Approximate String Matching using Backtracking over Suffix Arrays*

Mohammadreza Ghodsi†

Abstract

We describe a simple backtracking algorithm that finds approximate matches of a pattern in a large indexed text. This algorithm theoretically takes sublinear time in the length of the text. We prove a lemma that helps us to prune a significant number of branches of search in practice. We show an implementation of a variant of this algorithm and that is used to find similar regions between sequences of two bacterial genomes.

1 Introduction

The *approximate string matching* problem is to find an approximate occurrence of a relatively short string called pattern in a long string called text. The pattern P and text T are strings of characters from a finite alphabet Σ. We denote the length of T and P and the size of Σ by n, m and σ.

The approximate occurrence can be defined using a variety of distance metrics over strings. A few popular metrics are Hamming distance, unit-cost edit distance and general edit distance based on a substitution cost matrix. Edit distance seems harder to work with that Hamming distance as for example theoretical results show embedding the edit distance into L_1 requires a distortion of $\Omega(\log n)$ [3], and practical techniques that work well for Hamming distance such as spaced seeds [8] cannot be directly applied to edit distance.

An string matching problem is called *offline* if we are allowed to pre-process the text and make an *index* data structure, and *online* if we are not allowed to pre-process the text. Table 1 contains an overview of major results for different versions of the problem. The exact alignment problems can be solved optimally. There has been a lot of research into online approximate matching problem. For a nice overview of online algorithms see [11].

Two of the most famous index data structures used for large strings are inverted k-mer index and family of indexes related to suffix trees (including

*This paper is submitted to Computer Science Department of University of Maryland at College Park as a scholarly paper to partially satisfy requirements for Masters degree in Spring 2009 academic semester.

†Department of Computer Science, University of Maryland, College Park, MD 20742, USA. Email: ghodsi@cs.umd.edu

	Online	Offline
Exact	KMP $O(n+m)$	Suffix Tree $O(m)$
Approximate	Dynamic Programming $O(n{\cdot}m)$, Landau+Vishkin $O(k \cdot n)$ [6]	Myers [10], Navarro+Baeza-Yates [12] $O(n^\gamma \log n)$, Ukkonen [13] $O(m^{k+3}\sigma^k)$

Table 1: Overview of major results for different versions of string matching problem.

most importantly suffix arrays and Burrows-Wheeler index). Inverted indexes are very fast in practice but are less suited for approximate matching although they have been used for this purpose [10].

In this paper we will focus on offline k-difference problem: Find all substrings Q of T such that the unit-cost edit distance between P and Q is $\leq k$. In practice our algorithm can easily and efficiently be generalized to use a substitution cost matrix, although some of our theoretical running time bounds depend on the unit-cost model.

We will focus on the case that $\log_\sigma n \leq m \ll n$. If $m < \log_\sigma n$ for sufficiently random like sequences many occurrences will happen by mere chance. For very large m the pattern can be broken into smaller pieces and each piece can be searched for independently.

1.1 Application Biological Sequences

There are two primary types of biological sequences of interest: DNA sequences and amino acid sequences. DNA sequences are strings over 4 alphabet $\Sigma_{\mathrm{DNA}} = \{\mathrm{A}, \mathrm{T}, \mathrm{C}, \mathrm{G}\}$. Mutations will cause substitutions, insertions and deletions in random positions of the DNA. Unit-cost edit distance captures the evolutionary distance of two sequences quite well. On the other hand protein sequences are from an alphabet of 20 amino acids. Some of these amino acids are quite similar and it is possible that two highly similar proteins to have quite different amino acid sequence and totally different DNA coding sequence. For this reason in sequence alignment of amino acids, a substitution matrix is used that describes the rate at which one amino acid changes to another over time.

Approximate search in amino acid databases is arguably more difficult in practice. For instance seed and extend algorithms like BLAST will be forced to use a much shorter seed length and will therefore have lower sensitivity and lower specificity. Large alphabets also pose a problem for algorithms that generate all strings similar to query and use and exact searching algorithm to find them (e.g. that of Myers [10]) since there could be lots of different sequences 'similar' to query.

Another important property of non-coding DNA sequences is that they resemble random sequence in the sense that for any given very short string of characters the number of times that it appears in DNA is close to the number

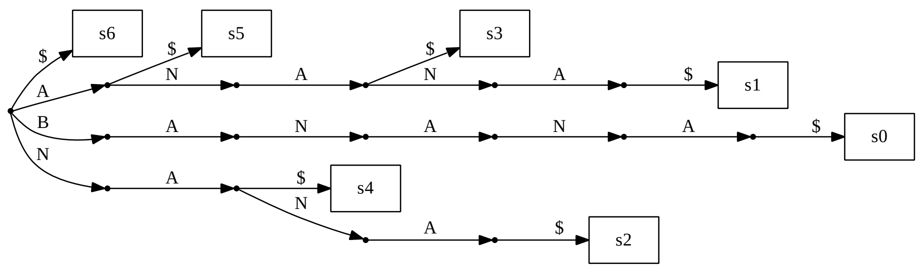


Figure 1: Sample Suffix Trie for string 'BANANA'

(a) String in memory

0	B
1	A
2	N
3	A
4	N
5	A
6	$

(b) Suffix array in memory

0	6	$
1	5	A$
2	3	ANA$
3	1	ANANA$
4	0	BANANA$
5	4	NA$
6	2	NANA$

Figure 2: Sample Suffix Array for string 'BANANA'

expected from a random sequence. This is in contrast to natural language text which is unlikely to contain certain sequence of letters.

To our knowledge two computational biology software use similar techniques: Bowtie [7] uses a Burrows-Wheeler index of the human (and other) genome and can align short reads with few errors to the reference using backtracking. Vmatch [5] is a closed source software that uses suffix arrays index (among others) and is able to perform approximate alignments.

2 Algorithm

2.1 DFS in Suffix Trie

It is easier to explain the algorithm using *Suffix Trie* data structure[1]. Consider a suffix trie of text as illustrated in Figure 1. An exact search algorithm will start at the root and follow corresponding character from pattern at each node. In case of approximate search however we may have to follow links labeled with characters that are different from the one that appear in the text.

A simple DFS traversal of this tree spells out all the suffixes of text in sorted order one character at a time. The general backtracking policy is to stop following down branches from an internal node as soon as the edit distance between the string corresponding to that node and any prefix of query keyword

[1]This representation may require $\omega(n)$ space.

is more than an specific threshold. The edit distance can be computed by maintaining the column of the dynamic programming matrix corresponding to the cost of aligning any prefix of query to the substring represented by each internal node.

The main algorithm is as follows: For every substring of P of length l (called a keyword W_p), search for an approximate occurrence of W_p in suffix array of T using DFS with backtracking. The simplest backtracking strategy is to stop as soon as the edit distance of current node's string and any prefix of query is greater than k. However for most practical texts the top levels of the suffix trie are nearly full (each node has nearly σ children), whereas deep nodes of the trie have about one child on average. The simple backtracking strategy will spend lots of time traversing top levels of the trie allowing many errors in the first few characters of the alignment. The main intuition is that we can look for words that do not have many differences on the left.

The main Lemma 1 shows that for any occurrence of P in T there exists a keyword that matches T in such a way that from left to right the number of editions necessary to match W_p to T is never more than $\frac{k}{m-l}$ times the length of the prefix.

In practice we use $\log_\sigma n \leq l \leq c \log_\sigma n$, Since for the shorter keywords we can expect better matches with fewer errors and still avoid random matches in expectation.

After finding the keyword hits one should verify that they can be extended to hits of length m. We will ignore this post-processing here and assume for sufficiently large keyword length $l \geq \log_\sigma n$ the number of false positives are small enough that the running time will be dominated by the keyword search phase.

Lemma 1. *If P matches Q (e.g. a substring of T) with $\leq k$ differences, then for any l, $1 \leq l < m$ there exists W_P a substing of P of length l and W_Q a substring of Q such that for every prefix of W_P of length j, $W_P[1..j]$ matches some prefix W_Q with $\leq \left(j \cdot \frac{k}{m-l}\right)$ differences.*

Proof. let $f(i)$ be a the minimum edit distance of $P[1..i]$ from any prefix of Q. See Figure 3. It is easy to see that f is a non-decreasing function, and by definition $f(m) \leq k$ and $f(0) = 0$.

As a way of contradiction assume for all $i = 0 \dots (m-l)$, there exists $j \in \{1 \dots l\}$ such that $f(i+j) - f(i) > \left(j \cdot \frac{k}{m-l}\right)$.

In particular there exists j_1 such that $f(0+j_1) > f(0) + \left(j_1 \cdot \frac{k}{m-l}\right)$, and there exists j_2 such that $f(0+j_1+j_2) > f(0) + \left(j_1 \cdot \frac{k}{m-l}\right) + \left(j_2 \cdot \frac{k}{m-l}\right)$, and

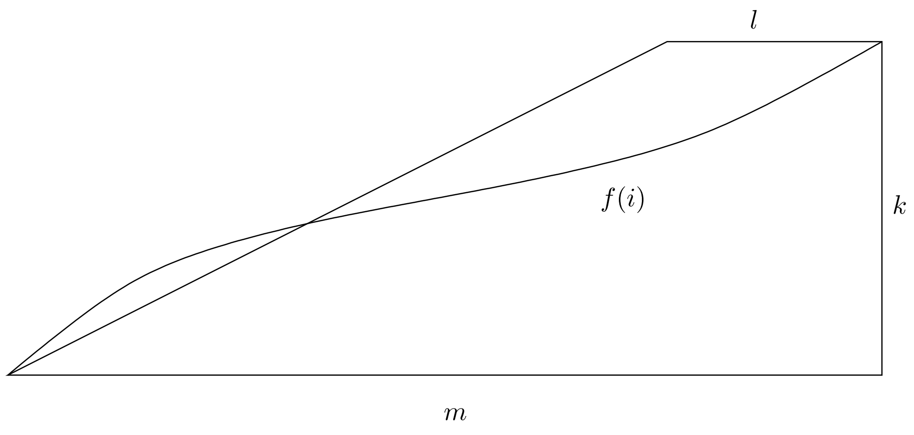


Figure 3: For any occurrence of P with $\leq k$ differences in T, define $f(i)$ as the minimum edit distance of $P[1..i]$ from any prefix of the occurrence.

so on up to some $m - l < 0 + j_1 + j_2 + \ldots + j_z \leq m$.

$$\begin{aligned} f(m) &\geq f(0 + j_1 + j_2 + \ldots + j_z) \\ &> f(0) + \left(j_1 \cdot \frac{k}{m-l}\right) + \left(j_2 \cdot \frac{k}{m-l}\right) + \ldots + \left(j_z \cdot \frac{k}{m-l}\right) \\ &= 0 + (j_1 + j_2 + \ldots + j_z) \cdot \left(\frac{k}{m-l}\right) \\ &\geq (m-l) \cdot \left(\frac{k}{m-l}\right) \\ &= k \end{aligned}$$

Which implies $f(m) > k$ which is a contradiction. □

2.2 Adapting the Algorithm to Suffix Arrays

A suffix array [9] is the list of indexes all suffixes of a string in lexicographically sorted order. A suffix array can be built in linear time [2] and occupies $n \log_2 n$ bits. Even though theoretically asymptotical memory requirements of suffix arrays and suffix trees are similar, in practice highly optimized implementations of suffix tree require over 10 bytes of memory per input character [4] whereas basic suffix array requires 4+1 [3] bytes. Space requirements of suffix arrays can be further reduced to $O(n)$ bits using compressed suffix arrays. Finally it has been shown that by storing some auxiliary tables, Enhanced Suffix Arrays [1]

[2] Assuming $\log n$ is smaller than the word size of the machine and operations on them takes constant time

[3] Assuming length of input string can be stored in a 32 bit integer.

can be used to simulate any type of traversal of suffix trees in the same time complexity.

Our algorithm over suffix arrays is inspired by the simplest exact search algorithm on a suffix array which is in essence a binary search. The backtracking algorithm described over suffix tries can be adapted to work on suffix arrays with a $\log_2 n$ factor increase in running time as will be shown in Lemma 2. The main algorithm is as follows: We maintain a window of the suffix array and the depth that we have matched so far, If the character at this depth from both sides of current window match, there is only one edge to follow down in this window, otherwise we simply divide the window in two equal windows and recursively find the matches on both halves.

Lemma 2. *The number of calls to the recursive suffix array search function is at most* $\log_2 n$ *times the number of suffix trie nodes visited by the backtracking DFS algorithm.*

Proof. let p be the number of nodes visited by the suffix trie backtracking DFS algorithm. The key point is to note that each internal node of the suffix trie corresponds to a window $[i_l \ldots i_r]$ on the suffix array. We need to bound the number of times we need to cut the suffix array of length n in half that is needed to realize any partitioning of the suffix array in p windows, denoted by $g\,(n, p)$.

$g(n, p)$ is given by the following recurrence:

$$g(n,p) = \begin{cases} 0, & n = 1 \text{ or } p = 0 \\ g\left(\frac{n}{2}, p_l\right) + g\left(\frac{n}{2}, p_r\right) + 1, & \text{otherwise, where } p \geq p_l + p_r \end{cases}$$

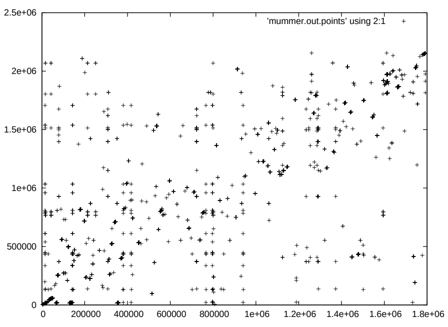
We will prove by induction that $g(n, p) \leq p \log_2 n$. Base case is trivial. For the induction step we have

$$\begin{aligned} g(n,p) &= g\left(\frac{n}{2}, p_l\right) + g\left(\frac{n}{2}, p_r\right) + 1 \\ &\leq p_l \log_2 \frac{n}{2} + p_r \log_2 \frac{n}{2} + 1 \\ &= (p_l + p_r) \log_2 \frac{n}{2} + 1 \\ &\leq p \log_2 \frac{n}{2} + 1 \\ &= p \log_2 n - p + 1 \\ &\leq p \log_2 n \end{aligned}$$

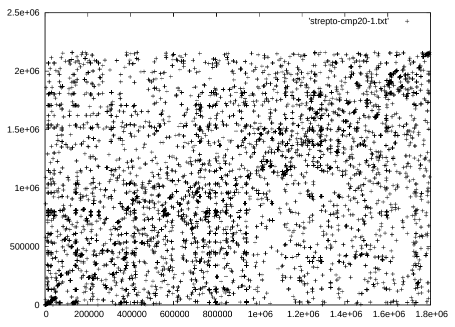
□

3 Analysis and Preliminary Experimental Results

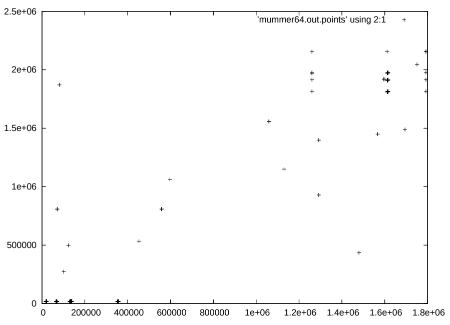
We will now prove the theoretical sublinear running time of our algorithm. Our time essentially matches that of Ukkonen [13].



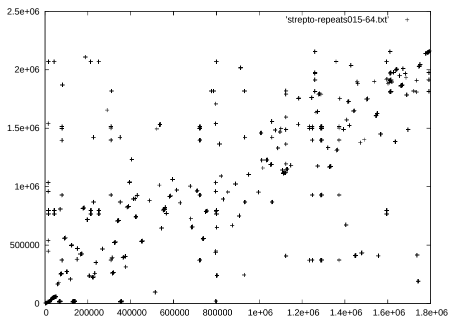
(a) MUMmer exact 20bp



(b) Approximate alignment 20bp with 1 error



(c) MUMmer exact 64bp



(d) Approximate alignment 85% similarity 64bp

Figure 4: Visual comparison with MUMmer exact alignment. Each + is a match at the corresponding coordinates in the genomes

Lemma 3. *Let l be the length of keyword and $d = l \cdot \frac{k}{m-l}$ be the maximum difference allowed for a keyword. Then the keyword search algorithm will take $O(ml^{d+2}\sigma^d \log_2 n) \subseteq O(m^{k+3}\sigma^k \log_2 n) \subseteq o(n)$.*

Proof. It is again easier to calculate the running time if we consider the algorithm over a suffix trie. Consider any internal node of the suffix trie that is visited during the execution of this algorithm. The string represented by the path from the root to such internal node should be a prefix of a string that matches keyword with at most d differences. But total number of such strings is $O(l^d\sigma^d)$. Therefore the total number of viable prefixes is $O(l^{d+1}\sigma^d)$.

If the alignment score is computed by maintaining the column of dynamic programming table the work for each node of the suffix trie is l (the height of a column of DP). By Lemma 2 our algorithm works on suffix arrays with a factor of $\log_2 n$ slowdown in running time. Also there are $O(m)$ keywords to search for in each pattern. Hence the total running time is $O(ml^{d+2}\sigma^d \log_2 n)$. $\square$

To examine the practical applicability of this algorithm we have implemented a variant of this algorithm in C++. There are two main differences between this implementation and the algorithm described above. First instead of maintaining a dynamic programming row we exhaustively search all possible edit operations at each internal node. Second this implementation does an all-vs-all search for any similar substrings between two fairly long sequences. The main search function

```
void sasearch_edit(int l1, int r1, int pos1,
                   int l2, int r2, int pos2,
                   int diff)
```

recursively finds all approximate matches between two windows in the two suffix arrays assuming the difference between the shared prefix of suffixes in first window and shared prefix of suffixes in the second window is given. The complete implementation of this function is listed in Appendix A to show the compactness of the implementation.

Figure 3 shows the results of running the algorithm on full sequence of genomes of two bacterial organisms: Streptococcus pneumoniae and Streptococcus thermophilus, in comparison to the exact algorithm of MUMmer [2]. By default MUMmer finds exact matches of length 20 between the two genomes, as can be seen in Figure 4(a). For comparison have included approximate matches of length 20 with one error Figure 4(b). Note that the number of false positives increases significantly if we allow just one error in a 20 base pair match. To decrease the number of false positives one can of course increase the length of the match, an exact match of 64 bases is shown in Figure 4(c). It is clear that we have decreased sensitivity considerably by increasing the length of the match. Finally our algorithm which looks for approximate (85%) matches of length 64 bases is shown in Figure 4(d). It can be seen that our algorithm has simultaneously better sensitivity and better specificity that all variations above.

References

[1] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] AL Delcher, S. Kasif, RD Fleischmann, J. Peterson, O. White, and SL Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369, 1999.

[3] Robert Krauthgamer and Yuval Rabani. Improved lower bounds for embeddings into l1. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 1010–1017, New York, NY, USA, 2006. ACM.

[4] S. Kurtz. Reducing the space requirement of suffix trees. *Software-Practice and Experience*, 29(13):1149–71, 1999.

[5] S. Kurtz. The Vmatch large scale sequence analysis software. *Ref Type: Computer Program*, pages 4–12, 2003.

[6] GM Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989.

[7] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[8] B. Ma, J. Tromp, and M. Li. PatternHunter: faster and more sensitive homology search, 2002.

[9] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327. Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 1990.

[10] EW Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4):345–374, 1994.

[11] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys (CSUR)*, 33(1):31–88, 2001.

[12] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, 1(1):205–239, 2000.

[13] E. Ukkonen. Approximate string-matching over suffix trees. *Lecture notes in computer science*, pages 228–228, 1993.

A All-vs-All Search Source Code

```
const float EPSILON = 0.15;
const int MIN_LENGTH = 40;

char *s1, *s2;
int *suffix1, *suffix2;

set<pair<int, int> > results;

void sasearch_edit(int l1, int r1, int pos1,
                   int l2, int r2, int pos2,
                   int diff)
{
  if(diff > min(pos1, pos2) * EPSILON)
    return;

  if(max(pos1, pos2) >= MIN_LENGTH){
    for(int i = l1; i <= r1; ++i)
      for(int j = l2; j <= r2; ++j)
        results.insert(make_pair(suffix1[i], suffix2[j]));
    return;
  }

  if(s1[suffix1[l1]+pos1] != s1[suffix1[r1] + pos1]){
    int mid1 = (l1 + r1)/ 2;
    sasearch_edit(l1      , mid1, pos1, l2, r2, pos2, diff);
    sasearch_edit(mid1 + 1, r1   , pos1, l2, r2, pos2, diff);
    return;
  }else if(s2[suffix2[l2] + pos2] != s2[suffix2[r2] + pos2]){
    int mid2 = (l2 + r2) / 2;
    sasearch_edit(l1, r1, pos1, l2      , mid2, pos2, diff);
    sasearch_edit(l1, r1, pos1, mid2 + 1, r2   , pos2, diff);
    return;
  }else{

    if((s1[suffix1[l1] + pos1] == '$') ||
       (s2[suffix2[l2] + pos2] == '$'))
      return;

    // match/ mismatch
    if(s1[suffix1[l1] + pos1] == s2[suffix2[l2] + pos2])
      sasearch_edit(l1, r1, pos1 + 1, l2, r2, pos2 + 1, diff);
    else
      sasearch_edit(l1, r1, pos1 + 1, l2, r2, pos2 + 1, diff + 1);


    // insert/ delete
    sasearch_edit(l1, r1, pos1 + 1, l2, r2, pos2    , diff + 1);
    sasearch_edit(l1, r1, pos1    , l2, r2, pos2 + 1, diff + 1);
    return;
  }
}
```