Inexact Pattern Matching Algorithms via Automata[1]

Chung W. Ng

BioChem 218

March 19, 2007

1. Introduction

Pattern matching occurs in various applications, ranging from simple text searching in word processors to identification of common motifs in DNA sequences in computational biology. The problem of exact pattern matching has been well studied and a number of efficient algorithms exist. However these exact pattern matching algorithms are of little help when they are applied to finding patterns in DNA sequences. The DNA sequence search is inheritably inexact in nature because there are acceptable equivalences of amino acids that made up of the sequence. Current inexact pattern matching algorithms are based on four approaches: (1) Dynamic Programming; (2) Automata; (3) Bit-Parallelism; (4) Filtering. This paper serves as an overview of the common existing inexact pattern matching algorithms, with the focus on automata approach.

2. Exact Pattern Matching

Exact pattern matching is the basis for all text searching applications and this section review some of the most common algorithm for exact match. The problem can be stated as: Given a pattern P of length m and a string (or text) T of length n ($m \leq n$), find all the occurrences of P in T. The match is an exact one, meaning that the exact word or pattern is found. In Unix environment, there is a useful command utility command "grep" which allows the user to search globally for lines matching the regular expression, and print them. Common exact matching algorithms are:

2.1. Naïve Brute Force Algorithm

This is the simplest method with the worst performance of O(mn). The first character of pattern P is compared with the first character of the string T. If it is a match, then pattern P and string T are matched character by character until a mismatch or the end of the pattern P is detected. If a mismatch is found, the pattern P is shifted one character to the right and the same matching process repeats. The algorithm is inefficient because, after a mismatch is detected, the shift-increment is only one character at a time.

[1] Participation in this course was partially supported by a Dept. of Education Minority Science and Engineering Improvement Program (MSEIP) grant to Morehouse College (Grant Number: P120A060097). It was also partially supported by cost-sharing by Morehouse College using funds from a Title III grant from the Department of Education.

2.2. Boyer-Moore Algorithm

The Boyer-Moore algorithm [BM1977] applies larger shift-increment for each mismatch detection A main modification to the naïve algorithm is that the matching of pattern P and string T is done from right to left, i.e. after aligning P and T, the last character of P will be matched to T first. If a mismatch is detected, and if the mismatch character, say c, in T is not in P, then P is shifted right to m positions and P is aligned to the next character after c. If c is part of P, then P is shifted right so that c is aligned with the rightmost occurrence of c in P. The worst complexity is still O(m+n). In practice the algorithm performs on average O(n/m), since the shifting-increment on average is the half of the length of P, (m/2).

2.3.Knuth-Morris_Pratt Algorithm

The Knuth-Morris-Pratt algorithm [KMP1977] is based on finite state automaton. The pattern P is pre-processed to create a finite state automaton M that accepts the pattern. The finite state automaton M is usually represented as a transition table. For example, if the pattern P is "genome", the following non-deterministic finite automata (NFA) will be created to accept the pattern:

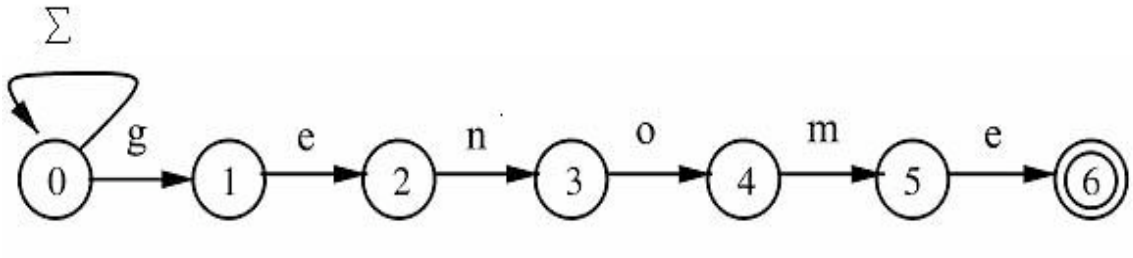

Figure 1. NFA that search "genome" exactly

In the classical implementation of algorithm, the NFA is converted to a deterministic finite automaton (DFA). The following is the equivalent DFA of the above NFA:

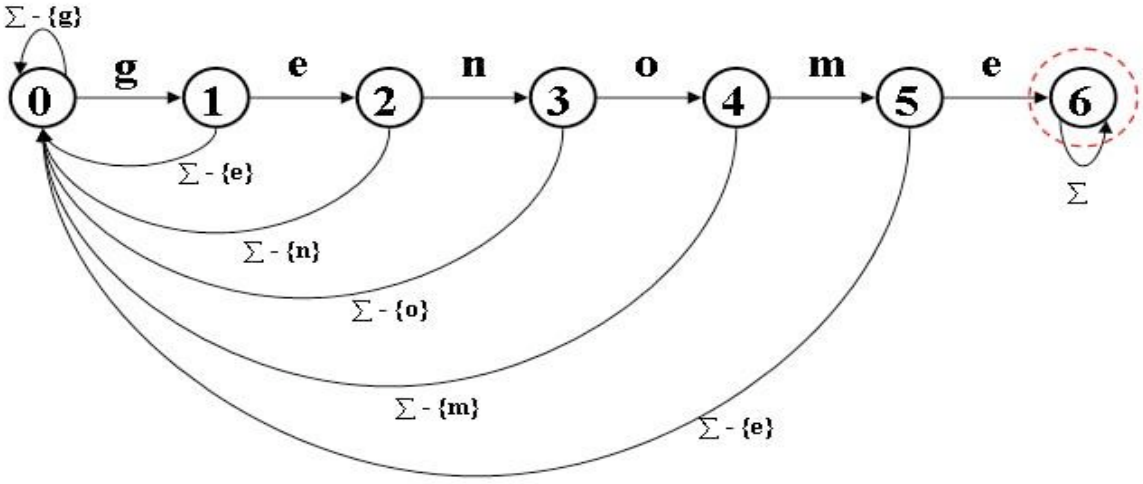

Figure 2. DFA that search "genome" exactly

The number of states of machine M required is (m+1). For each state, the number of transitions is $|\Sigma|$, where Σ is the alphabet set of characters for T. Both the average and worst case performance is O(m+n). The size of the transition table required is O(m.$|\Sigma|$).). This approach can easily be extended to handle search with a larger set of patterns or even regular expressions, including inexact pattern matching.

3. Inexact Pattern Matching

The need to align inexact sequence data arises in various fields and applications such as computational biology, signal processing and text processing. In particular, in DNA sequence analysis, exact sequence matching is rare. Due to possible DNA mutation, the biological inference does not expect an identical match, but rather a high sequence similarity usually implies significant functional or structural similarity.

Inexact pattern matching is sometimes referred as "approximate pattern matching" or "matching with k mismatches/differences". This problem, in the general form, can be stated as: Given a pattern P of length m and a string (or text) T of length n ($m \leq n$), find all the occurrences of substrings X in T that are "similar" to P, allowing a limited number, say k, of "errors" in the "similarity" matches. The "errors" are the total cost of transforming the pattern P so that P and X are equal. The common allowable edit/transformation operations are insertion, deletion and substitution. The common error model is called "edit distance". The edit distance is the minimal number of edit operations required to transform the first sequence into the second.

Inexact pattern matching algorithms can be classified into four main categories:

3.1. Dynamic Programming Approach

This is the oldest among the four approaches and the most commonly used approach, especially in the area of biological sequence analysis. Examples are the Needleman–Wunsch algorithm and Smith-Waterman algorithm. These algorithms are much more complex than the ones for exact pattern matching. It involved solving successive recurrence relations recursively. i.e. smaller problems are solved in succession to solve the main problem. The classical dynamic programming algorithm can also be thought of as a column-wise "parallelization" of the automaton.

The major advantage of dynamic programming approach is its flexibility in adapting to different edit distance functions. In general, the worst case complexity is O(mn). Over the past two decades, a number of improved solutions have been proposed to lower the worst case complexity to O(kn) and average complexity of O(kn/$\sqrt{|\Sigma|}$).

3.2. Automata Approach

This approach is also rather old. Though automata approach doesn't offer time advantage over Boyle-Moore algorithm for exact pattern matching, this approach does offer better running time for inexact pattern matching. Both the average and worst case performance remain O(m+n).

3.2.1 General Approach

This automaton approach is first proposed by Ukkonen in 1985 [Ukk1985]. Consider a non-deterministic automaton (NFA) for k=2 errors, in Figure.3:

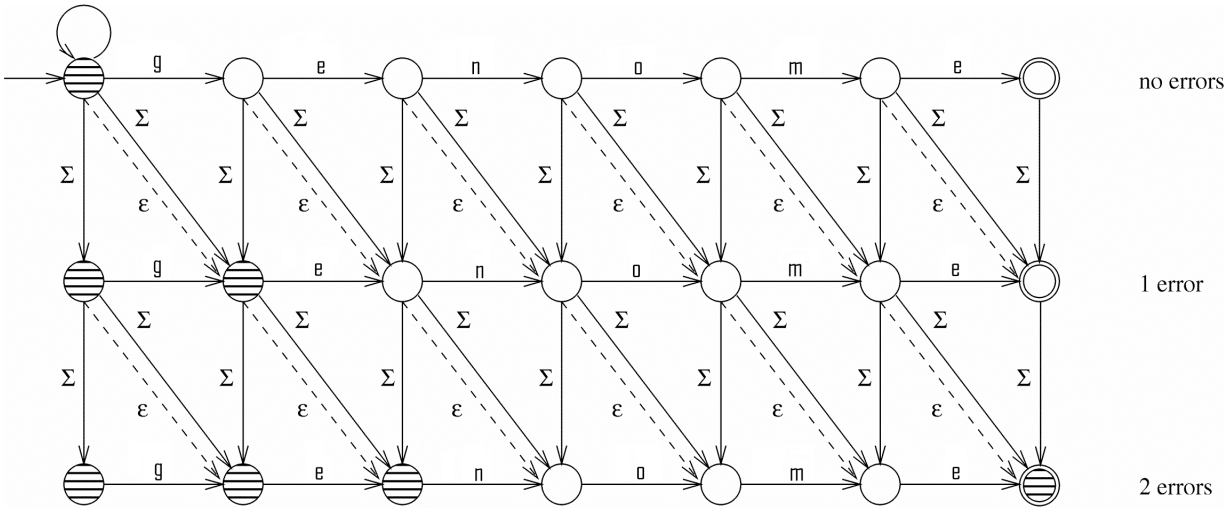

Figure 3. NFA for approximate string matching of the pattern "genome" with two errors. The shaded states are those active after reading the text "genames".

Every row denotes the number of errors seen (the first row zero, the second row one, etc.). Every column represents matching a pattern prefix. Horizontal arrows represent matching a character. All the others increment the number of errors and move to the next row. Vertical arrows represent insertion of a character (or known as a gap) into the pattern (string T is advanced without advancing pattern P). Solid diagonal arrows represent a single character substitution (both string T and pattern P are advanced). Dashed diagonal arrows ("ε-transitions) represent deletion of a character from the pattern (pattern P is advanced without advancing in the string T). Alternatively, we can also consider the last case as insertion of a gap to the string T.

The total number of states for this NFA is (k+1)(m+1). The initial self-loop allows a match to start anywhere in the string. The automaton signals a match whenever one of the rightmost states in the last column is active. It is easy to see that once a state in the automaton is active; all the states of the same column and higher numbered rows are active too.

Deterministic automata exhibit an O(n) worst-case search time. However, the main problem to this approach is the construction of the DFA from NFA which takes exponential time and space. An alternative solution is based on simulating the NFA instead of making it deterministic as described in some of the algorithms below.

3.2.2 Algorithms Based on Automata

3.2.2.1 [Ukk1985]

In 1985, Ukkonen first proposed the idea using a DFA for solving the inexact matching problem. A state of the DFA corresponds to the possible set of values for the columns of the dynamic programming matrix.

A big problem with this scheme was that the DFA had a huge number of states, which had to be built and stored. To improve space usage, Ukkonen proved that all the elements in the columns that were larger than k could be replaced by k+1 without affecting the output of the search. This reduced the potential number of different columns. He also showed that adjacent cells in a column differed in at most one. By applying these modifications, he obtained a nontrivial bound, $O(\min(3^m, m(2m|\Sigma|)^k))$, on the number of states of the automaton. This size, although much better than the obvious O((k+1)m), is still very large except for short patterns or very low error levels.

3.2.2.2. [WMM1996]

Their idea was to trade time for space using a Four Russians technique. The columns were partitioned into blocks of r cells (called "regions") which took 2r bits each. Instead of pre-computing the transitions from a whole column to the next, the transitions from a region to the next region in the column were pre-computed, although the current region could now depend on three previous regions. Since the regions were smaller than the columns, much less space was necessary. The total amount of work was O(m/r) per column in the worst case and O (k/r) on average. The space requirement was exponential in r. By using O(n) extra space, the algorithm was O(kn / log n) on average and O(mn / log n) in the worst case. This work was later extended to handle regular expressions allowing errors. The technique for exact regular expression searching is to pack portions of the deterministic automaton in bits and compute transition tables for each portion. The few transitions among portions are left nondeterministic and simulated one by one. To allow errors, each state is no longer active or inactive, but they keep count of the minimum number of errors that makes it active, in O(log k) bits.

3.2.2.3 [Mel1996]

Melichar further studied the size problem of the deterministic automaton. By considering the properties of the NFA of Figure 3, he improved the bound of [Ukk1985] to $O(\min(3^m, m(2mt)^k, (k+2)^{m-k}(k+1)!))$, where $t = \min(m+1, |\Sigma|)$. The space complexity and preprocessing time of the automaton is t times the number of states.

3.2.2.4 [Kur1996]

In 1996, Kurtz proposed another way to reduce the space requirements to at most O(mn). The idea was to build the automaton in lazy form, i.e. build only the states and transitions actually reached in the processing of the text. The automaton starts as just one initial state and the states and transitions are built as needed. Those

transitions that were not necessary were not built. Kurtz also proposed building only the initial part of the automaton, which should be the most commonly traversed states, to save space. However, the growth of the lazy automata is a function of m, k and n. Empirical results showed that the lazy automaton grows with the text at a rate of $O(n^{\beta})$, for $0 < \beta < 1$, depending on $|\Sigma|$, m, and k.

3.2.2.5 [Hol1996]

Holub showed how to reduce the number of states of nondeterministic finite automata to (k+1)(m+1-k) for approximate string matching when it is not necessary to know how many mismatches are in the found string. This algorithm is based on Shift-Or algorithm and thus it reduces the length of the state vectors that needed to be computed by the Shift-Or algorithm.

3.3. Bit-Parallelism

This approach is rather new (after 1990) and is based on exploiting the intrinsic parallelism of the bit operations inside a computer word. The basic idea is to "parallelize" another algorithm, possibly those algorithms from the previous two approaches in sections 3.1 and 3.2, using bits. In general, the number of operations that an algorithm performs can be cut down by a factor of at most w, where w is the number of bits in a computer word. Since in current computer architectures, w is 32 or 64, the speedup is very significant in practice. The results are especially significant when short patterns are involved. They may work effectively for any error level.

The first bit-parallel algorithm is known as "Shift-Or" which searches a pattern in a text (without errors) by parallelizing the operation of a nondeterministic finite automaton that looks for the pattern. Figure 1 illustrates this automaton. This automaton has m+1 states, and can be simulated in its nondeterministic form in O(mn) time. For patterns longer than the computer word (i.e. m>w), the algorithm uses (m/w) computer words for the simulation. The algorithm is O(n) on average.

Bit-parallelism has become a general way to simulate simple nondeterministic automata instead of converting them to deterministic form. It has the advantage of being much simpler, in many cases faster, and easier to extend in handling complex patterns than its classical counterparts. Its main disadvantage is the limitation it is imposed by the size of the computer word. In many cases its adaptations for longer pattern search are not very efficient.

There are two main trends in bit-parallelism approach: (1) parallelize the work of the dynamic programming matrix; or (2) parallelize the work of the nondeterministic automaton.

3.3.1 Parallelizing the Dynamic Programming Matrix

3.3.1.1 [Wri1994]

In 1994, Wright considered secondary diagonals (i.e. those that run from the upper-right to the bottom-left) of the dynamic programming matrix. The main observation is that the elements of the new secondary diagonal can be computed using the two previous ones. The algorithm packs many patterns and text characters in a computer word and performs in parallel a number of patterns versus text comparisons, then uses the vector of the results of the comparisons to update many cells of the diagonal in parallel. Since it has to store characters of the alphabet in the bits, the algorithm is

O(nmlog(|Σ|)/w) in the worst and average case. This was efficient for small alphabets (e.g. DNA). However it is difficult to adapt this algorithm for other distance functions.

3.3.1.2 [Mye1999]

Myers proposed a better way to parallelize the computation of the dynamic programming matrix. He represented the differences along columns instead of the columns themselves, so that two bits per cell were enough. The idea is to keep packed binary vectors representing the current values of the differences, and finding the way to update the vectors in a single operation. This results in a better uses of the bits of the computer word, with a worst case of O((m/w).n) and an average case of O((k/w).n). The algorithm adapts better to longer patterns and search on extended patterns.

3.3.2 Parallelizing Nondeterministic Automata

3.3.2.1 [WM1992]

The idea is to simulate, using bit-parallelism, the NFA of Figure 3, so that each row i of the automaton fits in a computer word (each state is represented by a bit). For each new text character, all the transitions of the automaton are simulated using bit operations among the k+1 computer words. All k+1 computer words have the same structure (i.e. the same bit is aligned on the same text position).

The cost of this simulation is O(k.(m/w).n) in the worst and average case, which is O(kn) for patterns typical in text searching (i.e. $m \leq w$). For short patterns, this is competitive to the best worst-case algorithms. This algorithm is also able to perform approximate string matching with sets of characters, wild cards, and regular expressions. Agrep (Approximate grep) was developed with all these search options.

3.3.2.2 [BYN1999]

In 1999, Baeza-Yates and Navarro proposed a bit-parallel formula for a diagonal parallelization of the computation of the automaton. They packed the states of the automaton along diagonals instead of rows or columns, which run in the same direction of the diagonal arrows (notice that this is totally different from the diagonals of the dynamic programming matrix).

The resulting algorithm is O(n) worst case time and very fast in practice if all the bits of the automaton fit in the computer word. In general, it is O((k(m-k)/w).n) worst case time, and O((k2/w).n) on average. The algorithm can handle classes of characters, wild cards and different integral costs in the edit operations.

3.4. Filtering Algorithms

This approach started in 1990 and has been most very active since. Most of the new algorithms proposed in recent years belong to this class [Nav2001]. Filtering is based on the fact that it may be much easier to tell that a text position does not match than to tell

that it matches. It is formed by algorithms that filter the text, quickly discarding text areas that do not match. Since the exact searching algorithms is much faster than approximate searching ones, most filtering algorithms take advantage of this fact by searching pieces of the pattern without errors.

Filtering algorithm, by itself, is normally unable to discover the matching text positions. Rather, it is used to discard large areas of the text that cannot contain a match. Filtering algorithms must couple with a process that verifies all those potential text matching positions. Any non-filtering algorithm can be used for this verification. The selection is normally independent, but the verification algorithm must behave well on short texts because it can be started at many different text positions to work on small text areas.

The major interest in this approach is the potential for algorithms that do not inspect all text characters. These filtering algorithms have a theoretical average running time O(n(k + log m)/m), which was proven optimal. In practice, filtering algorithms are among the fastest too.

The main drawback of this approach is that the performance of filtering algorithms is very sensitive to the error level α. Most filters work very well on low error levels and very badly otherwise. This is related to the amount of text that the filter is able to discard. When evaluating filtering algorithms, it is important not only to consider their time efficiency but also their tolerance for errors.

4. Empirical Results

Navarro [Nav2001] implemented most of the algorithms discussed in this paper and set up some experiments. He ran the algorithms using text and data from DNA application. It is noted that for short pattern ($m \leq 10$) and varying k, the best performance algorithm is [BYN1999], then [Mye1999], [Kur1996] and [WM1992]. In the case of longer pattern (m=30), most of the observations are still valid, except [Mye1999] performs better than [BYN1999]. It is also noted that [Mye1999] is more stable over different values of m because the entire problem fits into a computer word. Another observation from the experiments is that filtering algorithms perform better than non-filtering algorithms only when the error level α is low. Otherwise, it is better to use a non-filtering algorithm directly without filter.

5. Conclusion

Exact searching algorithms are simpler and much faster than the approximate searching ones. While most of exact pattern matching algorithms do not adapt well for inexact matching, automata approach adapts very well for inexact pattern matching. This makes automata approach a potentially good candidate for solving inexact pattern matching problem. For a given pattern, it is easy to create a NFA for it, especially with the ε-transitions. However, these ε-transitions also cause the "zero-time" dependencies, i.e. the current values of two rows or columns depend on each other, and therefore cannot easily be computed in parallel. Direct conversion of NFA to DFA is exponentially costly in

terms of time and space requirements. Bit-Parallelism and Filtering provide potential means of speedup to make automata approach feasible. In spite of these difficulties, automata approach remains a potentially attractive in biological sequence analysis. Since some motifs are expressed in regular expressions, and every regular expression can be converted to a NFA with ε-transitions, it is interesting to investigate this approach as a way to detect the common motifs. A recent algorithm that is based on union and intersection of automata to find the common motifs with gaps was proposed in [AHI2006].

Appendix: Comparison Table of Algorithms in this Paper

Algorithms	Worst Time/Space Complexity	Advantages	Limitations
DP algorithms	O(mn), worst O(kn) on average	Flexibility – algorithm is easy to adapt to other distance functions	Time requirement is high
Automata algorithms	O(m+n) if automata is deterministic Space O((k+1)m)	Fast running time, close to O(n)	Space requirement could increase exponentially. Direct conversion of NFA to DFA take exponential time
[Ukk1985]	Space $O(\min(3^m, m(2m\|\Sigma\|)^k))$	Reduce the number of states and compute as DP matrix	Number of states is huge except for short pattern or small error level k.
[WMM1996]	O(mn / log n)	Reduce space requirement. Handle regular expressions with errors	Trade time for space requirement.
[Mel1996]	Space $O(\min(3^m, m(2mt)^k, (k+2)^{m-k}(k+1)!))$, where $t = \min(m+1, \|\Sigma\|)$.	Reduce the space requirement even more	Processing time is t times the number of states
[Kur1996]	Space O(mn)	States and transitions are built as needed Reduce space to at most O(mn)	Growth of lazy automata is a function of m, k and n
[Hol1996]	Number of NFA states = (k+1)(m+1-k)	Reduce the number of states of NFA	The number of mismatches is not required.
Bit-Parallelism algorithms	O(n) on average	Significant speed up by a factor or 32 or 64 Effective at any error level k	Size of computer word. Adaptation to longer pattern is not efficient
[Wri1994] DP	O(nmlog(|Σ|)/w)	Especially good for very small alphabets (e.g. DNA)	Difficult to adapt to other distance functions.

[Mye1999] DP	O((m/w).n)	Adapts better to longer patterns. Allows extended pattern search	Difficult to adapt to other distance functions
[WM1992] NFA	O(k.(m/w).n)	Competitive to the best worst-case algorithms for short pattern Search with extended pattern, wild cards or regular expressions (Agrep)	Limited capability to search a set of patterns at the same time
[BNY1999] NFA	O((k(m-k)/w).n)	Can handle search with classes of characters, wild cards and different integral costs in the edit operations.	Unstable performance. Show non-monotonic behavior.
Filtering algorithms	O(n(k + log m)/m) on average	Fastest among all, close to the proven optimal	Performance is sensitive to the error level α.

References

[AHI2006] Antoniou P, Holub, J., Illiopoulos, C.S., etc. Finding Common Motifs with Gaps using Finite Automata. Proceeding of the 11th International Conference on Implementation and Application of Automata *(CIAA'06)*, Taiwan, 2006, 69-77

[BM1977] Boyer R., and Moore J. A Fast String Searching Algorithm, Comm. ACM 20, 1977, 762-72.

[BYN1999] Baeza-Yate, R. and Navarro, G. Faster approximate string matching. Algorithmica 23, 2, 1999, 127–158

[Hol1996] Holub, J. Reduced Nondeterministic Finite Automata for Approximate String Matching. Proceedings of the Prague Stringologic Club Workshop, 1996, 19-27

[KMP1977] Knuth D., Morris J. and Pratt V., Fast Pattern Matching in Strings, SIAM Journal on Computing 6(1), 1977, 323-350.

[Kur1996] Kurtz, S. Approximate string searching under weighted edit distance. In Proceedings of the 3rd South American Workshop on String Processing (WSP '96). Carleton Univ. Press, 1996, 156–170.

[Mel1996] Melichar, B., String matching with k differences by finite automata. In Proceedings of the International Congress on Pattern Recognition (ICPR '96). IEEE CS Press, Silver Spring, MD, 1996. 256–260.

[Mye1999] Myers, G. A fast bit-vector algorithm for approximate string matching based on dynamic programming. J. ACM 46, 3, 1999, 395–415.

[Nav2001] Navarro, G, A Guided Tour to Approximate String Matching", ACM Computing Survey, 33(1), 2001, 31-88

[Ukk1985] Ukkonen, E., Finding approximate patterns in strings. J. Algor. 6, 1985, 132–137.

[WM1992] Wu, S. and Manber, U. Fast text searching allowing errors. Commun. ACM 35, 10, 1992, 83–91.

[WMM1996] Wu, S., Manber, U., and Myer, E., A subquadratic algorithm for approximate limited expression matching. Algorithmica 15, 1996, 1, 50–67.

[Wri1994] Wright, A. Approximate string matching using within-word parallelism. Software Practice Exper. 24, 4, 1994, 337–362.