New Bit-Parallel Indel-Distance Algorithm

Heikki Hyyrö[1], Yoan Pinzon[2]* and Ayumi Shinohara[1,3]

[1] PRESTO, Japan Science and Technology Agency (JST), Japan
heikki.hyyro@gmail.com
[2] Department of Computer Science, King's College, London, UK
pinzon@dcs.kcl.ac.uk
[3] Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan
ayumi@i.kyushu-u.ac.jp

Abstract. The task of approximate string matching is to find all locations at which a pattern string p of length m matches a substring of a text string t of length n with at most k differences. It is common to use Levenshtein distance [5], which allows the differences to be single-character insertions, deletions, or substitutions. Recently, in [3], the IndelMYE, IndelWM and IndelBYN algorithms where introduced as modified version of the bit-parallel algorithms of Myers [6], Wu&Manber [10] and Baeza-Yates&Navarro [1], respectively. These modified versions were made to support the indel distance (only single-character insertions and/or deletions are allowed). In this paper we present an improved version of IndelMYE that makes a better use of the bit-operations and runs 24.5 percent faster in practice. In the end we present a complete set of experimental results to support our findings.

1 Introduction

The *approximate string matching problem* is to find all locations in a text of length n that contain a substring that is similar to a query pattern string p of length m. Here we assume that the strings consist of characters over a finite alphabet. In practice the strings could for example be English words, DNA sequences, source code, music notation, and so on. The most common similarity measure between two strings is known as Levenshtein distance [5]. It is defined as the minimum number of single-character insertions, deletions and substitutions needed in order to transform one of the strings into the other. In a comprehensive survey by Navarro [7], the $O(k\lceil m/w\rceil n)$ algorithm of Wu and Manber (WM) [10], the $O(\lceil (k+2)(m-k)/w\rceil n)$ algorithm of Baeza-Yates and Navarro (BYN) [1], and the $O(\lceil m/w\rceil n)$ algorithm of Myers (MYE) [6] were identified as the most practical verification capable approximate string matching algorithms under Levenshtein distance. Here w denotes the computer word size. Each of these algorithms is based on so-called *bit-parallelism*. Bit-parallel algorithms make use of the fact that a single computer instruction operates on bit-vectors of w bits,

* Part of this work was done while visiting Kyushu University. Supported by PRESTO, Japan Science and Technology Agency (JST).

where typically $w = 32$ or 64 in the current computers. The idea is to achieve gain in time and/or space by encoding several data-items of an algorithm into w bits so that they can be processed in parallel within a single instruction (thus the name bit-parallelism).

In [3] the three above-mentioned bit-parallel algorithms were extended to support the indel distance. In this paper we improve the running time of one of those algorithms, namely, IndelMYE. IndelMYE is a modified version of Myers' algorithm [6] that supports indel distance instead of the more general Levenshtein distance. The new version (called IndelNew) is able to compute the horizontal differences of adjacent cell in the dynamic programming matrix more efficiently. Hence, the total number of bit-operations decreases from 26 to 21. We run extensive experiments and show that the new algorithm has a very steady performance in all cases, achieving a speedup of up to 24.5 percent compared with its previous version.

This paper is organised as follows. In Section 2 we present some preliminaries. In Sections 3 we explain the main bit-parallel ideas used to create the new algorithm presented in Section 4. In Section 5 we present extensive experimental results for the three bit-parallel variants presented in [3] and two dynamic programming algorithms. Finally, in Section 6 we give our conclusions.

2 Preliminaries

We will use the following notation with strings. We assume that strings are sequences of characters from a finite character set Σ. The alphabet size, *i.e.* the number of distinct characters in Σ, is denoted by σ. The ith character of a string s is denoted by s_i, and $s_{i..j}$ denotes the substring of s that begins at its ith position and end at its jth position. The length of string s is denoted by $|s|$. The first character has index 1, and so $s = s_{1...|s|}$. A length-zero empty string is denoted by ε.

Given two strings s and u, we denote by $ed(s, u)$ the edit distance between s and u. That is, $ed(s, u)$ is defined as the minimum number of single-character insertions, deletions and/or substitutions needed in order to transform s into u (or vice versa). In similar fashion, $id(s, u)$ denotes the indel distance between s and u: the minimum number of single-character insertions and/or deletions needed in transforming s into u (or vice versa).

The problem of approximate searching under indel distance can be stated more formally as follows: given a length-m pattern string $p_{1..m}$, a length-n text string $t_{1..n}$, and an error threshold k, find all text indices j for which $id(p, t_{j-h..j}) \leq k$ for some $1 \leq h < j$. Fig. 1 gives an example with $p =$"`ACGC`", $t =$"`GAAGCGACTGCAAACTCA`", and $k = 1$. Fig. 1(b) shows that under indel distance t contains two approximate matches to p at ending positions 5 and 11. In the case of regular edit distance, which allows also substitutions, there is an additional approximate occurrence that ends at position 17 (see Fig. 1(a)). Note that Fig. 1 shows a minimal *alignment* for each occurrence. For strings s and u, the characters of s and u that correspond to each other in a minimal transforma-

tion of s into u are vertically aligned with each other. In case of indel distance and transforming s into u, s_i corresponds to u_j if s_i and u_j are matched, s_i corresponds to ε if s_i is deleted, and ε corresponds to u_j if u_j is inserted to s. In case of Levenshtein distance, s_i corresponds to u_j also if s_i is substituted by u_j.

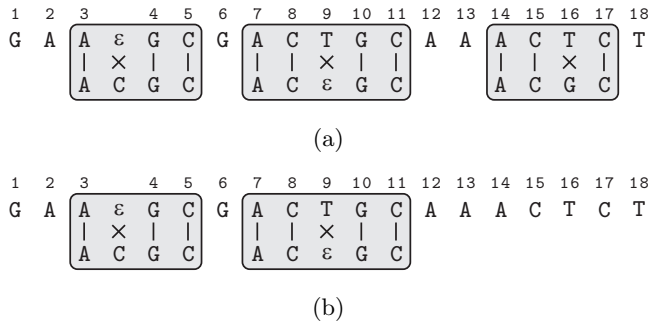


Fig. 1. Example of approximate string matching with $k = 1$ difference under (a) Levenshtein distance and (b) indel distance. Grey boxes show the matches and corresponding alignments. In the alignments we show a straight line between corresponding characters that match, and a cross otherwise. Hence the number of crosses is equal to the number of differences.

We will use the following notation in describing bit-operations: '&' denotes bitwise "AND", '|' denotes bitwise "OR", '^' denotes bitwise "XOR", '∼' denotes bit complementation, and '<<' and '>>' denote shifting the bit-vector left and right, respectively, using zero filling in both directions. The ith bit of the bit vector V is referred to as $V[i]$ and bit-positions are assumed to grow from right to left. In addition we use superscript to denote bit-repetition. As an example let $V = 1001110$ be a bit vector. Then $V[1] = V[5] = V[6] = 0$, $V[2] = V[3] = V[4] = V[7] = 1$, and we could also write $V = 10^2 1^3 0$. Fig. 2 shows a simple high-level scheme for bit-parallel algorithms. In the subsequent sections we will only show the sub-procedures for preprocessing and updating the bit-vectors.

Algo-BitParallelSearch($p_1 \ldots p_m$, $t_1 \ldots t_n$, k)
1. ▷ Preprocess bit-vectors
2. **Algo-PreprocessingPhase()**
3. **For** $j \in 1 \ldots n$ **Do**
4. ▷ Update bit-vectors at text character j and check if a match was found
5. **Algo-UpdatingPhase()**

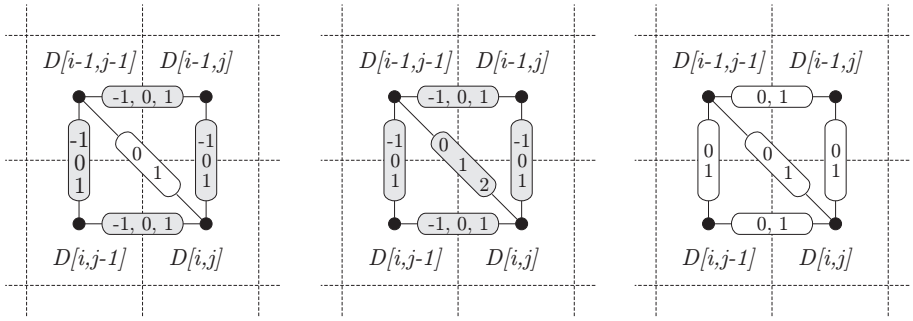
Fig. 2. A high-level template for bit-parallel approximate string matching algorithms.

3 Bit-parallel dynamic programming

During the last decade, algorithms based on bit-parallelism have emerged as the fastest approximate string matching algorithms in practice for the Levenshtein edit distance [5]. The first of these was the $O(kn(m/w))$ algorithm of Wu & Manber [10], where w is the computer word size. Later Wright [9] presented an $O(mn \log_\sigma /w))$ algorithm, where σ is the alphabet size. Then Baeza-Yates & Navarro followed with their $O((km/w)n)$ algorithm. Finally Myers [6] achieved an $O((m/w)n)$ algorithm, which is an optimal speedup from the basic $O(m/n)$ dynamic programming algorithm. With the exception of the algorithm of Wright, the bit-parallel algorithms dominate the other verification capable algorithms with moderate pattern lengths [7].

The $O(\lceil m/w \rceil n)$ algorithm of Myers [6] is based on a bit-parallelization of the dynamic programming matrix D. The $O(k\lceil m/w \rceil n)$ algorithm of Wu and Manber [10] and the $O(\lceil (k+2)(m-k)/w \rceil n)$ algorithm of Baeza-Yates and Navarro [1] simulate a non-deterministic finite automaton (NFA) by using bit-vectors.

For typical edit distances, their dynamic programming recurrence confines the range of possible differences between two neighboring cell-values in D to be small. Fig. 3 shows the possible difference values for some common distances. For both Levenshtein and indel distance, {-1,0,1} is the possible range of values for vertical differences $D[i,j]-D[i-1,j]$ and horizontal differences $D[i,j]-D[i,j-1]$. The range of diagonal differences $D[i,j]-D[i-1,j-1]$ is {0,1} in the case of Levenshtein distance, but {0,1,2} in the case of indel distance.



(a) edit distance (b) indel distance (c) lcs distance

Fig. 3. Differences between adjacent cells. White/grey boxes indicate that one/two bit-vectors are needed to represent the differences.

The bit-parallel dynamic programming algorithm of Myers (MYE) makes use of the preceding observation. In MYE the values of matrix D are expressed implicitly by recording the differences between neighboring cells. And moreover, this is done efficiently by using bit-vectors. In [4], a slightly simpler variant of

MYE, the following length-m bit-vectors Zd_j, Nh_j, Ph_j, Nv_j, and Pv_j encode the vertical, horizontal and diagonal differences at the current position j of the text:

— $Zd_j[i] = 1$ iff $D[i,j] - D[i-1,j-1] = 0$
— $Ph_j[i] = 1$ iff $D[i,j] - D[i,j-1] = 1$
— $Nh_j[i] = 1$ iff $D[i,j] - D[i,j-1] = -1$
— $Pv_j[i] = 1$ iff $D[i,j] - D[i-1,j] = 1$
— $Nv_j[i] = 1$ iff $D[i,j] - D[i-1,j] = -1$

The crux of MYE is that these difference vectors can be computed efficiently. The basic idea is that, given the vertical difference $D[i-1,j] - D[i-1,j-1]$ (left vertical difference in Fig. 4), the diagonal difference $D[i,j] - D[i-1,j-1]$ fixes the value of the horizontal difference $D[i,j] - D[i,j-1]$. And subsequently, in symmetric fashion, the diagonal difference also fixes the vertical difference $D[i,j] - D[i-1,j]$ after the previous horizontal difference $D[i,j] - D[i,j-1]$ is known. These observations determine the order in which MYE computes the difference vectors. The overall scheme is as follows. The algorithm maintains only the value of interest, $D[m,j]$, explicitly during the computation. The initial value $D[m,0] = m$ and the initial vectors $Pv_0 = 1^m$ and $Nv_0 = 0^m$ are known from the dynamic programming boundary values. When arriving at text position $j > 0$, MYE first computes the diagonal vector Zd_j by using Pv_{j-1}, Nv_{j-1} and $M(t_j)$, where for each character λ, $M(\lambda)$ is a precomputed length-m match vector where $M(\lambda)_i = 1$ iff $p_i = \lambda$. Then the horizontal vectors Ph_j and Nh_j are computed by using Zd_j, Pv_{j-1} and Nv_{j-1}. Finally the vertical vectors Pv_j and Nv_j are computed by using Zd_j, Nh_j and Ph_j. The value $D[m,j]$ is maintained incrementally during the process by setting $D[m,j] = D[m,j-1] + (Ph_h[m] - Nh_h[m])$ at text position j. A match of the pattern with at most k errors is found at position j whenever $D[m,j] \leq k$. Fig. 5 shows the complete MYE algorithm.

At each text position j, MYE makes a constant number of operations on bit-vectors of length-m. This gives the algorithm an overall time complexity $O(\lceil m/w \rceil n)$ in the general case where we need $\lceil m/w \rceil$ length-w bit-vectors in order to represent a length-m bit-vector. This excluded the cost of preprocessing the $M(\lambda)$ vectors, which is $O(\lceil m/w \rceil \sigma + m)$. The space complexity is dominated by the $M(\lambda)$ vectors and is $O(\lceil m/w \rceil \sigma)$. The difference vectors require $O(\lceil m/w \rceil)$ space during the computation if we overwrite previously computed vectors as soon as they are no longer needed.

4 IndelNew Algorithm

In this section we will present IndelNew, our faster version for IndelMYE which at the same time was a modification of MYE to use indel distance instead of Levenshtein distance.

As we noted before, indel distance allows also the diagonal difference $D[i,j] - D[i-1,j-1] = 2$. Fig. 4 is helpful in observing how this complicates the compu-

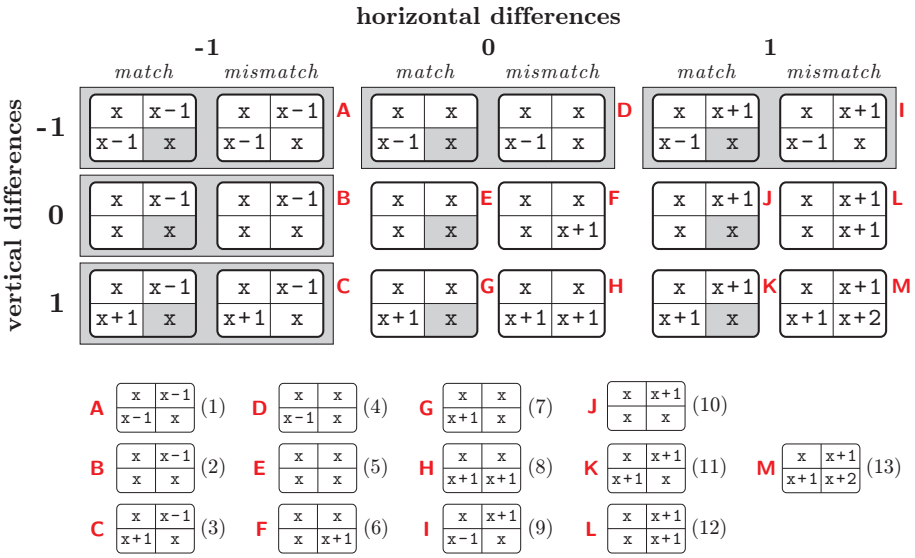


Fig. 4. The 13 possible cases when computing a D-cell.

tation of the difference vectors. It shows the 13 different cases that can occur in a 2×2 submatrix $D[i-1..i, j-1..j]$ of D. The cases are composed by considering all 18 possible combinations between the left/uppermost vertical/horizontal differences ($D[i,j-1]-D[i-1,j-1]/D[i-1,j]-D[i-1,j-1]$) and a match/mismatch between the characters p_i and t_j. Some cases occur more than once, so only 13 of them are unique.

We note that **M** is the only case where the diagonal difference is $+2$, and further that **M** is also the only case that is different between indel and Levenshtein distances: in all other cases the value $D[i,j]$ is the same regardless of whether substitutions are allowed or not. And since the diagonal, horizontal and vertical differences in the case **M** have only positive values, IndelNew can compute the 0/-1 difference vectors Zd_j, Nh_j, and Nv_j exactly as MYE. In the case of Levenshtein distance, the value $D[i,j]$ would be $x+1$ in case **M**, and hence the corresponding low/rightmost differences $D[i,j]-D[i,j-1]$ and $D[i,j]-D[i-1,j]$ would be zero. This enables MYE to handle the case **M** implicitly, as it computes only the -1/+1 difference vectors. But IndelNew needs to explicitly deal with the case **M** when computing the $+1$ difference vectors Ph_j and Pv_j, unless these vectors are computed implicitly/indirectly. The latter approach was employed in IndelMYE algorithm [3] by using vertical and horizontal zero difference vectors Zv_j and Zh_j, where $Zv_j[i] = 1$ iff $D[i,j] - D[i-1,j] = 0$, and $Zh_j[i] = 1$ iff $D[i,j] - D[i,j-1] = 0$. Then, solutions were found for computing Zv_j and Zh_j, and the positive difference vectors were then computed simply as $Ph_j = \sim (Zh_j \mid Nh_j)$ and $Pv_j = \sim (Zv_j \mid Nv_j)$. For IndelNew we propose

MYE-PreprocessingPhase

1. **For** $\lambda \in \Sigma$ **Do** $M(\lambda) \leftarrow 0^m$
2. **For** $i \in 1 \ldots m$ **Do** $M(p_i) \leftarrow M(p_i) \mid 0^{m-i}10^{i-1}$
3. $Pv_0 \leftarrow 1^m,\ Nv_0 \leftarrow 0^m,\ currDist \leftarrow m$

MYE-UpdatingPhase

1. $Zd_j \leftarrow (((M(t_j)\ \&\ Pv_{j-1}) + Pv_{j-1})\ ^\wedge\ Pv_{j-1})\ \mid M(t_j) \mid Nv_{j-1}$
2. $Nh_j \leftarrow Pv_{j-1}\ \&\ Zd_j$
3. $Ph_j \leftarrow Nv_{j-1} \mid\ \sim (Pv_{j-1} \mid Zd_j)$
4. $Nv_j \leftarrow (Ph_j << 1)\ \&\ Zd_j$
5. $Pv_j \leftarrow (Nh_j << 1) \mid\ \sim ((Ph_j << 1) \mid Zd_j)$
6. **If** $Ph_j\ \&\ 10^{m-1} \neq 0^m$ **Then** $currDist \leftarrow currDist + 1$
7. **If** $Nh_j\ \&\ 10^{m-1} \neq 0^m$ **Then** $currDist \leftarrow currDist - 1$
8. **If** $currDist \leq k$ **Then** Report a match at position j

Fig. 5. MYE algorithm. Variable $currDist$ keeps track of the value $D[m, j]$. The algorithm representations could be optimized to reuse the value $Ph_j << 1$ so that it is computed only once.

the following more efficient solution for computing Ph_j and Pv_j directly. The discussion assumes that $0 < i \leq m$ and $0 < j \leq n$.

Computing Ph_j We may observe from Fig. 4 that $Ph_j[i] = 1$ in the six cases **A, D, I, F, L,** and **M**. Cases **A, D,** and **I** arise from the negative vertical difference in column $j-1$, *i.e.* $Nv_{j-1}[i] = 1$. Cases **F** and **L** arise from a zero vertical difference in column $j-1$, i.e. $Nv_{j-1}[i] = 1$ and $Pv_{j-1}[i] = 0$, together with a positive diagonal difference, *i.e.* $Zd_j[i] = 0$. Hence the formula

$$Nv_{j-1} \mid (\sim Nv_{j-1}\ \&\ \sim Pv_{j-1}\ \&\ \sim Zd_j) = Nv_{j-1} \mid\ \sim (Pv_{j-1} \mid Zd_j)$$

covers the first five cases for the complete vectors, and this is enough for MYE under Levenshtein distance. Case **M** arises from having a positive difference in column $j-1$, a positive horizontal difference in row $i-1$, and a non-zero diagonal difference. This translates into the formula $Pv_{j-1}\ \&\ (Ph_j << 1)\ \&\ \sim Zd_j$, which contains a slightly problematic self-reference to Ph_j. We solve it as follows.

The self-reference states that case **M** can be true on row i only if one of the other five cases has happened above i. Let X be an auxiliary length-m bit-vector that covers the five cases, that is,

$$X = Nv_{j-1} \mid\ \sim (Pv_{j-1} \mid Zd_j).$$

Let Y be another auxiliary bit-vector so that

$$Y = Pv_{j-1}\ \&\ \sim Zd_j.$$

Now each set bit $Ph_j[i] = 1$ can be assigned to a distinct region $Ph_j[a..b] = 1^{b-a+1}$ of consecutive set bits in such manner, that $1 \leq a \leq i \leq b \leq m$, $X[a] = 1$,

$Y[a+1..b] = 1^{b-a}$ if $a < b$, and $Y[b+1] = 0$ if $b < m$. Moreover, the conditions $Y[a+1..b] = 1^{b-a}$ and $X[a] = 1$ are sufficient to imply that $Ph_j[a..b] = 1^{b-1+1}$. If we now shift the bit region $Y[a+1..b]$ one step right to overlap the positions $a \ldots b-1$ and then perform an arithmetic addition $Y[a..b] + X[a..b]$, the result is that the bits $Y[a..b-1]$ will change from 1 to 0 and the bit $Y[b]$ from 0 to 1. These changed bits can be set to 1, and thus to be correct values for $Ph_j[a..b]$, by performing XOR. Hence we have the formula

$$Ph_j = (X + Y) \wedge Y,$$

where Y has already been shifted one step right. We further note that if $Nh_j = Pv_{j-1} \,\&\, Zd_j$ has already been computed, we may set $Y = Pv_{j-1} \,\&\, \sim Zd_j = Pv_{j-1} - Nh_j$ in the beginning.

Computing Pv_j This step is diagonally symmetric with the case of Ph_j. After similar observations from Fig. 4 as before, the six relevant cases are seen to be **A, B, C, F, H**, and **M**, and the first five of these are covered by the formula $(Nh_j << 1) \mid \sim ((Ph_j << 1) \mid Zd_j)$. This time, case **M** has the formula $(Ph_j << 1) \,\&\, Pv_{j-1} \,\&\, \sim Zd_j$, which is straighforward to compute. As with the auxiliary variable Y, we may again use the fact that $Pv_{j-1} \,\&\, \sim Zd_j = Pv_{j-1} - Nh_j$. Then the complete formula for Pv_j becomes

$$Pv_j = (Nh_j << 1) \mid \sim ((Ph_j << 1) \mid Zd_j) \mid ((Ph_j << 1) \,\&\, (Pv_{j-1} - Nh_j)).$$

Fig. 7 shows the complete algorithm IndelNew for computing the difference vectors Zd_j, Nh_j, Ph_j, Nv_j, and Pv_j at text position j under indel distance. Obviously IndelNew has the same asymptotical time and space complexities as IndelMYE. Fig. 6 shows the complete algorithm IndelMYE as presented in [3]. IndelNew algorithm is able to compute the positive vectors directly. IndelMYE algorithm's main drawback is the way the horizontal solution is computed. All in all, the total number of bit-operations is 26 for IndelMYE versus 21 for IndelNew, so we have a more efficient implementation for a bit-parallel indel algorithm.

5 Experiments

We compare IndelNew against several other approximate string matching algorithms for indel distance. They are: IndelWM (our own implementation), IndelMYE (our own implementation), IndelBYN (a modification of the original code by Baeza-Yates and Navarro), and IndelUKK (our own implementation of the cutoff version of Ukkonen [8]). We also implemented a plain dynamic programming algorithm (without bit-parallelism) but it was too slow for the pattern lengths we used, therefore we removed it from the final test.

The computer used for testing was a 3.2Ghz AMD Athlon64 with 1.5 GB RAM running Linux. The computer word size was w=64. All code was compiled with GCC 3.3.2 and optimization switched on. We tested on three different 40MB texts. The first was composed by repeating the yeast genome. The second

IndelMYE-UpdatingPhase

1. $D' \leftarrow (((K_{T_j} \& Pv) + Pv) \wedge Pv) \mid K_{T_j} \mid Nv$
2. $X \leftarrow (Pv \ \& \ (\sim D')) >> 1$
3. $Y \leftarrow (Zv \ \& \ D') \mid ((Pv \ \& \ (\sim D')) \ \& \ 0^{m-1}1)$
4. $Zh' \leftarrow (X' + Y') \wedge X'$
5. $Nh' \leftarrow Pv \ \& \ D'$
6. $Ph' \leftarrow \ \sim (Zh' \mid Nh')$
7. $Zv' \leftarrow (((Zh' << 1) \mid 0^{m-1}1) \ \& \ D') \mid ((Ph' << 1) \ \& \ Zv \ \& \ (\sim D'))$
8. $Nv' \leftarrow (Ph' << 1) \ \& \ D'$
9. $Pv' \leftarrow \ \sim (Zv' \mid Nv')$
10. **If** $Ph' \ \& \ 10^{m-1} \neq 0^m$ **Then** $currDist \leftarrow currDist + 1$
11. **If** $Nh' \ \& \ 10^{m-1} \neq 0^m$ **Then** $currDist \leftarrow currDist - 1$
12. **If** $currDist \leq k$ **Then** Report a match at position j

Fig. 6. IndelMYE algorithm as presented in [3].

IndelNew-UpdatingPhase

1. $Zd_j \leftarrow (((M(t_j) \ \& \ Pv_{j-1}) + Pv_{j-1}) \wedge Pv_{j-1}) \ \mid M(t_j) \mid Nv_{j-1}$
2. $Nh_j \leftarrow Pv_{j-1} \ \& \ Zd_j$
3. $X \leftarrow Nv_{j-1} \mid \ \sim (Pv_{j-1} \mid Zd_j)$
4. $Y \leftarrow (Pv_{j-1} - Nh_j) >> 1$
5. $Ph_j \leftarrow (X + Y) \wedge Y$
6. $Nv_j \leftarrow (Ph_j << 1) \ \& \ Zd_j$
7. $Pv_j \leftarrow (Nh_j << 1) \mid \ \sim ((Ph_j << 1) \mid Zd_j) \ \mid ((Ph_j << 1) \ \& \ (Pv_{j-1} - Nh_j))$
8. **If** $Ph_j \ \& \ 10^{m-1} \neq 0^m$ **Then** $currDist \leftarrow currDist + 1$
9. **If** $Nh_j \ \& \ 10^{m-1} \neq 0^m$ **Then** $currDist \leftarrow currDist - 1$
10. **If** $currDist \leq k$ **Then** Report a match at position j

Fig. 7. IndelNew algorithm. The value $Pv_{j-1} - Nh_j$ could be reused.

was built from a sample of Wall Street Journal articles taken from the TREC collection. The third text was random with alphabet size $\sigma = 120$. The tested pattern lengths were $m = 16$, 32, and 64, and we tested over k=1 $\ldots m-2$. The patterns were selected randomly from the text, and each (m, k) combination was timed by taking the average time over searching for 100 patterns.

Fig. 8 shows the results. It can be seen that IndelWM is competitive with low k, being always the best when k=1. The performance of IndelBYN depends highly on the effectiveness of its "cutoff" mechanism, which in turns depends on the alphabet size σ. With DNA its performance becomes poor quite quickly when k grows (when $m = 16$, the NFA of IndelBYN requires more than $w = 64$ bits when $k = 3, 4, 5, 6, 7, 8, 9, 10, 11$. This is a remnant from how the case $m = 8$ did always fit into $w = 32$ bits). On the other hand, IndelBYN performs well with random text and moderately large alphabet size $\sigma = 120$. As expected due to its independence on k, IndelMYE/IndelNew has a very steady performance in

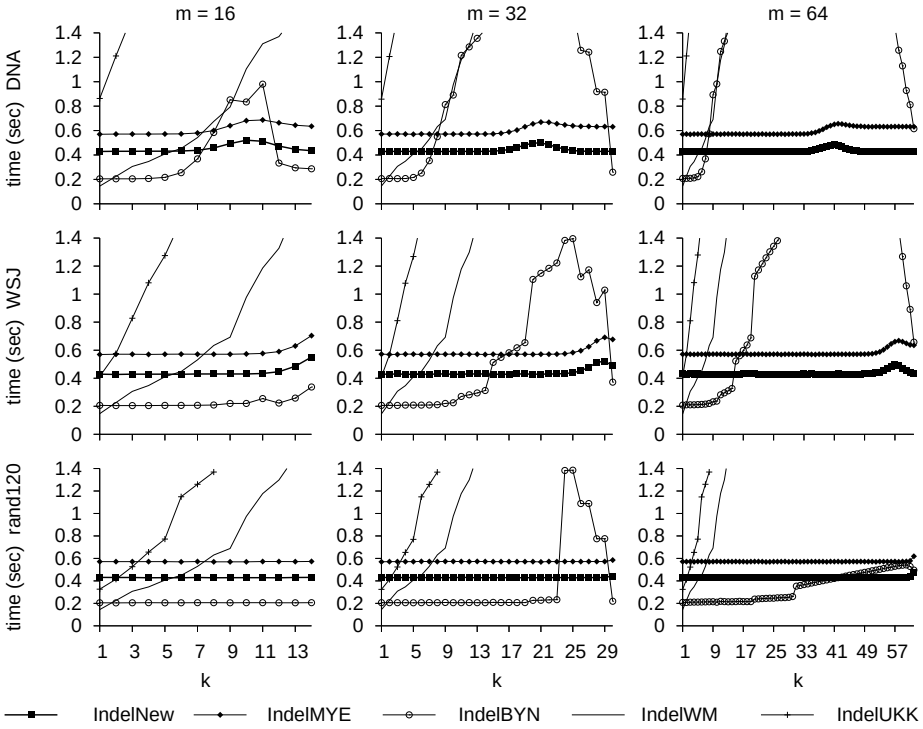


Fig. 8. The average time for searching for a pattern in a 40 MB text. The first row is for DNA (a repeated yeast genome), the second row for a sample of Wall Street Journal articles taken from TREC-collection, and the third row for random text with alphabet size $\sigma = 120$.

all cases. But IndelNew was 24.5 percent faster than IndelMYE. And IndelNew is the fastest of all in those cases where m and k are moderately large.

6 Conclusions

We have presented a new algorithm based on bit-parallelism that solves the problem of approximate string matching problem with k differences under the indel edit distance measure, namely, IndelNew. IndelNew is a more thought version of the early IndelMYE version in [3]. In our experiments the speedup gain by the new version was higher (24.5 percent) than the improvement in the number of bit-operations (about 19 percent – $26 \rightarrow 21$). IndelNew showed a very steady performance in all cases due to its independence on k. It is the fastest in those cases where m and k are moderately large and the cutoff scheme of IndelBYN does not work well.

We plan to use some of the ideas presented in [2] to search several text segments in parallel by encoding several copies of the pattern (or its prefixes) into a single bit-vector. This is left as a future work.

References

1. R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.
2. H. Hyyrö, K. Fredriksson and G. Navarro. Increased Bit-Parallelism for Approximate String Matching in *Proc. 3rd Workshop on Efficient and ExperimentalAlgorithms (WEA 2004)*, LNCS 3059, 285–298, 2004.
3. H. Hyyrö, Y. Pinzon and A. Shinohara. Fast Bit-Vector Algorithms for Approximate String Matching under Indel Distance in *Proc. 31st Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2005)*, LNCS 3381, 380–384, 2005.
4. H. Hyyrö. Explaining and extending the bit-parallel approximate string matching algorithm of Myers. Technical Report A-2001-10, Dept. of Computer and Information Sciences, University of Tampere, Tampere, Finland, 2001.
5. V. I. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones (original in Russian). *Russian Problemy Peredachi Informatsii 1*, 12–25, 1965.
6. G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *Journal of the ACM*, 46(3):395–415, 1999.
7. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
8. Esko Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.
9. A. Wright. Approximate string matching using within-word parallelism. *Software Practice and Experience*, 24(4):337–362, April 1994.
10. S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, October 1992.