Multiple Pattern Matching

Stephen Fulwider and Amar Mukherjee
College of Engineering and Computer Science
University of Central Florida
Orlando, FL USA
Email: {stephen,amar}@cs.ucf.edu

***Abstract*—In this paper, we consider certain generalizations of string matching problems. The multiple pattern matching problem is that of finding all occurrences of a set of patterns in a fixed text. This is a well studied problem, and several popular Unix utilities (grep, agrep, and nrgrep, to name a few) implement a host of algorithms to solve this problem. In this paper, we present both exact and approximate multiple pattern matching, using a uniform paradigm that generalizes the Shift-AND method of Baeza-Yates and Gonnet. Our algorithm is able to achieve better performance in certain searching scenarios when compared with the Unix utilities agrep and nrgrep, and should be considered being added to the grep family of searching algorithms.**

***Keywords*-string matching, approximate string matching, multiple approximate string matching.**

I. Introduction

In this paper, we consider certain generalizations of string matching problems. The most well known exact string matching algorithms are the Knuth-Morris-Pratt algorithm [1] and the Boyer-Moore algorithm [2]. A generalization to multiple pattern matching using keyword trees was proposed by Aho-Corasick [3]. Baeza-Yates and Gonnet [4] proposed a very fast and practical exact pattern matching algorithm in which the length of the pattern n does not exceed the length of a typical integer word (typically 32) in a computer so that bit shift and logical AND operations can be assumed to take a constant amount of time. The worst case time complexity is $O(m)$ where m is the length of the text and the algorithm takes $O(n)$ storage. They also proposed a generalization of the algorithm to handle arbitrary patterns with errors, called *approximate string matching*. Based on this and other independent ideas, Wu and Manber [5], [6] developed a whole suite of fast algorithms that can handle exact multiple patterns and approximate matches, patterns with unlimited wild card characters, patterns expressed by *regular expressions* and patterns with arbitrary cost for different edit operations (replacement, insertion and deletion operations). More recently, Külekci [7] developed a competitive exact multi-pattern matching algorithm for fixed length patterns by exploiting bit-parallelism. Additionally, many approximate string matching algorithms for multiple patterns have been developed which use a wide range of techniques and vary in performance based on the size and type of input [8], [9], [10]. Many different approximate string matching algorithms have been proposed in the literature including the class of algorithms called the *sequence alignment* algorithms using dynamic programming formulations. Interested readers are referred to the book by Dan Gusfield [11] and a recent research monograph by Adjeroh, Bell and Mukherjee [12].

An earlier version of this paper due to a regrettable oversight did not mention the following reference [5] which essentially presents the same approach we describe here. Our implementation has some advantages to their approach which will be described in Section III and IV.

We formulate the problem discussed in this paper as follows: The input to the problem is a set of patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_s\}$ of size $n = \sum |P_i|$ and a text T of size m, both over a finite alphabet Σ. We will consider a generalization of the approximate string matching problem. The output is all substrings (a set of consecutive characters in T) and subsequences (a set of not necessarily consecutive characters in T) that are close to the set of patterns $\mathcal{P}$ under some similarity measures. We adopt the *edit distance* or the *Levenshtein* distance as the similarity measure. In particular, we want to find all patterns in T such that the number of edit operations do not exceed $\{k_1, k_2, \ldots, k_s\}$ from patterns $\{P_1, P_2, \ldots, P_s\}$, respectively. A string P_i, $(1 \leq i \leq s)$, is said to be at an edit distance k_i to a subsequence Q in T if we can transform P_i to Q with a sequence of k_i insertions of single characters in arbitrary places in P_i, deletions of single characters in P_i, or replacements of a character in P_i by a character in Q. If all edit operations are replacements, it is equivalent to the so-called *Hamming distance or mismatch* measure. The quantity k_i is typically a small positive integer $(0 \leq k_i \leq c)$ where c is a constant. If all k_i's are integer 0, the problem is reduced to the *exact multiple pattern* matching problem. Most approximate string matching algorithms reported in the literature assume k_i to be constant for all patterns.

We develop the exact and approximate algorithms and all their generalizations using a uniform paradigm that generalizes the Shift-AND method of Baeza-Yates and Gonnet. In this respect, it is the same approach used by Wu and Manber [5], but we allow k_i to be different for different patterns. We also use a simplified formulation of the problem using only six recurrence expressions and provide formal proofs of correctness of all the algorithms presented. We

 ISBN: 978-1-61208-111-3

Table I
COMPLETE M MATRIX FOR $\mathcal{P} = \{abc, axa, bc\}$ AND $T = \epsilon baxabcx$

	T	ϵ	b	a	x	a	b	c	x
P		0	1	2	3	4	5	6	7
a	1	0	0	1	0	1	0	0	0
b	2	0	0	0	0	0	1	0	0
c	3	0	0	0	0	0	0	1	0
a	4	0	0	1	0	1	0	0	0
x	5	0	0	0	1	0	0	0	0
a	6	0	0	0	0	1	0	0	0
b	7	0	1	0	0	0	1	0	0
c	8	0	0	0	0	0	0	1	0

Table II
COLUMN 5 TO COLUMN 6 OF M WITH INTERMEDIATE STEPS SHOWN

Column 5	Shift-Or(5)	$U(c)$	Column 6
0	1	0	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	0	0
0	0	0	0
1	1	0	0
0	1	1	1

show that such an approach leads to very fast and practical performance exceeding the performance of the best algorithms available in the Unix utilities. We use the Baeza-Yates paradigm of using a *binary* valued matrix M in which a few simple operations allow us to extend the algorithm to multiple pattern matching, both in the exact and approximate case.

Exact and approximate string matching algorithms find applications in database search, data mining, text processing and editing, lexical analysis of computer programs, data compression and cryptography. In recent years, string matching and sequence alignment algorithms have been used extensively in the study of comparative genomics, proteomics, disease identification, drug design and molecular evolution theory. The remainder of the paper is organized as follows: Section II lays out the exact multiple pattern matching algorithm, Section III gives its approximate matching counterpart, Section IV gives our experimental results, and we make our conclusions in Section V.

II. EXACT MULTIPLE PATTERN MATCHING

This section is included for the sake of completeness and to set up our notations. We also include a formal proof of correctness.

Let $\mathcal{P} = \{P_1, P_2, \ldots, P_s\}$ be a set of patterns of size $n = \sum |P_i|$ and T be the text of size m preceded by a character ϵ which does not occur in any pattern. Define $P = P_1 P_2 \cdots P_s$ to be the concatenated patterns and M to be an $n \times (m+1)$ *binary* valued matrix with i running from 1 to n and j running from 0 to m. Entry $M(i, j)$ is 1 iff characters indexed by r through i of P exactly match the characters indexed by $i - r + 1$ of T ending at the character indexed by j, where r is the starting index of the pattern containing the character indexed by i.

The complete M matrix for $\mathcal{P} = \{abc, axa, bc\}$ and $T = \epsilon baxabcx$ is shown in Table I.

Notice that $M(5, 3)$ is 1, indicating that ax of the second pattern ($r = 4$, $i = 5$) matches the last 2 characters of T ending at position 3, ax. Anywhere a 1 exists at the end of a pattern indicates an occurrence of that pattern being found in the text (for example, $M(6, 4)$, $M(3, 6)$, and $M(8, 6)$). Hence, computing the s rows ending each pattern solves the exact multiple pattern matching problem.

The algorithm first constructs an n-length binary vector $U(x)$ for each character x of the alphabet. $U(x)$ is set to 1 for the positions in P where character x appears. From the example above, $U(a) = 10010100$.

The algorithm also constructs S and F, n-length binary vectors giving the start and end indices of all the patterns, respectively. Formally, $S(i)$ is 1 iff a pattern from $\mathcal{P}$ begins at i in P. Similarly, $F(i)$ is 1 iff a pattern from $\mathcal{P}$ ends at i in P. From the example above, $S = 10010010$ and $F = 00100101$.

Define $Shift\text{-}Or(j-1)$ as the vector derived by shifting the vector for *column* $j - 1$ down by one position and setting all *on* bits in S to 1. The previous bit in position n disappears. In other words, shift column $j - 1$ down by 1 and perform a bitwise OR with S.

A. Algorithm

M is constructed by a very simple algorithm. Initialize column 0 to all 0. Column $j \geq 1$ is obtained by taking the bitwise AND of $Shift\text{-}Or(j-1)$ with the U vector for character $T(j)$. If we let $M(j)$ denote the jth column of M, then $M(j) = Shift\text{-}Or(j-1)$ AND $U(T(j))$. Table II shows one iteration of the algorithm from column 5 to column 6.

Once column j has been obtained, it can be checked for any found matches with the aid of the F vector. Let Z be the bitwise AND of $M(j)$ with F. All locations where Z is 1 indicate found matches, and can be extracted efficiently using the following bit trick. Given a binary number $X > 0$, in order to find the lowest order bit of X which is turned on, simply perform the bitwise AND of X with $\sim (X - 1)$, where $\sim$ is the bitwise complement. This will give the number 2^a, where a is the index of the lowest order bit turned on. Finally, this value can be subtracted from X and this process repeated until $X = 0$, meaning all matching locations have been found.

Notice that at any given time the algorithm only ever needs the previous column of M in memory when computing the next, so this algorithm is efficient in terms of memory. The number of bit operations is $\Theta(mn)$. However,

 ISBN: 978-1-61208-111-3

when n is less than the size of a single computer word, each operation can be done very efficiently as single-word operations. Even when n is larger than a single computer word, each operation can be done as just a few single-word operations. Hence, for reasonably sized sets of patterns, this algorithm is efficient in both time and space regardless of the size of the text.

B. Proof of Correctness of Exact Multiple Pattern Matching Algorithm

We prove correctness of the algorithm described by induction on the columns of M. Column 0 of M is computed correctly, since it is set to all 0, and it corresponds to character ϵ which does not occur in any pattern in $\mathcal{P}$. Assume that all columns $j-1 < m$ are computed correctly. We will prove that column j is computed correctly.

Recall that $M(j) = Shift\text{-}Or(j-1)$ AND $U(T(j))$. By assumption, $M(j-1)$ is computed correctly. Thus $Shift\text{-}Or(j-1)$ is correct from column $j-1$. Namely, this vector will contain a 1 only in locations which match up until the $(j-1)-th$ character of T with some prefix of a pattern P_i *or* in locations which are the first character of some pattern. Note that because we only shift down by one, it is only possible for the last character of one pattern to interfere with the first character of another pattern. However, the first character of each pattern will always be set to 1 by the definition of $Shift\text{-}Or$, so no actual interference will take place.

Now this value is bitwise ANDed with $U(T(j))$, meaning only those locations which match with the current character will stay on. Thus, all previous matches that continue to match will stay 1, and any previous match that no longer matches will become 0, and any previous mismatch will stay 0. Therefore, $M(j)$ is correctly computed, and the algorithm correctly computes all columns of M.

III. Approximate Multiple Pattern Matching

We now address the l-edits problem of finding all approximate matches to a set $\mathcal{P}$ of patterns with at most l edit operations (replacements, insertions, or deletions). Recall that r is the starting index of the pattern containing character $P(i)$. Define $M^l(i,j)$ as a natural extension of $M(i,j)$, where $M^l(i,j)$ is 1 iff $P[r \ldots i]$ can be converted to some suffix of $T[1 \ldots j]$ with no more than l edit operations. The exact multiple pattern matching problem is a special case of this problem where $l = 0$.

The complete M matrices for $\mathcal{P} = \{abc, wxz, qrs\}$ with k values {2,2,2} and $T = \epsilon abdwxyzqt$ are shown in Table III.

Table III
COMPLETE M MATRICES FOR $\mathcal{P} = \{abc, wxz, qrs\}$ WITH k VALUES {2,2,2} AND $T = \epsilon abdwxyzqt$

M^0

	T	ϵ	a	b	d	w	x	y	z	q	t
P		0	1	2	3	4	5	6	7	8	9
a	1	0	1	0	0	0	0	0	0	0	0
b	2	0	0	1	0	0	0	0	0	0	0
c	3	0	0	0	0	0	0	0	0	0	0
w	4	0	0	0	0	1	0	0	0	0	0
x	5	0	0	0	0	0	1	0	0	0	0
z	6	0	0	0	0	0	0	0	0	0	0
q	7	0	0	0	0	0	0	0	0	1	0
r	8	0	0	0	0	0	0	0	0	0	0
s	9	0	0	0	0	0	0	0	0	0	0

M^1

	T	ϵ	a	b	d	w	x	y	z	q	t
P		0	1	2	3	4	5	6	7	8	9
a	1	1	1	1	1	1	1	1	1	1	1
b	2	0	1	1	1	0	0	0	0	0	0
c	3	0	0	1	1	0	0	0	0	0	0
w	4	1	1	1	1	1	1	1	1	1	1
x	5	0	0	0	0	1	1	1	0	0	0
z	6	0	0	0	0	0	1	1	1	0	0
q	7	1	1	1	1	1	1	1	1	1	1
r	8	0	0	0	0	0	0	0	0	1	1
s	9	0	0	0	0	0	0	0	0	0	0

M^2

	T	ϵ	a	b	d	w	x	y	z	q	t
P		0	1	2	3	4	5	6	7	8	9
a	1	1	1	1	1	1	1	1	1	1	1
b	2	1	1	1	1	1	1	1	1	1	1
c	3	0	1	1	1	1	0	0	0	0	0
w	4	1	1	1	1	1	1	1	1	1	1
x	5	1	1	1	1	1	1	1	1	1	1
z	6	0	0	0	0	1	1	1	1	1	0
q	7	1	1	1	1	1	1	1	1	1	1
r	8	1	1	1	1	1	1	1	1	1	1
s	9	0	0	0	0	0	0	0	0	1	1

Notice that $M^1(5,4)$ is 1, indicating that wx of the second pattern ($r = 4$, $i = 5$) matches a suffix of T ending at position 4 with at most 1 edit operation, in this case the suffix being w and the edit operation being to delete $P(5) = x$ from P. $M^1(6,7)$ is 1, indicating that an occurrence of wxz is found in the text ending at position 7. Here, the suffix of T is $wxyz$, and the edit operation is to insert $T(6) = y$ into the pattern between x and z. Anywhere a 1 exists at the end of a pattern in M^l indicates an occurrence of that pattern being found in the text with at most l edit operations.

Formally, in order to compute $M^l(j)$ (for $l \geq 1$), the following six recurrences may be used, an improvement to the seven recurrences given in [5]. Simply take the bit-wise OR of the following expressions, where $A \Downarrow x$ denotes shifting column A down by x bits and discarding any bits which shift past bit n:

1) $M^{l-1}(j)$
2) $Shift\text{-}Or(M^l(j-1))$ AND $U(T(j))$
3) $M^{l-1}(j-1) \Downarrow 1$
4) $M^{l-1}(j-1)$
5) $M^{l-1}(j) \Downarrow 1$
6) $(S \Downarrow l)$ AND $U(T(j))$

Essentially this says that $P[r \ldots i]$ will match a suffix of $T[1 \ldots j]$, with at most l edit operations, iff at least one of the following conditions hold:

1) $P[r \ldots i]$ matches a suffix of $T[1 \ldots j]$, with at most $l-1$ edit operations
2) $P[r \ldots i-1]$ matches a suffix of $T[1 \ldots j-1]$, with at most l edit operations, and $P[i] = T[j]$
3) $P[r \ldots i-1]$ matches a suffix of $T[1 \ldots j-1]$, with at most $l-1$ edit operations, and $P[i]$ is replaced by $T[j]$
4) $P[r \ldots i]$ matches a suffix of $T[1 \ldots j-1]$, with at most $l-1$ edit operations, and $T[j]$ is inserted into P after character $P[i]$
5) $P[r \ldots i-1]$ matches a suffix of $T[1 \ldots j]$, with at most $l-1$ edit operations, and $P[i]$ is deleted from P
6) $P[r \ldots l]$ may be deleted and $P[i] = T[j]$

There is only one exception to this recurrence, which occurs when computing M^1. For M^1, you must also OR this result with the binary vector S to allow the first character to be replaced or deleted.

As in the exact case, we must take care to show that this recurrence does not cause any interference between patterns. The second, third, and fifth expressions all cause the current column to be shifted down by 1. But for all $l \geq 1$, $M^l(j)$ is 1 for all on positions in the S vector. Since shifting down by 1 can only cause the last character of some pattern to shift into the first character of the next pattern, then any 1 shifted into the first position of a pattern would have been set to 1 by the S vector. The other possible shift occurs in the sixth expression, where S is shifted down by l bits. The only way for this shift to overlap with other patterns is when $l \geq |P_i|$ for some i. In this case, the shift by l bits overlaps into bit $b = l - |P_i| + 1$ of P_{i+1}. But since $|P_i| \geq 1$, then $b \leq l$, and so this bit will be on for P_{i+1} since these b characters of P may freely be deleted. In fact, it is possible for the shift by l to overlap with patterns past P_{i+1}, but the same argument holds for why this overlap does not cause interference for all subsequent patterns. Thus, bit shifting does not cause interference between the patterns.

A. Algorithm

After establishing the recurrence for approximate pattern matching, the algorithm for computing all approximate matches from a set $\mathcal{P}$ of patterns to a text T follows quite naturally. The only substantial change from the exact matching case is that instead of a single F vector, we now have a set of F vectors. Let $K = max\{k_1, k_2, \ldots, k_s\}$. Then compute $F_0, F_1, \ldots, F_K$, where $F_j(i)$ is 1 iff pattern p from $\mathcal{P}$ ends at i in P *and* $j \leq k_p$.

There are several ways to organize computation of the M matrix, but it makes the most sense to compute each column j for all M^l before computing any column $j+1$ for any M^l. Of course, each column will be computed in increasing order of l from 0 to K. Each column of M^0 is computed using the recurrence given in the exact case, and each column of M^l for $l \geq 1$ is computed using the recurrence given for the approximate case.

Once column j is computed for all M^l, all matches can be obtained with the aid of the F vectors. Let Z_l be the bitwise AND of $M^l(j)$ with F_l, and Z be the bitwise OR of $Z_0, Z_1, \ldots, Z_K$. All locations where Z is 1 indicate found approximate matches, and can be extracted efficiently using the same bit trick described for the exact matching case. In this way we allow each pattern to have a unique number of edit operations allowed, which is a new contribution by our algorithm.

B. Proof of Correctness of Approximate Multiple Pattern Matching Algorithm

The proof follows by induction on M^l. M^0 is the exact case and so is correct from before. M^1 only allows one character to be edited. The first character can be a replacement by the special case for M^1. Otherwise, the first expression lets M^1 stay a match if M^0 was a match. The second expression allows 1-edits to continue to be 1-edits when $P[i] = T[j]$. The third expression allows character $P[i]$ to be replaced by character $T[j]$ after an exact match. The fourth expression allows a single character to be inserted into the pattern after an exact match. The fifth and sixth expressions allow a single character to be deleted from the pattern after an exact match or a single character to be deleted from the beginning of the pattern if $P[r+1] = T[j]$, respectively. This list handles all the ways a pattern can be a 1-edit from the text, and so M^1 is correctly computed.

Now assume that for some $l-1$, M^2 through M^{l-1} are computed correctly. We prove that M^l is computed correctly. When computing M^l, the possible events are that $(l-1)$-edits continue to be l-edits, l-edits match at the next pair of characters and may be kept as l-edits, or $P[i]$ is replaced, $T[j]$ is inserted after $P[i]$, or $P[i]$ is deleted. The first two cases are handled exactly by the first and second expressions of the recurrence, respectively. The next three cases are handled by the last four expressions, allowing for $(l-1)$-edits to continue to be l-edits by replacement, insertion of a character into a pattern, or deletion of a character (or set of initial characters) from a pattern, respectively. Since M^{l-1} is assumed to have been computed correctly, this correctly computes M^l.

Similar to the exact case, the number of bit operations is $\Theta(mnK)$, and the memory usage is $O(nK)$. However, when K is a small constant and n is the size of only a few computer words, this algorithm is very practical.

IV. Experimental Results

We have developed the algorithms described in this paper and written C code to implement the ideas presented. In this section we give some timing results, comparing our method with the current methods which are used in practice.

Our exact matching algorithm is competitive with fgrep (invoked using the grep -F command), the standard Unix utility for doing exact pattern matching on a set of patterns.

 ISBN: 978-1-61208-111-3

Table IV
TIMES COMPARING PATS TO FGREP

Text File	Text Size	Pattern File	Time	
English Text[1]	12MB	10 4–6 length words	PATS	.25s
			fgrep	.321s
English Text[2]	115MB	30 common English words	PATS	1.92s
			fgrep	2.83s
Random Text	100MB	30 common English words	PATS	2.1s
			fgrep	2.5s
Genome[3]	215MB	6 common motifs (short strings)	PATS	3.6s
			fgrep	3.8s

Table V
TIMES COMPARING APATS TO AGREP AND NRGREP

Text File	Text Size	Pattern File	k	Time	
English Text[1]	12MB	30 common English words	1	APATS	.29s
				agrep	1.4s
				nrgrep	1.4s
English Text[2]	115MB	100 common English words	2	APATS	2s
				agrep	153s
				nrgrep	177s
Random Text	100MB	100 common English words	2	APATS	1.15s
				agrep	78s
				nrgrep	104s
Genome[4]	1.3GB	12 common motifs (short strings)	1	APATS	82s
				agrep	144s
				nrgrep	297s

fgrep implements the Aho-Corasick algorithm, a linear time algorithm for searching for a set of patterns in a given text. Table IV shows our timing results comparing our exact matching algorithm, PATS, to fgrep. Our results show a modest improvement over fgrep in certain searching scenarios, especially when n is small and many patterns can be expressed as just a few computer words. Of course, when the number of patterns gets too large (and hence n starts to grow to more than just a few computer words), fgrep becomes faster and would be the preferred method.

Where our algorithm shows its true usefulness is when doing approximate pattern matching against a set of patterns. The Unix utility agrep written by Manber and Wu [6] is known to be one of the fastest for doing approximate pattern matching. Another popular utility for fast approximate pattern matching is nrgrep, written by Navarro[0] [13]. Our method does not always match the times of these methods when searching against a single pattern, but when looking for approximate matches against a set of patterns we can beat these current methods, sometimes by very large margins in appropriate search settings. This is largely due to the fact that existing tools for multiple pattern matching must be re-run for each approximate match, where our algorithm is able to do multiple pattern approximate matching in a single pass. Table V shows our timing results comparing our approximate matching algorithm, APATS, to agrep and nrgrep.

We also ran our algorithm against 2 sets of English texts, one small text[1] and one large text[2], varying k over all values from 0 to 8. The results are shown in Fig. 1 and Fig. 2.

As seen, when $k = 0$, agrep and nrgrep both exceed the performance of APATS. This is due to these algorithms being tailored to perform special exact pattern matching algorithms for the $k = 0$ case. However, for $k \geq 1$, APATS shows excellent performance, doing far better than both agrep and nrgrep for all values tested. Testing was limited to 8 due to limitations of the agrep software.

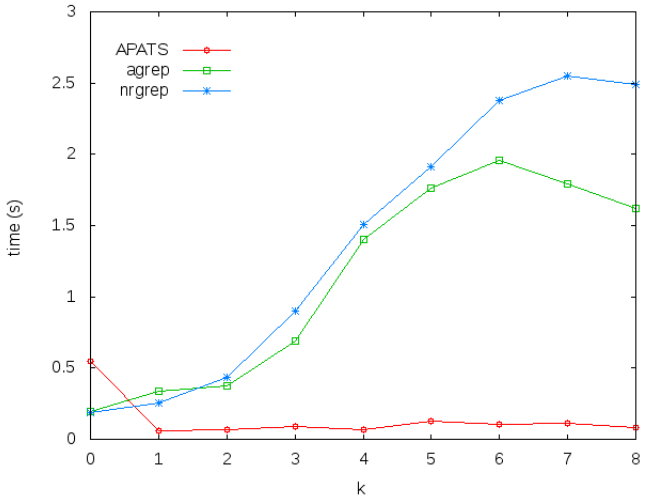

Figure 1. 12MB text searching for 30 common English words

Our implementation can be downloaded at http://www.cs.ucf.edu/∼stephen/pats-apats.

V. CONCLUSIONS

We have developed a very fast utility for exact and approximate pattern matching on a set of patterns. It is our hope that this algorithm would be added to the grep family of pattern matching algorithms and used in cases where it is expected to perform better than the current implementations, especially in cases where approximate pattern matching is

[0]We thank Dr. Navarro deeply for his personal communication and providing source code for our testing

[1]War and Peace from Project Gutenberg

[2]Selected works by Jane Austen, William Blake, Thornton W. Burgess, Sarah Cone Bryant, Lewis Carroll, G. K. Chesterton, Maria Edgeworth, King James Bible, Herman Melville, John Milton, William Shakespeare, and Walt Whitman from Project Gutenberg

[3]Chromosome 1 of Celera Genome from NCBI

[4]Chromosomes 1–6 of Human Genome from NCBI

 ISBN: 978-1-61208-111-3

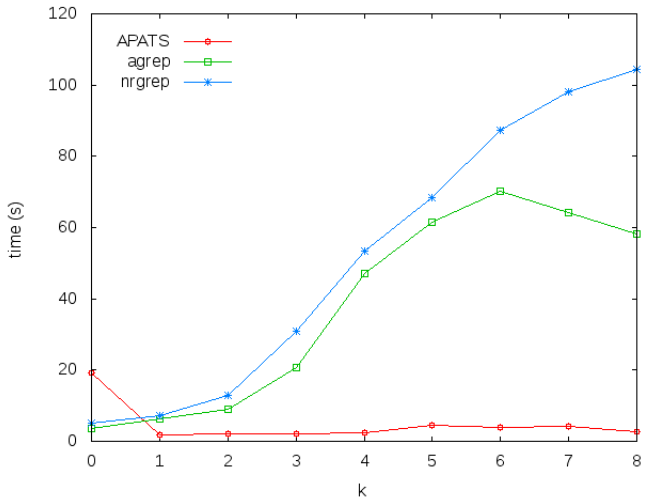

Figure 2. 115MB text searching for 20 English words of length 9–12

desired against a reasonably sized set of patterns. Another advantage of our algorithm is that it is very simple, both in concept and implementation. The run-times presented in this paper could no doubt be improved with a focus on optimizing implementation details. We have only implemented the algorithms exactly as they are presented.

REFERENCES

[1] D. E. Knuth, Jr, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[2] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[3] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[4] R. A. Baeza-Yates and G. H. Gonnet, "A new approach to text searching," *SIGIR Forum*, vol. 23, no. SI, pp. 168–175, 1989.

[5] S. Wu and U. Manber, "Fast text searching: allowing errors," *Commun. ACM*, vol. 35, no. 10, pp. 83–91, 1992.

[6] ——, "Agrep - a fast approximate pattern-matching tool," in *In Proc. of USENIX Technical Conference*, 1992, pp. 153–162.

[7] M. Kulekci, "Tara: An algorithm for fast searching of multiple patterns on text files," *Computer and information sciences, 2007. iscis 2007. 22nd international symposium on computer and information sciences*, pp. 1–6, 2007.

[8] R. Baeza-Yates and G. Navarro, "New and faster filters for multiple approximate string matching," *Random Struct. Algorithms*, vol. 20, no. 1, pp. 23–49, 2002.

[9] K. Fredriksson and G. Navarro, "Average-optimal single and multiple approximate string matching," *ACM Journal of Experimental Algorithmics*, vol. 9, 2004.

[10] H. Hyyrö, K. Fredriksson, and G. Navarro, "Increased bit-parallelism for approximate and multiple string matching," *ACM Journal of Experimental Algorithmics*, vol. 10, 2005.

[11] D. Gusfield, *Algorithms on strings, trees, and sequences: computer science and computational biology*. New York, NY, USA: Cambridge University Press, 1997.

[12] D. Adjeroh, T. Bell, and A. Mukherjee, *The Burrows-Wheeler Transform:: Data Compression, Suffix Arrays, and Pattern Matching*, 1st ed. Springer, July 2008.

[13] G. Navarro, "Nr-grep: A fast and flexible pattern matching tool," *Software Practice and Experience (SPE*, vol. 31, p. 2001, 2000.

 ISBN: 978-1-61208-111-3