6

Approximate matching

6.1 Basic concepts

Approximate string matching, also called "string matching allowing errors," is the problem of finding a pattern p in a text T when a limited number k of *differences* is permitted between the pattern and its occurrences in the text.

From the many existing models defining a "difference," we focus on the most popular one, called *Levenshtein distance* or *edit distance* [Lev65]. Other more complex models exist, especially in computational biology, but the edit distance model has received the most attention and the most effective algorithms have been developed for it. Some of these algorithms can be extended to more complex models.

Under edit distance, one difference equals one *edit operation*: a character insertion, deletion, or substitution. That is, the edit distance between two strings x and y, $ed(x, y)$, is the minimum number of edit operations required to convert x into y, or vice versa. For example, $ed(\texttt{annual,annealing}) = 4$. The approximate string matching problem becomes that of finding all occurrences in T of every p' that satisfies $ed(p, p') \leq k$. To ensure a linear size output it is customary to report only the starting or ending positions of the occurrences.

Note that the problem only makes sense for $0 < k < m$, because otherwise every text substring of length m can be converted into p by substituting the m characters. The case $k = 0$ corresponds to exact string matching. We call $\alpha = k/m$ the "error level." It gives a measure of the "fraction" of the pattern that can be altered.

We concentrate on algorithms that are the fastest in the cases that are likely to be of use in some foreseeable application, particularly text retrieval and computational biology. In particular, $\alpha < 1/2$ in most cases of interest.

We present four approaches. The first approach, which is also the oldest

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

and most flexible, adapts a dynamic programming algorithm that computes edit distance. The second uses an automaton-based formulation of the problem and deals with the ways to simulate the automaton. The third, one of the most successful approaches, is based on the bit-parallel simulation of other approaches. Finally, the fourth approach uses a simple necessary condition to filter out large text areas and another algorithm to search the areas that cannot be discarded. Filtration is the most successful approach for low error levels.

6.2 Dynamic programming algorithms

The oldest solution to the problem relies on dynamic programming. Discovered and rediscovered many times since the 1960s, the final search algorithm is attributed to Sellers [Sel80]. Although the algorithm is not very efficient, taking $O(mn)$ time, it is among the most adaptable to more complex distance functions.

We first show how to compute the edit distance between two strings. Later, we extend the algorithm to search for a pattern in a text allowing errors. We then show how this algorithm can be made faster on average. Finally, we discuss alternative algorithms based on dynamic programming.

6.2.1 Computing edit distance

We need to compute $ed(x, y)$. A matrix $M_{0\ldots|x|,0\ldots|y|}$ is filled, where $M_{i,j}$ represents the minimum number of edit operations needed to match $x_{1\ldots i}$ to $y_{1\ldots j}$, that is, $M_{i,j} = ed(x_{1\ldots i}, y_{1\ldots j})$. This is computed as follows:

$$\begin{aligned} M_{0,0} &\leftarrow 0 \\ M_{i,j} &\leftarrow \min(M_{i-1,j-1} + \delta(x_i, y_j), M_{i-1,j} + 1, M_{i,j-1} + 1) \end{aligned}$$

where $\delta(a, b) = 0$ if $a = b$ and 1 otherwise, and M is assumed to take the value ∞ when accessed outside its bounds. At the end, $M_{|x|,|y|} = ed(x, y)$.

The rationale of the formula is as follows. $M_{0,0}$ is the edit distance between two empty strings. For two strings of length i and j, we assume inductively that all the edit distances between shorter strings have already been computed, and try to convert $x_{1\ldots i}$ into $y_{1\ldots j}$.

Consider the last characters x_i and y_j. If they are equal, we do not need to consider them, just convert $x_{1\ldots i-1}$ into $y_{1\ldots j-1}$ at a cost $M_{i-1,j-1}$. On the other hand, if they are not equal, we must deal with them. Following the three allowed operations, we can substitute x_i by y_j and convert $x_{1\ldots i-1}$ into $y_{1\ldots j-1}$ at a cost $M_{i-1,j-1}+1$, delete x_i and convert $x_{1\ldots i-1}$ into $y_{1\ldots j}$ at

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

a cost $M_{i-1,j}+1$, or insert y_j at the end of $x_{1\ldots i}$ and convert $x_{1\ldots i}$ into $y_{1\ldots j-1}$ at a cost $M_{i,j-1}+1$. Note that the insertions in one string are equivalent to deletions in the other.

Therefore, the algorithm is $O(|x||y|)$ time in the worst and average cases. An alternative formulation that yields faster coding is as follows:

$$M_{i,0} \leftarrow i, \qquad M_{0,j} \leftarrow j \tag{6.1}$$

$$M_{i,j} \leftarrow \begin{cases} M_{i-1,j-1} & \text{if } x_i = y_j \ , \\ 1 + \min(M_{i-1,j-1}, M_{i-1,j}, M_{i,j-1}) & \text{otherwise} \end{cases}$$

which is equivalent to the previous one because neighboring cells in M differ at most by 1. Therefore, when $\delta(x_i, y_j) = 0$, we have that $M_{i-1,j-1}$ cannot be larger than $M_{i-1,j}+1$ or $M_{i,j-1}+1$.

From the matrix it is possible to determine an *optimal path*, that is, a minimum cost sequence of matrix cells that goes from cell $M_{0,0}$ to $M_{|x|,|y|}$. Multiple paths may exist. Each path is related to an *alignment*, which is a mapping between the characters of x and y that shows how characters should be matched, substituted, and deleted to make x and y equal. A complete reference on alignments is [Gus97].

Figure 6.1 illustrates the algorithm to compute ed(`annual`, `annealing`).

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	**0**	1	2	3	4	5	6	7	8
n	2	1	**0**	1	2	3	4	5	6	7
n	3	2	1	**0**	1	2	3	4	5	6
u	4	3	2	1	**1**	2	3	4	5	6
a	5	4	3	2	2	**1**	2	3	4	5
l	6	5	4	3	3	2	**1**	**2**	**3**	**4**

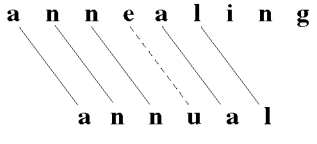

Fig. 6.1. Example of the dynamic programming algorithm to compute the edit distance between `"annual"` and `"annealing"`. The path in bold yields the only optimal alignment. On the right we show the alignment, where the dashed line means a substitution.

6.2.2 Text searching

Searching a pattern p in a text T is basically similar to computing edit distance, with $x = p$ and $y = T$. The only difference is that we must allow an occurrence to begin at any text position. This is achieved by setting $M_{0,j} = 0$ for all $j \in 0 \ldots n$. That is, the empty pattern occurs with zero errors at any text position because it matches with a text substring of length zero.

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

The resulting algorithm needs $O(mn)$ time. If we use a matrix M, it also needs $O(mn)$ space. However, we can work with just $O(m)$ space. The key observation is that to compute $M_{*,j}$ we only need the values of $M_{*,j-1}$. Therefore, instead of building the whole matrix M, we process T character by character and maintain a column C of M, which is updated after reading each new text position j to keep the invariant $C_i = M_{i,j}$.

The algorithm initializes its column $C_{0...m}$ with the values $C_i \leftarrow i$ and processes the text character by character. At each new text character t_j, its column vector is updated to $C'_{0...m}$. The update formula is

$$C'_i \leftarrow \begin{cases} C_{i-1} & \text{if } p_i = t_j \ , \\ 1 + \min(C_{i-1}, C'_{i-1}, C_i) & \text{otherwise} \end{cases}$$

and the text positions where $C_m \leq k$ are reported as ending positions of occurrences. Observe that since $C = M_{*,j-1}$ is the old column and $C' = M_{*,j}$ is the new one, C_{i-1} corresponds to $M_{i-1,j-1}$, C'_{i-1} to $M_{i-1,j}$ and C_i to $M_{i,j-1}$ in formula (6.1).

Figure 6.2 applies this algorithm to search for the pattern `"annual"` in the text `"annealing"` with at most $k = 2$ errors. In this case there are three occurrences.

		a	n	n	e	a	l	i	n	g
	0	0	0	0	0	0	0	0	0	0
a	1	0	1	1	1	0	1	1	1	1
n	2	1	0	1	2	1	1	2	1	2
n	3	2	1	0	1	2	2	2	2	2
u	4	3	2	1	1	2	3	3	3	3
a	5	4	3	2	2	1	2	3	4	4
l	6	5	4	3	3	**2**	**1**	**2**	3	4

Fig. 6.2. Example of the dynamic programming algorithm to search for `"annual"` in the text `"annealing"` with two errors. Each column of this matrix is a value of the column C at some point in time. Bold entries indicate ending positions of occurrences in the text.

6.2.3 Improving the average case

A simple twist to the dynamic programming algorithm [Ukk85], which retains all its flexibility, takes $O(kn)$ time on average [CL92, BYN99]. We call it **DP**. The idea is that, since a pattern does not normally match in the text, the values at each column read from top to bottom quickly reach $k+1$ (i.e., mismatch), and that if a cell has a value larger than $k+1$, the result of the search does not depend on its exact value. A cell is called *active* if

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

its value is at most k. The algorithm keeps count of the last active cell and avoids working on subsequent cells.

The last active cell must be recomputed for each new column. When moving from one text position to the next, the last active cell can be incremented by at most one since neighbors in M differ by at most one, so we check in constant time whether we have activated the next cell. However, it is also possible that the formerly last active cell becomes inactive now. In this case we have to search upwards in the column for the new last active cell. Although we can work $O(m)$ at a given column, we cannot work more than $O(n)$ overall, because there are at most n increments of this value in the whole process, and hence there are no more than n decrements. So, the last active cell is maintained at $O(1)$ worst-case amortized cost per column.

Figure 6.3 shows pseudo-code for this algorithm. Its basic idea of avoiding to compute some inactive cells has been used extensively in other algorithms. In particular, the bit-parallel algorithms that we cover later profit from this technique to reduce their average search time.

```
DP (p = p_1 p_2 ... p_m, T = t_1 t_2 ... t_n, k)
1.    Preprocessing
2.        For i ∈ 0...m Do C_i ← i
3.        lact ← k + 1 /* last active cell */
4.    Searching
5.        For pos ∈ 1...n Do
6.            pC ← 0, nC ← 0
7.            For i ∈ 1...lact Do
8.                If p_i = t_pos Then nC ← pC
9.                Else
10.                   If pC < nC Then nC ← pC
11.                   If C_i < nC Then nC ← C_i
12.                   nC ← nC + 1
13.               End of if
14.               pC ← C_i, C_i ← nC
15.           End of for
16.           While C_lact > k Do lact ← lact − 1
17.           If lact = m Then report an occurrence at pos
18.           Else lact ← lact + 1
19.       End of for
```

Fig. 6.3. An $O(kn)$ expected time dynamic programming algorithm. Note that it works with just one column vector.

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

6.2.4 Other algorithms based on dynamic programming

There are many other algorithms based on this scheme. From the practical point of view, the most interesting is "column partitioning" [CL92], which obtains $O(kn/\sqrt{|\Sigma|})$ expected time [Nav01a]. This is the fastest algorithm based on dynamic programming. But it is hard to extend to more complex distance functions, and in this case newer bit-parallel algorithms are faster.

From the theoretical point of view, some of the most important algorithms are based on dynamic programming. If we are restricted to polynomial space in m and k, then the best existing algorithms use this technique and achieve $O(kn)$ worst-case search time with $O(m)$ extra space. The most competitive in practice are [GP90, CL94], which are still slower than algorithms that do not offer such a worst-case guarantee. When k is much smaller than m, an $O(n(1+k^4/m))$ time algorithm [CH98] becomes of interest. The worst-case lower bound for the problem when only polynomial space in m and k can be used is an open issue.

6.3 Algorithms based on automata

An alternative and very useful way to consider the approximate search problem is to model the search with a nondeterministic finite automaton (NFA). This automaton, in its deterministic form, was proposed first in [Ukk85], and later explicitly presented in [BY91, WM92b, BYN99].

Consider the NFA for $k = 2$ errors under edit distance shown in Figure 6.4. Every row denotes the number of errors seen. Every column represents matching a pattern prefix. Horizontal arrows represent matching a character (i.e., if the pattern and text characters match, we advance in the pattern and in the text). All the others increment the number of errors by moving to the next row: Vertical arrows insert a character in the pattern (we advance in the text but not in the pattern), solid diagonal arrows substitute a character (we advance in the text and pattern), and dashed diagonal arrows delete a character of the pattern (they are ε-transitions, since we advance in the pattern without advancing in the text). The initial self-loop allows an occurrence to start anywhere in the text. The automaton signals (the end of) an occurrence whenever a rightmost state is active.

It is not hard to see that once a state in the automaton is active, all the states in the same column and higher numbered rows are active too. Moreover, at a given text position, if we collect the smallest active rows at each NFA column, we obtain the current vertical vector of the dynamic programming matrix. For example, after reading the text `"anneal"`, the

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

seventh column in Figure 6.2 shows that $C = [0, 1, 1, 2, 3, 2, 1]$. Compare it with the least active row per NFA column in Figure 6.4.

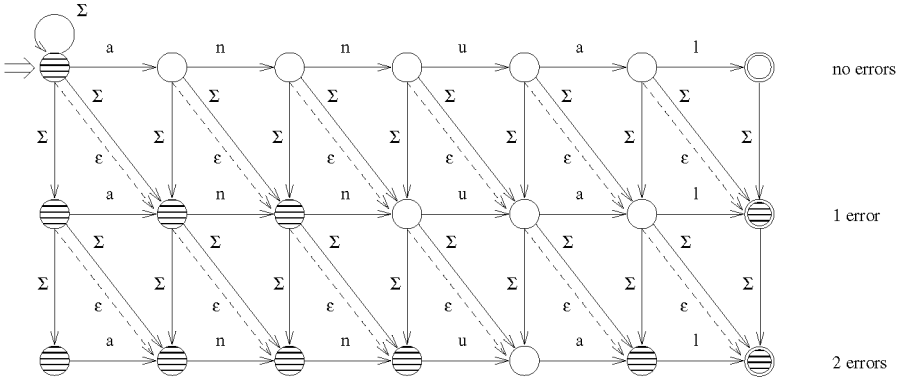

Fig. 6.4. An NFA for approximate string matching of the pattern `"annual"` with two errors. The shaded states are those active after reading the text `"anneal"`.

The original proposal of [Ukk85] was to make this automaton deterministic using the classical algorithm to convert an NFA into a DFA. This way, $O(n)$ worst-case search time is obtained, which is optimal. The main problem then becomes the construction and storage requirements of the DFA. An upper bound to the number of states of the DFA is $O(\min(3^m, m(2m|\Sigma|)^k))$ [Ukk85]. In practice, this automaton cannot be used for $m > 20$, and nowadays it is not the best choice even for small m: Bit-parallel algorithms are simpler and faster thanks to their higher locality of reference.

An alternative way to look at the DFA is to consider that each DFA state is a possible column of the dynamic programming matrix, so the preprocessing precomputes the transitions among columns for each possible input character.

Later developments [WMM96] based on the Four-Russians approach tackled the space and preprocessing cost problem by cutting columns into "regions" and building a DFA of regions. Figure 6.5 shows a schematic view of the automaton.

Given $O(s)$ space, the algorithm obtains $O(kn/\log s)$ expected time and $O(mn/\log s)$ worst-case time. Although it is the fastest choice in practice for long patterns and high error level ($\alpha > 0.7$), we do not include the details of this algorithm because it is complicated and because it is not the fastest in the most interesting cases: 0.7 is too high an error level for most applications.

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

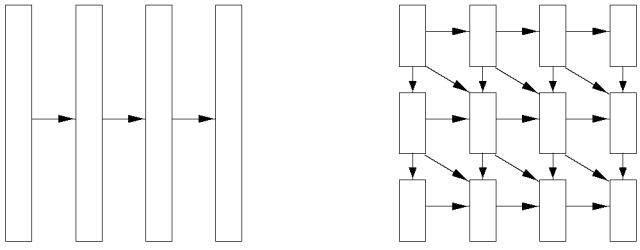

Fig. 6.5. On the left is the full DFA, where each column is a state. On the right is the Four-Russians version, where each region of a column is a state. The arrows show dependencies between consecutive regions.

6.4 Bit-parallel algorithms

Bit-parallelism has been heavily used for approximate searching, and many of the best results are obtained using this approach. The results are most useful for short patterns, and in many cases these are the patterns of interest. In cases when the representation does not fit in a single computer word, standard techniques permit the simulation of a virtual computer word formed from a number of physical words. Then, the techniques developed in Section 6.2.3 for the algorithm **DP** can be applied, so that only the computer words holding "active" data are updated.

Bit-parallel algorithms simulate "classical" algorithms. In approximate searching we find some that parallelize the work of the NFA and others that parallelize the work of the dynamic programming matrix.

6.4.1 Parallelizing the NFA

If we consider the first row of Figure 6.4, we are left with an NFA for exact string matching, the same one that is simulated using the **Shift-And** approach (Section 2.2.2). Different techniques have been proposed to extend this idea to the more general automaton.

6.4.1.1 Row-wise bit-parallelism

The simplest technique [WM92b], which we call **BPR**, packs each row i of the NFA in a different machine word R_i, with each state represented by a bit. For each new text character, all the transitions of the automaton are simulated using bit operations among the $k+1$ bit masks. Notice that all $k+1$ bit masks have the same structure, that is, the same bit is aligned to the same text position. The update formula to obtain the new R'_i values at

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

text position j from the current R_i values is

$$\begin{aligned} R'_0 &\leftarrow ((R_0 << 1) \mid 0^{m-1}1) \ \& \ B[t_j] \\ R'_i &\leftarrow ((R_i << 1) \ \& \ B[t_j]) \ \mid \ R_{i-1} \ \mid \ (R_{i-1} << 1) \ \mid \ (R'_{i-1} << 1) \end{aligned}$$

where B is the table from the **Shift-And** algorithm. We start the search with $R_i = 0^{m-i}1^i$, which is equivalent to $C_i = i$ in the **DP** algorithm. As expected, R_0 undergoes a simple **Shift-And** process, while the other rows receive ones (i.e., active states) from previous rows as well. The formula for R'_i expresses horizontal, vertical, diagonal, and dashed diagonal arrows, respectively. Figure 6.6 gives pseudo-code for this algorithm.

BPR $(p = p_1p_2 \ldots p_m,\ T = t_1t_2 \ldots t_n,\ k)$
1. `Preprocessing`
2. **For** $c \in \Sigma$ **Do** $B[c] \leftarrow 0^m$
3. **For** $j \in 1 \ldots m$ **Do** $B[p_j] \leftarrow B[p_j] \mid 0^{m-j}10^{j-1}$
4. `Searching`
5. **For** $i \in 0 \ldots k$ **Do** $R_i \leftarrow 0^{m-i}1^i$
6. **For** $pos \in 1 \ldots n$ **Do**
7. $oldR \leftarrow R_0$
8. $newR \leftarrow ((oldR << 1) \mid 0^{m-1}1) \ \& \ B[t_{pos}]$
9. $R_0 \leftarrow newR$
10. **For** $i \in 1 \ldots k$ **Do**
11. $newR \leftarrow ((R_i << 1) \ \& \ B[t_{pos}]) \mid oldR \mid ((oldR \mid newR) << 1)$
12. $oldR \leftarrow R_i$, $R_i \leftarrow newR$
13. **End of for**
14. **If** $newR \ \& \ 10^{m-1} \neq 0^m$ **Then** report an occurrence at pos
15. **End of for**

Fig. 6.6. Row-wise bit-parallel simulation of the NFA. The length of the pattern must be less than w.

The cost of this simulation is $O(k\lceil m/w \rceil n)$ in the worst and average cases, which is $O(kn)$ for patterns typical in text searching (i.e., $m \leq w$). Notice that for short patterns this is competitive with the best worst-case algorithms. As we see next, one can do much better, but this algorithm has maximum flexibility when it comes to adapting it to more complex cases such as wild cards or regular expression searching allowing errors.

Example of BPR We search for the string `"annual"` in the text `"annealing"` allowing $k = 2$ errors. Note that, at any point, the bit representations of R_0, R_1, and R_2 resemble the active states in the NFA of Figure 6.4, provided we read the states right to left and discard the first column of the NFA, which is never represented because it is known to be active all the

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

time. In particular, compare the bit map after reading the text `"anneal"` with the active states of Figure 6.4. It is also interesting to compare the bit maps to the column values of Figure 6.2 to check that C_i is the least active row at NFA column i.

$$B = \begin{cases} \texttt{a} & 0\,1\,0\,0\,0\,1 \\ \texttt{l} & 1\,0\,0\,0\,0\,0 \\ \texttt{n} & 0\,0\,0\,1\,1\,0 \\ \texttt{u} & 0\,0\,1\,0\,0\,0 \\ * & 0\,0\,0\,0\,0\,0 \end{cases}$$

$R_0 = 0\,0\,0\,0\,0\,0$
$R_1 = 0\,0\,0\,0\,0\,1$
$R_2 = 0\,0\,0\,0\,1\,1$

1. Reading `a` $0\,1\,0\,0\,0\,1$
   $R_0 = 0\,0\,0\,0\,0\,1$
   $R_1 = 0\,0\,0\,0\,1\,1$
   $R_2 = 0\,0\,0\,1\,1\,1$

2. Reading `n` $0\,0\,0\,1\,1\,0$
   $R_0 = 0\,0\,0\,0\,1\,0$
   $R_1 = 0\,0\,0\,1\,1\,1$
   $R_2 = 0\,0\,1\,1\,1\,1$

3. Reading `n` $0\,0\,0\,1\,1\,0$
   $R_0 = 0\,0\,0\,1\,0\,0$
   $R_1 = 0\,0\,1\,1\,1\,1$
   $R_2 = 0\,1\,1\,1\,1\,1$

4. Reading `e` $0\,0\,0\,0\,0\,0$
   $R_0 = 0\,0\,0\,0\,0\,0$
   $R_1 = 0\,0\,1\,1\,0\,1$
   $R_2 = 0\,1\,1\,1\,1\,1$

5. Reading `a` $0\,1\,0\,0\,0\,1$
   $R_0 = 0\,0\,0\,0\,0\,1$
   $R_1 = 0\,1\,0\,0\,1\,1$
   $R_2 = 1\,1\,1\,1\,1\,1$

   The last bit of R_2 is set, so we mark an occurrence.

6. Reading `l` $1\,0\,0\,0\,0\,0$
   $R_0 = 0\,0\,0\,0\,0\,0$
   $R_1 = 1\,0\,0\,0\,1\,1$
   $R_2 = 1\,1\,0\,1\,1\,1$

   The last bit of R_2 is set again, so we mark an occurrence. Note that the position matches even in R_1, i.e., with $k = 1$ errors.

7. Reading `i` $0\,0\,0\,0\,0\,0$
   $R_0 = 0\,0\,0\,0\,0\,0$
   $R_1 = 0\,0\,0\,0\,0\,1$
   $R_2 = 1\,0\,0\,1\,1\,1$

   The last bit of R_2 is set, so we mark an occurrence. Note that this occurrence is just a consequence of having matched with $k < 2$ errors at the previous positions, since no more matches of pattern letters are involved.

8. Reading `n` $0\,0\,0\,1\,1\,0$
   $R_0 = 0\,0\,0\,0\,0\,0$
   $R_1 = 0\,0\,0\,0\,1\,1$
   $R_2 = 0\,0\,0\,1\,1\,1$

9. Reading `g` $0\,0\,0\,0\,0\,0$
   $R_0 = 0\,0\,0\,0\,0\,0$
   $R_1 = 0\,0\,0\,0\,0\,1$
   $R_2 = 0\,0\,0\,1\,1\,1$

6.4.1.2 Diagonal-wise bit-parallelism

In light of the row-wise parallelization presented above, the classical dynamic programming algorithm can be thought of as a column-wise parallelization of the automaton where, as explained, each NFA column corresponds to a cell in C that stores the smallest active row at that column. Neither algorithm is able to increase the parallelism even if all the NFA states fit in a computer word, because the ε-transitions of the automaton cause *zero-*

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

time dependencies. That is, the current values of two rows or two columns depend on each other and hence cannot be computed in parallel.

In [BYN99] the bit-parallel formula for a *diagonal-wise* parallelization was found. We call **BPD** the resulting algorithm. They pack the states of the automaton along diagonals instead of rows or columns, running in the direction of the diagonal arrows. There are $m - k + 1$ complete diagonals, which are numbered left to right from 0 to $m - k$. Let D_i be the row number of the first active state in diagonal i. All the subsequent states in the diagonal are active because of the ε-transitions. The new D'_i values after reading text position j are

$$D'_i \quad \leftarrow \quad \min\ (\ D_i + 1,\ \ D_{i+1} + 1,\ \ g(i-1, t_j)\ ) \tag{6.2}$$

where the first term represents the substitutions, the second term the insertions, and the last term the matches. Deletions are implicit since only the lowest-row active state of each diagonal is represented. The main problem is how to compute the function g, defined as

$$g(i,c) \quad = \quad \min(\ \{k+1\}\ \cup\ \{\ r,\ r \geq D_i \text{ AND } p_{i+1+r} = c\ \}\ )$$

which expresses the fact that from all active states at diagonal i, namely, $r \in \{D_i,\ D_i + 1,\ \ldots,\ k\}$, those that can follow a horizontal arrow (i.e., $p_{i+1+r} = c$) move to diagonal $i + 1$. We take the minimum over those r. Another way to understand g is to note that an active state that crosses a horizontal edge has to propagate all the way down along the diagonal.

This process is simulated in [BYN99] by representing the D_i values in unary and using arithmetic operations on the bits to produce the desired propagation effect (in Section 4.4 a similar flooding problem is solved in detail). The update formula can be understood either numerically (operating on the D_i) or logically (simulating the arrows of the automaton). A computer word D holds $m - k$ blocks, one per diagonal excluding D_0 because it is known to be always active. From left to right, D_1 to D_{m-k} are represented. Inside each block there are $k + 2$ bits. The rightmost bit is always zero to avoid propagation of arithmetic operations to adjacent blocks, and the other $k + 1$ bits are used to represent D_i in unary: The leftmost D_i bits of block i are 1 and the others are 0. The typical B table is used, except that its bits are reversed. A table BB is computed from B in order to align the corresponding horizontal arrows to the arrangement made in D.

Figure 6.7 shows the algorithm. The representation does not include the states to the right of the last full diagonal. As a result, some occurrences are lost. However, those occurrences are uninteresting in most applications since they are trivial extensions of occurrences already found, in the sense

that no new pattern characters match the text (such as the one found after processing `"anneali"` in the example of **BPR**). To ensure that those occurrences are consistently discarded, line 14 removes all the active states in the last diagonal after an occurrence is reported. Hence the algorithm reports any occurrence that ends with a text character matching the pattern.

Line 11 updates D according to formula (6.2), by AND-ing four expressions, an operation that corresponds to minimization in unary. The first expression represents $D_i + 1$, the second $D_{i+1} + 1$, the third $g(i-1, t_{pos})$, and the last cleans up separators. About the third expression, note that x holds the states of the previous diagonal that arrive by horizontal transitions, and we make the last zero flood the block to the right.

BPD $(p = p_1 p_2 \ldots p_m,\ T = t_1 t_2 \ldots t_n,\ k)$

```
1.    Preprocessing
2.        For c ∈ Σ Do B[c] ← 1^m
3.        For j ∈ 1...m Do B[p_j] ← B[p_j] & 1^{m-j} 0 1^{j-1}
4.        For c ∈ Σ Do
5.            BB[c] ← 0 s_{k+1}(B[c],0)  0 s_{k+1}(B[c],1)  ...  0 s_{k+1}(B[c],m-k-1)
6.        End of for
7.    Searching
8.        D ← (0 1^{k+1})^{m-k}
9.        For pos ∈ 1...n Do
10.           x ← (D >> (k+2)) | BB[t_pos]
11.           D ← ((D << 1) | (0^{k+1} 1)^{m-k})
                      & ((D << (k+3)) | (0^{k+1} 1)^{m-k-1} 0 1^{k+1})
                      & (((x + (0^{k+1} 1)^{m-k}) ∧ x) >> 1) & (0 1^{k+1})^{m-k}
12.           If D & 0^{(m-k-1)(k+2)} 0 1^k 0 = 0^{(m-k)(k+2)} Then
13.               Report an occurrence at pos
14.               D ← D | 0^{(m-k-1)(k+2)} 0 1^{k+1}
15.           End of if
16.       End of for
```

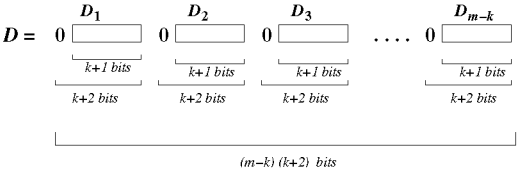

Fig. 6.7. Diagonal-wise bit-parallel simulation of the NFA. It requires that $(m-k)(k+2) \leq w$. The function $s_\ell(D, j)$ extracts the j-th to the $(j+\ell-1)$-th bits of D, that is, $s_\ell(D, j) = (D >> j)\ \&\ 0^{(m-k)(k+2)-\ell} 1^\ell$. On the bottom we show how the unary D_i values are arranged in the mask D.

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

The resulting algorithm is $O(n)$ worst-case time and very fast in practice if all the bits of the automaton fit in a computer word, while the row-wise simulation remains $O(kn)$. In general, it is $O(\lceil k(m-k)/w \rceil n)$ worst-case time. It can be made $O(\lceil k^2/w \rceil n)$ on average by updating only the computer words holding active states, using an adaptation of the technique for active cells presented for **DP**. The scheme can handle classes of characters, wild cards, and different integral costs in the edit operations [BYN99], but it is less flexible than row-wise simulation.

Example of BPD We search for the string `"annual"` in the text `"annealing"` allowing $k = 2$ errors. This time the bit representation for D is harder to relate visually to the NFA of Figure 6.4. The rule is to read full NFA diagonals, excluding the first, and to map them to blocks. Each diagonal must be read from top to bottom and its active states mapped to the zeros of its block, read from right to left.

$B =$

a	1 0 1 1 1 0
l	0 1 1 1 1 1
n	1 1 1 0 0 1
u	1 1 0 1 1 1
*	1 1 1 1 1 1

$D = 0\,1\,1\,1 \;\; 0\,1\,1\,1 \;\; 0\,1\,1\,1 \;\; 0\,1\,1\,1$

Table BB

a	0 1 0 1 0 0 1 1 0 1 1 1 0 1 1 0
l	0 0 1 1 0 1 1 1 0 1 1 1 0 1 1 1
n	0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 1
u	0 1 1 0 0 1 0 1 0 0 1 1 0 1 1 1
*	0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1

1. Reading `a`

BB[a]	0 1 0 1 0 0 1 1 0 1 1 1 0 1 1 0
$D =$	0 0 0 0 0 1 1 1 0 1 1 1 0 1 1 1

2. Reading `n`

BB[n]	0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 1
$D =$	0 0 0 1 0 0 0 0 0 1 1 1 0 1 1 1

3. Reading `n`

BB[n]	0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 1
$D =$	0 0 0 1 0 0 0 1 0 0 0 0 0 1 1 1

4. Reading `e`

BB[e]	0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1
$D =$	0 0 1 1 0 0 0 1 0 0 0 1 0 1 1 1

5. Reading `a`

BB[a]	0 1 0 1 0 0 1 1 0 1 1 1 0 1 1 0
$D =$	0 0 0 0 0 0 1 1 0 0 1 1 0 0 0 1

The highest bit of $D_{m-k} = D_4$ (third bit read right to left in D) is zero, so we mark an occurrence and clean the last diagonal:

$D = 0\,0\,0\,0 \;\; 0\,0\,1\,1 \;\; 0\,0\,1\,1 \;\; 0\,1\,1\,1$

6. Reading `l`

BB[l]	0 0 1 1 0 1 1 1 0 1 1 1 0 1 1 1
$D =$	0 0 0 1 0 1 1 1 0 0 1 1 0 0 1 1

The highest bit of D_4 is zero again, so we mark an occurrence and clean the last diagonal:

$D = 0\,0\,0\,1 \;\; 0\,1\,1\,1 \;\; 0\,0\,1\,1 \;\; 0\,1\,1\,1$

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

7. Reading `i`

BB[i]	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1
$D =$	0	0	1	1	0	1	1	1	0	1	1	1	0	1	1	1

Unlike the classical algorithm, we do not mark an occurrence here, because it does not involve any new matching pattern character. This is a consequence of having cleaned the last diagonal in the previous step.

8. Reading `n`

BB[n]	0	1	1	1	0	1	1	0	0	1	0	0	0	0	0	1
$D =$	0	0	0	1	0	1	1	1	0	1	1	1	0	1	1	1

9. Reading `g`

BB[g]	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1
$D =$	0	0	1	1	0	1	1	1	0	1	1	1	0	1	1	1

6.4.2 Parallelizing the DP matrix

A better way to parallelize the computation [Mye99] is to represent the differences between consecutive rows or columns of the dynamic programming matrix instead of the absolute values. Let us call

$$\begin{aligned} \Delta h_{i,j} &= M_{i,j} - M_{i,j-1} \in \{-1, 0, +1\} \\ \Delta v_{i,j} &= M_{i,j} - M_{i-1,j} \in \{-1, 0, +1\} \\ \Delta d_{i,j} &= M_{i,j} - M_{i-1,j-1} \in \{0, 1\} \end{aligned}$$

the horizontal, vertical, and diagonal differences among consecutive cells. Their range of values comes from the properties of the dynamic programming matrix.

We present a version [Hyy01] that differs slightly from that of [Mye99]: Although both perform the same number of operations per text character, the one we present is easier to understand.

Let us introduce the following boolean variables. The first four refer to horizontal/vertical positive/negative differences and the last to the diagonal difference being zero:

$$\begin{aligned} VP_{i,j} &\equiv \Delta v_{i,j} = +1 \qquad VN_{i,j} \equiv \Delta v_{i,j} = -1 \\ HP_{i,j} &\equiv \Delta h_{i,j} = +1 \qquad HN_{i,j} \equiv \Delta h_{i,j} = -1 \\ & D0_{i,j} \equiv \Delta d_{i,j} = 0 \end{aligned}$$

Note that $\Delta v_{i,j} = VP_{i,j} - VN_{i,j}$, $\Delta h_{i,j} = HP_{i,j} - HN_{i,j}$, and $\Delta d_{i,j} = 1 - D0_{i,j}$. It is clear that these values completely define $M_{i,j} = \sum_{r=1\ldots i} \Delta v_{r,j}$.

The key idea is to notice some dependencies among the above values:

- If $HN_{i,j}$, then $\Delta h_{i,j} = -1$. Therefore, the only possibility is that $\Delta v_{i,j-1} = +1$ and hence $\Delta d_{i,j} = 0$, otherwise the Δ ranges of values would be violated. The last two conditions are equivalent to $VP_{i,j-1}$ AND $D0_{i,j}$. On the other hand, if these two conditions hold, $HN_{i,j}$ holds.

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

- By symmetric arguments it can be seen that $VN_{i,j}$ is logically equivalent to $HP_{i-1,j}$ AND $D0_{i,j}$.
- If $HP_{i,j}$ holds, then $VP_{i,j-1}$ cannot hold without violating the ranges of the Δ values. So the choices for $\Delta v_{i,j-1}$ are -1 and 0. In the first case we have $VN_{i,j-1}$, whereas in the second we have that neither $VP_{i,j-1}$ nor $D0_{i,j}$ hold. Moreover, this is a logical equivalence: If $VN_{i,j-1}$, then $HP_{i,j}$ has to hold; and if both $VP_{i,j-1}$ and $D0_{i,j}$ are false, then $HP_{i,j}$ has to hold as well.
- Symmetrically, we can see that $VP_{i,j}$ is logically equivalent to $HN_{i-1,j}$ OR (NOT $HP_{i-1,j}$ AND NOT $D0_{i,j}$).
- Finally, $D0_{i,j}$ can be true for three possible reasons, which correspond to formula (6.1). First, it may happen that $P_i = T_j$. Second, it may be the case that $M_{i,j} = 1 + M_{i,j-1} = M_{i-1,j-1}$, which means $HP_{i,j}$ AND $VN_{i,j-1}$. Third, it may occur that $M_{i,j} = 1 + M_{i-1,j} = M_{i-1,j-1}$, which means $VP_{i,j}$ AND $HN_{i-1,j}$. From these conditions we use only $(P_i = T_j)$ OR $VN_{i,j-1}$ OR $HN_{i-1,j}$. Note again that if any of these three conditions hold, then $D0_{i,j}$ holds, so we have a logical equivalence.

Hence we have proved the following equivalences:

$$\begin{aligned}
HN_{i,j} &\equiv VP_{i,j-1} \text{ AND } D0_{i,j} \\
VN_{i,j} &\equiv HP_{i-1,j} \text{ AND } D0_{i,j} \\
HP_{i,j} &\equiv VN_{i,j-1} \text{ OR NOT } (VP_{i,j-1} \text{ OR } D0_{i,j}) \\
VP_{i,j} &\equiv HN_{i-1,j} \text{ OR NOT } (HP_{i-1,j} \text{ OR } D0_{i,j}) \\
D0_{i,j} &\equiv (P_i = T_j) \text{ OR } VN_{i,j-1} \text{ OR } HN_{i-1,j} \ .
\end{aligned}$$

The algorithm traverses the text and, at each text position j, keeps track of the five values above for every i. Since each value needs only one bit, we keep bit masks HN, VN, HP, VP, and $D0$ and update them for every new text character T_j read. Hence, for example, the i-th bit of the bit mask HN will correspond to the value $HN_{i,j}$. The index $j-1$ refers to the previous value of the bit mask (before processing T_j), whereas j refers to the new value, after processing T_j.

Under this light, it is clear that we can first compute $D0$, then HN and HP, and finally VN and VP. However, there is a circular dependency regarding $D0_{i,j}$; it depends on $HN_{i-1,j}$, which in turn depends on $D0_{i-1,j}$. That is, current $D0_{i,j}$ values depend on other current $D0_{i',j}$ values. This corresponds, again, to the zero-time dependency problem and complicates computing $D0$ in one shot. However, a solution exists.

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

Let us expand the formula for $D0_{i,j}$:

$$D0_{i,j} \equiv (P_i = T_j) \text{ OR } VN_{i,j-1} \text{ OR } (VP_{i-1,j-1} \text{ AND } D0_{i-1,j}) ,$$

which has the form $D_i \equiv X_i \text{ OR } (Y_{i-1} \text{ AND } D_{i-1})$. Unrolling the first values we get

$$\begin{aligned} D_1 &\equiv X_1 \\ D_2 &\equiv X_2 \text{ OR } (Y_1 \text{ AND } X_1) \\ D_3 &\equiv X_3 \text{ OR } (Y_2 \text{ AND } X_2) \text{ OR } (Y_2 \text{ AND } Y_1 \text{ AND } X_1) \\ D_i &\equiv \text{ OR }_{r=1}^{i}(X_r \text{ AND } Y_r \text{ AND } Y_{r+1} \text{ AND } \ldots \text{ AND } Y_{i-1}) . \end{aligned}$$

Let s be such that $Y_s \ldots Y_{i-1} = 1$ and $Y_{s-1} = 0$. It should be clear that D_i will be activated if $X_r = 1$ for some $s \leq r \leq i$. In other words, D_i will be activated if there is a bit set in X in the area covered by the last contiguous block of bits set in Y. If we compute $(Y + (X \ \& \ Y))$, the result is that every $X_r = 1$ that is aligned to a $Y_r = 1$ will propagate a change until one position after the end of the block. This covers all the positions i that should be set in D because of X_r being aligned to a block of 1's in Y. If we compute $(Y + (X \ \& \ Y)) \wedge Y$, the bits that changed will be on. Note that, since there may be several X_r bits under the same block of Y, all but the first such r positions will remain unchanged and hence not marked by the XOR operation. To fix this and to account for the case $X_i = 1$, we OR the final result with X. An example is as follows:

$$\begin{aligned} Y &= 0\,0\,0\,1\,1\,1\,1\,1\,0\,0\,0\,0\,1\,1 \\ X &= 0\,0\,0\,0\,1\,0\,1\,0\,0\,0\,0\,1\,0\,1 \\ X \ \& \ Y &= 0\,0\,0\,0\,1\,0\,1\,0\,0\,0\,0\,0\,0\,1 \\ (Y + (X \ \& \ Y)) &= 0\,0\,1\,0\,1\,0\,0\,1\,0\,0\,0\,1\,0\,0 \\ (Y + (X \ \& \ Y)) \wedge Y &= 0\,0\,1\,1\,0\,1\,1\,0\,0\,0\,0\,1\,1\,1 \\ D0 = ((Y + (X \ \& \ Y)) \wedge Y) \mid X &= 0\,0\,1\,1\,1\,1\,1\,0\,0\,0\,0\,1\,1\,1 \end{aligned}$$

Once the solution to $D0$ is obtained, the rest flows easily. Figure 6.8 gives pseudo-code. The value *err* stores $C_m = M_{m,j}$ explicitly and is updated using $HP_{m,j}$ and $HN_{m,j}$. Note that the shifts correctly introduce zeros.

We call this algorithm **BPM**. It uses the bits of the computer word better than the previous bit-parallel algorithms, with a worst case of $O(\lceil m/w \rceil n)$ and an average case of $O(\lceil k/w \rceil n)$, achieved by updating only the computer words having "active" cells, as for **DP**. The update formula is a little more complex than that of **BPD** and the algorithm is a bit slower, but it adapts better to longer patterns because fewer computer words are needed. On the

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

BPM $(p = p_1 p_2 \ldots p_m,\ T = t_1 t_2 \ldots t_n,\ k)$
1. `Preprocessing`
2. **For** $c \in \Sigma$ **Do** $B[c] \leftarrow 0^m$
3. **For** $j \in 1 \ldots m$ **Do** $B[p_j] \leftarrow B[p_j] \mid 0^{m-j}10^{j-1}$
4. $VP \leftarrow 1^m$, $VN \leftarrow 0^m$
5. $err \leftarrow m$
6. `Searching`
7. **For** $pos \in 1 \ldots n$ **Do**
8. $X \leftarrow B[t_{pos}] \mid VN$
9. $D0 \leftarrow ((VP + (X \,\&\, VP)) \ \wedge\ VP) \mid X$
10. $HN \leftarrow VP \,\&\, D0$
11. $HP \leftarrow VN \mid \sim (VP \mid D0)$
12. $X \leftarrow HP << 1$
13. $VN \leftarrow X \,\&\, D0$
14. $VP \leftarrow (HN << 1) \mid \sim (X \mid D0)$
15. **If** $HP \,\&\, 10^{m-1} \neq 0^m$ **Then** $err \leftarrow err + 1$
16. **Else If** $HN \,\&\, 10^{m-1} \neq 0^m$ **Then** $err \leftarrow err - 1$
17. **If** $err \leq k$ **Then** report an occurrence at pos
18. **End of for**

Fig. 6.8. Bit-parallel simulation of the dynamic programming matrix. It requires $m \leq w$.

other hand, **BPM** is even less flexible than **BPD** when it comes to searching for complex patterns or different distance functions.

Note that the algorithm can be adapted to compute edit distance simply by adding "$\mid 0^{m-1}1$" at the end of line 12 in Figure 6.8, since this time there is a horizontal increment at row zero (not represented in the bit masks).

Example of BPM We search for the string `"annual"` in the text `"anne-aling"` allowing $k = 2$ errors. The easiest way to understand what is going on is to relate the bit masks to the Δ values and these in turn to those of the dynamic programming matrix of Figure 6.2.

It is interesting to verify that err correctly maintains the value of the last cell of the current column of the **DP** matrix.

$B =$

a	0 1 0 0 0 1
l	1 0 0 0 0 0
n	0 0 0 1 1 0
u	0 0 1 0 0 0
*	0 0 0 0 0 0

$VN = 0\,0\,0\,0\,0\,0$
$VP = 1\,1\,1\,1\,1\,1$
$err = 6$

1. Reading a 0 1 0 0 0 1

$D0 =$	1 1 1 1 1 1
$HN =$	1 1 1 1 1 1
$HP =$	0 0 0 0 0 0
$VN =$	0 0 0 0 0 0
$VP =$	1 1 1 1 1 0
$err =$	5

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

2.

Reading `n`	0 0 0 1 1 0
$D0 =$	1 1 1 1 1 0
$HN =$	1 1 1 1 1 0
$HP =$	0 0 0 0 0 1
$VN =$	0 0 0 0 1 0
$VP =$	1 1 1 1 0 1
$err =$	4

3.

Reading `n`	0 0 0 1 1 0
$D0 =$	1 1 1 1 1 0
$HN =$	1 1 1 1 0 0
$HP =$	0 0 0 0 1 0
$VN =$	0 0 0 1 0 0
$VP =$	1 1 1 0 0 1
$err =$	3

4.

Reading `e`	0 0 0 0 0 0
$D0 =$	0 0 0 1 0 0
$HN =$	0 0 0 0 0 0
$HP =$	0 0 0 1 1 0
$VN =$	0 0 0 1 0 0
$VP =$	1 1 0 0 1 1
$err =$	3

5.

Reading `a`	0 1 0 0 0 1
$D0 =$	1 1 0 1 1 1
$HN =$	1 1 0 0 1 1
$HP =$	0 0 1 1 0 0
$VN =$	0 1 0 0 0 0
$VP =$	1 0 0 1 1 0
$err =$	2

We mark an occurrence since $err \leq 2$.

6.

Reading `l`	1 0 0 0 0 0
$D0 =$	1 1 0 0 0 0
$HN =$	1 0 0 0 0 0
$HP =$	0 1 1 0 0 1
$VN =$	1 1 0 0 0 0
$VP =$	0 0 1 1 0 1
$err =$	1

We mark an occurrence since $err \leq 2$.

7.

Reading `i`	0 0 0 0 0 0
$D0 =$	1 1 0 0 0 0
$HN =$	0 0 0 0 0 0
$HP =$	1 1 0 0 1 0
$VN =$	1 0 0 0 0 0
$VP =$	0 0 1 0 1 1
$err =$	2

We mark an occurrence since $err \leq 2$.

8.

Reading `n`	0 0 0 1 1 0
$D0 =$	1 0 0 1 1 0
$HN =$	0 0 0 0 1 0
$HP =$	1 1 0 0 0 0
$VN =$	1 0 0 0 0 0
$VP =$	0 1 1 1 0 1
$err =$	3

9.

Reading `g`	0 0 0 0 0 0
$D0 =$	1 0 0 0 0 0
$HN =$	0 0 0 0 0 0
$HP =$	1 0 0 0 1 0
$VN =$	0 0 0 0 0 0
$VP =$	0 1 1 0 1 1
$err =$	4

6.5 Algorithms for fast filtering the text

The idea behind filtration algorithms is that it may be easier to tell that a text position does *not* match than to tell that it does. So these algorithms filter the text, discarding areas that cannot match. They are unable on their own to tell that there *is* a match, so every filtration algorithm needs to be coupled with a nonfiltration algorithm to check the nondiscarded text areas for potential occurrences.

Filtering algorithms only improve the average-case performance, and their major attraction is the potential for algorithms that do not inspect every text character. The performance of filtration algorithms is related to the amount of text that they are able to discard, and it is very sensitive to the error level. Most filters work very well on low error levels and poorly otherwise,

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

so when evaluating filtration algorithms it is important to consider not only their time efficiency but also their tolerance to errors.

There are many filtration algorithms, among which we have selected the two that are the best in most cases. The first, **PEX**, is the best when the alphabet size is not too small, for example, on English text. The second one, **ABNDM**, is the best on DNA and other small-alphabet texts.

6.5.1 Partitioning into $k+1$ pieces

The idea behind this algorithm, which we call **PEX**, is that if a pattern is cut into $k+1$ pieces, then at least *one* of the pieces must appear unchanged in an approximate occurrence. This is evident, since k errors cannot alter $k+1$ pieces, at least under the edit distance model. Indeed, a more general lemma turns out to be useful [Mye94, BYN99]:

Lemma 1 *Let Occ match p with k errors, $p = p^1 \dots p^j$ be a concatenation of subpatterns, and $a_1 \dots a_j$ be nonnegative integers such that $A = \sum_{i=1}^{j} a_i$. Then, for some $i \in 1 \dots j$, Occ includes a substring that matches p^i with $\lfloor a_i k/A \rfloor$ errors.*

To see this, note that if each p^i matches only with $1+\lfloor a_i k/A \rfloor$ errors, then the whole p cannot match with less than $k+1$ errors. If we set $A = j = k+1$ and $a_i = 1$, then the simpler case shows up.

The proposal in [WM92b, BYN99, NBY99] is to split the pattern into $k+1$ approximately equal length pieces, search the pieces in the text with a multipattern search algorithm, and then check the neighborhood of their occurrences. Some care has to be exercised to report the occurrences in order and to avoid reporting the same occurrences more than once.

The "neighborhood" must be large enough to hold any occurrence. Occurrences are of length at most $m+k$ under edit distance. If pattern piece $p_{i_1 \dots i_2}$ matches at text position $t_{j \dots j+(i_2-i_1)}$, then the occurrence can start at most $i_1 - 1 + k$ positions before t_j since the insertions can all occur at the beginning of the occurrence, and it can finish at most $m - i2 + k$ positions after $t_{j+(i_2-i_1)}$ since the insertions can all occur at the end of the occurrence. Hence we need to check the text area $T_{j-(i_1-1)-k \dots j+(m-i_1)+k}$, which is of length $m+2k$. Note that if two pieces happen to be equal, each occurrence must trigger two verifications with different areas.

Two choices need to be made to obtain a concrete algorithm. The most important one is which multipattern search algorithm to use (Chapter 3). **Multiple Shift-And** is used in [WM92b], while [BYN99, NBY99] use **Set Horspool**.

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

The second choice is the verification algorithm. Although many authors care little about this choice and resort to plain dynamic programming, a faster technique such as **BPM** reduces the cost per verification from $O(m^2)$ to $O(m^2/w)$.

It is shown in [BYN99] that the cost of the multipattern search dominates for $\alpha < 1/(3 \log_{|\Sigma|} m)$. Above that error level, the cost of verifying candidate text positions starts to dominate and the filter efficiency deteriorates abruptly. In the area where the filter behaves well, its search cost is about $O(kn \log_{|\Sigma|}(m)/m)$.

Hierarchical verification To reduce unnecessary verification costs, "hierarchical verification" is introduced in [NBY99]. The idea is that, since the verification cost is quadratic in the pattern length, we pay too much verifying the whole pattern each time a small piece matches. We could reject the occurrence with a cheaper test for a shorter pattern piece.

Assume that the pattern is partitioned into $j = k+1 = 2^r$ pieces. Instead of splitting it into $k+1$ pieces in one shot, we do it hierarchically. The pattern is first split in half, each half to be searched with $\lfloor k/2 \rfloor$ errors due to Lemma 1. The halves are then recursively split in two, until the number of errors allowed becomes zero.

Figure 6.9 illustrates the resulting tree. The leaves of this tree are the pieces actually searched. When a leaf occurs in the text, instead of checking the whole pattern as in the basic technique, the parent of the leaf is checked (with $k = 1$ errors in the example) in a small area around the piece that matched. The extension of this area is computed as before, according to the piece length and the error level permitted. If that parent node is not found, then the verification stops and the multipattern scanning resumes. Otherwise the verification continues with the grandparent of the leaf and so on, until the root (i.e., the whole pattern) is found.

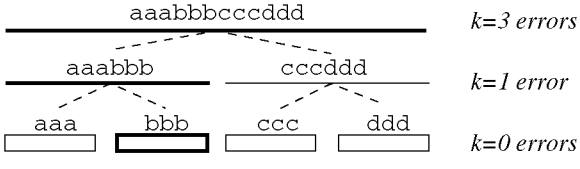

Fig. 6.9. The hierarchical verification method for a pattern split into four parts. The boxes (leaves) are the elements that are really searched, and the root represents the whole pattern. At least one pattern at each level must appear in any occurrence of the complete pattern. If the bold box is found, all the bold lines may be verified.

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

This technique is correct because Lemma 1 applies to each level of the tree: The grandparent cannot appear if none of its children appear, even if a grandchild appeared.

Let us go back to Figure 6.9. If one searches for the pattern `"aaabbbcc-cddd"` with three errors in the text `"xxxbbbxxxxxx"`, and splits the pattern into four pieces to be searched for without errors, then the piece `"bbb"` will be found in the text. In the original approach, one would verify the complete pattern with $k = 3$ errors in the text area, while with hierarchical verification one checks only its parent `"aaabbb"` with one error and immediately determines that there cannot be a complete occurrence. This latter check is much cheaper.

The analysis in [NBY99] shows that with hierarchical verification the area of applicability of the algorithm grows to $\alpha < 1/\log_{|\Sigma|} m$.

When $k+1$ is not a power of 2, it is advisable to keep the binary tree as balanced as possible. For example, if $k+1 = 3$, then we split the pattern into three pieces (leaves) of length $\lfloor m/3 \rfloor$. In the binary tree, the left child of the root has length $2\lfloor m/3 \rfloor$ and is searched with $\lfloor 2k/3 \rfloor = 1$ errors, while the second child is the leftmost leaf with length $\lfloor m/3 \rfloor$ to be searched with $\lfloor k/3 \rfloor = 0$ errors. The node that is searched with one error is then split into its two leaves. Pseudo-code for the algorithm that builds this tree is shown in Figure 6.10 together with the resulting tree for the pattern `"annual"` with $k = 2$. Pseudo-code for the **PEX** algorithm is shown in Figure 6.11.

Example of PEX We search for the string `"annual"` in the text `"any_annealing"` allowing $k = 2$ errors. The corresponding partition is given in Figure 6.10. As can be seen, the same occurrences can be found many times.

1. Found [an] `y_annealing`

Search for	`"annu"` with $k = 1$
inside	[any_a] `nnealing`
failed	(so abort verification)

Search for	`"annual"` with $k = 2$
inside	`an` [y_annealin] `g`
found	(report positions 9,10,11)

2. Found `any_` [an] `nealing`

Search for	`"annu"` with $k = 1$
inside	`any` [_annea] `ling`
found	(so go upper in the tree)

3. Found `any_anne` [al] `ing`

Search for	`"annual"` with $k = 2$
inside	`an` [y_annealin] `g`
found	(report positions 9,10,11)

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

CreateTree $(p = p_i p_{i+1} \dots p_j,\ k,\ myParent,\ idx,\ plen)$
1. Create new *node*
2. $from(node) \leftarrow i$
3. $to(node) \leftarrow j$
4. $left \leftarrow \lceil (k+1)/2 \rceil$
5. $parent(node) \leftarrow myParent$
6. $err(node) \leftarrow k$
7. **If** $k = 0$ **Then** $leaf_{idx} \leftarrow node$
8. **Else**
9. **CreateTree**$(p_{i \dots i+left \cdot plen-1},\ \lfloor (left \cdot k)/(k+1) \rfloor,\ node,\ idx,\ plen)$
10. **CreateTree**$(p_{i+left \cdot plen \dots j},\ \lfloor ((k+1-left) \cdot k)/(k+1) \rfloor,$ $node,\ idx + left,\ plen)$
11. **End of if**

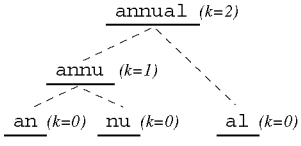

Fig. 6.10. Recursive algorithm to build the hierarchical verification tree on $p_{i \dots j}$ with k errors. The other variables are *myParent* (parent of the node to be built), *idx* (next leaf index to assign), and *plen* (length of the pieces). At the bottom is an example for the pattern `"annual"` with $k = 2$.

6.5.2 Approximate BNDM

Just as **BDM/BNDM** is better than **Boyer-Moore** algorithms for small alphabet sizes, an extension of **BNDM** proposed in [NR00] works better than **PEX** on DNA. We call it **ABNDM**.

The modification is to build an NFA to search the *reversed* pattern allowing errors, modify it to match any pattern suffix, and apply essentially **BNDM** (Section 2.4.2) using this automaton. Figure 6.12 shows the resulting automaton.

This automaton recognizes any reverse prefix of p allowing k errors. The text window will be abandoned when no pattern factor matches with k errors what was read. At that point, the window is shifted to the next pattern prefix found with k errors (position *last*).

The occurrences must start exactly at the initial window position. This makes it easier to report initial rather than final positions of the pattern occurrences, although with some care we can report the sorted final positions without repetitions.

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

```
PEX (p = p_1 p_2 ... p_m, T = t_1 t_2 ... t_n, k)
1.    Preprocessing
2.        CreateTree(p, k, θ, 0, ⌊m/(k+1)⌋)
3.        Preprocess multipattern search for
              {p_{from(node)} ... p_{to(node)}, node = leaf_i, i ∈ {0...k}}
4.    Searching
5.        For (pos, i) ∈ output of multipattern search Do
6.            node ← leaf_i
7.            in ← from(node)
8.            node ← parent(node)
9.            cand ← TRUE
10.           While cand = TRUE AND node ≠ θ Do
11.               p_1 ← pos − (in − from(node)) − err(node)
12.               p_2 ← pos + (to(node) − in + 1) + err(node)
13.               Verify text area T_{p_1...p_2} for pattern piece p_{from(node)...to(node)}
                      allowing err(node) errors
14.               If pattern piece was not found Then cand ← FALSE
15.               Else node ← parent(node)
16.           End of while
17.           If cand = TRUE Then
18.               Report the positions where the whole p was found
19.           End of if
20.       End of for
```

Fig. 6.11. Filtration algorithm based on partitioning into exact searches. It assumes that the multipattern search algorithm delivers its results in the form $(text\_position, piece\_that\_matched)$.

The window length is $m-k$, not m, to ensure that if there is an occurrence starting at the window position then a factor of the pattern occurs in any suffix of the window.

Reaching the beginning of the window does not guarantee an occurrence, however. Since the occurrences are of varying length, we only know that a factor of the pattern has occurred with at most k errors. In particular, if no pattern *prefix* has been read with k errors or less, no match can start at the initial window position. On the other hand, if we found such a pattern prefix, we would have to check the area by computing the edit distance from the beginning of the window, reading at most $m+k$ text characters.

This verification can be done with the algorithm to compute edit distance given in Section 6.2.1. Another choice is to use **BPR**, where we remove the initial self-loop in Figure 6.4. The formula is the same except for R_0, where it becomes

$$R'_0 \leftarrow (R_0 << 1) \ \& \ B[t_j]$$

The other bit-parallel algorithms are more complicated to adapt.

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

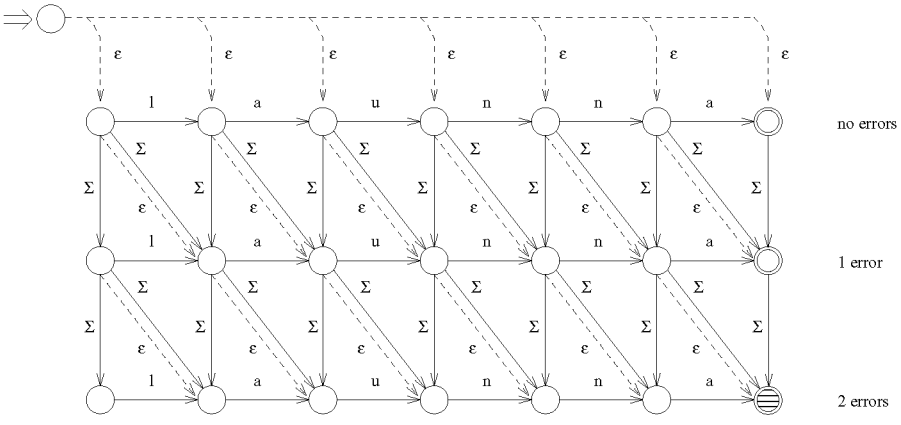

Fig. 6.12. The NFA to search for any reverse prefix of `"annual"` allowing two errors. We show the active states after reading the text window `"any_"`.

As with the original NFA of Figure 6.4, there are many ways to simulate the automaton of Figure 6.12. Given that this algorithm works well for small k values, using row-wise parallelization is a good choice. In particular, specializing the code for constant k values is a good idea. The only change is that we have to initialize the automaton with all the states active and remove the self-loop.

Other schemes, such as **BPD** and **BPM**, are more complicated to use. **BPD** needs more bits than for searching, because the whole automaton needs to be represented, not just the full diagonals. In principle $(m + k + 2)(k + 2)$ bits are necessary. **BPM** was not designed to tell whether the automaton has any active state (there is, however, recent work on this [HN01]). This is the first example where the flexibility of **BPR** pays off.

Figure 6.13 shows the algorithm. We initialize it after reading the first character of the window.

The algorithm works well for small alphabets and short patterns; it needs $m \leq w$ because of bit-parallelism. With longer patterns it is possible to use more computer words, but the results quickly deteriorate because the trick of only updating the computer words holding active states does not work well. The reason is that, since we initialize the NFA with all the states activated, the active states tend to be distributed uniformly over the whole pattern. On the other hand, making the automaton deterministic as with **BDM** generates an exponential number of states, just as the DFA construction reviewed in Section 6.3.

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

```
ABNDM (p = p_1 p_2 ... p_m, T = t_1 t_2 ... t_n, k)
1.    Preprocessing
2.        For c ∈ Σ Do B[c] ← 0^m
3.        For j ∈ 1...m Do B[p_j] ← B[p_j] | 0^{j-1}10^{m-j}
4.    Searching
5.        pos ← 0
6.        While pos ≤ n − (m − k) Do
7.            j ← m − k − 1, last ← m − k − 1
8.            R_0 ← B[t_{pos+m-k}]
9.            newR ← 1^m
10.           For i ∈ 1...k Do R_i ← newR
11.           While newR ≠ 0^m AND j ≠ 0 Do
12.               oldR ← R_0
13.               newR ← (oldR << 1) & B[t_{pos+j}]
14.               R_0 ← newR
15.               For i ∈ 1...k Do
16.                   newR ← ((R_i << 1) & B[t_{pos+j}])
                                | oldR | ((oldR | newR) << 1)
17.                   oldR ← R_i, R_i ← newR
18.               End of for
19.               j ← j − 1
20.               If newR & 10^{m-1} ≠ 0^m Then /* prefix recognized */
21.                   If j > 0 Then last ← j
22.                   Else check a possible occurrence starting at pos + 1
23.               End of if
24.           End of while
25.           pos ← pos + last
26.       End of while
```

Fig. 6.13. The extension of **BNDM** to approximate searching. It assumes $m - k > 1$.

Example of ABNDM We search for the string `"annual"` in the text `"any_annealing"` allowing $k = 1$ errors. We have reduced the error level because $k = 2$ is too high to be illustrative.

$B =$

a	1 0 0 0 1 0
l	0 0 0 0 0 1
n	0 1 1 0 0 0
u	0 0 0 1 0 0
*	0 0 0 0 0 0

1. [any_a] nnealing

Reading a	1 0 0 0 1 0
$R_0 =$	1 0 0 0 1 0
$R_1 =$	1 1 1 1 1 1

$last \leftarrow 4$.

Reading _	0 0 0 0 0 0
$R_0 =$	0 0 0 0 0 0
$R_1 =$	1 0 0 1 1 0

The last bit of R_1 is set, so $last \leftarrow 3$.

Reading y	0 0 0 0 0 0
$R_0 =$	0 0 0 0 0 0
$R_1 =$	0 0 0 0 0 0

The automaton runs out of active states, so we shift the window by $last = 3$.

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

2. `any` `_anne` `aling`

Reading `e`	0 0 0 0 0 0
$R_0 =$	0 0 0 0 0 0
$R_1 =$	1 1 1 1 1 1

$last \leftarrow 4$.

Reading `n`	0 1 1 0 0 0
$R_0 =$	0 0 0 0 0 0
$R_1 =$	0 1 1 0 0 0

Reading `n`	0 1 1 0 0 0
$R_0 =$	0 0 0 0 0 0
$R_1 =$	0 1 0 0 0 0

Reading `a`	1 0 0 0 1 0
$R_0 =$	0 0 0 0 0 0
$R_1 =$	1 0 0 0 0 0

The last bit of R_1 is set, so $last \leftarrow 1$.

Reading `_`	0 0 0 0 0 0
$R_0 =$	0 0 0 0 0 0
$R_1 =$	0 0 0 0 0 0

The automaton runs out of active states, so we shift by $last = 1$.

3. `any_` `annea` `ling`

Reading `a`	1 0 0 0 1 0
$R_0 =$	1 0 0 0 1 0
$R_1 =$	1 1 1 1 1 1

$last \leftarrow 4$.

Reading `e`	0 0 0 0 0 0
$R_0 =$	0 0 0 0 0 0
$R_1 =$	1 0 0 1 1 0

The last bit of R_1 is set, so $last \leftarrow 3$.

Reading `n`	0 1 1 0 0 0
$R_0 =$	0 0 0 0 0 0
$R_1 =$	0 0 1 0 0 0

Reading `n`	0 1 1 0 0 0
$R_0 =$	0 0 0 0 0 0
$R_1 =$	0 1 0 0 0 0

Reading `a`	1 0 0 0 1 0
$R_0 =$	0 0 0 0 0 0
$R_1 =$	1 0 0 0 0 0

The last bit of R_1 is set and $j = 0$, so we compute edit distance between the pattern and prefixes of the text `"anneali"`. Since we find a match ($k \leq 1$) against the prefix `"anneal"`, we report the text position 5.
We shift the window by $last = 3$.

4. `any_ann` `ealin` `g`

Reading `n`	0 1 1 0 0 0
$R_0 =$	0 1 1 0 0 0
$R_1 =$	1 1 1 1 1 1

$last \leftarrow 4$.

Reading `i`	0 0 0 0 0 0
$R_0 =$	0 0 0 0 0 0
$R_1 =$	1 1 1 0 0 0

The last bit of R_1 is set, so $last \leftarrow 3$.

Reading `l`	0 0 0 0 0 0
$R_0 =$	0 0 0 0 0 0
$R_1 =$	0 0 0 0 0 0

The automaton runs out of active states, so we shift by $last = 3$.

Since the window falls out of the text, we stop.

6.5.3 Other filtration algorithms

There are many proposals for filtration. In particular, we have left out some algorithms that are slightly faster than **PEX** and **ABNDM** for a few $(m, k, |\Sigma|)$ combinations [TU93, JTU96, BYN99, CL94, ST95]. In general, however, the differences in performance do not justify the programming effort.

There exist filtration algorithms that are optimal on average. It was proved in [CM94] that a lower bound for the expected time of approximate searching is $O((k + \log_{|\Sigma|} m)n/m)$. In the same paper, a filtration algorithm

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

with that complexity is obtained. The complexity is valid for $\alpha < 1-e/\sqrt{|\Sigma|}$, a limit shown impossible to improve [BYN99] since at that point there are too many *real* occurrences in the text. Although it is optimal in theory, the algorithm is not fast in practice. Whether a practical algorithm with optimal complexity exists is still an open issue.

On the other hand, most filters achieve $O(k \log_{|\Sigma|}(m)n/m)$ time for $\alpha = O(1/\log_{|\Sigma|} m)$. The central issue is that, in order to break this barrier, it seems necessary to reduce the problem to pieces that are searched with *fewer* errors instead of with *zero* errors. This is precisely what is done in [CM94], as well as in other filters [BYN99] that reach the limit $\alpha < 1-e/\sqrt{|\Sigma|}$. This last technique does not skip text characters, but it is a reasonable alternative in practice for medium error levels.

6.6 Multipattern approximate searching

A natural extension to the approximate search problem is that of searching multiple patterns simultaneously. Not many algorithms have been proposed for this, and all of them are filters that lose efficiency for high enough error levels.

6.6.1 A hashing based algorithm for one error

A good solution for $k = 1$ proposed in [MM96], which we call **MultiHash**, is based on the observation that if p matches p' with one error, then there are $m-1$ characters that match. The idea is to obtain m strings from p, which we call "signatures," by removing one character at a time, that is, $\{p_2p_3 \ldots p_m,\ p_1p_3 \ldots p_m,\ p_1p_2p_4 \ldots p_m,\ \ldots,\ p_1p_2 \ldots p_{m-1}\}$. We define the j-th signature of a string x of length m as

$$S_{x,j} = x_1x_2 \ldots x_{j-1}x_{j+1} \ldots x_m$$

For example, for the pattern $p =$`"annual"` the signatures are $S_{p,1} =$ `"nnual"`, $S_{p,2} = S_{p,3} =$ `"anual"`, $S_{p,4} =$ `"annal"`, $S_{p,5} =$ `"annul"`, and $S_{p,6} =$ `"annua"`.

If we search for r patterns, then we obtain m signatures from each, for a total of rm signatures. All the patterns have to be the same length. If this is not the case, they are truncated to the length of the shortest pattern.

Those rm signatures are stored in a hash table, which will be used for exact searching. To search the text, all m-length windows $t_it_{i+1} \ldots t_{i+m-1}$ are considered.

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

For each such window, all m signatures are obtained: $t_{i+2}t_{i+3}\dots t_{i+m-1}$, $t_{i+1}t_{i+3}\dots t_{i+m-1}$, $\dots$, $t_{i+1}t_{i+2}\dots t_{i+m-2}$. We abreviate the notation and call $S_{i,j} = S_{t_i\dots t_{i+m-1},j}$. Each such signature is searched for in the hash table and, if it is found, an occurrence is reported.

We now show that the method is correct. If p and a text occurrence p' match with one substitution error, so that $p_1p_2\dots p_{j-1}ap_{j+1}\dots p_m = p'_1p'_2\dots p'_{j-1}bp'_{j+1}\dots p'_m$, then p' and p are equal after removing a and b, and hence the occurrence will be found because the j-th signatures of p and the text window are the same. If they match with an insertion in p', $p_1p_2\dots p_{j-1}p_jp_{j+1}\dots p_m = p'_1p'_2\dots p'_{j-1}p'_jbp'_{j+1}\dots p'_{m-1}$, then since the text windows are of length m there will be a window $x = p'_1p'_2\dots p'_{j-1}p'_jp'_{j+1}\dots p'_{m-1}c$, and the signatures $S_{x,m} = S_{p,j+1}$ match. Finally, p and p' may match after a deletion in p', $p_1p_2\dots p_{j-1}p_jp_{j+1}\dots p_m = p'_1p'_2\dots p'_{j-1}p'_{j+1}\dots p'_m p'_{m+1}$. In this case the text window of interest is $x = p'_1p'_2\dots p'_{j-1}p'_jp'_{j+1}\dots p'_m$, since $S_{x,j} = S_{p,m}$.

The way to compute the hash function is important. A formula like

$$h(x_1\dots x_{m-1}) = \sum_{i=1}^{m-1} x_i d^{i-1} \mod s$$

for relative primes (d, s), as used in [KR87], is known to distribute the strings fairly uniformly in a table $H[0\dots s-1]$. Moreover, it permits computing the hash value of each signature of the new window in $O(1)$ time using those of the previous window. Say that the new window is $t_{i+1}t_{i+2}\dots t_{i+m}$. Then, its j-th signature is $S_{i+1,j} = t_{i+1}t_{i+2}\dots t_{i+j-1}t_{i+j+1}\dots t_{i+m}$, which is obtained from the $(j+1)$-th signature of the previous region $S_{i,j+1} = t_it_{i+1}\dots t_{i+j-1}t_{i+j+1}\dots t_{i+m-1}$ by the formula $S_{i,j+1}t_{i+m} = t_iS_{i+1,j}$. Hence, $h(S_{i+1,j}) = ((h(S_{i,j+1}) - t_i)/d + t_{i+m}d^{m-2}) \mod s$, which can be computed in constant time.

Since the hash values can be computed in constant time and we have to perform m searches per text window, the search time is $O(mn)$, independent of the number of patterns. We are not accounting for the collision problem in the hashing scheme. On average, the search time remains constant if the size of the hash table is proportional to the number of signatures inserted. Hence the method takes on average $O(mn)$ time and $O(rm)$ space.

The scheme works well in practice even for thousands of patterns. In this respect the method is unbeatable. On the other hand, it is costly to extend to more than one error. For k errors we should consider the $O(m^k)$ alternatives of removing k characters from every pattern and every text window, for a total average cost of $O(m^kn)$ time and $O(m^kr)$ space.

Figure 6.14 shows the algorithm. At any point in the execution it holds $h_j = h(S_{pos,j})$. The time of the preprocessing and the initial filling of h can be reduced by noticing that $S_{x,j+1} = (S_{x,j} + (x_j - x_{j+1})d^j) \mod s$. Hence the algorithm takes time $O(rm + mn)$ plus collisions.

```
MultiHash (P = {p^1, p^2, ..., p^r}, T = t_1 t_2 ... t_n, k = 1)
1.    Preprocessing
2.        H ← empty hash table
3.        For i ∈ 1...r Do
4.            For j ∈ 1...m Do insert (S_{p^i,j}, i) in H[h(S_{p^i,j})]
5.    Searching
6.        For j ∈ 1...m − 1 Do h_j ← h(S_{1,j})
7.        For pos ∈ 1...n − m + 1 Do
8.            oldh_1 ← h_1
9.            For j ∈ 1...m Do
10.               For (x, i) ∈ H[h_j] Do
11.                   If x = S_{pos,j} Then report pattern p^i at pos
12.               End of for
13.               h_j ← ((h_{j+1} − t_pos)/d + t_{pos+m} d^{m−2}) mod s
14.           End of for
15.           h_m ← oldh_1
16.       End of for
```

Fig. 6.14. Hashing-based scheme to search for multiple patterns with one error. We assume that H is a hash table where we insert pairs of the form $(key, value)$ and then retrieve the set of pairs associated with a given cell. We also assume that t_{n+1} can be accessed, although its value is irrelevant.

6.6.2 Partitioning into $k+1$ pieces

The algorithm **PEX** described in Section 6.5.1 is easily extended to multiple patterns [BYN97]. We call it **MultiPEX**. Given r patterns, we split each pattern into $k+1$ pieces. Then we proceed exactly as before: We perform a multipattern exact search for all those $r(k+1)$ pieces (Chapter 3), and each time a piece is found we check the corresponding pattern in the candidate text area. If a piece belongs to more than one pattern, then all the owners have to be checked. Hierarchical verification can be used as well.

The algorithm performs well under a wide range of cases. It is shown in [BYN97] that it can be applied whenever basic **PEX** can be applied, that is, $\alpha < 1/\log_{|\Sigma|} m$. The code is basically the same as that of Figure 6.11.

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

6.6.3 Superimposed automata

A third idea, which we call **MultiBP**, is based on the NFA of Figure 6.4. Given r patterns of the same length to be searched for with k errors, we build the NFA for each of them and then we *superimpose* their automata [BYN97]. Superimposition means that the j-th horizontal arrow can be crossed with the j-th character of *any* pattern. Figure 6.15 shows an example with the patterns `"annual"` and `"binary"`.

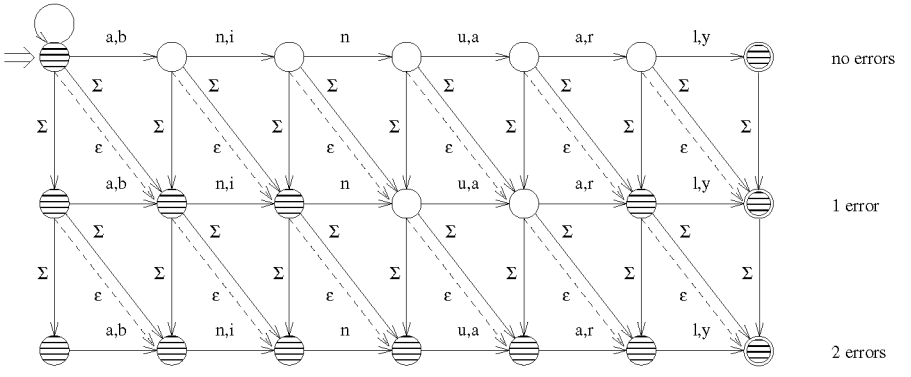

Fig. 6.15. An NFA for approximate string matching of the patterns `"annual"` and `"binary"` with two errors. The shaded states are those that are active after reading the text `"binual"`.

In particular we are interested in a bit-parallel simulation of the superimposed NFA. Let $B_i[c]$ be the bit-parallel table for the i-th pattern. Then we build a new table B, where

$$B[c] \quad = \quad B_1[c] \;\mid\; B_2[c] \;\mid\; \ldots \;\mid\; B_r[c]$$

and apply any of the algorithms suitable for single patterns, such as **BPR**, **BPD**, **BPM**, or **ABNDM**.

The result is equivalent to searching for a single pattern with classes of characters: We convert the search for $\{p^1,\ p^2,\ \ldots,\ p^r\}$ into the search for

$$\{p_1^1, p_1^2, \ldots, p_1^r\}\ \{p_2^1, p_2^2, \ldots, p_2^r\}\ \ldots\ \{p_m^1, p_m^2, \ldots, p_m^r\}$$

So it is not really necessary to use NFAs: Any bit-parallel algorithm can be used, in particular **BPM**.

Of course this is only a filter: If we search for `"annual"` and `"binary"`, then `"binual"` will be found with *zero* errors. Each time our relaxed search mechanism reports a match we have to check the area for all the patterns involved.

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

A new hierarchical verification mechanism is advisable here. If we have superimposed the patterns $\{p^1, p^2, p^3, p^4\}$ and found a possible occurrence, then we can check for $\{p^1, p^2\}$ and $\{p^3, p^4\}$ instead of checking for all four patterns. In many cases we will avoid performing r checks just by testing two superimposed sets. If, say, the superimposed set $\{p^1, p^2\}$ matches, then we have to check for p^1 and p^2 separately.

Compared to **PEX** (Section 6.5.1), this hierarchical verification mechanism is top-down rather than bottom-up. If we find the superimposition of $\{p^1, \ldots, p^r\}$, then we recursively check the relevant text area for the two superimposed sets $\{p^1, \ldots, p^{\lfloor r/2 \rfloor}\}$ and $\{p^{1+\lfloor r/2 \rfloor}, \ldots, p^r\}$. Of course all the $2r-1$ possible superimpositions are precomputed. The process finishes when we do not find the pattern set in the area or when a set of just one pattern is found.

As we superimpose more patterns, it becomes easier to cross the horizontal arrows. Indeed, the probability of crossing raises from $1/|\Sigma|$ to about $r/|\Sigma|$. Therefore, it is not advisable to superimpose too many patterns. The optimal number of patterns to superimpose is shown in [BYN97] to be $r' = |\Sigma|(1-\alpha)^2$. If there are more patterns, one should split them into groups of r' patterns and search each group separately.

Figure 6.16 shows the preprocessing and Figure 6.17 gives search pseudocode for this algorithm. The code is independent of how we simulate the bit-parallel search. Our recommendation is to use **BPD** if the patterns fit in a computer word, and **BPM** otherwise. We assume that all the preprocessing information is stored in an object B and that "joining" two such objects produces a new one that reflects their superimposition. For example, "joining" tables B_1 and B_2 into table B (in line 10 of **CreateSuperp**) is translated, for **BPR**, **ABNDM**, and **BPM**, into

For $c \in \Sigma$ **Do** $B[c] \leftarrow B_1[c] \mid B_2[c]$

BPD also needs to reflect these changes in table BB.

6.7 Searching for extended strings and regular expressions

Sometimes one would like to search for complex patterns allowing errors. There are three classes of algorithms addressing this issue: One extends classical dynamic programming for simple strings to regular expressions, a second is based on a Four-Russians approach, and the third uses bit-parallelism. We explain all three approaches but concentrate on bit-parallelism because it is simpler and yields the best results in most cases.

Since classes of characters are trivially solved by either approach, we focus

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

```
CreateSuperp (p^i, ..., p^j)
1.    Create new node
2.    If i = j Then
3.        B(node) ← preprocess single pattern p^i
4.        idx(node) ← i
5.        left(node) ← θ
6.        right(node) ← θ
7.    Else
8.        left(node) ← CreateSuperp(p^i ... p^⌊(i+j)/2⌋)
9.        right(node) ← CreateSuperp(p^(1+⌊(i+j)/2⌋) ... p^j)
10.       B(node) ← join B(left(node)) and B(right(node))
11.   End of if
12.   Return node
```

Fig. 6.16. Preprocessing for hierarchical verification of the superimposed search for multiple patterns.

```
Verify (node, from, to)
1.    B ← B(node)
2.    For pos ∈ occurrences reported with B in T_from...to Do
3.        If left(node) = θ Then report p^idx(node) at pos
4.        Else
5.            Verify(left(node), pos − m − k + 1, pos)
6.            Verify(right(node), pos − m − k + 1, pos)
7.        End of if
8.    End of for

MultiBP (P = {p^1, p^2, ..., p^r}, T = t_1 t_2 ... t_n, k)
9.    Preprocessing
10.       tree ← CreateSuperp(p^1 ... p^r)
11.   Searching
12.       Verify(tree, 1, n)
```

Fig. 6.17. Superimposition scheme to search for multiple patterns with errors.

on more complex extensions such as gaps, optional, and repeatable characters, and regular expressions.

6.7.1 A dynamic programming based approach

This is the oldest solution to the problem [MM89], and a beautiful yet complicated one. To understand it we need to come back to the basic dynamic programming algorithm (Section 6.2).

Consider the graph of Figure 6.18. Each node corresponds to a cell of

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

the dynamic programming matrix of Figure 6.1. The arrows between nodes represent the cost of insertion in the pattern (horizontal), deletion in the pattern (vertical), or matching/substitution (diagonal) among neighboring cells. The cost of the diagonal arrows is 0 or 1, depending on whether the corresponding characters are equal (match) or different (substitution).

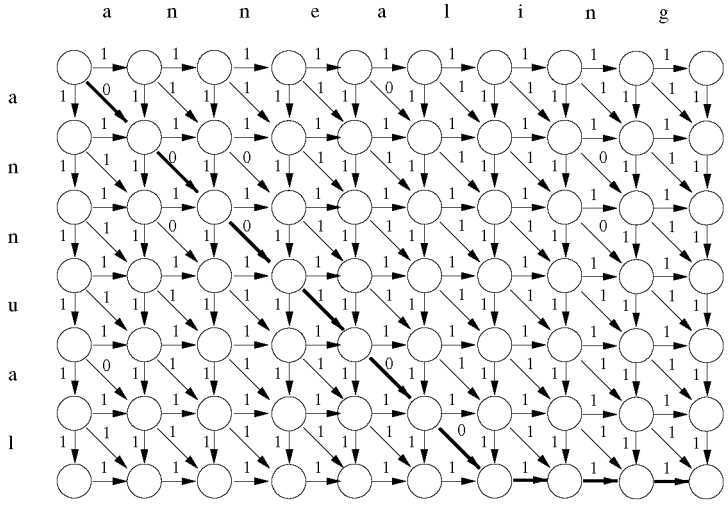

Fig. 6.18. Converting the edit distance problem into a shortest path problem. Bold arrows show the optimum path, of cost 4.

The edit distance problem can be converted into the problem of finding a shortest path from the upper left to the lower right node. If we are interested in approximate searching rather than in computing edit distance, then we assign zero cost to the horizontal arrows of the first row and consider minimum distances to every node of the last row.

Since the graph is acyclic, the optimum path can be computed in $O(mn)$ time. This is just another view of the classical dynamic programming algorithm, but this view is more flexible and can be extended to more complex patterns. In the simplest case, the pattern is represented by the vertical columns of nodes of the graph.

Figure 6.19 shows a graph over the text `"baa"`, where each "vertical" row of nodes has been replaced by the NFA of the regular expression `"(a|b)a*"` (Thompson's construction; see Section 5.2.1). It can be seen that in this case the distance is zero (i.e., the regular expression matches the text exactly), and that the best path is achieved thanks to an ε-transition.

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

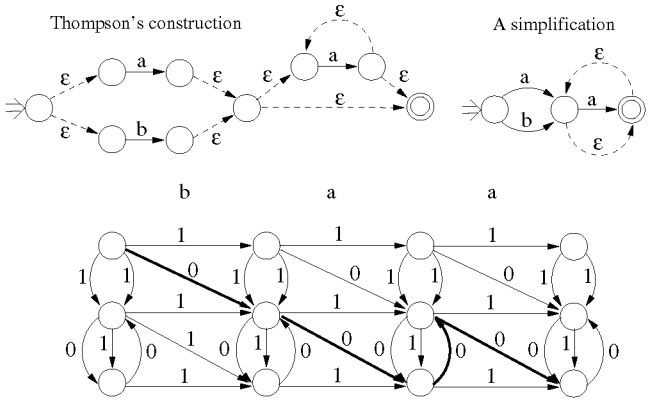

Fig. 6.19. The graph for the regular expression `"(a|b)a*"` on the text `"baa"`. Bold arrows show an optimal path, of cost zero.

The idea of the shortest path can still be applied quite easily if the graph is acyclic, that is, if the regular expression does not contain the "$*$" or the "$+$" operator. On acyclic regular expressions we can find a topological order to evaluate the graph so as to find the shortest paths in overall time $O(mn)$. This requires Thompson's guarantee that there are $O(m)$ edges on an automaton of m nodes.

Cycles in the NFA pose a problem because no suitable order can be found. The problem appears when we combine cycles with deletion (vertical) arrows, because a deletion can propagate through a cycle and influence the departing node. One of the most important results of [MM89] is that those "back edges" coming from the "$*$" or "$+$" operators can be ignored in a first pass, and then a second pass considering the deletion arrows is enough to obtain the correct result. For more details we refer the reader to [MM89].

6.7.2 A Four-Russians approach

We have already seen Four-Russians approaches that deal with regular expression searching without errors (Section 5.3.3) [Myc92] and with simple string matching allowing errors (Section 6.3) [WMM96]. Both methods obtain $O(mn/\log s)$ worst-case time provided $O(s)$ space is available, and the second method obtains $O(kn/\log s)$ average time with the technique of Section 6.2.3.

Both methods are based on similar ideas: An NFA of $O(m)$ states is split into r "regions" of m/r states each. For simple patterns [WMM96] a region is a contiguous pattern substring, while for regular expressions [Mye92] it is

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

some subset of the NFA states. Each region can be represented using $O(m/r)$ bits: For approximate searching of simple patterns we need $2m/r$ bits since each cell differs from the previous one by -1, 0, or $+1$, and hence two bits are enough to represent its value; for exact regular expression searching one bit per state (active or inactive) is enough.

A deterministic automaton, that is, a table storing all the outputs, is precomputed for each region, requiring $2^{O(m/r)}$ space per region. A nondeterministic automaton *of regions* simulates the original NFA arrows that connect different regions. Those arrows are simulated one by one. Either for simple patterns (where there are three arrows leading to each state; recall Figure 6.5) or for regular expressions (where regions are properly chosen and Thompson's construction guarantees $O(m)$ edges), there are $O(r)$ edges across modules. They have to be updated one by one, so the time is $O(rn)$. If we have $s = O(r)2^{O(m/r)}$ space, then we have $O(mn/\log s)$ time.

These ideas can be extended to the more general case of approximate searching for regular expressions [WMM95]. The idea is identical to that of exact searching, except that the states of the NFA are not just active or inactive, but store the minimum error level necessary to make each state active. Since we search with k errors, the value $k+1$ is used to denote any value larger than k. So for each state we need to store a number in the range $0 \ldots k+1$, and therefore a deterministic automaton on m/r states needs $O((k+2)^{m/r})$ space. Hence, given $O(s)$ space, the algorithm obtains $O(mn/\log_{k+2} s)$ time.

When faced with approximate searching, a new problem appears that does not exist with exact searching, namely, the problem of dependencies derived from ε-transitions in the regular expression. Just as in Section 6.7.1, a two-sweep algorithm guarantees that all the arrows are considered correctly.

A related work [Mye96] considers "spacers," which are what we have called "gaps" on PROSITE expressions (Section 4.3), except that a spacer can have a negative length. This means that a piece of the regular expression may overlap approximately with the next one in the occurrence. The idea is to search for one of the regular expressions and use its adjacent spacers to define the areas where its neighbor expressions should be searched for. The occurrence is extended until the complete pattern is found. The paper shows an optimal search order that considers the length of the spacers and the probability of matching the regular expressions.

The same work shows that if regular expressions are restricted to "network expressions," that is, no "$*$" or "$+$" is permitted, then it is possible to define the regions in increasing distance from the initial state and to apply

a technique similar to that of Section 6.2.3 to obtain an $O(kn/\log_{k+2} s)$ average time algorithm.

Note that positive-length gaps can be handled by converting them into regular expressions, but the resulting DFAs are unnecessarily large.

In general, the Four-Russians approach gives the best results for large regular expressions, but they are difficult to implement. A simpler approach that works well on reasonable-sized patterns is presented next.

6.7.3 A bit-parallel approach

Extending the bit-parallel algorithms we have seen in order to handle errors is quite straightforward [WM92b, Nav01b].

If we want to permit wild cards, then only **BPR** and **BPD** are able to handle them efficiently. And only **BPR** is flexible enough to handle all the extensions we are interested in. This is the algorithm we consider now.

Let us go back to Figure 6.4. Each row of the NFA is a replica of the nondeterministic automaton that searches for a single pattern. The replicas are linked together using the rule: "Vertical" arrows link the same states from row i to row $i+1$; while "diagonal" arrows, either dashed or not, link each state s at row i to the states, in row $i+1$, that can be reached from s in one transition (the "next" states).

This idea can be generalized to more complex automata [WM92b]. In particular, if we replace each row by the specialized bit-parallel automata developed in Chapters 4 and 5, the result is an NFA that is able to search for the corresponding extended pattern or regular expression with k errors. Moreover, this automaton can be searched in a "forward" manner as in Section 6.4.1 [WM92b] or in a "backward" manner as in Section 6.5.2 [Nav01b]. The only change with respect to the algorithms presented in this chapter is the bit-parallel simulation of the automata; the general mechanism is the same. Figure 6.20 shows an example for a regular expression.

To implement the "diagonal" transitions, we compute a table T_d, which for each state set D gives the bit mask of all the states reachable from D in one step. We have already built this table for regular expression searching (Section 5.4.2). For simple patterns it is simply $T_d[D] = (D << 1)$.

Assume that 1 represents active and 0 inactive. Let $f(c, D)$ be the pattern-type-dependent update function used to search without errors *without* the self-loop, and $f_0(c, D)$ *with* the self-loop.

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

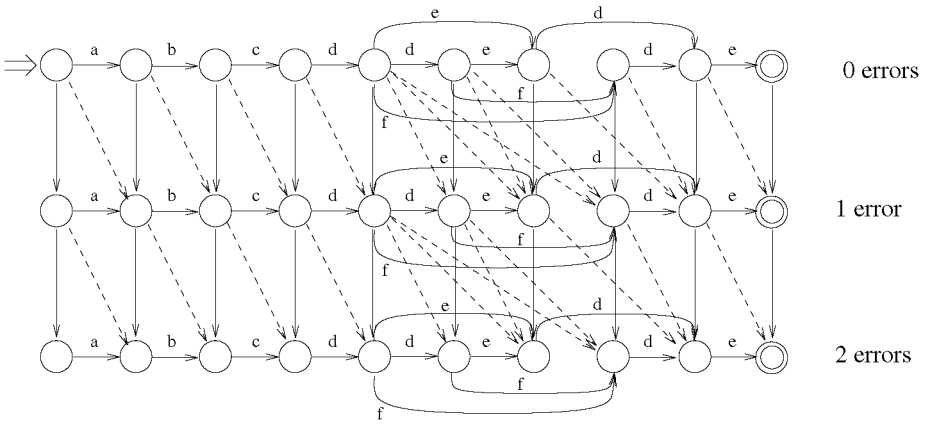

Fig. 6.20. Glushkov's NFAs for the regular expression `"abcd(d|ε)(e|f)de"` searched with two insertions, deletions, or substitutions. To simplify the figure, the dashed lines represent deletions and substitutions (i.e., they move by $\Sigma \cup \{\varepsilon\}$), while the vertical lines represent insertions (i.e., they move by Σ).

For example, for simple patterns, f corresponds to **Shift-And** (Section 2.2.2) and we have

$$\begin{aligned} f(c, D) &= (D << 1) \ \& \ B[c] \\ f_0(c, D) &= ((D << 1) \mid 0^{m-1}1) \ \& \ B[c] \end{aligned}$$

Note that it holds $T_d[D] = |_{c \in \Sigma} f(c, D)$.

Now, to update the rows after reading text character t_{pos}, we use

$$\begin{array}{l} R'_0 \leftarrow f_0(t_{pos}, R_0) \\ \textbf{For } i \in 1 \ldots k \textbf{ Do } R'_i \leftarrow f(t_{pos}, R_i) \mid R_{i-1} \mid T_d[R_{i-1} \mid R'_{i-1}] \end{array}$$

The formula can be plugged into the **BPR** and **ABNDM** algorithms.

It is also possible to deal with very limited cases of multipattern extended searches allowing errors by combining **BPR** or **ABNDM** with the multipattern technique explained in Section 4.6.

6.8 Experimental map

We now present a map of the most efficient approximate string matching algorithms, for single and multiple strings, leaving aside extended patterns and regular expressions.

There exist about 40 algorithms for approximate string searching. The best choices, however, are just a handful of them in most cases. We are

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

leaving aside algorithms that happen to be the best by a slight margin in a few cases in order to present a reasonably simple recommendation.

To give an idea of the areas where each algorithm dominates, Figure 6.21 shows the cases of English text and DNA. Since every filtration algorithm needs a nonfiltration algorithm for verification, we have presented the nonfiltration algorithms and superimposed in gray the area where the filters dominate. Therefore, in the grayed area the best choice is to use the corresponding filter with the dominating nonfilter as its verification engine. In the nongrayed area it is better to use directly the dominating nonfiltering algorithm.

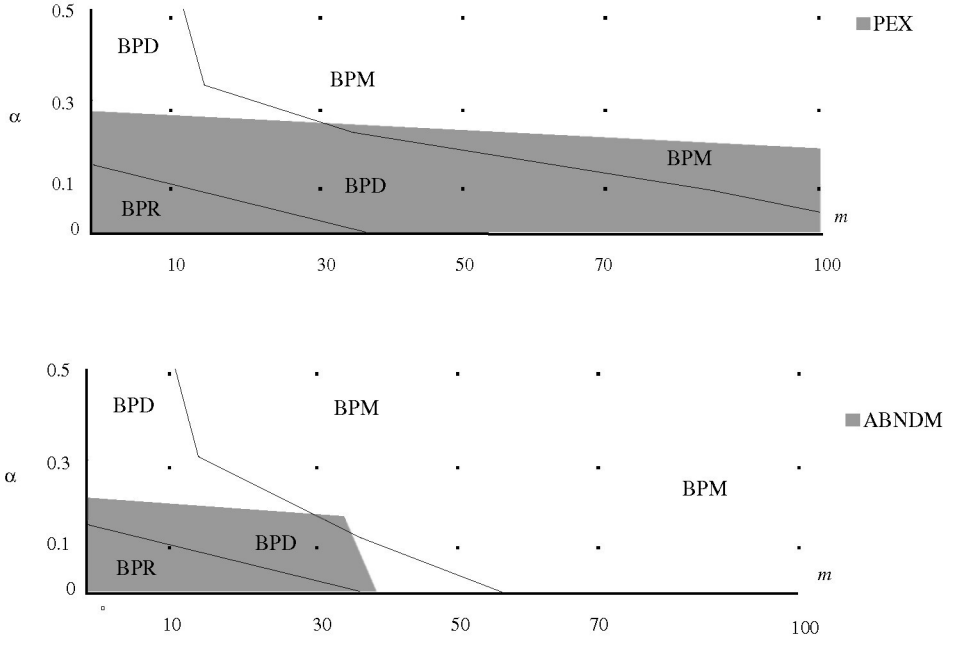

Fig. 6.21. The areas where each single pattern matching algorithm is best. Areas for filtering algorithms are gray. English text is on top and DNA on the bottom. The figures correspond to a word size of $w = 32$ bits. For $w = 64$ bits, the areas of **ABNDM** and **BPD** would grow on the m-axis.

Figure 6.22 shows the case of multipattern searching. On English text, **MultiPEX** is the best algorithm for $\alpha \leq 0.3$, **MultiBP** for $0.3 \leq \alpha \leq 0.4$, and for higher error levels no algorithm is known that improves over sequentially searching all r patterns. **MultiHash** is better for $k = 1$ and a large number of patterns. For longer computer word sizes the area of **MultiBP** would grow to the right along the m-axis.

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

On DNA there are few choices: For $k = 1$ **MultiHash** is in general the best option, while for $k > 1$ and low α value **MultiPEX** is of some interest. **MultiBP**, on the other hand, is in general not applicable because $|\Sigma| = 4$ and hence superimposing as few as 4 patterns means matching almost every text position.

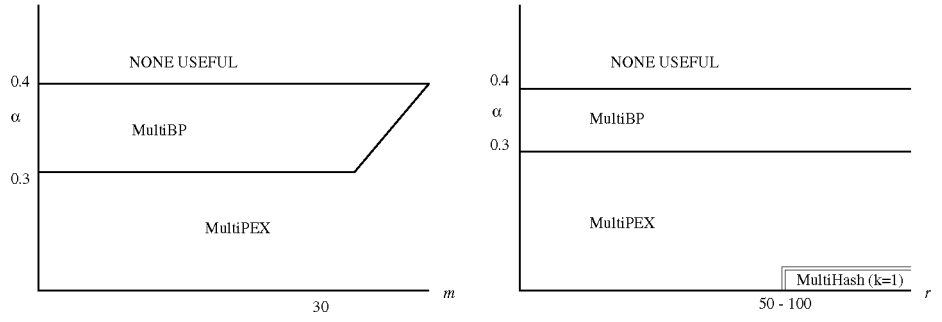

Fig. 6.22. The areas where each multipattern algorithm is best on English text, as a function of α, m, and r. In the left plot (varying m), we have assumed an r less than 50, while in the right plot we have assumed a pattern less than w characters.

6.9 Other algorithms and references

If one is interested in more complex distance functions, then the dynamic programming approach is the most flexible. For example, if the operations have different costs, we add the cost instead of adding 1 when computing $M_{i,j}$, that is,

$$\begin{aligned} M_{0,0} &\leftarrow 0 \\ M_{i,j} &\leftarrow \min(M_{i-1,j-1} + \delta(x_i, y_j), M_{i-1,j} + \delta(x_i, \varepsilon), M_{i,j-1} + \delta(\varepsilon, y_j)) \end{aligned}$$

where $\delta(x, \varepsilon)$ and $\delta(\varepsilon, y)$ are the cost of inserting and deleting characters.

For distances that do not allow some operations, we just take them out of the minimization formula or, equivalently, we assign ∞ to their δ cost. For transpositions (i.e., permitting $ab \to ba$ in one operation), we introduce a fourth rule that says that $M_{i,j}$ can be $M_{i-2,j-2} + 1$ if $x_{i-1}x_i = y_j y_{j-1}$ [LW75].

The automata approach can handle different *integer* costs for the operations, and some simplifications of the edit distance can be modeled by changing or removing arrows. For instance, if insertions cost 2 instead of 1, we make the vertical arrows go from rows i to rows $i + 2$ in Figure 6.4. Transpositions are more complex but can be modeled as well [Mel96, Nav01b].

https://doi.org/10.1017/CBO9781316135228.006 Published online by Cambridge University Press

All this can be expressed in **BPR** and **ABNDM**. Some restricted cases of different integral costs can be expressed in **BPD**. There is also some very recent work on extending **BPM** to accommodate different costs for the edit operations [BH01], to include transpositions [Hyy01], and to integrate it into **ABNDM** [HN01].

The **PEX** filter can be adapted, with some care, to other distance functions. The main issue is to determine how many pieces an edit operation can destroy and how many edit operations can be made before surpassing the error threshold. For example, a transposition can destroy two pieces in one operation, so we would need to split the pattern into $2k+1$ pieces to ensure that one is unaltered. A more clever solution [Nav01a] is to leave a hole of one character between consecutive pairs of pieces, so that one transposition cannot alter both.

Readers seeking a deeper coverage of approximate search issues for single patterns are referred to a recent survey [Nav01a]. For those interested in the distances and patterns used in biological applications, see [SK83, KM95].

There are models of approximate searching that deviate significantly from those we have covered. For example, there are totally different distance functions, such as Hamming distance (short survey in [Nav98]), reversals [KS95] (which allow reversing substrings), block distance [Ukk92, LT97] (which allows rearranging and permuting the substrings), swaps [KLPC99] (which are transpositions between nonadjacent characters), and so on. Although Hamming distance is a simplification of edit distance, specialized algorithms exist for it that go beyond our algorithms for edit distance.

With regard to the objects searched, they need not be only sequences of symbols. Extensions such as approximate searching in multidimensional texts (short survey in [BYN00b]) or in graphs [ALL97, Nav00] exist. Approximate searching of context-free grammars also has been pursued [Mye95]. None of these areas is well developed and the algorithms rely on the classical ones.

Finally, there are nonstandard algorithms, such as approximate (not to be confused with our exact algorithms for approximate searching), probabilistic, and parallel algorithms [TU88, AGM+90, LV89]. A good survey on the open theoretical problems in nonstandard stringology, including some results on Hamming distance, is [MP94].