CHAPTER

1 Background

1.1 Introduction

1.1.1 Motivation

This work has been motivated by recent advances of molecular genetics. The human genome has been sequenced in 2001. Also mouse, drosophila, etc. Nowadays # reference model genomes are available in genbank.

Next-generation sequencing has been the second revolution. NGS produces billions of reads for 1000$ dollars. Why should one re-sequence a known genome? Resequencing applications include variant calling, etc. So NGS impacts biomedicine.

Given a set of reads, two approaches are possible: assembly and mapping.

Assembly methods are based on overlaps, de brujin graphs, or...

Read mapping methods work on a previously assembled reference genome.

The typical SNPs analysis pipeline 1.1 consists of...

In this work we focus on read mapping, although many core algorithms considered are also applicable to assembly, as well as to later pipeline stages.

Figure 1.1: *NGS pipeline.*

1.1.2 Organization of this manuscript

1.2 Stringology preliminaries

We now introduce fundamental definitions and problems of stringology, in order to keep the manuscript self-contained. The reader familiar with basic stringology can skip this section and proceed to section **??**.

1.2.1 Definitions

Let us start by defining primitive objects of stringology: alphabets and strings. An alphabet is a finite set of symbols (or characters); a string (or word) over an alphabet is a finite sequence of symbols from that alphabet. We denote the length of a string s by $|s|$, and by

ϵ the empty string s.t. $|\epsilon| = 0$. Given an alphabet Σ, we define $\Sigma^0 = \{\epsilon\}$ as the set containing the empty string, Σ^n as the set of all strings over Σ of length n, and $\Sigma^* = \cup_{n=0}^{\infty}\Sigma^n$ as the set of all strings over Σ. Finally, we call any subset of Σ^* a language over Σ.

We now define concatenation, the most fundamental operation on strings. The concatenation operator of two strings is denoted with $\cdot$ and defined as $\cdot : \Sigma^* \times \Sigma^* \to \Sigma^*$. Given two strings, $x \in \Sigma^m$ with $x = x_1x_2 \dots x_m$, and $y \in \Sigma^n$ with $y = y_1y_2 \dots y_n$, their concatenation $x \cdot y$ (or simply denoted xy) is the string $z \in \Sigma^{m+n}$ consisting of the symbols $x_1x_2 \dots x_my_1y_2 \dots y_n$.

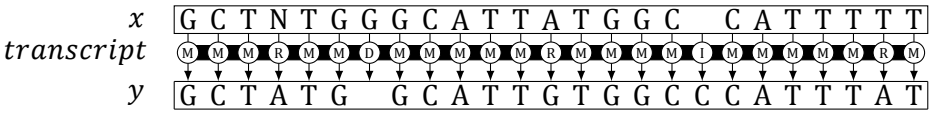
From concatenation we can derive the notion of prefix, suffix, and substring. A string x is a prefix of y iff there is some string z s.t. $y = x \cdot z$. Analogously, x is a suffix of y iff there is some string z s.t. $y = z \cdot x$. Moreover, x is a substring of y iff there is some string w, z s.t. $y = w \cdot x \cdot z$, and then we say that x occurs within y at position $|w|$.

Example 1.1. These definitions allow us to model basic biological sequences. Let us consider the alphabet consisting of DNA bases: $\Sigma = \{A, C, G, T\}$. Examples of strings over Σ are x =A, y =AGGTAC, z =TA. For instance, $y \in \Sigma^6$ and $|y| = 6$. Moreover, the concatenation $x \cdot z$ produces ATA. The string x is a prefix of y, and the string z is a substring of y occurring at position 4 in y.

1.2.2 Alignments

The next step is to define the minimal set of edit operations to transform one string into another: substitutions, insertions and deletions. Given two strings x, y of equal length n, the string x can be transformed into the string y by substituting (or replacing) all symbols x_i s.t. $x_i \neq y_i$ into y_i, for $1 \leq i \leq n$. If the given strings have different lengths, insertion and deletion of symbols from x become necessary to transform it into y. Therefore, given any two strings x, y, we define as edit transcript for x, y any finite sequence of substitutions, insertions and deletions transforming x into y. See Figure 1.2 for an example.

Figure 1.2: *Example of edit transcript transforming the string $x = AAAA$ into $y = CCCC$. The transcript character M indicates a match, R a replacement, I an insertion, and D a deletion.*

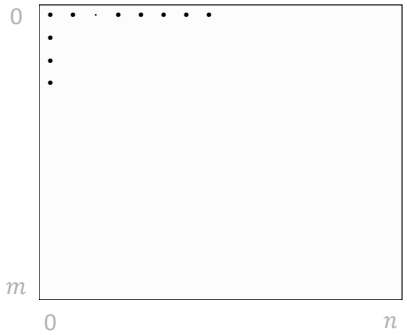


An alignment is an alternative yet equivalent way of visualizing a transformation between strings. While an edit transcript provides an explicit sequence of edit operations transforming one string into another, an alignment relates pairs of corresponding symbols between two strings. Because some symbols in one string are not related to any symbol in the other string, i.e. some symbols are inserted or removed, we first need to introduce a gap symbol $-$, which is not part of the string alphabet Σ. Subsequently, we can define the alignment of two strings of length m, n over Σ to be a string of length between $\min\{m, n\}$ and $m + n$ over the pair alphabet $(\Sigma \cup \{-\}) \times (\Sigma \cup \{-\})$.

Example 1.2. An alignment of the strings $x = AAAA$ and $y = CCCC$ is given by the string $z = \binom{A}{A}\binom{A}{-}\binom{A}{C}\binom{G}{G}$

A dotplot is a way to visualize any alignment between two strings and highlight their similarities. Given two string x, y of length m, n, a dotplot is a $m \times n$ matrix containing a dot at position (i, j) iff the symbol x_i matches symbol y_j. We define a dotplot trace to be a monotonical path in the matrix connecting non-decreasing positions of the matrix. A dotplot trace corresponds to an alignment and vice versa. In a trace, match and mismatch columns of the corresponding alignment appear as diagonal stretches, while insertions and deletions are horizontal or vertical stretches. See Figure **??**.

Figure 1.3: *Example of dotplot of the strings $x = AAAA$ and $y = CCCC$. The highlighted trace corresponds to the alignment of Example 1.2.*



1.2.3 Distance functions

We can assign a cost to any alignment and to its associated edit transformation by defining a weight function $\omega : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \to \mathbb{R}$, where:

- $\omega(\alpha, \beta)$ for all $(\alpha, \beta) \in \Sigma \times \Sigma$ defines the cost of substituting α with β,
- $\omega(\alpha, -)$ for all $\alpha \in \Sigma$ defines the cost of deleting the symbol α,
- $\omega(-, \beta)$ for all $\beta \in \Sigma$ defines the cost of inserting the symbol β,

and by defining the total cost $C(z)$ of an alignment z between two strings as the sum of the weights of all its alignment symbols:

$$C(z) = \sum_{i=0}^{|z|} \omega(z_i) \tag{1.1}$$

Consequently, we can define the distance function $d : \Sigma^* \times \Sigma^* \to \mathbb{R}$ by taking the minimum cost over all possible alignments of x, y:

$$d(x, y) = \sum_{z \in \mathbb{A}(x,y)} C(z) \tag{1.2}$$

In particular, the edit or *Levenshtein distance* between two strings $x, y \in \Sigma^*$ is defined as the function $d_E : \Sigma^* \times \Sigma^* \to \mathbb{N}$ counting the *minimum* number of edit operation necessary to transform x into y. It is obtained by defining for all $(\alpha, \beta) \in \Sigma \times \Sigma$, $\omega(\alpha, \beta) = 1$ iff $\alpha \neq \beta$ and 0 otherwise, and $\omega(\alpha, -)$ and $\omega(-, \beta)$ as 1. The *Hamming distance* between two strings $x, y \in \Sigma^n$ is defined as the function $d_H : \Sigma^n \times \Sigma^n \to \mathbb{N}$ counting the number of substitutions necessary to transform x into y. We obtain it by defining $\omega(\alpha, \beta)$ as in the edit distance, and by setting all $\omega(\alpha, -)$ and $\omega(-, \beta)$ to be ∞ in order to disallow indels.

Example 1.3. TODO: example of edit and hamming distance.

1.2.4 Optimal alignments

The problem of finding an optimal alignment between two strings is equivalent to the problem of finding their minimum distance [**?**]. A solution to this optimization problem can be efficiently computed via dynamic programming (DP). Below we describe the three essential components of the DP approach: the recurrence relation, the DP table, and the traceback.

Given two strings x, y of length m, n, for all $1 \leq i \leq m$ and $1 \leq j \leq n$ we define with $d(x_{1..i}, y_{1..j})$ the distance between their prefixes $x_{1..i}$ and $y_{1..j}$. The base conditions of the recurrence relation are:

$$d(\epsilon, \epsilon) = 0 \tag{1.3}$$

$$d(x_{1..i}, \epsilon) = \sum_{l=1}^{i} \omega(x_l, -) \text{ for all } 1 \leq i \leq m \tag{1.4}$$

$$d(\epsilon, y_{1..j}) = \sum_{l=1}^{j} \omega(-, y_l) \text{ for all } 1 \leq j \leq n \tag{1.5}$$

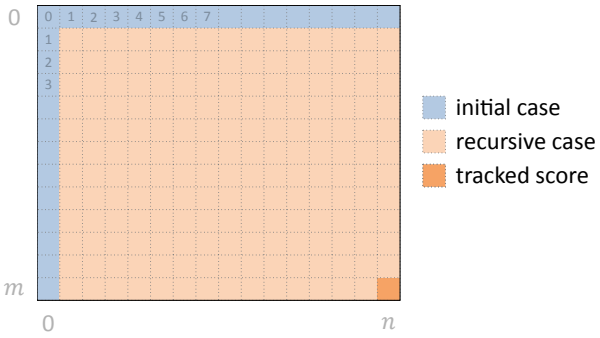
and the recursive case is defined as follows:

$$d(x_{1..i}, y_{1..j}) = \min \begin{cases} d(x_{1..i-1}, y_{1..j}) & + \quad \omega(x_i, -) \\ d(x_{1..i}, y_{1..j-1}) & + \quad \omega(-, y_j) \\ d(x_{1..i-1}, y_{1..j-1}) & + \quad \omega(x_i, y_j) \end{cases} \tag{1.6}$$

We can compute the above recurrence relation in time $\mathcal{O}(nm)$ using a dynamic programming table D of $(m+1)\times(n+1)$ cells, where cell $D[i, j]$ stores the value of $d(x_{1..i}, y_{1..j})$. The sole distance without any alignment can be computed in space $\mathcal{O}(\min\{n, m\})$, as we only need column $D[: j-1]$ to compute column $D[: j]$ (or row $D[i-1 :]$ to compute $D[i :]$), and we can fill the table D either column-wise or row-wise[1]. An optimal alignment can be computed in time $\mathcal{O}(m+n)$ via *traceback* on the table D: We start in the cell $D[m, n]$ and go backwards (either left, up-left, or up) to the previous cell by deciding which condition of Equation 1.6 yielded the value of $D[m, n]$.

[1] Note that D can be filled also diagonal-wise or antidiagonal-wise.

Figure 1.4: *DP table representing the computation of the edit distance* $d_E(x_{1..5}, y_{1..4})$.



1.3 Overview of string matching

1.3.1 Problem definition

We can now define *exact string matching*, perhaps the most fundamental problem in stringology. Given a string p (with $|p| = m$) called the *pattern* and a longer string t (with $|t| = n$) called the *text*, the exact string matching problem is to find all occurrences, if any, of pattern p into text t [?]. This problem has been extensively studied from the theoretical standpoint and is well solved in practice.

The definition of distance functions between strings let us generalize exact string matching into a more challenging problem: *approximate string matching*. Given a text t, a pattern p, and a *distance threshold* $k \in \mathbb{N}$, the approximate string matching (a.s.m.) problem is to find all occurrences of p into t within distance k. The a.s.m. problem under the Hamming distance is commonly referred as the *k-mismatches* problem and under the edit distance as the *k-differences* problem. For k-mismatches and k-differences, it must hold $k > 0$ as the case $k = 0$ corresponds to exact string matching, and $k < m$ as a pattern occurs at any position in the text if we substitute all its m characters. Under these distances, we define the *error rate* as $\epsilon = \frac{k}{m}$, with $0 < \epsilon < 1$ given the above conditions, and we alternatively refer to these a.s.m. problems as ϵ-mismatches and ϵ-differences.

We can classify string matching problems in two categories, *online* and *offline*, depending on which string, the pattern or the text, is given first. Algorithms for online string matching work by preprocessing the pattern and scanning the text from left to right (or right to left); their worst-case runtime complexity ranges from $\mathcal{O}(nm)$ to $\mathcal{O}(n)$ while their worst-case memory complexity ranges from $\mathcal{O}(\sigma^k m^k)$ to $\mathcal{O}(m)$. Algorithms for offline string matching are instead allowed to preprocess the text; their worst-case runtime complexity ranges from $\mathcal{O}(m)$ to $\mathcal{O}(\sigma^k m^k)$ while their worst-case memory complexity is usually $\mathcal{O}(n)$. In practice, if the text is long, static and searched frequently, offline methods largely outperform online methods in terms of runtime, provided the necessary amount of memory. Therefore, we concentrate on offline algorithms.

We can subdivide algorithms for offline string matching in two categories: *fully-indexed* and *filtering*. Fully-indexed algorithms work solely on the index of the text, while filtering

methods first use the index to discard uninteresting portions of the text and subsequently use an online method to verify narrow areas of the text. Filtering methods outperform fully-indexed methods for a vast range of inputs[2] and are thus very interesting from a practical standpoint. Nonetheless, filtering methods are just opportunistic combinations of online and fully-indexed methods.

In the following of this section we thus give a brief and non-exhaustive overview of the fundamental techniques for online and (both fully-indexed and filtering) offline string matching. This overview serves as an introduction to the more involved algorithms presented in chapter **??** and directly used in applications of part **??**. For an extensive treatment of the subject, we refer the reader to complete surveys on exact [**?**] and approximate [**?**] online string matching methods, as well as to a succint survey on indexed methods [**?**].

1.3.2 Online methods

We consider two classes of algorithms for online string matching, those based on automata and those based on dynamic programming.

Automata

Exact search of one pattern. Knuth-Morris-Pratt automaton.

Exact search of multiple patterns. Aho-corasick automaton.

Approximate search of one pattern. Ukkonen automaton.

Dynamic programming

The dynamic programming algorithm **??** to compute the distance of two strings can be easily turned into a string matching algorithm. Since an occurrence of the pattern can start and end anywhere in the text, a.s.m. consists of computing the edit distance between the pattern and all substrings of the text. The problem can be thus solved by computing the edit distance between the text and the pattern without penalizing leading and trailing deletions in the text.

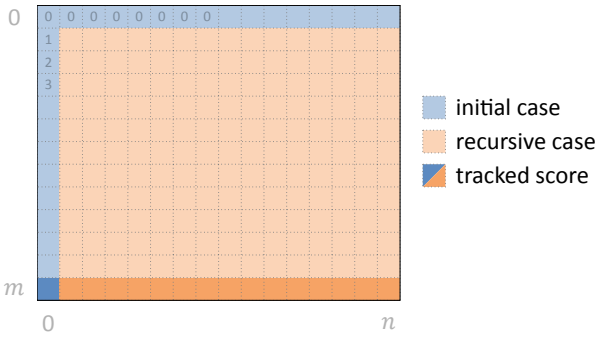
We pose $x = p$ and $y = t$ and consider Equation 1.6. Since an occurrence of the pattern can start anywhere in the text, we change the initialization of the top row $D[0 :]$ of the DP matrix according to the condition:

$$d(\epsilon, y_{1..j}) = 0 \text{ for all } 1 \leq j \leq n \tag{1.7}$$

and since an occurrence of the pattern can end anywhere in the text, we check all cells $D[m, j]$ for all $1 \leq j \leq n$ in the bottom row of D for the condition $D[m, j] \leq k$.

[2] When the error rate is low.

Figure 1.5: *DP table representing the match of $p = ...$ in $t = ...$*



1.3.3 Indexed methods

No matter how efficient online methods can be, these approaches quickly become impractical when the text is long and searched frequently. If the text is static and given in advance, we are allowed to preprocess it. Therefore we build an index of the text beforehand and use it to speed up subsequent searches. To this intent we introduce the *suffix tree*, an optimal data structure to index all substrings of a text. We take the suffix tree as an idealized data structure to elegantly expose our indexed algorithms solving string matching problems. In chapter **??** we will consider other substring indices to replace the suffix tree in practice.

Suffix tree and suffix trie

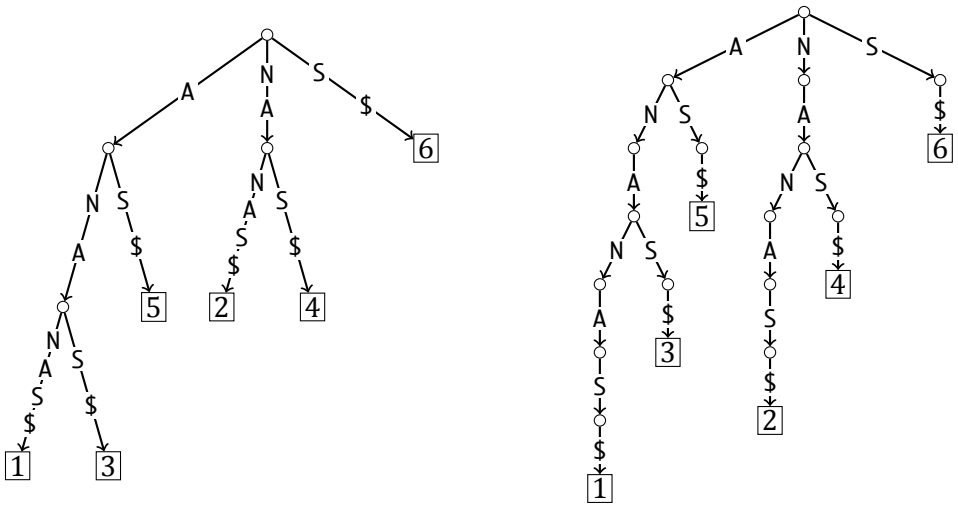
The suffix tree [**?**] is a lexicographically ordered tree data structure representing all suffixes of a string. Assume w.l.o.g. a string s of length n, padded with a *terminator symbol* \$ not being part of the string alphabet Σ[3]. The suffix tree $\mathcal{S}$ of the string s has one node designated as the root and n leaves, each one pointing to a distinct suffix of s, denoted as $l_1 ... l_n$. Each internal node has more than one child, and each edge is labeled with a non-empty substring of s. Each path from the root to a leaf l_i spells the suffix $s_{i..n}$. Figure 1.6 illustrates.

In the following of this manuscript we consider w.l.o.g. *suffix tries* instead of suffix trees. On suffix tries, internal nodes can have only one child and each edge is labeled by one single character (see Figure **??**). This fact simplifies the exposition of all given algorithms without affecting their runtime complexity nor their result. However, we remark that all given algorithms can be generalized to work on trees.

Therefore, from now on we assume the text t to be indexed using a suffix trie $\mathcal{T}$. Given a node x of $\mathcal{T}$, we denote with $label(x)$ the label of the edge entering into x, with $\mathbb{C}(x)$ the set of children of x being internal nodes, with $\mathbb{E}(x)$ the set of children of x being leaves, and with $\mathbb{L}(x)$ the set of all leaves of the subtree rooted in x. We remark that entering edges of internal nodes in $\mathbb{C}(x)$ are always labeled with symbols in Σ, while entering edges

[3] The terminator symbol is necessary to ensure that no suffix $s_{i..n}$ is a prefix of another suffix $s_{j..n}$.

Figure 1.6: *Suffix tree and suffix trie for the string ANANAS.*



of leaves in $\mathbb{L}(x)$ and $\mathbb{E}(x)$ are always labeled with terminator symbols.

Exact string matching

Using the suffix trie $\mathcal{T}$ of the text t, we can find all occurrences of a pattern p into t in optimal time $\mathcal{O}(m)$, thus independently of n. Algorithm 1.1 searches a pattern p by starting in the root node of $\mathcal{T}$ and following the path spelling the pattern. If the search ends up in a node x, each leaf $l_i \in \mathbb{L}(x)$ points to a distinct suffix $t_{i..n}$ such that $t_{i..i+m} = p$. Algorithm 1.1 is correct since each path from the root to any internal node of the suffix trie $\mathcal{T}$ spells a different unique substring of t; consequently all equal substrings of t are represented by a single common path.

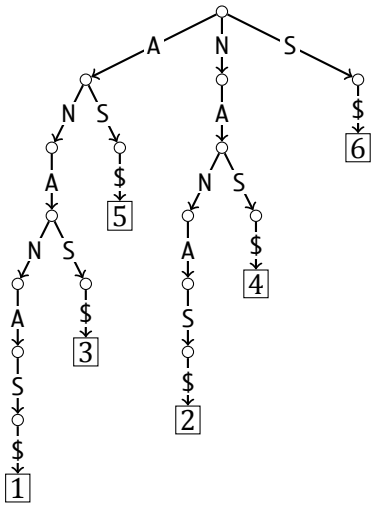
Algorithm 1.1 Exact string matching on a suffix trie.

```
1: procedure ExactSearch(x, p)
2:     if p = ϵ then
3:         report 𝕃(x)
4:     else if ∃ c_x ∈ ℂ(x) : label(c_x) = p_1 then
5:         ExactSearch(c_x, p_{2..|p|})
6:     end if
7: end procedure
```

Backtracking k-mismatches

We can solve k-mismatches by backtracking [**??**] on the suffix trie $\mathcal{T}$, in average time sublinear in n [**?**]. A top-down traversal on $\mathcal{T}$ spells incrementally all distinct substrings of t. While traversing each branch of the trie, we incrementally compute the distance between the query and the spelled string. If the computed distance exceeds k, we stop

Figure 1.7: *Exact string matching on a suffix tree. The pattern NA is searched exactly in the text ANANAS.*



the traversal and proceed on the next branch. Conversely, if we completely spelled the pattern p, and we ended up in a node x, each leaf $l_i \in \mathbb{L}(x)$ points to a distinct suffix $t_{i..n}$ such that $d_H(t_{i..i+m}, p) \leq k$. See algorithm 1.2.

Algorithm 1.2 k-mismatches on a suffix trie.

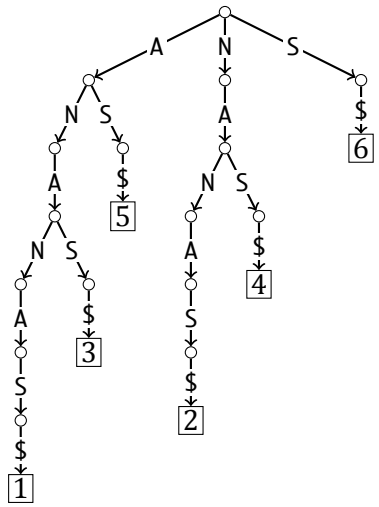
```
1:  procedure KMismatches(x, p, e)
2:      if e = 0 then
3:          ExactSearch(x, p)
4:      else
5:          for all c_x ∈ 𝕔(x) do
6:              if label(c_x) = p_1 then
7:                  KMismatches(c_x, p_{2..|p|}, e)
8:              else
9:                  KMismatches(c_x, p_{2..|p|}, e − 1)
10:             end if
11:     end if
12: end procedure
```

Backtracking k-differences

We can compute k-differences on a suffix tree in two different ways. Algorithm 1.3 explicitly enumerates errors by recursing on the suffix trie. Algorithm 1.4 computes the edit distance on the suffix trie.

In algorithm 1.4, any online method can be used to compute the edit distance. However, for theoretical considerations, it is important to consider an algorithm wich com-

Figure 1.8: *Approximate string matching on a suffix tree.*



Algorithm 1.3 k-differences on a suffix trie.

```
1:  procedure KDifferences(x, p, e)
2:      if e = 0 then
3:          ExactSearch(x, p)
4:      else
5:          KDifferences(x, p_{2..|p|}, e − 1)
6:          for all c_x ∈ ℂ(x) do
7:              KDifferences(c_x, p, e − 1)
8:              if label(c_x) = p_1 then
9:                  KDifferences(c_x, p_{2..|p|}, e)
10:             else
11:                 KDifferences(c_x, p_{2..|p|}, e − 1)
12:             end if
13:     end if
14: end procedure
```

putes in $\mathcal{O}(1)$ per node. Note that it is sufficient to have an algorithm capable of checking whether the current edit distance is within the imposed threshold k.

Algorithm 1.3 reports more occurrences than algorithm 1.4. Discuss neighborhood, condensed neighborhood, and super-condensed neighborhood.

1.3.4 Filtering methods

The goal of filtering methods is to obtain algorithms that have favorable average running times. The principle under which they work is that large and uninteresting portions of the text can be quickly discarded, while narrow and highly similar portions can be verified with a conventional online method. We remark that any filtering method can either

Algorithm 1.4 k-difference on a suffix trie.

```
1: procedure KDIFFERENCES(x, p, e)
2:     for all c_x ∈ ℂ(x) do
3:         KDIFFERENCES(c_x, p_{2..|p|}, e − d_E(repr(x), p))
4: end procedure
```

work in a online fashion or take advantage of an index of the text to speed up the filtration phase. Here we always consider the filtration phase to be indexed.

Filtering methods work under the assumption that given patterns occurs in the text with a *low average probability*. Such probability is a function of the error rate ϵ, in addition to the alphabet size σ, and can be computed or estimated under the assumption of the text being generated by a specific random source. Under the uniform Bernoulli model, where each symbol of Σ occurs with probability $\frac{1}{\sigma}$, **?** estimates that $\epsilon < 1 - \frac{1}{\sigma}$ is a conservative bound on the error rate which ensures few matches, and for which filtering algorithms are effective. For higher error rates, non-filtering online and indexed methods work better.

We call a filter *lossless* or *full-sensitive* if it guarantees not to discard any occurrence of the pattern, otherwise we call it *lossy*. Lossy filters can be designed to solve approximately ϵ-differences. We focus our attention on lossless filters.

We now consider two classes of filtering methods: those based on *q-grams* and those based on *seeds*. Filters based on q-grams consider all *overlapping* substrings of the pattern having length q, the so-called q-grams. Eventually, simple lemmas gives us lower bounds on the number of q-grams that must be present in a narrow window of the text as necessary condition for an occurrence of the pattern. Instead, seed based filters partition the pattern into *non-overlapping* factors called seeds. We can derive full-sensitive partitioning strategies by application of the pigeonhole principle.

q-Gram filters

q-Gram filters are based on the so-called q-gram similarity measure $\tau_q : \Sigma^* \times \Sigma^* \to \mathbb{N}$, defined as the number of substrings of length q common to two given strings. The following lemma relates the q-gram similarity to edit distance[4]. It gives a lower bound on the q-gram similarity $\tau_q(x, y)$ for any two strings having $d_E(x, y) = k$. This means that $\tau_q(x, y) \geq k$ is a necessary but not sufficient condition for $d_E(x, y) \leq k$.

Lemma 1.1. *[?] Let x, y be two strings with edit distance k and $min\{|x|, |y|\} = m$, then x and y have q-gram similarity $\tau_q(m, k) \geq m - q + 1 - kq$.*

Proof. By induction on k. □

Example 1.4.

[4] Thus also to Hamming distance.

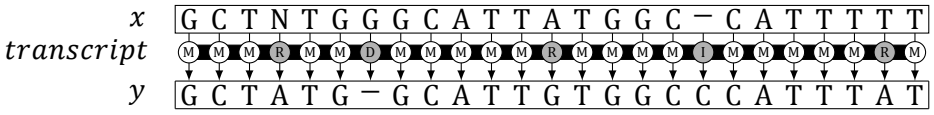
How can we use this result to solve approximate string matching? The lemma itself does not give us the direct solution, indeed it considers the edit distance between two arbitrary strings, while in a.s.m. the pattern can match any substring of the text. In the case of Hamming distance, if the pattern matches any substring s of t, then s must have length m. In the case of edit distance, it must hold $m - k \leq |s| \leq m + k$. The dot-plot representation helps us to visualize this concept. Hamming distance occurrences cover one single diagonal of the dot-plot, while edit distance occurrences are enclosed inside a parallelogram of side $2k + 1$.

Overlapping parallelograms?

We can design a filtration algorithm that scans the text and counts how many q-grams of the pattern falls into each parallelograms. Only the parallelograms exceeding the threshold $\tau_q(m, k)$ have to be verified with an online method, e.g. standard DP. To speed up the filtration phase, the q-grams can be counted with the help of a substring index.

Which length of q makes filtration lossless? $q = \left\lfloor \frac{m}{k+1} \right\rfloor$.

Figure 1.9: *Filtration with q-grams.*



Seed filters

Again, we start by considering the case of two arbitrary strings x, y s.t. $d_E(x, y) \leq k$. If we partition w.l.o.g. y into $k + 1$ non-overlapping seeds, then at least one seed y^i will occur as a factor of x.

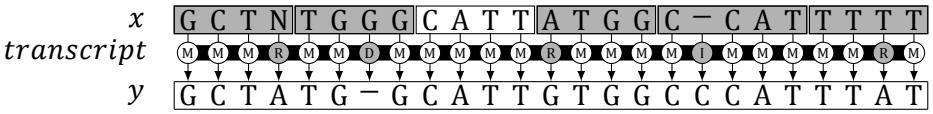
Lemma 1.2. *[?] If $y = y^1 y^2 \ldots y^{k+1}$ then $x = a y^i b$ for some a, b.*

Proof. Again, by induction on k? □

Thus, we solve k-differences by partitioning p into $k + 1$ seeds and searching them into t, e.g. with the help of a substring index. Note that we are reducing one approximate search into many smaller exact searches. Again, this lemma gives us a necessary but not sufficient condition, consequently we must verify whether any candidate location induced by an occurrence of some seed corresponds to an approximate occurrence of p in t. Thus, we verify any substring s of the text of length $m - k \leq |s| \leq m + k$ containing one seed of p.

Which length of q makes filtration lossless?

Figure 1.10: *Filtration with exact seeds.*



Analysis of filters

1.4 Related problems

1.4.1 Dictionary search

Dictionary search is a restriction of string matching. Given a set of database strings $\mathbb{D}$ and a query string q, the approximate dictionary search problem is to find all strings in $\mathbb{D}$ within distance k from q. Note that usually the query string q has length similar to strings in $\mathbb{D}$, as $||d| - |q|| \leq k$ is a necessary condition for $d_E(d, q) \leq k$.

Online methods

The problem can be solved by checking whether $d_E(d, q) \leq k$ for all $d \in \mathbb{D}$. Answering the question whether the distance $d_E(d, q) \leq k$ is an easier problem than computing the edit distance $d_E(d, q)$: a band of size $k + 1$ is sufficient.

Lemma 1.3. *The k-differences global alignment problem can be solved by computing only a diagonal band of the DP matrix of width $k+1$, where the leftmost band diagonal is $\lfloor \frac{m-n+k}{2} \rfloor$ cells left of the main diagonal (see Figure* **??***).*

Proof. Indirect. Assume that a cell outside the band is part of a global alignment with at most k errors. If the cell is left of the band, the traceback that starts in the top left corner would go down at least $c = \lfloor \frac{m-n+k}{2} \rfloor + 1$ cells. Then it needs to go right at least $n - m + c$ cells to end in the bottom right corner. Hence it contains at least $n - m + 2c > n - m + 2\frac{m-n+k}{2} = k$ errors. The assumption that the cell is right of the band can be falsified analogously. □

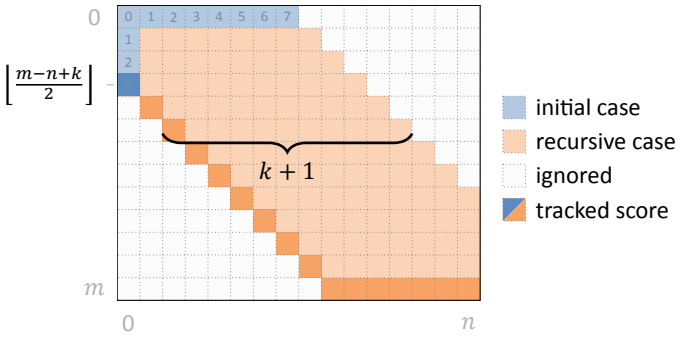
Indexed methods

Using a radix tree $\mathcal{D}$ we can find all strings in $\mathbb{D}$ equal to a query string q, in optimal time $\mathcal{O}(|q|)$ and independently of $||\mathbb{D}||$.

Filtering methods

Filtering methods of section 1.3.4 can be directly applied to solve the dictionary search problem. Database strings satisfying the filtering condition can be verified with algorithm **??**.

Figure 1.11: *DP table representing the match of $p = ...$ in $t = ...$*



Algorithm 1.5 Exact dictionary search on a radix trie.

```
1: procedure ExactSearch(x, p)
2:     if p = ϵ then
3:         report 𝔼(x)
4:     else if ∃ c_x ∈ ℂ(x) : label(c_x) = p_1 then
5:         ExactSearch(c_x, p_{2..|p|})
6:     end if
7: end procedure
```

1.4.2 Local similarity search

Define score and scoring scheme.
Define local similarity.

Online methods

Give dynamic programming solution.

Indexed methods

Backtracking over substring index. BWT-SW.

Filtering methods

SWIFT/Stellar is based on the q-gram lemma.

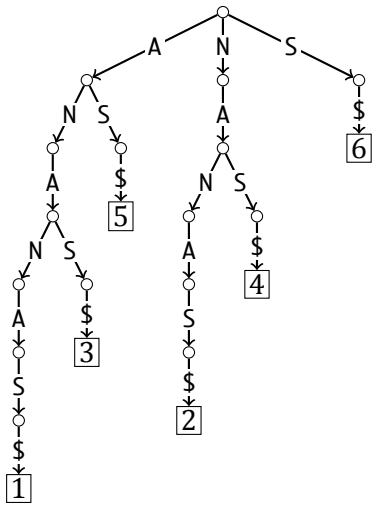
1.4.3 Overlaps computation

Define problem.

Online methods

DP solution.

Figure 1.12: *Exact dictionary search on a suffix trie.*



Indexed methods

Indexed solution, exact and approximate.