rows in $\hat{M_T}$ start with the same character and are therefore lexicographically sorted. Compare these two lists of rows. The rows in M_T beginning with a c are sorted lexicographically and they are cyclic permutations of $T\#$. We just saw that the rows of $\hat{M_T}$ starting with a c are also sorted lexicographically. By construction they are also cyclic permutations of $T\#$. So both lists of rows contain the cyclic permutations of $T\#$ beginning with a c and the lists are sorted. The only possible conclusion is that both sets are exactly the same, including order. Since we did nothing to $\hat{M_T}$ except for a simple cyclic shift we have shown that the order of the corresponding c's in both F and L is the same.

In order to formalize this property and some subsequent results we need to introduce some additional notation. This notation is the same as the one used by Ferragina and Manzini in [8].

- Let $C[c]$ be the number of occurring text characters that are alphabetically smaller than c. So $C[\cdot]$ denotes an array of length $|\Sigma| + 1$. In our example in figure 3.1 $C[\#] = 0$, while $C[\mathrm{n}] = 8$, since the hash, the three e's, the two g's and the two i's are alphabetically smaller than 'n'.

- $Occ(c, q)$ gives the number of occurences of c in $L[1, q]$. So in our example $Occ(n, 5) = 2$.

Lemma 3.1. *The Last-to-First column mapping LF that assigns to each $L[i]$ its corresponding location $LF(i)$ in F is given by $C[L[i]] + Occ(L[i], i)$.*

Proof. This lemma follows directly from the preceding text. $C[L[i]]$ is the row in M_T preceding the ones starting with the sought after character $L[i]$. The second term, $Occ(L[i], i)$ is the number of $L[i]$'s in L upto this one. Since order is preserved we find the required index in F by adding them. □

3.2 Compressing the BWT

We have seen in the previous section that the BWT often produces long runs of identical characters or at least few different characters. See for a somewhat lengthier example Figure 3.2. Ferragina and Manzini proposed the following compression algorithm. The first step involves converting all characters in L to small numbers. Imagine we process L character by character. After each character we output a counter: the number of distinct characters since the previous occurrence of that character. We bootstrap this process by assuming the alphabet is processed first in alphabetic order, since then everything is well-defined.

"Oh, grandmother, what big ears you have!" "All the better to hear you with." "Oh, grandmother, what big eyes you have!" "All the better to see you with." "Oh, grandmother, what big hands you have!" "All the better to grab you with!" "Oh, grandmother, what a horribly big mouth you have!"	␣””””””””teeetttyggo,,,,guuuuoagolllrrr ,,,,uuuhsssbreeeeeh!!!!!..␣␣␣␣#␣␣hhhhrr rrhh””””””””␣rrrrrheehhhhhhhha␣␣␣␣␣␣ ␣innnnnhhhevvvvh␣sttthhhhybbb␣iiii␣ ␣␣␣␣ttOOOOtt␣wwww␣␣␣␣ttt␣tttt␣rbb bbwwwlllAAAbdddd␣aaaaattthmmm myyyyyyymeeeaeeeeggggggroadre␣aaaa tttuiii␣␣␣oooo␣␣␣eeeoooooooooaaaa␣␣␣␣ ␣␣␣le␣␣␣␣␣␣␣

Figure 3.2: The BWT applied to an excerpt from Grimm's Little Red Riding Hood, the spaces have been visualized by '␣' since they are in the alphabet as well. Notice the long runs of identical characters.

Formally this process is called a move-to-front (MTF) encoding [4]. It uses a list with all characters of the alphabet, ordered by recency of occurrence. This list is called the MTF-list. Since we supposed that the alphabet was processed first, initially this list contains all characters in reverse alphabetical order. For every new character we output its position in the list. The character then is moved to the front of the list and the next character is processed. The resulting string is L^{mtf}, which is a list of, hopefully, small numbers.

Applying the move-to-front encoding on the transformed text L has significant consequences. Every sequence of identical characters is converted to a sequence of zeroes. In addition if L contains only a few characters locally these will be converted to small numbers in L^{mtf} since it is likely we have seen them before.

Ferragina and Manzini subsequently get rid of these sequences of zeroes by applying a run-length encoder, see for example [14]. They replace a sequence of zeroes by their length in binary. Finally both the normal numbers as well as the binary numbers representing the runs of zeroes are converted into zeroes and ones using a variable-length prefix code, see for example [14]. These three steps combined will be referrred to as `BW_RLX`. For a more technicial exposition see Ferragina's and Manzini's paper [8].

It is shown in [8] that the compression rate can be bounded by the kth order empirical entropy $H_k(T)$ of T,

$$|\texttt{BW\_RLX}(T)| \leq 5nH_k(T) + O(\log n) \tag{3.1}$$

for any $k \geq 0$. Remember that when using the kth order entropy one should take the additional cost of describing the codewords into account. In (3.1) this term is hidden inside the big-O notation. We thus assume that $|\Sigma|$ is constant with respect to n.

3.3 Counting Occurrences

Ferragina and Manzini identify two phases in the searching process. The first phase counts the number of occurences, the second phase locates these occurrences. This is useful because "it simplifies the presentation and shows that the locating phase builds on top of the counting phase."

In this section we will concern ourselves with the first phase: counting the occurrences of a pattern $P[1,p]$ in text T. This number equals the number of rows in M_T that are prefixed by $P[1,p]$. Since the suffixes are sorted, these rows will be consecutive. Let the first of these rows have index `First` and the last of these index `Last`. Then the number of occurences is ($\texttt{Last} - \texttt{First} + 1$).

For the duration of the section we shall forget about the compression described in the previous section. We will reintroduce this aspect in the next section.

```
1  **Algorithm** backward_search($P[1,p]$)
2  $i \leftarrow p, c \leftarrow P[p], \mathsf{First} \leftarrow C[c] + 1, \mathsf{Last} \leftarrow C[c+1]$;
3  **while** (($\mathsf{First} \leq \mathsf{Last}$) **and** ($i \geq 2$)) **do**
4    $c \leftarrow P[i-1]$;
5    $\mathsf{First} \leftarrow C[c] + \mathsf{Occ}(c, \mathsf{First} - 1) + 1$;
6    $\mathsf{Last} \leftarrow C[c] + \mathsf{Occ}(c, \mathsf{Last})$;
7    $i \leftarrow i - 1$;
8  **if** ($\mathsf{Last} < \mathsf{First}$) **then return** "No rows prefixed by$P[1,p]$" **else return** ($\mathsf{First}, \mathsf{Last}$).
```

Listing 3.1: Algorithm backward_search locates the rows in M_T that are prefixed by pattern P in p steps.

To find these rows prefixed by $P[1,p]$ we use Lemma 3.1. First we find the rows starting with just the last character of P, $P[p]$, this we can find using just C. Then we add characters step by step from the back of P until we have processed all of P. See Listing 3.1 for the pseudocode of this algorithm. We only give an intuitive account of the workings of this algorithm here, see Ferragina and Manzini [8] for a formal proof. Suppose we have determined that the rows $\texttt{First}, \ldots, \texttt{Last}$ start with $P[i,p]$, for some i. We will now show how to find the rows prefixed by $P[i-1,p]$. Let c be $P[i-1]$. Some of the selected rows may end with a c, these rows are the interesting ones. Remember that $LF(i) = C[L[i]] + Occ(L[i], i)$ is the position of the ith character of L in the first column F, since it was the $Occ(L[i], i)$-th character $L[i]$ after $C[L[i]]$. We can adapt this slightly to find the first row starting with $cP[i,p] = P[i-1,p]$. The number of c's before row `First` in L is $Occ(c, \texttt{First} - 1)$, so our c, if it exists, should be the first c after that, so it is at $C[c] + Occ(c, \texttt{First} - 1) + 1$, see also line 5. Similarly the last row prefixed by $cP[i,p]$ is $C[c] + Occ(c, \texttt{Last})$. Note that if c is not in $L[\texttt{First}, \texttt{Last}]$ then $\texttt{Last} < \texttt{First}$ so the guard is no longer true and the algorithm will correctly report that no matches could be found.

Until now we have not yet talked about how to calculate $Occ(c, q)$. Examining L each time we need to know $Occ(c, q)$ seems a bit expensive. Assume for the moment we build a huge array `OCC` such that $\texttt{OCC}[c][q] = Occ(c, q)$, then `backward_search` runs in $O(p)$ time. Do note that `OCC` requires $O(|\Sigma| n \log n) = O(n \log n)$ bits.

3.4 A Four Russians Trick

Using the array `OCC` is useful, but its size completely defeats the purpose of any compression applied to L, since L is only $O(n)$ bits. We will illustrate here a solution produced by Ferragina and Manzini based on the Four Russians trick. The trick is named after four Russians [13]: Arlazarov, Dinic, Kronrod and Faradzev, since they used it first in their paper [2]. This trick is used to reduce the amount of storage required for calculating Occ efficiently. The trick itself resembles a divide and conquer approach. This time however the trick suggests to stop when the problem is 'small enough'. The outcome is then determined by looking it up in a precomputed table.

We start by partitioning L in so-called buckets of length $l = \Theta(\log n)$. So we have n/l buckets $BL_i = L[(i-1)l + 1, il], i = 1, \ldots, n/l$ of length l. For simplicity we will ignore a lot of details. For a more precise discussion we refer to Ferragina's and Manzini's paper. The partitioning induces a partitioning of L^{mtf} into buckets $BL_1^{mtf}, \ldots, BL_{n/l}^{mtf}$ as well. And finally it also induces a partition on the compressed text $Z = \texttt{BW\_RLX}(T)$ into buckets $BZ_1, \ldots, BZ_{n/l}$, provided some assumptions are met. Note that the latter buckets do not have equal size.

Recall that $Occ(c, q)$ is the number of occurrences of c in $L[1, q]$. We apply the Four Russians trick. Split $L[1, q]$ in three substrings (of which the last ones might be empty), see also figure 3.3. The first part is the longest prefix of $L[1, q]$ that has length a multiple of l^2. The second part is the longest prefix of the remainder that has length a multiple of l. The third part is the remainder of $L[1, q]$. Note that by construction the last part will be the prefix of a single bucket. The idea of determining $Occ(c, q)$ is now as follows. First get the number of occurrences of c in substring one. Then get the number of occurrences in substring two. We are left with a substring of L with length less than l. The number of occurrences of c in this last part can only be obtained by examining the correct compressed bucket. This is where the precomputed table of solutions comes in. We just look up the required information in a table indexed by the bucket, the character c and the length of the remaining substring. Adding these three numbers gives $Occ(c, q)$.

The previous paragraph already alluded to two data structures for counting the occurrences in the first and second substring, as well as the lookup table. Note that all of these data structures can be precomputed. We brushed over two important elements. First of all we need to find the last bucket in Z containing a piece of $L[1, q]$, that is we need to know where it begins and ends. In order to get this information we store for every substring of type one as well as for every substring of type two its compressed size in bits. Note that in the latter case we only need to store its length up to the first multiple of l^2. These two data

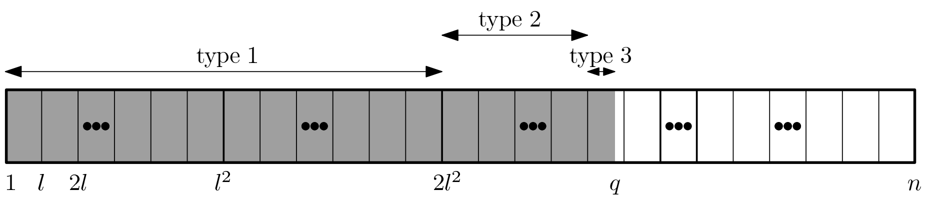


Figure 3.3: A possible splitting of $L[1,q]$ into the three substrings. Type 1 has length a multiple of l^2, type 2 has a length a multiple of l and finally type 3 has length less than l.

structures may be precomputed as well. Now we are able to locate the bucket and we come to the second important element: we need the MTF-list before we can say anything about the content of the bucket. So we also store the content of the MTF-list for every bucket and index the lookup table with this list as well. Table 3.1 summarizes the various data structures and their sizes.

Substrings of length a multiple of l^2	
number of occurrences	$O(\|\Sigma\|n/l^2 \log n) = O(n/\log n)$
compressed bucket sizes	$O(n/l^2 \log n) = O(n/\log n)$
Substrings of length a multiple of l	
number of occurrences	$O(\|\Sigma\|n/l \log(l^2)) = O(n/(\log n)\log\log n)$
compressed bucket sizes	$O(n/l \log(l^2)) = O(n/\log n)$
Substrings of length less than l	
MTF-lists	$O(\|\Sigma\|n/l \log\|\Sigma\|) = O(n/\log n)$
lookup table	$O(\|\Sigma\|l2^{l'}2^{\|\Sigma\|\log\|\Sigma\|})$

Table 3.1: An overview of the various additional data structures in Opp(T) and the corresponding sizes.

It is proven in [8] that this set of data structures allows us to compute $Occ(c,q)$ in $O(1)$ time using $|Z| + O(n(\log\log n)/(\log n))$ bits of storage. Combining this fact with bound 3.1 we get the following major result from Ferragina's and Manzini's paper.

Theorem 3.2. *Using procedure* backward_search *we can compute the number of occurrences of a pattern $P[1,p]$ in $T[1,n]$ in $O(p)$ time. It needs at most $5nH_k(T) + O(n\frac{\log\log n}{\log n})$ bits, for any $k \geq 0$, to store the precomputed data structures.*

We will need this set of data structures a couple of times more, so denote this set by `Opp`(T).

3.5 Locating the Patterns

Recall how the result of the backward_search algorithm is a set of rows [First,Last in M_T that are prefixed by the pattern P. We are however interested in the location of these (`Last` $-$ `First` $+ 1$) occurrences in the original text T. Note that every row starts with a suffix of T, we are interested in the position of these suffixes in T. However it is not completely trivial to find the position of these suffixes since their location in M_T is determined by sorting cyclic permutations of T. Rebuilding this mapping every time costs $O(n^2 \log n)$ time because of sorting n cyclic permutations (comparing two permutations takes $O(n)$ time as well). This is not acceptable from a performance point of view. On the other hand, storing the mapping in a table is not acceptable from a storage point of view since that tabel takes $O(n \log n)$ bits.

Ferragina and Manzini propose a different solution. Remember we can step along T by means of Lemma 3.1. They propose to 'logically mark' a limited set of rows with their position in T. Now when a position of a row is requested there are two possibilities: either we know its position because the row is marked, or we do not. In the latter case we can always step to the row that has a prefix that starts one position earlier because of Lemma 3.1 and try again until we find a marked row.

Let $\mathsf{Pos}(i)$ be the position of the suffix of T starting in row i of M_T in the original text T. In our example in Figure 3.1 we have $\mathsf{Pos}(4) = 7$ since suffix `ering` starts at position 7 in T. To step back one position from row i we need to find j such that $\mathsf{Pos}(j) = \mathsf{Pos}(i) - 1$. Or in other words: we need to find j such that $T[\mathsf{Pos}(i) - 1, n] = T[\mathsf{Pos}(j), n]$. Using Lemma 3.1 one may conclude this is as simple as writing $j = C[L[i]] + Occ(L[i], i)$, however we do not yet know what $L[i]$ is. Fortunately we can find this out by calculating the difference between $Occ(c, i)$ and $Occ(c, i - 1)$ for all characters c. The result is algorithm backward_step; see Listing 3.2. For a formal proof of the correctness see [8].

```
Algorithm backward_step($i$)
Compute $L[i]$ comparing $\mathsf{Occ}(c, i)$ with $\mathsf{Occ}(c, i-1)$ for every $c \in \Sigma \cup \#$.
if ($L[i] = \#$) then return "Pos(i) = 1";
else return $C[L[i]] + \mathsf{Occ}(L[i], i)$;
```

```
Algorithm get_position($i$)
$i' \leftarrow i, t \leftarrow 0$;
while row $i'$ is not marked do
  $i' \leftarrow$ backward_step($i'$);
  $t \leftarrow t + 1$;
return $\mathsf{Pos}(i') + 1$;
```

Listing 3.2: Algorithms backward_step and get_position.

Now on to the marking of the rows. Here we have a tension between two of our requirements. If we mark more rows queries will be faster, however marking more rows requires more space. We solve this by introducing a parameter ϵ. Let the distance between two markers be $\eta = \lceil \log^{1+\epsilon} n \rceil$. Mark every row r_j such that $\mathsf{Pos}(r_j) = 1 + j\eta$ for $j = 0, 1, \ldots, \lfloor n/\eta \rfloor$. So within η steps we always find a marker. The algorithm to do this is get_position, see Listing 3.2. Since backward_step takes constant time, every iteration of get_position takes constant time as well. So finding the position of one occurrence takes $O(\log^{1+\epsilon} n)$ time. Finding occ occurrences of P in text T then takes $O(occ \log^{1+\epsilon} n)$ time.

We have assumed that checking if a row is marked takes constant time. This can be done using a Packet B-tree, see [8] for the details. This data structure uses $O(n/\log^{\epsilon} n)$ bits. The results are formalized in the following theorem.

Theorem 3.3. *For any text $T[1,n]$ we can build a compressed index such that all the occ occurrences of $P[1,p]$ in T can be retrieved in $O(p + occ \log^{1+\epsilon} n)$ time and at most $5nH_k(T) + \frac{n}{\log^{\epsilon} n}$ bits space, for any $k \geq 0$.*

3.6 LZ78 Parsing

The previous result in Theorem 3.3 showed a time complexity of $O(p+occ \log^{1+\epsilon} n)$. In the last part of their paper Ferragina and Manzini show that this bound can in fact be lowered to the theoretical lower bound of $O(p + occ)$. To do this they adapted the manner in which rows are marked and combined this in a new way with the previous results. The improvement in marking is brought about by considering a compression technique developed by Lempel and Ziv.

In 1978 Lempel and Ziv described in their paper [17] an adaptive dictionary encoder for compressing text. A dictionary encoder uses a dictionary to encode a string [3]. First the text is split into words that are in the dictionary. This is called parsing (note that in general there are a lot of ways to split a text into dictionary words). Then each word is replaced by a reference to this word in the dictionary. However dictionary encoders with a fixed dictionary do not perform very well. The version by Lempel and Ziv is adaptive and it therefore usually performs better.

Ferragina and Manzini only used the parsing part of the dictionary encoder. The parsing method of Lempel and Ziv will henceforth be known as the LZ78 parsing. The most important aspect of the LZ78 parsing is that it splits the input text T in a sequence of d words $T_1, T_2, \ldots, T_d$ such that $T_1 T_2 \ldots T_d = T$. Each of these words (except possibly the last word) is bound by the following constraint: it is either

1. a single new character that is not one of the previous words, or
2. it is an existing word followed by one additional character.

To make this parsing unique we require that we always make the next word as long as possible. By construction all words are unique, with the possible exeption of the last word because it is not always possible to add sufficient characters to the last word to make it distinct from the previous ones. As an example the LZ78 parsing of $T =$ `engineering` is `e, n, g, i, ne, er, in, g`. In the remainder of the chapter we shall assume, for simplicity's sake, that the last word is unique as well.

Let $T = T_1 T_2 \ldots T_d$ be the LZ78 parsing of T. The set $\mathcal{D} = T_1, \ldots, T_d$ is called the dictionary. It can easily be seen that the dictionary is *prefix-complete*. That is, every nonempty prefix of a word in $\mathcal{D}$ is again contained in $\mathcal{D}$. This property will be very important later on.

Theorem 3.4. *Let $T = T_1 T_2 \ldots T_d$ be the LZ78 parsing of text T then $d = O(n/\log n)$.*

Proof. This prove is due to Wim Hesselink. Another, somewhat similar proof appears in [11]. We know $T, T_1, \ldots, T_d \in \Sigma^*$ and $|T| = n$. To prove the order we need to show that $d \leq A\frac{n}{\log n}$ for some $A > 0$. Since we would like to bound for d it suffices to consider only the maximum. Define d_k as the number of T_i's of length k and let x be $|\Sigma|$.

It is clear that d is maximal when for some integer $m > 0$ $d_k = x^k$ for $k < m$, $d_k = 0$ for $k > m$ and $0 \leq d_m \leq x^m$, since the smaller the words, the more we can fit into T. Now we find an upperbound for d

$$d = \sum_{k \in \mathbb{N}} d_k \leq \sum_{k \leq m} x^k = \frac{x^{m+1} - 1}{x - 1} \leq \frac{x^{m+1}}{x - 1} \tag{3.2}$$

and a lowerbound for n

$$\begin{aligned} n &= \sum_{k \in \mathbb{N}} k d_k \geq \sum_{k<m} k x^k = x \frac{\mathrm{d}}{\mathrm{d}x} \sum_{k<m} k x^k = x \frac{\mathrm{d}}{\mathrm{d}x} \left(\frac{x^m - 1}{x - 1} \right) \\ &= \frac{1}{(x-1)^2} \left(x(x-1)mx^{m-1} - x^{m-1}x^2 + x \right) \\ &\geq \frac{1}{(x-1)^2} \left(x(x-1)mx^{m-1} - x^{m-1}x^2 \right) \end{aligned}$$

put $y := x - 1$ and suppose $x \geq 2$ and $m \geq 3$ to obtain

$$\begin{aligned} n &\geq \frac{1}{(x-1)^2} \left(x(x-1)mx^{m-1} - x^{m-1}x^2 \right) \\ &= \frac{x^{m-1}}{y^2} \left((m-1)y^2 + (m-2)y - 1 \right) \geq (m-1)x^{m-1} \end{aligned} \tag{3.3}$$

Let $r := m - 1$ to get $rx^r \leq n$ and $d \leq Bx^r$ for $B = \frac{x^2}{x-1}$. We would like to get rid of the B as well so let $z = d/B$, this yields $z \leq x^r$ and $x^r \leq n/r$.

Remember we wanted to show that $d \leq A\frac{n}{\log n}$, or equivalently, $z \leq A'\frac{n}{\log n}$. We get this for free from the last inequality if we take $r \geq \frac{\log n}{1+\epsilon}$ for any $\epsilon > 0$ since this implies $z \leq \frac{(1+\epsilon)n}{\log n}$. Now consider $r < \frac{\log n}{1+\epsilon}$ we then get:

$$z \leq x^{\frac{\log n}{1+\epsilon}} = n^{\frac{\log x}{1+\epsilon}} < \frac{n}{\log n}$$

where the last inequality follows from:

$$\log n < n^{1 - \frac{\log x}{1+\epsilon}}$$

given that $1 + \epsilon > \log x$. So we have found that $z \leq (1+\delta)\log x \frac{n}{\log n}$, with $\delta > 0$. It follows directly that $d \leq A\frac{n}{\log n}$ for $A = \frac{x^2}{x-1}(1+\delta)\log x$. □

Because of the complexity of the solution we are forced to repeat some of the notation of Ferragina and Manzini. Introduce a new string

$$T_\$ = T_1\$T_2\$ \cdots \$T_d\$ \tag{3.4}$$

where \$ is a character that is not in Σ and is alphabetically smaller than all characters except #. This string is introduced because the \$'s in $T_\$$ will take the role of the markers in the previous section. For every \$ following T_i we store its position $1 + |T_1| + |T_2| + \cdots + |T_i|$ in T. Suppose now we have a pattern P that overlaps such a marker, i.e. it crosses the boundary between two words in the parsing of T. If we know its relative position to the marker we know its location in T. This well be the main idea in the remainder of this chapter.

Not every pattern will however overlap a word boundary. These occurrences are what Ferragina and Manzini call *internal occurrences.* We deal with those in the next section. Another possibility is that a pattern P overlaps one or more words, as was alluded to in the previous paragraph. These occurrences are called *overlapping occurrences.* Finding these is a bit more tricky. Ferragina and Manzini first showed a straightforward, but suboptimal, algorithm which they then optimized. We will cover the simple method here but only describe the changes made to get the optimal version. If possible we will omit the troublesome details.

We need some more notation before we can continu. Let $T_\$^R$ be the string $T_\$$ reversed, that is $T_\$^R = \$T_d^R\$ \cdots \$T_2^R\$T_1^R$. Applying the Burrows-Wheeler transform gives us the cyclic shift matrix $M_{T_\$^R}$ corresponding to $T_\$^R$. Using $\mathtt{Opp}(T_\$^R)$ we can find in $O(|P|)$ time the rows of $M_{T_\$^R}$ that are prefixed by P. Ferragina and Manzini showed that $\mathtt{Opp}(T_\$^R)$ is bounded by $5nH_k(T) + O(n\frac{\log\log n}{\log n})$.

Note that in the next sections we will often call words in $\mathcal{D}$ just 'words'. Do not confuse these words with ordinary words in a written language, they are rarily the same.

3.7 Internal Occurrences

We first focus our attention on the internal occurrences of a given pattern P. So the occurrences where the pattern P is completely contained within a word T_i. Remember that $\mathcal{D}$ is prefix complete. Our overall strategy is as follows:

1. find all words T_i such that P is a suffix of T_i and report the position of P in T for each of them, then

2. find all T_j such that T_i is a prefix of T_j and report the positions of P in T for each T_j as well.

Clearly every word found in step one contains P so an occurrence is correctly reported. Subsequently every word found in step two has a prefix that ends with P, so the word itself contains P as well and is therefore correctly reported. Note that every internal occurrence will be found. If P is contained in a T_k, that is $wPw' = T_k$ for some $w, w' \in \Sigma^*$ then wP is also in $\mathcal{D}$ by the prefix completeness of $\mathcal{D}$ and it will be found in step one. In step two the T_k will be correctly reported.

Let us first focus our attention to the words found in step one. How can they be found? Remember the string $T_\$ = T_1\$T_2\$\cdots\$T_d$. Searching for $P\$$ seems to be a good solution to finding all words ending with pattern P. However knowing where $P\$$ can be found in $T_\$$ still leaves us with the question of which words in $\mathcal{D}$ contain the occurrences. It is possible to create a mapping that maps the position of the \$'s in $T_\$$ to the corresponding T_i. It is hard (if not impossible) to store this mapping efficiently. There exists a better solution to finding the words from step one.

The improved solution makes itself apparent when we examine not $T_\$$ but its reverse, $T_\$^R$. Again we start by first locating the occurrences of the pattern at word boundaries. This time however we have to search for $\$P^R$ instead. In Section 3.3 we showed that in $O(p)$ time we can find the rows in $M_{T_\$^R}$ prefixed by $\$P^R$. Furthermore, matrix $M_{T_\$^R}$ is sorted so the rows beginning with a \$ will be contiguous. Note that there exists a one-to-one correspondence between the rows starting with a \$ and the words in $\mathcal{D}$. More precisely the row beginning with $\$T_i^R\$$ corresponds to the word T_i in $\mathcal{D}$ and vice versa.

We wanted to know to which words the occurrences of $\$P^R$ belong. Create an array $\mathcal{N}[1,d]$ that stores in $\mathcal{N}[i]$ the word in $\mathcal{D}$ that corresponds to the ith row in $M_{T_\$^R}$ beginning with a \$. Now for every row returned by backward_search we can use $\mathcal{N}[1,d]$ to look up the corresponding word in $\mathcal{D}$.

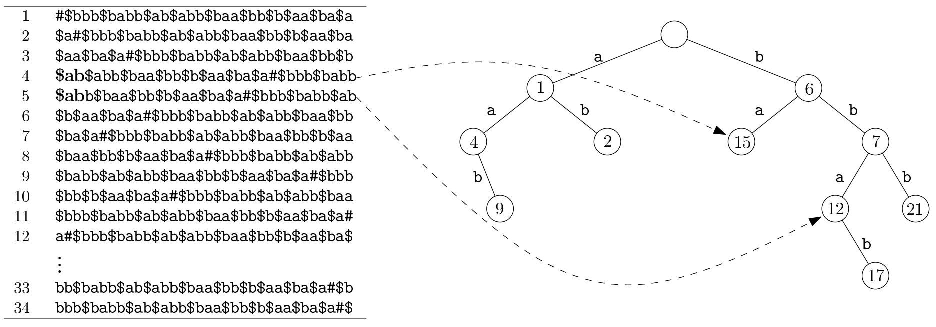
The algorithm described in the previous two paragraphs correctly reports the words in step one. Let's say these words are $T_{i_1},\ldots,T_{i_k}$. Now we need to find the words that have one of the $T_{i_1},\ldots,T_{i_k}$ as a prefix. The fact that $\mathcal{D}$ is prefix-complete suggests that this can be solved efficiently by representing $\mathcal{D}$ as a trie $\mathcal{T}$. We label every edge with a character. Consider a node u in the trie. The path from the root to u always spells out one of the words in $\mathcal{D}$. Henceforth every node in the trie will just be denoted by the word it spells out.

We dit not yet describe how to report an occurrence. To do so we need the index v_i in T where word T_i begins. Then the occurrences in step one have positions $v_i + (|T_i| - p)$. The trie $\mathcal{T}$ is a prefix tree, see [9] and is ideally suited for storing these indices v_i. Just add it as a label to the corresponding node in the trie $\mathcal{T}$.

```
1  **Algorithm** get_internal($P[1,p]$)
2  Search for $\$P^R$ in $M_{T_\$^R}$ thus determining the words $T_{i_1},\ldots,T_{i_k}$ which have $P$ as a
   suffix.
3  For $l = 1,\ldots,k$, visit the subtrie of $\mathcal{T}$ rooted at the corresponding node $T_{i_l}$. For
   each visited word $T_j$ **return** the value of $v_j + (|T_{i_l}| - p)$, where $v_j$ is the starting
   position of $T_j$ in $T$.
```

Listing 3.3: Algorithm get_internal retrieves all internal occurrences of pattern P in text T.

Using the trie we can now easily find all words that have T_i as a prefix. We simply travers the subtree rooted at T_i. All of these nodes have T_i as a prefix and should indeed be reported in step two. It is now also clear that the elements of $\mathcal{N}$ should refer to the corresponding node in the trie to make this process go smoothly. The algorithm described above is named get_internal, see Listing 3.3.



1	#\$bbb\$babb\$ab\$abb\$baa\$bb\$b\$aa\$ba\$a
2	\$a#\$bbb\$babb\$ab\$abb\$baa\$bb\$b\$aa\$ba
3	\$aa\$ba\$a#\$bbb\$babb\$ab\$abb\$baa\$bb\$b
4	**\$ab**\$abb\$baa\$bb\$b\$aa\$ba\$a#\$bbb\$babb
5	**\$ab**b\$baa\$bb\$b\$aa\$ba\$a#\$bbb\$babb\$ab
6	\$b\$aa\$ba\$a#\$bbb\$babb\$ab\$abb\$baa\$bb
7	\$ba\$a#\$bbb\$babb\$ab\$abb\$baa\$bb\$b\$aa
8	\$baa\$bb\$b\$aa\$ba\$a#\$bbb\$babb\$ab\$abb
9	\$babb\$ab\$abb\$baa\$bb\$b\$aa\$ba\$a#\$bbb
10	\$bb\$b\$aa\$ba\$a#\$bbb\$babb\$ab\$abb\$baa
11	\$bbb\$babb\$ab\$abb\$baa\$bb\$b\$aa\$ba\$a#
12	a#\$bbb\$babb\$ab\$abb\$baa\$bb\$b\$aa\$ba\$
	⋮
33	bb\$babb\$ab\$abb\$baa\$bb\$b\$aa\$ba\$a#\$b
34	bbb\$babb\$ab\$abb\$baa\$bb\$b\$aa\$ba\$a#\$

Figure 3.4: Let $T =$ aabaabbbaabbbababbabbb then the relevant piece of $M_{T_\$^R}$ is shown on the left. On the right the corresponding trie is shown. The edges are labeled with the characters. The nodes are labeled with the corresponding location in T.

Example 3.5. Let us consider a somewhat lengthier example. Let $\Sigma = \{a, b\}$ be the alphabet and let $T =$ aabaabbbaabbbababbabbb. When beginning the parsing we have no dictionary words yet. So the only possibility is to add a to the dictionary. For the next word we can extend the a with a b to obtain the longest possible extention: ab. Our dictionary now contains [a, ab]. Continuing in this fashion gives the following parsing [a, ab, aa, b, bb, aab, bba, ba, bbab, bbb]. So we have found that $T_\$ = a\$ab\$aa\$b\$bb\$aab\$bba\$ba\$bbab\$bbb\$$ and $T_\$^R = \$bbb\$babb\$ab\$abb\$baa\$bb\$b\$aa\$ba\$a$.

In Figure 3.4 we can see a piece of $M_{T_\$^R}$. Suppose we want to locate all internal occurrences of pattern ba. Algorithm get_internal dictates that we search for the rows beginning with $\$P^R =$ \$ab. These are rows 4 and 5. Using array $\mathcal{N}$ we can locate the corresponding parsing words T_{i_1} and T_{i_2}. In the trie these are the nodes pointed to by the arrows, let's call them p and q. This completes step 1 of the algorithm.

In step 2 we need to visit each node in the subtries of both p and q. Let's begin with p. The only node in the subtrie is itself so we report $v_{i_1} + (|T_{i_1}| - p) = 15 + (2 - 2) = 15$. The subtrie of node q contains another node, corresponding to word T_{i_3}. For this node we report $v_{i_3} + (|T_{i_2}| - p) = 17 + (3 - 2) = 18$. For the root node q we report $v_{i_2} + (|T_{i_2}| - p) = 12 + (3 - 2) = 13$. Indeed the positions 13, 15 and 18 are the only positions in T that have an internal occurrence of ba.

The following theorem summarized the results obtained in this section. See [8] for a proof of the space bounds.

Theorem 3.6. *Let occ_1 denote the number of internal occurrences of $P[1, p]$ in $T[1, n]$. The algorithm* get_internal *retrieves the internal occurrences in $O(p + occ_1)$ time. Algorithm* get_internal *uses a precomputed data structure with space bounded by $O(nH_k(T)) + O((n \log\log n)/\log n)$ bits.*

3.8 Overlapping Occurrences

As promised we will now concern ourselves with occurrences of the pattern P that overlap more than one dictionary word. We first present a relatively simple algorithm. Having done this we will show how these ideas can be optimized. The core of the simple algorithm is based on the following observation.

Property 3.7. *An overlapping occurrence of the pattern $P[1,p]$ starts inside the dictionary word T_{j-1}, fully overlaps $T_j \cdots T_{j+h-1}$ and ends inside T_{j+h} for some $h \geq 0$, if and only if there exists an m, $1 \leq m < p$ such that $P[1,m]$ is a suffix of T_{j-1} and $P[m+1,p]$ is a prefix of $T_j \cdots T_d$.*

Let us linger on the meaning of this property a bit longer. It states that if an occurrence of P overlaps some successive words in $\mathcal{D}$ then that is equivalent to stating that we can find a splitting point m such that $P[1,m]$ is a suffix of the first overlapped word and the remainder of P, $P[m+1,p]$ is a prefix of the remainder of T. This suggests the following approach to finding the overlapping occurrences. For every splitting point m do the following:

1. Find all dictionary words ending with $P[1,m]$.
2. Find all suffixes of T that begin with $P[m+1,p]$.
3. Check which of the dictionary words immediately precede one of the suffixes and report those occurrences.

Since we check every possible splitting and every possible concatenation this approach will give all overlapping occurrences.

The previous section suggests searching for $\$P[1,m]^R$ using backward_search to find all dictionary words ending with $P[1,m]$. Remember how backward_search, when searching for $\$P[1,m]^R$, first finds all rows of $M_{T^R_\$}$ prefixed by $P[1]$, then the rows prefixed by $P[1,2]^R$, then the rows prefixed by $P[1,3]^R$ and so on until it finds the rows prefixed by $P[1,m]^R$. Looking at the approach above we see we would need the rows prefixed by $\$P[1]^R, \$P[1,2]^R, \ldots, \$P[1,m]^R$ one for each m we choose. By changing backward_search a little bit we can obtain all these sets of rows in one pass. After finding the rows prefixed by $P[1,k]^R$ first find the rows prefixed by $\$P[1,k]^R$, this yields one of the desired ranges. Then backtrack and find the rows prefixed by $P[1,k+1]^R$, and so on. In doing so we find all ranges $[f^*_m, l^*_m]$ of rows prefixed by $\$P[1,m]^R$ for all $1 \leq m < p$. Similarly we can find all ranges $[f_m, l_m]$ of rows in M_T that are prefixed by $P[m,p]$. Now we have two sets of ranges that somehow represent the points found at steps one and two for every m. We only need to combine these in order to complete step 3.

The property above is stated in such a way that we only need to consider joining at the word boundary, there are only d of them. Ferragina and Manzini proposed a geometric approach. For every splitting point m we have ranges $[f^*_m, l^*_m]$ and $[f_{m+1}, l_{m+1}]$. The approach is based on the following observation. If pattern P starts in T_i, say $T_i = wP[1,m]$ for some $w \in \Sigma^*$, then the row of $M_{T^R_\$}$ prefixed by $\$T_i^R\$$ should be in the range $[f^*_m, l^*_m]$. Similarly the row of M_T prefixed by

$L_{i+1} \cdots L_D$ should be in the range f_{m+1}, l_{m+1}. For every splitting point we thus need to check which, if any, of the word-boundaries are inside these ranges.

Suppose we draw a 2D grid. On the x-axis we have the rows in $M_{T^R_\$}$. On the y-axis we have the rows in M_T, see also Figure 3.6 for an example. For every word boundary $T_i\$T_{i+1}$ we draw a point in this grid. The x coordinate x_i is the row in $M_{T^R_\$}$ prefixed by $\$T_i^R\$$. The y-coordinate y_i is the row in M_T prefixed by $T_{i+1} \cdots T_d\#$. So we have a set of 2D points $\mathcal{Q} = (x_1, y_1), \ldots, (x_{d-1}, y_{d-1})$, each corresponding to a word boundary. Now for every region $[f^*_m, l^*_m] \times [f_m, l_m]$ we find all points of $\mathcal{Q}$ that lie inside this region. We have thus found all overlapping occurrences.

Finding all the points from $\mathcal{Q}$ that lie in a given range is called an orthogonal range query. Note that the dataset we query, the one containing $\mathcal{Q}$, may be precomputed. Ferragina and Manzini use a result by Alstrup [1] to make these queries sufficiently fast. The corresponding data structure is reffered to as $\mathcal{RT}(\mathcal{Q})$. The resulting algorithm is shown in Listing 3.4. This data structure supportes orthogonal range queries in $O(\log\log n + q)$ time, where q is the number of retrieved points.

```
1 Algorithm get_overlapping($P[1,p]$)
2 For $m = p, p-1, \ldots, 1$, search for $P[m,p]$ in $\mathsf{Opp}(T)$ thus retrieving the range
  $[f_m, l_m]$ of rows in $M_T$ prefixed by $P[m,p]$.
3 For $m = 1, 2, \ldots, p$, search for $\$P[1,m]^R$ in $\mathsf{Opp}(T^R_\$)$ thus retrieving the range
  $[f^*_m, l^*_m]$ of rows in $M_{T^R_\$}$ prefixed by $\$P[1,m]^R$.
4 For $m = 1, 2, \ldots, p-1$ use the data structure $\mathcal{RT}(\mathcal{Q})$ to retrieve the points of $\mathcal{Q}$
  which lie inside the rectangle $[f^*_m, l^*_m] \times [f_m, l_m]$.
5 For each point $(x_j, y_j)$ retrieved at the $m-$th iteration of the previous step **return**
  the value $v_j - m$, where $v_j = 1 + |T_1| + \cdots + |T_{j-1}|$ is the starting position of the
  word $T_j$ inside $T$.
```

Listing 3.4: Algorith get_overlapping retrieves all overlapping occurrences of the pattern P in T.

Example 3.8. *Continuation of Example 3.5.* We are considering the string T = aabaabbbaabbbababbabbb over the alphabet $\Sigma = \{a, b\}$. We will use algorithm get_overlapping to find all overlapping occurrences of pattern P = abb. In step 2 we find the ranges of rows of M_T, see Figure 3.5, beginning with a suffix of P, see the first two rows of Table 3.2. In step 3 we find the rows of $M_{T^R_\$}$, see Figure 3.4, beginning with $\$P[1,m]^R$ for $m = 1, 2, 3$. The results are shown in the last two rows of Table 3.2.

In step 4 we need to examing the rectangles $[f^*_1, l^*_1] \times [f_1, l_1]$ and $[f^*_2, l^*_2] \times [f_2, l_2]$. They are shown in Figure 3.6 together with small squares corresponding to the points in $\mathcal{Q}$. We immediately see there are four such points inside the rectangles. These correspond to the four overlapping occurrences at positions 5, 10, 16 and 19.

The time and space bounds, as proven by Ferragina and Manzini, are summarized in the following theorem.

m	3	2	1
$P[m,p]$	b	bb	abb
$[f_m, l_m]$	[11,24]	[17,24]	[7,10]
$\$P[1,m]^R$	$bba	$ba	$a
$[f^*_m, l^*_m]$	$\emptyset$	[7,9]	[2,5]

Table 3.2: Shows the results of steps 1 and 2 when searching for $P =$ `abb` in $T =$ `aabaabbbaabbbababbabbb`.

1	#aabaabbbaabbbababbabbbb
2	aabaabbbaabbbababbabbbb#
3	aabbbaabbbababbabbbb#aab
4	aabbbababbabbbb#aabaabbb
5	abaabbbaabbbababbabbbb#a
6	ababbabbbb#aabaabbbaabbb
7	abbabbbb#aabaabbbaabbbab
8	abbbaabbbababbabbbb#aaba
9	abbbababbabbbb#aabaabbba
10	abbbb#aabaabbbaabbbababb
11	b#aabaabbbaabbbababbabbb
12	baabbbaabbbababbabbbb#aa
13	baabbbababbabbbb#aabaabb
14	bababbabbbb#aabaabbbaabb
15	babbabbbb#aabaabbbaabbba
16	babbbb#aabaabbbaabbbabab
17	bb#aabaabbbaabbbababbabb
18	bbaabbbababbabbbb#aabaab
19	bbababbabbbb#aabaabbbaab
20	bbabbbb#aabaabbbaabbbaba
21	bbb#aabaabbbaabbbababbab
22	bbbaabbbababbabbbb#aabaa
23	bbbababbabbbb#aabaabbbaa
24	bbbb#aabaabbbaabbbababba
	M_T

Figure 3.5: The Burrows-Wheeler transform of $T =$ `aabaabbbaabbbababbabbb`.

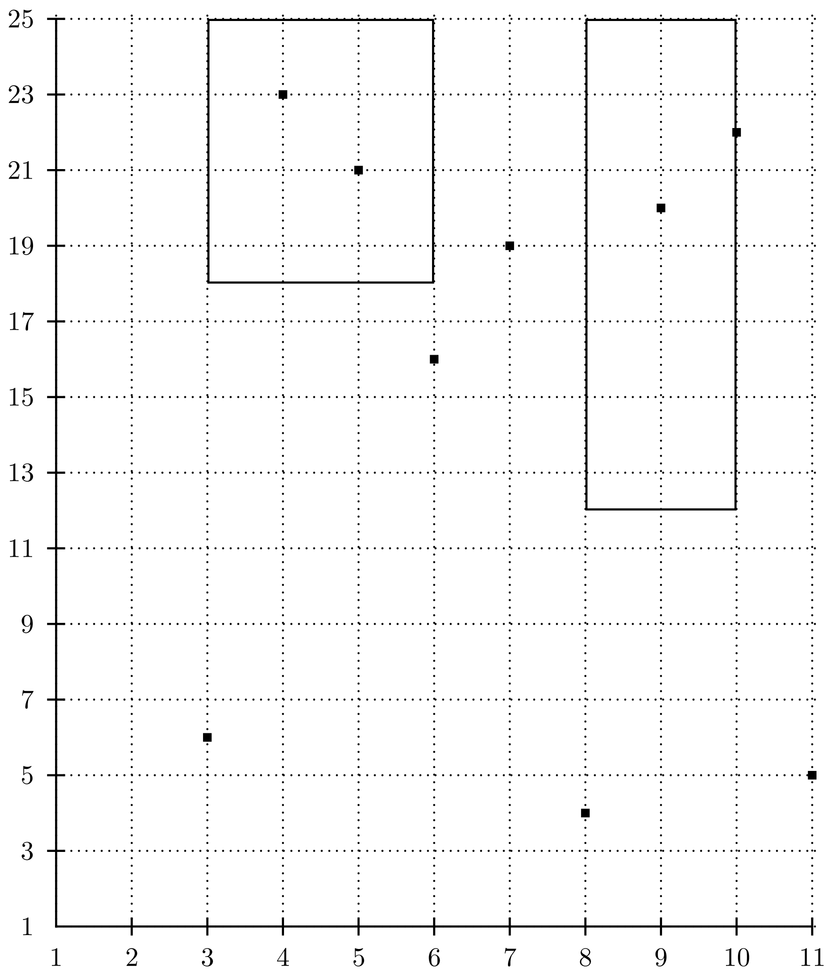


Figure 3.6: The 2D grid displaying all point (x_i, y_i) as small squares. The orthogonal ranges are shown as rectangles. Along the x-axis are the rows in $M_{T^R_\$}$, while along the y-axis are the rows in M_T.

Theorem 3.9. *Let occ_O denote the number of overlapping occurrences of $P[1,p]$ in $T[1,n]$. Algorithm* get_overlapping *retrieves them in $O(p \log\log n + occ_O$ time. The precomputed data structures used by* get_overlapping *take $O(nH_k(T)\log^{\epsilon} n) + O(n/log^{1-\epsilon}n)$ bits of storage overall, where $0 < \epsilon < 1$ is an arbitrary constant chosen when building the data structures.*

3.8.1 A Faster Approach

As mentioned before the theoretical lower bound of finding all occurrences of a pattern $P[1,p]$ is $O(p + occ)$, so the previous result is off by a factor $\log\log n$. Ferragina and Manzini removed this factor by reducing the number of orthogonal range queries, since they are responsible for this unnecessary factor. Instead of trying all m as possible splitting points we only examine points of the form $m = 1 + i\log\log n$, where we just assume that $\log\log n$ is an integer.

Again we will employ a defining property to obtain a correct algorithm. Let T_j^{-k} be the word T_i without the last k characters, that is T_j^{-k} is the prefix of T_i with length $|T_i| - k$. Similarly let T_i^{+k} be the suffix of length k of T_i. The following property states that there is always a splitting point of the form $m = 1 + i\log\log n$ within $\log\log n$ characters of a word boundary, provided of course that a character overlaps multiple words.

Property 3.10. *An overlapping occurrence of pattern $P[1,p]$ starts inside the dictionary word T_{j-1} and ends inside T_{j+h}, for some $h \geq 0$, if and only if there exist $i \geq 0$ and $k \in [0, \log\log n - 1]$ such that $P[1, i\log\log n + 1]$ is a suffix of T_{j-1}^{-k} and $P[i\log\log n + 2, p]$ is a prefix of $T_{j-1}^{+k} T_j \cdots T_d$.*

This property has an interesting reading. If we not only consider our anchoring points right at the splitting of a word but also in a small neighborhood of those word-boundaries we only have to test $p/(\log\log n)$ possible splitting points of P. Our general approach is identical to the previous attempt. Only this time we have to take into account the extra anchoring points and what to do with them. Create a new data structure $\mathcal{Q}'$ that contains points (x, y) for every anchoring point, just like they did for $\mathcal{Q}$. For each i, k let $x_{i,k}$ be the row of $M_{T_\$^R}$ prefixed by $(\$T_i^{-k}\$)^R$ and $y_{i,k}$ the row of M_T prefixed by $T_i^{+k}T_{i+1}\cdots T_d\#$, provided that $k \leq |T_i|$. Note that by the prefix-completeness of $\mathcal{D}$ the string T_i^{-k} is also in $\mathcal{D}$. We associate the value $v_{i,k} = 1 + |T_1| + \cdots + |T_{i-1}| - k$ with the point $(x_{i,k}, y_{i,k})$ and add each of these points to $\mathcal{Q}'$.

Now that we have created additional anchoring points we know by the property stated above that we only need to test splitting the pattern $P[1,p]$ at points of the form $m = i\log\log n + 1$. The resulting algorithm can be seen in Listing 3.5. The query time for the data structure $\mathcal{RT}(\mathcal{Q}')$ is again $O(\log\log n + q)$ where q is the number of retrieved points. The results are summarized in the following theorem.

Theorem 3.11. *Let occ_O denote the number of overlapping occurrences of a long pattern $P[1,p]$ in $T[1,n]$. Algorithm* get_overlapping_fast *retrieves them in $O(p + occ_O)$ time. It requires $O(nH_k(T)\log^{\epsilon} n) + O(n/\log^{1-\epsilon})$ bits of storage for the precomputed data structures. Here ϵ is an arbitrary constant, $0 < \epsilon < 1$.*

```
1 Algorithm get_overlapping_fast(P[1,p])
2 For $m = p, p-1, \ldots, 1$, search for $P[m,p]$ in $\mathsf{Opp}(T)$ thus retrieving the range
  $[f_m, l_m]$ of rows in $M_T$ prefixed by $P[m,p]$.
3 For $m = 1, 2, \ldots, p$, search for $\$P[1,m]^R$ in $\mathsf{Opp}(T_\$^R)$ thus retrieving the range
  $[f_m^*, l_m^*]$ of rows in $M_{T_\$^R}$ prefixed by $\$P[1,m]^R$.
4 For $j = 0, 1, \ldots, \lfloor (p-2)/(\log\log n) \rfloor$, set $h = j \log\log n + 1$ and use the data
  structure $\mathcal{RT}(\mathcal{Q}')$ to retrieve the points of $\mathcal{Q}'$ which lie inside the rectangle
  $[f_m^*, l_m^*] \times [f_m, l_m]$.
5 For each point $(x,y)$ retrieved at the $j-$th iteration of the previous step return the
  value $v_{i,k} - (j \log\log n + 1)$, where $v_{i,k}$ is the value associated with $(x,y)$.
```

Listing 3.5: Algorith get_overlapping_fast retrieves all overlapping occurrences of the pattern $P(1,p)$ in $T(1,n)$.

3.8.2 Summary of Results

In this section and the preceding one we have seen how we can handle both internal and overlapping occurrences. Searching for an occurrence takes $O(p + occ_I)$ time in the internal case and $O(p + occ_O)$ time in the overlapping case. Combining them yields an algorithm of order $O(p + occ)$, where p is the length of the pattern P and occ the number of occurrences of P in T. We summarize the overall results in the following theorem.

Theorem 3.12. *For any text $T[1,n]$ we can create a full-text index that supports queries for a pattern $P[1,p]$ in $O(p + occ)$ time. The size of the precomputed index is bounded by $O(nH_k(T)\log^\epsilon n) + O(n/\log^{1-\epsilon})$ bits for any $k \geq 0$. The constant $0 < \epsilon < 1$ is an arbitrary constant chosen when building the index.*

Chapter 4

Regular expressions

The full-text index described in the previous chapter is perfectly fine for locating specific patterns $P[1, p]$. However we might envision asking, for a given text, a query like: where are all the quoted pieces in this text? Or maybe even: what are all the quoted pieces in this text? And what about finding all the numeric expressions in a text? None of these questions can easily be answered by a query for a pattern, since we a priori do not know what the pattern looks like. A query language that is more suited to answer these kinds of questions is that of regular expressions.

Ordinarily regular expression engines operate by examining the entire text character by character, while performing some simple bookkeeping. It is our goal to create an engine that does not have to scan the entire text, but instead uses the full-text index described in the previous chapter. We will begin by recalling the formal definition of a regular expression. Over the years quite a number of extensions have been made to this formal definition to make application of these regular expressions easier. We will briefly touch upon them as well, since we will implement some of these extensions.

We have chosen to find all matches of a given regular expressions instead of just the longest-leftmost match. This choice does have major repercussions in the sense that for some regular expressions the result will contain a great many overlapping and nested results. The rule of always picking the longest-leftmost match is however not infallible either. In [6] these problems are discussed in greater detail. The authors propose an alternative to the longest-leftmost rule that is more suitable in most practical applications involving large written texts. As can be seen there are many different options about which match to pick, in order to allow the most flexibility we have opted to just find them all.

After giving the formal definition in Section 4.1, we will examine a set-based implementation of this formal definition in Section 4.2. We derive both a time and a space bound for this method. Most of the current fast implementations are based on some kind of NFA method. We will examine this method in Section 4.3 and then design a competitive implementation of this NFA method that utilizes the full-text index. Finally we consider the difficulties that occur when retrieving a match in Section 4.4.

4.1 Formal Definition

Regular expressions offer a way of describing so-called regular languages. In the formal sense a language is a set of strings over a given alphabet. The set $L = \{s \in \Sigma^* : s = ba^n, n \in 0, 1, 2, \ldots\}$ is an example of such a language over the alphabet $\Sigma = \{a, b\}$. The corresponding regular expression R is given by `ba*`, in other words, a `b` followed by zero or more `a`'s. Note how we have written the literal parts of the regular expressions in typewriter font to distinguish it from the other text.

It is useful to describe regular expressions in an inductive fashion, see for example [10] for a classical account. We summarize those results here. The main idea is that every regular expression has an associated language. We will write $L(R)$ for the language associated with the regular expression R. We will shortly show how operators on regular expressions operate. First we need to examine the basic building blocks:

1. The constants ϵ and $\emptyset$ are regular expressions. They represent respectively the language ϵ, containing only the empty string, and the language $\emptyset$, the empty language. Summarizing, $L(\epsilon) = \{\epsilon\}$ and $L(\emptyset) = \emptyset$.
2. If a is a character in the alphabet then `a` is a regular expression denoting this character, that is $L(\mathtt{a}) = \{a\}$.
3. A variable L representing a language is a regular expression.

There are three valid operators: union, concatenation and the Kleene closure. In the following let both A and B be regular expressions. We have the following induction steps:

1. The union between A and B is denoted $A|B$ and is a regular expression. The corresponding language is the union of the languages defined by A and B, that is $L(A|B) = L(A) \cup L(B)$.
2. The concatenation of A and B is denoted by AB and is a regular expression. The corresponding language is obtained by taking the concatenation of all strings in A with all strings in B, that is $L(AB) = L(A)L(B) = \{ab \in \Sigma^* : a \in L(A), b \in L(B)\}$.
3. The Kleene closure of A is denoted by A^* and is a regular expression. The corresponding language is obtained by taking the closure of $L(A)$, $L(A)^*$. Each element in $L(A)^*$ can be obtained by taking an arbitrary number of strings from $L(A)$, it is allowed to take one string multiple times, and concatenate them. Note that the empty string is also in $L(A)^*$.
4. The parenthesized regular expression (A) denotes the same language as A, that is $L((A)) = L(A)$.

In practice we will never use the regular expression $\emptyset$. As for notation, we have deviated a bit from Hopcroft et al. [10] here, in favour of the more common notation employed by many Unix tools. Let us look at the precedence of the operators. The star binds strongest, it binds to the smallest complete regular expression to the left. The next strongest operator is the concatenation. After grouping the closures we group the adjacent expressions. The weakest operator is the union operator, it binds least. When it is not clear how to group expressions we associate them from the left.

Example 4.1. The regular expression `aaba` is grouped as `(((aa)b)a)`. The regular expression `1(1|0)*|0` is grouped as `(1((1|0)*))|0`. It represents all positive binary numbers without leading zeroes.

When applying regular expressions in real-world scenarios the semantics change a bit. When a string *matches* a regular expression it usually does not imply that the entire string is in the language represented by the regular expression, but rather that a substring is. As a result we can only speak of 'a match' for the regular expression. We will just try to find all of them.

4.1.1 Modern extensions

Sometimes writing expressions using the building blocks above becomes a bit tedious. Suppose we want to match an arbitrary character followed by the word 'tree'. The only possible solution is to create a big union of the form $(a|b|\cdots|y|z)$ to match an arbitrary character, provided that the alphabet $\Sigma = \{a, b, \ldots, y, z\}$. The solution is to introduce the regular expression `.`, pronounced dot, such that $L(.) = \Sigma$. Furthermore we introduce a couple of variants for repetition. The operator `+` means one or more times, so $A+$ equals $AA*$. The operator `?` means zero or one times, so $A?$ equals $(\epsilon|A)$.

4.2 Set-based implementation

In the set-based approach a set of pairs will represent all matches of a corresponding regular expression. Initially these sets are generated by using the full-text index to searching for the strings and find all matches. Then we apply operators to these sets to obtain new sets, this continues until the entire regular expression has been processed.

Each pair will denote a match (a, b), the first component, a, corresponds to the starting position of the match in T. The second component, b, is the first character after the match, so (a, b) matches $T[a, b-1]$. Note that in this context an empty match is just of the form (a, a). Denote the set corresponding to a regular expression A by $\mathcal{S}(A)$.

The basis elements of a regular expression as defined in the previous section are not entirely suited for this approach. Imagine the embarrassingly simple regular expression `foobar`. Decomposing it using the previous definition gives six sets, one for each of the characters, and five operations on these sets. This is of course far from optimal since using the full-text index to just find `foobar` gives only one sets and no operations. The latter is obviously preferable. So somehow we want to consider a string as one unit, instead of a concatenation of characters. Make the following change to the definition of the basic building blocks.

2'. If w is a nonempty string, that is $w \in \Sigma^+$, then w is a regular expression denoting this string, so $L(w) = w$.

To remove the ambiguity caused by the fact that concatenating two strings is possible as well change the induction step to:

2'. The concatenation of regular expressions A and B is denoted by AB and is a regular expression unless both A and B are as in basis step 2'. The corresponding language is $L(AB) = L(A)L(B)$.

Translating ϵ and $\emptyset$ into sets is rather easy. Let for a text $T[1, n]$ the set $\xi = \{(a, a) : a \in 1, \ldots, n\}$ represent the regular expression ϵ, that is $\mathcal{S}(\epsilon) = \xi$. In a way ξ denotes that at any point in T we can match the empty string ϵ. Since T is never empty, $\emptyset$ is represented by itself, so $\mathcal{S}(\emptyset) = \emptyset$.

The operations are rather easy to define on the sets. Suppose we have two regular expressions A and B, with corresponding sets of matches $\mathcal{S}(A)$ and $\mathcal{S}(B)$. Then

1. The set corresponding to the union $A|B$ is given by

$$\mathcal{S}(A|B) = \mathcal{S}(A) \cup \mathcal{S}(B) = \{(p, q) : (p, q) \in \mathcal{S}(A) \vee (p, q) \in \mathcal{S}(B)\}, \quad (4.1)$$

since it may contain matches of A as well as matches of B.

2. The set corresponding to the concatenation AB is given by

$$\mathcal{S}(AB) = \mathcal{S}(A) \circ \mathcal{S}(B) = \{(p, q) : (p, r) \in \mathcal{S}(A) \wedge (r, q) \in \mathcal{S}(B)\}, \quad (4.2)$$

since matches in B have to follow matches in A.

3. The set corresponding to the closure $A*$ is given by

$$\mathcal{S}(A^*) = \xi \cup \{(p, q) : (p, q) \in A^n, n \in \mathbb{P}\}, \quad (4.3)$$

since every match for $A*$ is obtained by repeating A arbitrarily often.

This definition gives a remarkable clean solution for finding all matches of a given regular expression R, it is however not very efficient. As mentioned before the full-text index from the previous chapter will take care of creating the sets corresponding to the strings. To facilitate reasonably quick operations we need to consider a suitable data structure for the sets. It is desirable to keep the sets as small as possible, since both the union and the closure can increase the number of elements significantly. A nice solution to this problem is by just demanding that the data structures should not contain any duplicates, we will see that we can do this without an additional penalty. Furthermore to calculate a concatenation we need to swiftly locate all elements of the form $(a, \cdot)$, for some $a \in \{1, \ldots, n\}$. As a solution create a linked list L with the elements sorted by

$$(a_1, b_1) \leq (a_2, b_2) \equiv a_1 < a_2 \vee (a_1 = a_2 \wedge b_1 < b_2). \quad (4.4)$$

In addition create a hashtable H indexed with keys in $\{1, \ldots, n\}$. If there exist elements $(a, \cdot)$ in L then $H(a)$ points to the first of those elements in L. This way we can easily find all elements $(a, \cdot)$ by simply looking up $H(a)$ and then following the linked list.

Let us examine the algorithm for calculating the union of two sets. It is union in Listing 4.1. Merging L_A and L_B is easy since they are both sorted and the linkedlist supports iterating. As a result step 3 takes only $O(|L_A| + |L_B|)$ time, the same timebound holds for step 4 so calculating the union can be done in $O(|L_A| + |L_B|)$ time.

Determining the concatenation is a bit more involved. Take a look at equation (4.2), for every $(p, r) \in L_A$ we need to add (p, q) for every $(r, q) \in L_B$. See Listing 4.1 for a more formalized version of concatenation. The algorithm takes $O(|L_A||L_B|)$ time since worst case every element has to be connected to every other element.

```
 1  Algorithm union(($L_A, H_A$), ($L_B, H_B$))
 2  Let $L$ be an empty linked list, and $H$ an empty hashtable.
 3  Merge the elements from $L_A$ and $L_B$ into $L$ by always adding the smallest element first.
 4  Fill hashtable $H$ by iterating over the elements of $L$.
 5  return $(L, H)$.

 1  Algorithm concatenation(($L_A, H_A$), ($L_B, H_B$))
 2  Let $L$ be an empty linked list, and $H$ an empty hashtable.
 3  for each element $(p, r) \in L_A$ do
 4    for each element $(r, q) \in L_B$ do
 5      Add $(p, q)$ to $L$
 6      Link $H(p)$ to $(p, q)$ if it does not exist already.
 7    end
 8  end
 9  return $(L, H)$.

 1  Algorithm closure(($L_A, H_A$))
 2  Let $L$ and $L_{\text{prev}}$ be empty sets, and $H$ and $H_{\text{prev}}$ be empty hashtables.
 3  $L \leftarrow \emptyset$;
 4  while $L \neq L_{\text{prev}}$
 5    $L_{\text{prev}} \leftarrow L, H_{\text{prev}} \leftarrow H$;
 6    $(L, H) \leftarrow$ union(($L_{\text{prev}}, H_{\text{prev}}$), concatenation($L, L_A$));
 7  end
 8  $L_{\text{prev}} \leftarrow L, H_{\text{prev}} \leftarrow H$;
 9  $(L, H) \leftarrow$ union(($L_{\text{prev}}, H_{\text{prev}}$), ($L_\xi, H_\xi$));
10  return $(L, H)$.
```

Listing 4.1: Algorithms union, concatenation and closure

The final algorithm in Listing 4.1 is closure. Notice how the algorithm just calculates

$$\xi \cup \bigcup_{i=1}^{\infty} A^i \tag{4.5}$$

which equals $\mathcal{S}(A^*)$. The only difference is that we do not take the infinite union but stop after the set does not change anymore. Suppose L_A has length m. The longest possible sequence of concatenations has length $m \leq n$, since we cannot go outside the text. A quick calculations shows that taking these m unions costs $O(1 + m + m^2 + \cdots + m^m = O(m^{m+1})$ time. Since the size of a set is at most n^2 and the maximum sequence has length n we have a slightly more optimistic bound. The algorithm is timebounded by $O(n \cdot n^2 n^2)$, since we take n unions of sets of size n^2. Neither of these bounds is very promising.

Even though the general ideo of this setbased implementation is simple, calculating the closure is prohibitively expensive. In the following sections we will see how we can improve these results. The reason we get a better bound is because we only determine the closure on demand.

4.3 An NFA based approach

The fastest [7] modern regular expression engines are based on the concept of a finite automaton. This method was first described by Thompson in 1968 in [16]. He did so without actually mentioning the finite automata. The term was not yet well established then. Finite automata are conceptual machines that have a finite number of states, hence their name. After reading a character the machine can go to a new state. The initial state is fixed. Some of the states are socalled accepting states. If an input sequence leads to such a state it is said to be accepted by the finite automaton, it is in the language. It turns out that regular expressions can accept the same class of languages as finite automata can. There exists an easy translation from a regular expression to the corresponding finite automaton. An excellent formal account of automata and the translation from regular expresions to automata is given in [10]. We do not require the rigorous introduction to automata given in [10], we will introduce the necessary concepts about automata in the following examples. At the same time we will illustrate how a regular expression can be converted into a corresponding automaton.

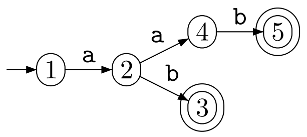


Figure 4.1: Automaton corresponding to the regular expression `a(b|ab)`.

Example 4.2. Consider the regular expression $R = \mathtt{a}(\mathtt{b}|\mathtt{ab})$. The corresponding automaton is shown in Figure 4.1. Each state is represented by a node. The initial state is marked by an incoming arrow. In this example it is state 1. Transitions of the automaton are represented by edges. For example, when the automaton is in state 2 it can go to state 4 when it reads an `a` or to state 3 when it reads a `b`. When there is no outgoing edge with the right character the automaton blocks. The accepting states are marked with a double circle. The sequences `ab` and `aab` bring the automaton into an accepting state, it is easy to see that these are the only input sequences leading to such a state. The only strings in $L(R)$ are `ab` and `aab`, so the automaton does indeed correspond with the regular expression.

In a way the previous finite automaton is simple. For every node it is always clear which transition to follow. The next example shows that this is not always the case.

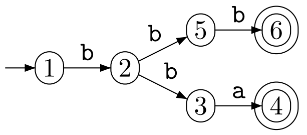


Figure 4.2: Automaton corresponding to the regular expression `b(bb|ba)`.

Example 4.3. Consider the regular expression `b(bb|ba)`. Drawing an automaton like the one in the previous example gives Figure 4.2. This automaton has a problem in state 2, if it reads a `b` it can go to state 3 as well as state 5. To deal with these kinds of transistions assume that the automaton will magically choose that transition that will lead to accepting state, if such a transition exists. Note that this really requires some kind of magic since the remainder of the input is not yet known at the moment the automaton has to make that decision. As an example, after reading `bb` of `bbb` the automaton will go to state 5 and not state 3. If it however reads `bb` as a beginning of `bba` it will go to state 3 instead.

Automata where there always is only one possible transition are called deterministic finite automata, or DFA's. When magic is required to decide which transition to take the automaton is called a nondeterministic finite automaton, or NFA. Note that it is not possible to build an NFA since we do not know how to implement the magic component. It is however possible to convert an NFA to a DFA, see [10] for the details.

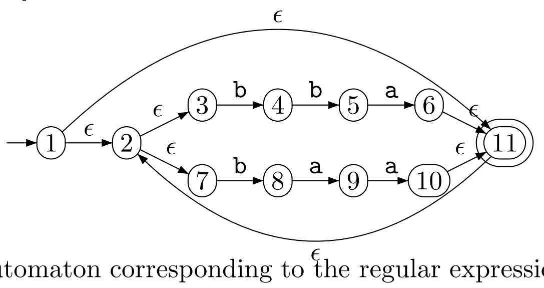


Figure 4.3: Automaton corresponding to the regular expression `(bba|baa)*`.

Example 4.4. Consider the regular expression `(bba|baa)*`. The corresponding automaton is shown in Figure 4.3. Notice the introduction of a new kind of transition, the ϵ-transitions, marked by an ϵ. Before and after every transition the automaton can follow an arbitrary number of ϵ transitions. Notice how the structure of the regular expression shows itself in the automaton. States 3, 4, 5 and 6 are responsible for matching `bba`, states 7, 8, 9 and 10 are responsible for matching `baa`. The split in state 2 corresponds to the union `(bba|baa)` and finally the surrounding ϵ-transitions are responsible for the closure.

An NFA that includes ϵ-transitions is often called an ϵ-NFA. Once again adding these special transitions does not make the total class of possible languages bigger. An ϵ-NFA is just as strong as a DFA. See once more [10] for an algorithm for converting an ϵ-NFA into a DFA that accepts the same language. Eventhough adding ϵ-transitions and nondeterminism doesn't increase what the automaton can do, it does make it easier to build automata.

The same induction steps as used originally to define regular expressions can be used to construct a corresponding ϵ-NFA from a regular expression. Each basis part of a regular expression gets converted to an ϵ-NFA. Combine these using the induction steps above to obtain a new automata.

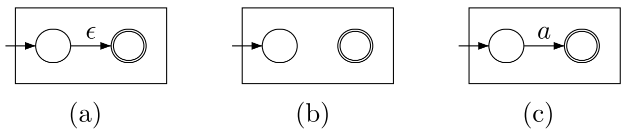


Figure 4.4: The basis constructions of a regular expression converted to an NFA.

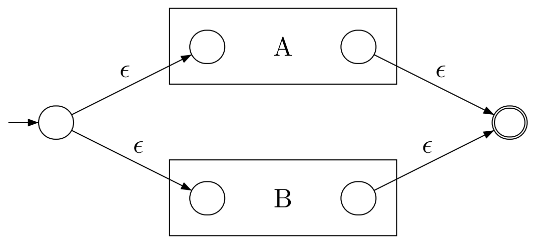
In Figure 4.4 the three basis components for a regular expression are shown in NFA-form. Subfigure (a) shows the automaton corresponding to the regular expression ϵ. It correctly accepts the language $\{\epsilon\}$, since that is the only input that leads to an accepting output. Subfigure (b) shows the automaton corresponding to the regular expression $\emptyset$. Since there are no transitions to the accepting state it does not accept anything, that is the language is $\emptyset$. Subfigure (c) show the automaton for accepting the character a.

How to combine these very simple automata is shown in Figure 4.5. The blocks with the A or B inside represent the automata corresponding to regular expressions A and B. In subfigure (a) we see the automaton corresponding to the regular expression $(A|B)$. As a result of the ϵ-transitions the only two possible paths to get to the accepting states are through either A or B, so it accepts the correct language. Subfigure (b) shows the concatenation of A and B, that is it corresponds to the regular expression AB. The construction of the automaton forces it to first read an acceptable input for A followed by an acceptable input for B before it reaches an accepting state. Again it accepts the correct language. Finally subfigure (c) shows an automaton for the closure of A. The ϵ-transitions make sure the automaton can reach the accepting state by passing A zero, one or more times. As a result the automaton accepts the correct input. For a formal account see [10].

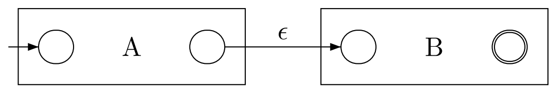
4.3.1 Simulating an automaton

We have seen how every regular expression can be converted to an ϵ-NFA. This is very useful since NFA's are very easy to simulate. We first show how to simulate the automaton. This gives an algorithm for using a regular expression in its orginal sense, the automaton will report for every input string if that string is in the alphabet. Later on we show how we can adapt this algorithm to report all possible matching substrings.

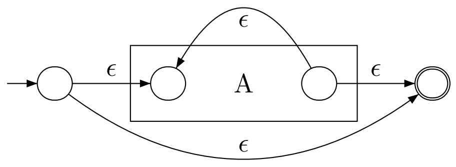
Simulating a DFA is straightforward. At every point the automaton is only in one, well-defined state. Reading a character brings the automaton to a next state that is also well-defined. So keeping track of the current state is sufficient



(a)



(b)



(c)

Figure 4.5: The inductive constructions of a regular expression converted to an NFA.

to simulate a DFA. Things change when simulating an NFA. As a result of the nondeterminism we don't know in which state the automaton is. The solution is to keep track of all possible states that the automaton can currently be in. When processing the next character the algorithm considers each state in turn and adds all possible new states to the set of current states. In this way we just simulate the automaton to be in a lot of states at once. Since it is a finite automaton, there are only a finite number of states, so this is perfectly doable.

```
Algorithm simulate_NFA(T, nfa)
Let $S_{old}$ be an empty set of states.
Let $S$ by a set of states containing only the initial state.
Replace $S$ by its $\epsilon-$closure.
while more characters
  Let $c$ be the next character.
  $S_{old} \leftarrow S; S \leftarrow \emptyset$.
  for each state $s$ in $S_{old}$
    Add states reachable from $s$ by an edge marked with a $c$ to $S$.
  end
end
return “accepted” if an accepting state is in $S$.
```

Listing 4.2: Algorithm simulate_NFA simulates an ϵ-NFA.

Let us examine Listing 4.2. Set S contains the current set of states. Notice how we immediately add the ϵ-closure to S. This closure includes all states reachable by just taking ϵ-transitions, this is necessary because all these states can be the current state of the NFA. Next we process each character in turn and advance the machine whenever possible. It is possible that one of the states has no transitions with a c. In that case that state is discarded. It is also possible that the same state is reached in more than one way, the fact that we deal with sets guarantees we only get that state once. At the end we correctly return that the string is accepted only if there is an accepting state in the final set of states. This is correct since the NFA would have ended up in this accepting state by means of its magic.

```
Algorithm find_all_matches(T, regexp)
Let nfa be the $\epsilon-$NFA corresponding to regexp.
For each suffix $P$ of $T$ do
  Run simulate_NFA($P$, nfa) to find all matching prefixes of $P$, and report them.
end
```

Listing 4.3: Algorithm find_all_matches uses simulate_NFA to find all matching substrings

The algorithm simulate_NFA can be used to report all possible matches by adapting it slightly. Suppose we run simulate_NFA on a suffix of the input text T. It will dutifully report if the entire suffix matches. Suppose now that after processing a character we check if one of the states in S is accepting. In doing so we can report all matching prefixes of the suffix of T. By running the algorithm for each suffix of T we thus examine all possible substrings and thus find all matches. The algorithm find_all_matches that implements this is shown in Listing 4.3. The following theorem shows the time complexity of this algorithm.

Theorem 4.5. *Algorithm* find_all_matches *finds all matching substrings of text T in $O(mn^2)$ time, where m is the length of the regular expression and n is the length of the text T.*

Proof. Take a look at the conversion of the regular expression to an NFA as shown in Figures 4.4 and 4.5. For every character in the basis steps only 2 states are necessary. An induction steps introduces at most 2 new states. So the number of states of the resulting NFA is $O(m)$.

Now let us look at the actual algorithm. The loop is of course $O(n)$ since T has length n and we consider every suffix. Algorithm simulate_NFA's outerloop loops over all characters of P. Since this is a suffix of T it is $O(n)$ as well. The inner loop loops over all states. Since the number of states is bounded by $O(m)$ the inner loop is $O(m)$ as well. As a result algorithm simulate_NFA is $O(mn)$ and algorithm find_all_matches is $O(mn^2)$. □

4.3.2 Using a full-text index

Notice how simulating an NFA is a more efficient method of finding all matches than using the set-based approach. However we need the original text to pull this off. That was not our intent. We opted to use the results of the full-text index just like in the set-based approach. This is possible but does have some problems of its own.

Taking a close look at the algorithms above we see that the transitions are determined by the next character. When using the full-text index we don't know what that character is. We propose the following approach. Change the way in which transitions work. In a sense we replace every character by the positions in the text where that character occurs. Suppose we have such an automaton that is at position k in the text. To get the next state we find all outgoing transitions marked with position k.

A side-effect is that we need to change the automaton for every new input string. We will see this is not that hard to do. Consider the following pathological example.

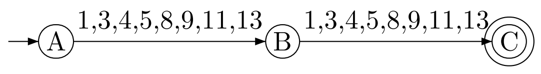


Figure 4.6: Automaton corresponding to the regular expression `aa` and text `abaabbaaababa`.

Example 4.6. Suppose $T =$ `abaabbaaababa` and our regular expression is `aa`. Following the above construction we get the automaton shown in Figure 4.6. Every transition is marked with the positions at which an `a` occurs in the text. Suppose we start this automaton in position 9. We can go from state A to state B, so we are in state B and in position 10. Now we cannot go any further because 10 is not an outgoing transition. If we start the automaton in position 8 we can transfer to B and then on to C since position 9 is a valid transition from B to C.

The above example clearly illustrates how such an automaton would work. However using it in this way is clearly an abuse of the full-text index. Again we don't take advantage of the fact that `aa` occurs less frequent than just an `a`. The lists for the transitions are enormous. Remember the solution we used in the set-based approach: change the basis components of a regular expression to remove concatenation of strings from the regular expressions. As a result we only have strings and operations on these strings in our parse tree. We again take advantage of this fact.

Our translation from regular expressions to NFA's needs to be updated a bit. The automata for the regular expression ϵ and $\emptyset$ as shown in Figure 4.4 remain the same. The last basis component is now not just a single character but a string w. Suppose $|w| = l$. Since we are building the NFA for a given input T we can use the full-text index to find all occurrences $i_1, i_2, \ldots, i_k$ of string w in T. The corresponding automaton is shown in Figure 4.7.

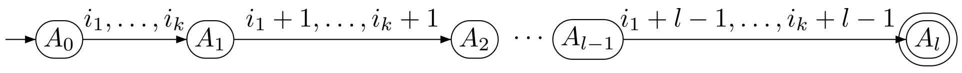


Figure 4.7: Automaton matching string w at positions $i_1, i_2, \ldots, i_k$ in text T

Example 4.7. Continuation of Example 4.6. Using the above definition we can build a better automaton for matching the regular expression `aa`. See Figure 4.8 for the automaton for this regular expression and input text `abaabbaaababa`. Notice the decrease in the amount of transitions.

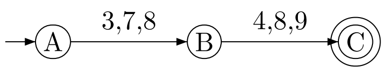


Figure 4.8: Improved automaton corresponding to the regular expression `aa` and text `abaabbaaababa`.

The next theorem summarizes the efficiency of this NFA-implementation.

Theorem 4.8. *The transformation described above will convert a regular expression to an NFA. This conversion will take $O(m+occ)$ where m is the length of the regular expression and occ the sum of the number of occurrences of the various strings in the regular expression. Finding all matches costs $O(mn^2)$ time and $O(m+occ)$ runtime memory.*

Proof. We saw before that converting the regular expression to the corresponding NFA takes $O(m)$ time. This time we also need to find all the occurrences of the substrings. For each substring w of length $|w|$ with occ_w occurrences this takes $O(|w| + occ_w)$ time. Combining all the substrings gives $O(m + occ)$ where occ is the sum of all the occ_w's. This proves the time complexity of $O(m + occ)$ for creating the automaton.

We have to be a bit more careful when simulating this NFA. We require an $O(1)$ operation to check if a transition is possible. This can be done using a hashtable. Storing these hashtables for all the substrings requires $O(occ)$ space. Using these hashtables does not slow down algorithm find_all_matches so we can find them all in $O(mn^2)$ time. $\square$

4.4 Retrieving Matches

Retrieving matches is one aspect of this new implementation that we have not yet considered. The result of both the set-based implementation as well as the NFA-based implemention is a set of indices. For example running find_all_matches with automaton in Example 4.7 gives indices $(3,4), (7,8)$ and $(8,9)$ for the substrings $T[3,4], T[7,8]$ and $T[8,9]$. Retrieving these substrings is not completely trivial when we only have the compressed text available.

Remember that we can retrieve T by stepping through L. Unfortunately we then need to know where to start. Remember that the last character of T is the first character of L, so we do have one starting point. This is not really optimal since to retrieve a substring at the beginning we need to decode almost all of T. We propose the following solution. Store for every position in T of the form $n - \eta i$ the corresponding location in L, for $i = 0, 1, \ldots, \lfloor (n-1)/\eta \rfloor$ and $\eta = \log^{1+\epsilon} n$ for some $\epsilon > 0$. Create a table $\mathcal{M}$ of length $\lfloor (n-1)/\eta \rfloor + 1$ containing these positions. In order to retrieve a match m we can now in η steps through T reach the end of the match m in T. In an additional $|m|$ steps we can find the corresponding string. So retrieving a match costs at most $O(m + \log^{1+\epsilon} n)$.

Chapter 5

Discussion

In Chapter 2 we have seen how Shannon defined the concept of entropy as a measure for the information density of a piece of text. We used this definition in Chapter 3 to give a bound for the compressed Burrows-Wheeler transform of a text. I have implemented the BWT and compression part of the algorithm to get a better grasp of the material. The few tests that I performed showed a compression ratio of about 50%. Ferragina and Manzini claim [8] that the compression is comparable to gzip compression.

Using some additional datastructures we have seen how Ferragina and Manzini manage to build a full-text index with the optimal time bound of $O(p+occ)$ for locating a pattern of length p. Ferragina and Manzini state [8] that the storage space required for the additional datastructures is neglegible. I have not tested this.

In Chapter 4 we have seen two implementations for regular expression engines that profit from the full-text index that we built in the previous chapter. Obviously the NFA-based implementation is more efficient than the set-based implementation, since the latter deals badly with closures. Unfortunately, both implementations suffer from some other problems. When one of the substrings is very short, say one, two or three characters it occurs very often, thereby causing the various sets to be very large. This reduces performance a lot. On the other hand, when dealing with reasonable length substrings both implementations will in practise be a lot faster in finding all matches in a large text than an ordinary engine is.

We should not forget however that these implementations are only useful when the full-text index has already been precomputed. In all other cases it is much easier to just use the existing engines, since they have a smaller overhead.

For testing purposes I build a simple implementation of the set-based engine. It used simple string searching as a full-text index. In the small number of tests that I have performed this implementation worked rather well.

Bibliography

[1] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. New data structures for orthogonal range searching. In *Proceedings of IEEE Symposium on Foundations of Computer Science*, pages 198–207, 2000.

[2] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economic construction of the transitive closure of a directed graph. *Soviet Math. Dokl.*, (11):1209–1210, 1975.

[3] Timothy Bell, Ian H. Witten, and John G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–591, December 1989.

[4] Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, April 1986.

[5] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital SRC Research Report, 1994.

[6] Charles L. A. Clarke and Gordon V. Cormack. On the use of regular expressions for searching text. *ACM Transactions on Programming Languages and Systems*, 19(3):413–426, May 1997.

[7] Russ Cox. Regular expression matching can be simple and fast, January 2007. http://swtch.com/~rsc/regexp/regexp1.html, accessed 10 july 2008.

[8] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–582, July 2005.

[9] Michael T. Goodrich and Robert Tamassia. *Algorithm Design.* John Wiley & Sons, 2002.

[10] John E. Hopcroft, Rajeev Motwani, and Jeffred D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Pearson Education International, 2nd international edition, 2001.

[11] Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, January 1976.

[12] David J.C. MacKay. *Information Theory, Inference, and Learning Algorithms.* Cambridge University Press, 2003.

[13] Gene Myers. A four Russians algorithm for regular expression pattern matching. *Journal of the ACM*, 39(4):430–448, April 1992.

[14] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1988.

[15] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, July and October 1948.

[16] Ken Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, June 1968.

[17] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, September 1978.