An Overview of Two Fast Bit-Vector Approximate String-Matching Algorithms

Konstantin Tretyakov

June 2, 2003

1 Introduction

The intent of the article is to give a brief overview and a rough comparison of two well-known bit-parallel algorithms for fast approximate string matching. The first one was described by Gene Myers in article [2] and is a bit-parallel version of the basic algorithm based on dynamic programming. The second one is an enhanced “Shift-Or” type algorithm by Sun Wu and Udi Manber [3]. For sake of simplicity, only the most basic versions of the algorithms will be considered here. Considering the existence of great articles [1], [2] and [3], the whole idea of this work is pretty senseless, therefore it was mostly done only for my own fun and satisfaction.

2 Approximate String Matching Basics

2.1 The Problem

Suppose we are given two sequences of characters—pattern $P = p_1p_2 \dots p_m$ and text $T = t_1t_2 \dots t_n$. The problem of approximate string matching is to find all the locations in the text T, that contain the pattern P “approximately”. That is, we wish to find all the substrings in the text, that are similar to P under some measure of similarity. The most common measure used is the *edit distance* (also *Levenshtein distance*)—the minimum number of character insertions, deletions or substitutions required to obtain one string from another.

2.2 Edit Distance

As already mentioned in the previous paragraph, edit distance between two strings s and t is the minimum number of basic edit transformations (i.e. character insertions, deletions and substitutions) required to convert s into t. For example, edit distance between strings `man` and `mad` is 1, because the second is obtained from the first by a single character substitution, and the edit distance between the strings `cost` and `cat` is 2, because the second can be obtained from the first by deleting `o` and replacing `s` with `t`, whereas 2 is the least possible number of such operations. Here, edit distance between two strings $s_1s_2\dots s_m$ and $t_1t_2\dots t_n$will be denoted by $d(s_1s_2\dots s_m, t_1t_2\dots t_n)$ or simply $d(s,t)$.

It is easy to notice, that edit distance is a metric, that is,

$$\begin{aligned} & d(s,t) = 0 \Leftrightarrow s = t \\ & d(s,t) = d(t,s) \\ \forall u \quad & d(s,t) \le d(s,u) + d(u,t) \end{aligned}$$

The way to calculate edit distance is given by the following result:

Lemma 1 *The recursive formula for calculating edit distance:*

$$d(s_1s_2\dots s_m, t_1t_2\dots t_n) = \min \left\{ \begin{array}{ll} d(s_1s_2\dots s_{m-1}, t_1t_2\dots t_n) & +1 \\ d(s_1s_2\dots s_m, \quad t_1t_2\dots t_{n-1}) & +1 \\ d(s_1s_2\dots s_{m-1}, t_1t_2\dots t_{n-1}) & +\delta_{mn} \end{array} \right\}$$

where $\delta_{mn} = 1$, *if* $s_m = t_n$ *or* 0 *otherwise.*

The idea of proof may be the following: suppose we have an arbitrary sequence of basic transformations of minimal length, that changes $s_1s_2\dots s_m$ to $t_1t_2\dots t_n$. Notice that the order of transformations in this sequence does not matter, and that the number of transformations equals $d(s,t)$. Then there are 4 possibilities:

1. One of the basic transformations deletes s_m. Then we may perform this transformations first, so the total number of transformations in the sequence will equal 1 plus the minimum number of transformations needed to change $s_1s_2\dots s_{m-1}$ to $t_1t_2\dots t_n$. So in this case

$$d(s_1s_2\dots s_m, t_1t_2\dots t_n) = 1 + d(s_1s_2\dots s_{m-1}, t_1t_2\dots t_n)$$

2. One of the transformations substitutes s_m to t_k, where $k < n$. Then there must be a tranformation that inserts t_n. Reasoning analogous to the previous case shows that then

$$d(s_1s_2\dots s_m, t_1t_2\dots t_n) = d(s_1s_2\dots s_m, t_1t_2\dots t_{n-1}) + 1$$

3. One of the transformations substitutes s_m to t_n (so $s_m \ne t_n$). This possibility corresponds to the case where

$$d(s_1s_2\dots s_m, t_1t_2\dots t_n) = 1 + d(s_1s_2\dots s_{m-1}, t_1t_2\dots t_{n-1})$$

4. At last, there may be no transformation that deletes or substitutes s_m. It means that s_m must be matched against some t_k (so $s_m = t_k$). If $k < n$, then the situation is similar to the possibility 2, and if $k = n$, then

$$d(s_1 s_2 \dots s_m, t_1 t_2 \dots t_n) = d(s_1 s_2 \dots s_{m-1}, t_1 t_2 \dots t_{n-1})$$

By taking the minimum of these 4 possibilities we shall obtain the minimum possible number of transformations to change s into t, which is by definition $d(s,t)$. Thus we come to the required result.

□

This result gives way for a simple dynamic programming algorithm to calculate edit distance between $s_1 s_2 \dots s_m$ and $t_1 t_2 \dots t_n$. The algorithm calculates a $(m+1) \times (n+1)$ *dynamic programming (d.p.) matrix* $C[0 \cdots m, 0 \cdots n]$, such that $C[i,j] = d(s_1 s_2 \dots s_i, t_1 t_2 \dots t_j)$. The matrix is initially filled with values $C[0,j] = j$ for all j and $C[i,0] = i$ for all i. The remaining values of the matrix are calculated by the formula of *Lemma 1*:

$$C[i,j] = \min(C[i-1,j] + 1, C[i,j-1] + 1, C[i-1,j-1] + \delta_{ij})$$

After calculating the entire matrix this way, the value of $C[m,n]$ will be the edit distance $d(s,t)$. Here is an example of such a matrix calculated for the words `booze` and `looser`:

```
      0 1 2 3 4 5 6
        L O O S E R
      -------------
0    |0 1 2 3 4 5 6
1 B|1 1 2 3 4 5 6
2 O|2 2 1 2 3 4 5
3 O|3 3 2 1 2 3 4
4 Z|4 4 3 2 2 3 4
5 E|5 5 4 3 3 2 3
```

Here we find that the edit distance between these strings is 3. Using this matrix it is also possible to determine what exactly are the minimum sequences of transformations required to transform one string into another, but this problem wont be considered here.

2.3 Basic Algorithm for Approximate String Matching

Now suppose that we change the boundary condition for the dynamic programming matrix of the previous algorithm to the following: the first row of the matrix will be initialized with zeroes. That is, $C[0,j] = 0, \quad 0 \le j \le n$.

By a reasoning similar to the proof of *Lemma 1* it is possible to show, that in this case, $C[i,j]$ contains the minimum of edit distances between $s_1 s_2 \ldots s_i$ and all possible suffixes of $t_1 t_2 \ldots t_j$. That is,

$$C[i,j] = \min_{1 \le g \le j} (d(s_1 s_2 \ldots s_i, t_g t_{g+1} \ldots t_j))$$

For example, here is the corresponding matrix for strings `word` and `ordinaryworld`:

```
       0 1 2 3 4 5 6 7 8 910111213
         O R D I N A R Y W O R L D
       ---------------------------
   0   |0 0 0 0 0 0 0 0 0 0 0 0 0 0
   1 W|1 1 1 1 1 1 1 1 1 0 1 1 1 1
   2 O|2 1 2 2 2 2 2 2 2 1 0 1 2 2
   3 R|3 2 1 2 3 3 3 2 3 2 1 0 1 2
   4 D|4 3 2 1 2 3 4 3 4 3 2 1 1 1
```

Examining the last row of the matrix we may, for example, tell, that some substrings of `ordinaryworld` that end at positions 3, 11, 12 and 13, are only 1 transformation away from the string `word`. That is, we have found a way to determine all the locations in the first string (`ordinaryworld`) where some substring ends, that is k transformations away from the second string (`word`). This is exactly the solution to the approximate matching problem raised above. Moreover, knowing the whole matrix it is possible to determine not only the *locations* in the text, but the *substrings* themselves, that match the word being searched (in the above example these substrings are `ord`, `wor`, `worl` and `world`). Therefore we have a $O(mn)$ time and $O(mn)$ space algorithm to find all substrings in text, that match the pattern with at most k differences. A simple observation shows, that it is not necessary to store the whole matrix in memory, because in order to calculate the next column of the matrix it suffices to know only the previous column. By keeping in memory only that column we obtain an $O(m)$ space algorithm. Truth is, in this case the algorithm would report primarily the *locations* in the text and we lose a simple way to find the *substrings* of the text that match the pattern. However, this limitation can be overcome in several ways. For example, storing $m + k$ columns of the matrix helps. Another interesting idea uses the fact, that by searching in forward direction we obtain all the locations, where matching substrings *end*, then by searching in the reverse direction we can determine the locations, where the matching substrings *begin*. Anyway, the problem of finding the locations only is considered in this article.

2.4 Summary

Given text $t_1 t_2 \ldots t_n$, pattern $p_1 p_2 \ldots p_m$, and threshold k, the following algorithm reports all the locations j in the text, where for some substring

$t_g t_{g+1} \dots t_j$ the edit distance $d(p_1 p_2 \dots p_m, t_g, t_{g+1} \dots t_j)$ is less than or equal to k:

```
approx_basic(text[1..n], pattern[1..m], int k)
{
  int c1[0..m], c2[0..m];
  int *prev_column = c1, *cur_column = c2;

  // Initialize prev_column
  for (i = 0..m)
    prev_column[i] = i;

  cur_column[0] = 0;

  // Recalculate the matrix column under each text symbol
  for (j = 1..n)
  {
    for (i = 1..m)
      cur_column[i] = min(prev_column[i] + 1,
                          prev_column[i - 1] +
                            (text[j] == pattern[i]?0:1),
                          cur_column [i - 1] + 1);
    if (cur_column[m] <= k)
      report match at position j

    swap(cur_column, prev_column);
  }
}
```

3 Fast Bit-Parallel Computation of the D. P. Matrix

3.1 Introduction

The technique of bit-parallelism is common in string-matching algorithms. The idea is to exploit the fact, that computers perform their operations on *words* of several bits. Usually there are 32 or 64 bits in a single computer word, that is, 32 or 64 bit-operations can be performed simultaneously by the computer hardware. Therefore by parallelizing algorithms cleverly, we may achieve a considerable speedup factor of 32 or 64. Two different approaches will be considered here. The first is a bit-parallel version of the algorithm presented above. Assuming that the length of the pattern is no more than the size of the computer word (which will be further denoted by w), it computes the whole column of the dynamic programming matrix in a single step. The second approach, described in the next section, also assumes that the length of the pattern is shorter than w, and utilizes this by emulating a non-deterministic automaton using bit-parallelism. Both algorithms can be easily adapted to patterns longer than w, but that case is omitted here for sake of simplicity.

3.2 The Algorithm

A very clever algorithm by Gene Myers [2] performs the calculation of the next column of the dynamic programming matrix in an $O(1)$ time and $O(1)$ space, assuming that the length of the pattern does not exceed w. A very good explanation of it is given in the original article. Here I'll just try to give the basic idea.

So, suppose that the length of the pattern is not longer than w. Consider the dynamic programming matrix:

```
    0 1 2 3 4 5 6 7 8 9
      R E M A C H I N E
   +-------------------
0  |0 0 0 0 0 0 0 0 0 0
   |
1 M|1 1 1 0 1 1 1 1 1 1
   |
2 A|2 2 2 1 0 1 2 2 2 2
   |
3 T|3 3 3 2 1 1 2 3 3 3
   |
4 C|4 4 4 3 2 1 2 3 4 4
   |
5 H|5 5 5 4 3 2 1 2 3 4
```

Note that the differences between the neighbor elements of the matrix are not greater than 1. These differences will be denoted here as "vertical deltas" ($\Delta v[i,j] = C[i,j] - C[i-1,j]$) and "horizontal deltas" ($\Delta h[i,j] = C[i,j] - C[i,j-1]$). A single column of the matrix can be stored as a sequence of vertical deltas. As it is really true, that all deltas are always either -1, 0 or 1, then 2 bits suffices to store one vertical delta. The whole column j can be stored therfore in two bit vectors Pv_j and Mv_j, such that

$$Pv_j[i] = 1 \Leftrightarrow \Delta v[i,j] = +1$$
$$Mv_j[i] = 1 \Leftrightarrow \Delta v[i,j] = -1$$

And the problem is to calculate Pv_{j+1} and Mv_{j+1} knowing Pv_j and Mv_j.

Consider a square of 4 adjacent matrix entries: $C[i,j]$, $C[i-1,j]$, $C[i,j-1]$ and $C[i-1,j-1]$. Such a square will be further referenced as *cell* (i,j):

```
    0 1 2 3 4 5 6 7 8 9
      R E M A C H I N E
   +-------------------
0  |0 0 0 0 0 0 0 0 0 0
   |       +---+
1 M|1 1 1 0|1 1|1 1 1 1
   |       |   |
2 A|2 2 2 1|0 1|2 2 2 2
   |       +---+
3 T|3 3 3 2 1 1 2 3 3 3
   |
4 C|4 4 4 3 2 1 2 3 4 4
   |
5 H|5 5 5 4 3 2 1 2 3 4
```

Let's denote by Δv_{in} the vertical delta on the left edge of the cell, by Δv_{out}—the vertical delta on the right edge, by Δh_{in}—the horizontal delta on the top edge and by Δh_{out}—the horizontal delta on the bottom edge. Also define $Eq = \delta_{ij} =$ (if $p_i = t_j$ then 1 else 0).

Note that knowing Δv_{in}, Δh_{in} and Eq suffices to determine Δv_{out} and Δh_{out} for any cell. Besides, as there are only $3 \times 3 \times 2$ possible values for $(\Delta v_{in}, \Delta h_{in}, Eq)$, then the function $(\Delta v_{in}, \Delta h_{in}, Eq) \to (\Delta v_{out}, \Delta h_{out})$ performed by the cell, may be expressed in terms of boolean logic. Namely, denote the bit values

$$\begin{aligned} Pv_{in} = 1 &\Leftrightarrow \Delta v_{in} = +1 \\ Mv_{in} = 1 &\Leftrightarrow \Delta v_{in} = -1 \end{aligned}$$

and analogously for Pv_{out}, Mv_{out}, Ph_{out}, Mh_{out}, Ph_{in}, Mh_{in}. Then the cell function may be expressed by the following relations:

$$\begin{aligned} Xv &= Eq \text{ or } Mv_{in} \\ Pv_{out} &= Mh_{in} \text{ or not } (Xv \text{ or } Ph_{in}) \\ Mv_{out} &= Ph_{in} \text{ and } Xv \\ Xh &= Eq \text{ or } Mh_{in} \\ Ph_{out} &= Mv_{in} \text{ or not } (Xh \text{ or } Pv_{in}) \\ Mh_{out} &= Pv_{in} \text{ and } Xh \end{aligned}$$

The formulas can be controlled by simply checking the results for all possible inputs. These formulas provide a way to calculate the output of a cell using bit operations. Now consider the whole column j of cells $(1, j), (2, j) \ldots (m, j)$:

```
    0 1 2 3 4 5 6 7 8 9
      R E M A C H I N E
   +-------+---+-------
0  |0 0 0 0|0 0|0 0 0 0
   |       |   |
1 M|1 1 1 0|1 1|1 1 1 1
   |       |   |
2 A|2 2 2 1|0 1|2 2 2 2
   |       |   |
3 T|3 3 3 2|1 1|2 3 3 3
   |       |   |
4 C|4 4 4 3|2 1|2 3 4 4
   |       |   |
5 H|5 5 5 4|3 2|1 2 3 4
   |       +---+
```

The main goal is to obtain a way to quickly calculate Δv_{out} of every cell in the column, knowing Δv_{in} of each cell, Δh_{in} of the topmost cell (it is always 0), and Eq for each cell. This can be done in the following way:

- First, we define a bit-vector Eq_j such that $Eq_j[i] = \delta_{ij}$. As we need to obtain this vector in $O(1)$ time, it will be precomputed before executing the main algorithm. More specifically, for each letter c of the alphabet, a vector $Eq(c)$ will be precomputed such that $Eq(c)[i] = 1 \Leftrightarrow p_i = c$. Then at any text position j, we shall have $Eq_j = Eq(t_j)$. This precomputation will take us $O(m)$ time.[1]

- Secondly, if (Pv_k, Mv_k) and (Ph_k, Mh_k) encode the vertical and horizontal deltas in column k (e.g. $Ph_k[i] = 1 \Leftrightarrow \Delta h_{ij} = +1$, etc), then, according to the relations shown above:

$$\begin{aligned}
Ph_j[0] &= Mh_j[0] = 0 \\
Ph_j[i] &= Mv_{j-1}[i] \text{ or not } (Xh_j[i] \text{ or } Pv_{j-1}[i]) \\
Mh_j[i] &= Pv_{j-1}[i] \text{ and } Xh_j[i] \\
Pv_j[i] &= Mh_j[i-1] \text{ or not } (Xv_j[i] \text{ or } Ph_j[i-1]) \\
Mv_j[i] &= Ph_j[i-1] \text{ and } Xv_j[i]
\end{aligned}$$

  And the required initial values are:

$$\begin{aligned}
Pv_j[i] &= 1 \\
Mv_j[i] &= 0
\end{aligned}$$

[1]Note that though strictly speaking precomputation requires $O(m|\Sigma|)$ time (where $|\Sigma|$ is the size of the alphabet), I assume that filling a region of memory with similar bytes by using a function like `memset()` is done by the hardware in $O(1)$ time. This assumption is not very far from the truth, and using it, the preprocessing time can be estimated as $O(m)$.

- What remains is to determine the formulas for calculating the "X-factors" Xh_j and Xv_j. From their definition we have

$$\begin{aligned} Xv_j[i] &= Eq_j[i] \text{ or } Mv_{j-1}[i] \\ Xh_j[i] &= Eq_j[i] \text{ or } Mh_j[i-1] \end{aligned}$$

  The tricky part is to unwind the circularly dependent formula for $Xh_j[i]$:

$$\begin{aligned} Xh_j[i] =& Eq_j[i] \text{ or } Mh_j[i-1] = Eq_j[i] \text{ or } (Pv_{j-1}[i-1] \text{ and } Xh_j[i-1]) \\ =& Eq_j[i] \text{ or } (Pv_{j-1}[i-1] \text{ and } (Eq_j[i-1] \text{ or } Mh_j[i-2])) = \\ =& Eq_j[i] \text{ or } (Pv_{j-1}[i-1] \text{ and } Eq_j[i-1]) \\ & \quad \text{or } (Pv_{j-1}[i-1] \text{ and } Mh_j[i-2]) = \\ =& Eq_j[i] \text{ or } (Pv_{j-1}[i-1] \text{ and } Eq_j[i-1]) \\ & \quad \text{or } (Pv_{j-1}[i-1] \text{ and } (Pv_{j-1}[i-2] \text{ and } Xh_j[i-2])) = \\ =& Eq_j[i] \text{ or } (Pv_{j-1}[i-1] \text{ and } Eq_j[i-1]) \\ & \quad \text{or } (Pv_{j-1}[i-1] \text{ and} \\ & (Pv_{j-1}[i-2] \text{ and } (Eq_j[i-2] \text{ or } Mh_j[i-3])) = \\ =& Eq_j[i] \text{ or } (Pv_{j-1}[i-1] \text{ and } Eq_j[i-1]) \\ & \quad \text{or } (Pv_{j-1}[i-1] \text{ and } Pv_{j-1}[i-2] \text{ and } Eq_j[i-2]) \\ & \quad \text{or } (Pv_{j-1}[i-1] \text{ and } Pv_{j-1}[i-2] \text{ and } Mh_j[i-3]) = \\ =& \cdots = \\ =& \exists k \le i : \ Eq_j[k] \ \& \ \forall x \in [k, i-1] \ Pv_{j-1}[x] \end{aligned}$$

  In order to calculate $Xh_j[i]$ in a single bit-parallel operation, some kind of "bit-propagation" should be used, similar to that of carry propagation when adding binary numbers. In fact, the following formula does the job:

$$Xh_j = (((Eq_j \text{ and } Pv_{j-1}) + Pv_{j-1}) \text{xor } Pv_{j-1}) \text{ or } Eq_j$$

  where the logical operations are performed on all the elements of the bit-vectors simultaneously. A detailed proof of this result is available in the original article [2].

Thus, there is a way to calculate the whole column of the dynamic programming matrix in $O(1)$ time using bit-parallelism. In order to check for a match, however, we need the value of the last element in the column, "the score", and due to the encoding of the column using deltas, it would take $O(m)$ time to sum all the deltas to determine the score, thus ruining all the benefit of bit-parallel computation. So we do not sum the vertical deltas in the column, but add the horizontal delta Δh_{mj} to the previous value of the score instead. This completes the description of the algorithm.

3.3 Summary

The following algorithm solves the approximate string matching problem in $O(m+n)$ time and $O(|\Sigma|)$ space (where $|\Sigma|$ is the size of the alphabet), assuming the length of the pattern is not greater than the size of the computer word w:

```
approx_myers(text[1..n], pattern[1..m], int k)
{
  unsigned long PEq[1..S];

  precompute PEq[i];  // This is O(m)

  unsigned long Pv = (unsigned long) -1;
  unsigned long Mv = 0;
  int Score = m;
  unsigned long Eq, Xv, Xh, Ph, Mh;

  for (j = 1..n)
  {
    Eq = PEq[text[j]];
    Xv = Eq | Mv;
    Xh = (((Eq & Pv) + Pv) ^ Pv) | Eq;

    Ph = Mv | ~ (Xh | Pv);
    Mh = Pv & Xh;

    if (Ph & (1 << (m-1)) Score++;
    else if (Mh & (1 << (m-1)) Score--;

    Ph <<= 1;
    Mh <<= 1;

    Pv = Mh | ~(Xv | Ph);
    Mv = Ph & Xv;

    if (Score <= k)
      report match at position j
  }
}
```

4 The “Shift-Or” Style Approximate String Matching

4.1 Introduction

A totally different approach to using bit-parallelism is exploited in the algorithm by Sun Wu and Udi Manber ([3]). The idea to extend a simple “Shift-Or” exact string matching algorithm lead to a very powerful approximate string matching solution, that was implemented in a well-known `agrep` package. The algorithm allows numerous extensions, up to matching general regular expressions approximately, but again, here only the most basic version is taken into consideration.

4.2 Shift-Or

This is the case of exact string matching. As before, we are given text $t_1t_2\dots t_n$, pattern $p_1p_2\dots p_m$, threshold k, whereas $m < w$. We define a bit-vector array $R[1..m]$ and denote by R_j the value of the array after reading j-th text symbol. The bits in the array are defined by the following relation:

$$R_j[i] = 0 \Leftrightarrow p_1p_2\dots p_i \text{ matches the suffix of } t_1t_2\dots t_j$$

The following formulas are easily controlled:

$$\begin{aligned} R_0[i] &= 1 \quad \text{for all } i = 1..m \\ R_{j+1}[1] &= 0 \Leftrightarrow p_1 = t_j \\ R_{j+1}[i] &= 0 \Leftrightarrow R_j[i-1] = 0 \;\&\; p_i = t_j \end{aligned}$$

These formulas allow to calculate the whole bit-array in a single step using bit-parallelism in the following way:

$$R_{j+1} = (R_j << 1) \text{ or } Eq[t_j]$$

where $Eq[t_j]$ is a bit-vector such that $Eq[t_j][i] = 0 \Leftrightarrow t_j = p_i$. This vector may be precomputed for each alphabet symbol before executing the main algorithm (in $O(m)$ time). Then the text will be scanned symbol by symbol, and the array R_j calculated for every text symbol in $O(1)$ time. If at any moment $R_j[m] = 0$, we have a match. This is obviously an $O(m + n)$ time, $O(|\Sigma|)$ space algorithm:

```
shift_or(text[1..n], pattern[1..m])
{
  precompute Eq[1..S];
  unsigned long R = (unsigned long) -1;
  for (j = 1..n)
  {
    R = (R << 1) | Eq[text[j]];
    if (~R & (1 << (m-1)))
      report match at position j
  }
}
```

Here is an illustration of how the algorithm works with text `abababc` and pattern `abab`:

```
text:       A B A B A B C         Eq[A]   Eq[B]   Eq[C]
       +----------------
      A| 1 0 1 0 1 0 1 1            0       1       1
 R:   B| 1 1 0 1 0 1 0 1            1       0       1
      A| 1 1 1 0 1 0 1 1            0       1       1
      B| 1 1 1 1 0 1 0 1            1       0       1
match:           *   *
```

Note that this algorithm may be considered as an emulation of a non-deterministic automaton for matching the pattern. (Then the zeroes in the array R_j correspond to active states of the automaton).

4.3 Extension to Approximate Matching

The idea presented above may be easily extended to approximate matching. For the beginning suppose the problem of finding all matches of the pattern with at most one difference. Then, in addition to bit vector R_j (which will now be denoted by R_j^0), one more bit vector, R_j^1, will be kept, such that

$R_j^1[i] = 0 \Leftrightarrow p_1p_2\dots p_i$ matches a suffix of $t_1t_2\dots t_j$ with at most 1 difference.

The transitions for the new array are the following:

1. $R_{j+1}^1[i] = 0$, if $p_1p_2\dots p_i$ matches exactly a suffix of $t_1t_2\dots t_j$. Because then obviously $p_1p_2\dots p_i$ matches a suffix of $t_1t_2\dots t_{j+1}$ with one insertion.

2. $R_{j+1}^1[i] = 0$, if $p_1p_2\dots p_{i-1}$ exactly matches a suffix of $t_1t_2\dots t_{j+1}$. This case corresponds to a match with one deletion (where p_i was deleted).

3. $R_{j+1}^1[i] = 0$, if $p_1p_2\dots p_{i-1}$ exacly matches a suffix of $t_1t_2\dots t_j$Then $p_1p_2\dots p_i$ matches at position $j+1$ with at most one substitution.

4. $R_{j+1}^1[i] = 0$, if $p_i = t_{j+1}$ and $p_1p_2\dots p_i$ matches a suffix of $t_1t_2\dots t_j$ with at most one difference.

It is possible to show that these cases are exhaustive. Moreover, they allow to exploit bit-parallelism in transition for the whole array. Here is a complete rule to calculate R_j^1 for all j:

$$\begin{aligned} R_0^1 =& 1\dots 110 = (-1) << 1 \\ R_{j+1}^1 =& R_j^0 \text{ and} \\ & (R_{j+1}^0 << 1) \text{ and} \\ & (R_j^0 << 1) \text{ and} \\ & (Eq[t_{j+1}] \text{ or } (R_j^1 << 1)) \end{aligned}$$

The more general case of finding all approximate matches of at most k differences is handled similarly, only this time we keep track of $k+1$ arrays: $R_j^0, R_j^1, \dots R_j^k$ such that

$R_j^d[i] = 0 \Leftrightarrow p_1 p_2 \dots p_i$ matches a suffix of $t_1 t_2 \dots t_j$ with at most d differences.

The transition rules for these k arrays can be derived by exactly the same reasoning as in the case of the array R_j^1. And the formula is basically the same, namely:

$$\begin{aligned} R_0^d =& 1\dots 110 \dots (d \text{ times}) \dots 0 = (-1) << d \\ R_{j+1}^d =& R_j^{d-1} \text{ and} \\ & (R_{j+1}^{d-1} << 1) \text{ and} \\ & (R_j^{d-1} << 1) \text{ and} \\ & (Eq[t_{j+1}] \text{ or } (R_j^d << 1)) \end{aligned}$$

Thus the following $O(m + nk)$ time and $O(|\Sigma| + k)$ space algorithm is obtained: the text is scanned character-by-character and for every character all the $k+1$ arrays are recalculated. If at some text position j it is true that $R_j^k[m] = 0$, then an approximate match is reported.

4.4 Summary

This is the complete algorithm implementation:

```
approx_wu_manber(text[1..n], pattern[1..m], int k)
{
  precompute Eq[1..S];
  unsigned long R1[0..k], R2[0..k];
  unsigned long *Rprev = R1; *Rnew = R2;
  for (d = 0..k) Rprev[d] = (unsigned long)(-1) << d;

  for (j = 1..n)
  {
    Rnew[0] = (Rprev[0] << 1) | Eq[text[j]];

    for(d = 1..k)
    {
      Rnew[d] = Rprev[d-1] & (Rnew[d-1] << 1) &
                (Rprev[d-1] << 1) & ((Rprev[d] << 1) | Eq[text[j]]);
    }

    if (~Rnew[k] & (1 << (m-1)))
      report approximate match at position j

    swap(Rprev, Rnew);
  }
}
```

5 Comparison

A very rough comparison of the described algorithms is presented here. The goal is rather to demonstrate the speedup factor approximately, than to perform a comprehensive analysis of the algorithms speed in practice. The algorithm implementations used for the comparison are those of my own and they lack any of the possible optimizations. Three different files were used for testing:

- `words.dat`, 2.0 M — a list of english words.
- `dna.dat`, 2.0 M — a file with DNA sequence data.
- `random.dat` 2.0 M — a file with random character data.

The point of interest was to check the speed of the algorithms when applied to different file types searching for patterns of different lengths with different thresholds. The following simple ideas were ascertained by the tests:

1. Basically, the speed of the algorithms does not depend on the file type. True it is, that searching for the same pattern in different files would take a slightly different time, but this difference in speed is well correlated with the number of reported matches, therefore it is most probably not related to the algorithms themselves.
2. The characters used in the pattern and the length of the pattern do not matter for the bit-vector algorithms (however, analogously to the previous case, the choice of the pattern affects the number of reported matches, which in turn affect the running time slightly).

3. The choice of threshold does not affect the speed of the Myers' algorithm.
4. The time of execution of the algorithms is strictly directly proportional to the size of the input file (parts of the input files named above were used to test this).

These facts allow to present a sufficiently consistent and fair summary of the tests that considers only one file of a fixed size (2.0 M) and that does not focus on the exact values of the pattern used in searching.

The tests were performed on an Intel Celeron 800 Mhz processor with 128M RAM and $16 + 128$ K cache. Algorithms were implemented in *C* and were compiled by `gcc` with maximum compiler optimizations turned on.

5.1 The Basic Algorithm

Here are the average running times of the basic dynamic programming algorithm. The numbers represent the time in seconds needed to perform a certain search 10 times. Two measurements are given for each pattern length—the least time (attained when the pattern consisted only of symbols not present in the text) and the "worst" time (attained when the symbols of the pattern were commonly present in the text). The value of threshold is not mentioned, because it does not affect the running time much.

Pattern length	Running time
1	1.1–1.5
3	2.5–3.8
5	3.85–5.25
7	4.7–7.2
10	8.0–9.5
15	11.3–13.2
20	15.0–16.9
32	23.0–25.0

5.2 The Myers' Algorithm

The running time of this algorithm is nearly independent of both the threshold and the pattern length. The main factor that affected the running time was the number of matches reported—the fastest running time was attained when no match for a pattern could be found, and the longest running time was attained when every byte in the file was a match.

Running time
0.8–1.1

5.3 The Algorithm of Wu & Manber

Running times of this algorithm depend on the value of threshold k. They are presented in the following table.

Threshold	Avg. running time
1	0.8
3	1.4
5	1.8
7	2.8
10	3.7
15	4.95
20	6.5
32	9.7

5.4 Conclusions

The speed comparison shows the superiority of the Myers' algorithm, the running time of which does not depend neither on the pattern length nor on the threshold. The "Shift-Or" algorithm is also rather fast, which makes it a decent rival, especially considering the fact that it is extensible to matching regular expressions. And on the whole, even though the data presented in this section is very rough, it is possible to see that bit-parallelism does give substantial speedup (up to 32 times, which is seen from the test results: note that $25/0.8 \approx 32$). And that is probably the main and the sole point of this whole article.

References

[1] Navarro, G. 2001. A Guided Tour to Approximate String Matching. *ACM Comp. Surveys 33*, 1, 31–88.

[2] Myers, G. 1997. A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming. *J. ACM 46*, 395–415.

[3] Wu, S., and Manber, U. 1992. Fast Text Searching Allowing Errors. *Commun. ACM 35*, 10, 83–91.