What's Behind Blast

Gene Myers

MPI for Cellular Molecular Biology and Genetics, 01307 Dresden, DE
myers@mpi-cbg.de

Abstract. The BLAST search engine was published and released in 1990. It is a heuristic that uses the idea of a neighborhood to find seed matches that are then extended. This approach came from work that this author was doing to lever these ideas to arrive at a deterministic algorithm with a characterized and superior time complexity. The resulting $O(en^{pow(e/p)}logn)$ expected-time algorithm for finding all e-matches to a string of length p in a text of length n was completed in 1991. The function $pow(\epsilon)$ is 0 for $\epsilon = 0$ and concave increasing, so the algorithm is truly sublinear in that its running time is $O(n^c)$ for $c < 1$ for ϵ sufficiently small. This paper reviews the history and the unfolding of the basic concepts, and it attempts to intuitively describe the deeper result whose time-complexity, to this authors knowledge, has yet to be improved upon.

1 The Meeting

The 1980's were an active decade for basic advances in sequence comparison algorithms. Michael Waterman, Temple Smith, Esko Ukkonen, Webb Miller, Gad Landau, David Lipman, Bill Pearson, and myself, among others, were all very active in this period of time and were working out the basic algorithms for comparing sequences, approximate pattern matching, and database searching (e.g. [1,2,3,4,5,6]). During this time, the BLAST heuristic was developed and deployed at the National Library of Medicine in 1990 and the paper that described it became one of the most highly cited papers in science [7]. This paper is about how the design for the algorithm came about, from my point of view, and its relationship to the theoretical underpinnings of sequence comparison that I was exploring at the time that ultimately lead to an efficient, deterministic, expected-time algorithm for finding approximate matches using a precomputed index [8].

In 1988, Webb Miller and I organized a small bioinformatics meeting in Bethesda, Maryland that included such notable figures as David Sankoff, Michael Waterman, Temple Smith, Eric Lander, Zvi Galil, Esko Ukkonen, David Lipman and other great investigators of that time. At the meeting Zvi Galil gave a talk about suffix trees [9] and raised two questions that ultimately were answered:

1. Can suffix trees be built in a way that is independent of alphabet size s?
2. Can a precomputed index such as a suffix tree of a large text be used to speed up searches for approximate matches to a string?

To understand the first question, one must recall that one can either use an s-element array in each suffix tree node to permit a search for a string of length p in a text of length n in $O(p)$ time but requiring $O(ns)$ space for the suffix tree, or one can use only $O(n)$ space by using binary trees to decide which edge to follow out of a node, but resulting in $O(plogs)$ time for the search. This question ultimately led Udi Manber and I to develop suffix arrays in late 1990 [10], where a suffix array occupies $O(n)$ space, independent of s, and takes $O(p+logn)$ time to search for a string, again independent of s. This data structure in turn enables the Burroughs-Wheeler Transform or BWT [11], that is now greatly in vogue for next-gen sequencing (NGS) applications [12], to be computed in $O(n)$ time.

2 Filters and Neighborhoods

But it was the second question on the use of an index, such as a suffix tree, to speed searches for approximate matches that captured my attention immediately after the meeting. Most algorithmicists work on *deterministic* search algorithms meaning that the method finds exactly the set of locations in the text where a query approximately matches within some specified threshold, whereas a *heuristic* is an algorithm that finds most of the matches sought, but may miss a few, called a *false negative*, and may further report a few locations where a match doesn't actually occur, called a *false positive*. In between these two types of algorithms, a *filter* is an algorithm that has no false negatives but may produce false positives. That is, it produces a superset of the instances sought, or equivalently it filters out most of the locations where matches do not occur. A filter can be the first step of a deterministic algorithm simply by running a deterministic checking algorithm on the subset of locations reported by the filter. If the filter is much more efficient than the deterministic checker, then one ends up with a much more efficient search.

At the time, there were surprisingly no published methods using the simple idea of finding *exact* matches to k-mers (strings of length k) from the query string [13,14] even though this was fairly obvious and had been used in the heuristic method of FASTA. Shortly after the Bethesda meeting, I had the first and most important idea of looking for exact matches to strings in the *neighborhood* of k-mers selected from the query string. Let δ be a sequence comparison measure that given two strings v and w returns a numeric measure $\delta(v, w)$ of the degree to which they differ (e.g. the generalized Levenshtein metric). Given a string w the τ-neighborhood of w with respect to δ, $\aleph_\tau^\delta(w)$ is the set of all strings v whose best alignment with w under scoring scheme δ is less than τ, i.e. $\{v : \delta(v, w) \leq \tau\}$.

For this paper, except where mentioned otherwise, we are focusing on the approximate match problem where δ is the *simple Levenshtein* metric, which is the minimum number of insertions, deletions, and substitutions possible in an alignment between the two strings in question. That is, we seek matches of a query of length p to a text of length n, where up to e differences are allowed. Another way to phrase this, which we will use interchangeably, is that we seek ϵ-matches where $\epsilon = e/p$ is the *length relative* fraction of differences allowed. To illustrate the idea of a neighborhood under this metric, the 1-neighborhood of *abba* (or 25%-neighborhood) is $\aleph_1(abba) = \{aaba, aabba, abaa, aba, abaa, ababa, abba,$ *abbaa*, *abbab*, *abb*,*abbb*, *abbba*, *babba*, *bba*, *bbba*$\}$.

For the example above, notice that wherever one finds *abaa* one will also finds *aba* as it is a prefix of the former. So to find all matches to neighborhood strings it suffices to look up in an index only those that are *not* an extension of a shorter string in the same neighborhood. Let the *condensed* τ-neighborhood of w be the subset of these strings, i.e. $\bar{\aleph}^{\delta}_{\tau}(w) = \{v : v \in \aleph^{\delta}_{\tau}(w)$ and $\nexists u \in \aleph^{\delta}_{\tau}(w)$ such that u is a prefix of $v\}$. For our example, the condensed 1-neighborhood of *abba* is $\bar{\aleph}_1(abba) = \{aaba, aabba, aba, abb$, *babba*, *bba*, *bbba*$\}$, a considerably smaller set of strings.

To illustrate the advantage of using (condensed) neighborhoods, consider looking for a match with 9 differences to a query of length say 100. If one partitions the query into 10 strings of length 10, then by the Pigeon Hole principle, one of the 10 strings must exactly match in the database. So one can filter the text by looking for one of these 10 strings of length 10. But if one partitions the query into 5 strings of length 20, then by the Pigeon Hole principle, a string in the 1-neighborhoods of the 5 query parts must exactly match in the database. A rough estimate for the number of strings in the condensed e-neighborhood of a string of length k is $\bar{\aleph}_e(k) = \binom{k}{e}(2s)^e$. Thus in our example we can filter the text by looking for one of 800 strings of length 20. Which filter is better? The probability of a random false positive for the k-mer filter is $10/s^{10}$ and for the neighborhood filter it is $800/s^{20}$. Thus the later filter produces $s^{10}/80$ fewer false positives. If s is 4 (e.g. the DNA alphabet) and n is 3×10^9 (e.g. the size of the human genome) then the neighborhood filter produces $13,000$ times fewer false positives, and reports in expectation 2.18 false positive, whereas the k-mer filter reports over $28,600$!

3 Version 0.1

For every true positive location, i.e. an approximate match is present, one must spend time proportional to the best algorithm available for aligning one sequence to another. While there are some quite sophisticated algorithms, in practice, one of the best is still the $O(pe)$ algorithm discovered by Ukkonen [2] and a year later by myself [6], where I further proved that the algorithm runs in $O(p + e^2)$ expected time, and can be modified with a suffix tree and $O(1)$ lca-finding [15] to take this much time in the worst-case. If a search results in h true hits, we will assume for simplicity that $O(hep)$ time will be taken to confirm and report all the matches and their alignments. The goal of a filter is to deliver the hits efficiently and to waste as little time as possible on false positives. That is, the goal is to optimize the time a filter would take on a random text that in expectation has no hits to the query. The initial simple idea that I was working with in early 1989 was as follows:

1. Partition the query into p/k k-mers.
2. Generate every string in the (ϵk)-neighborhood of the query k-mers and find all the exact matches to these strings using an index.
3. Check each location reported above with the $O(ep)$ algorithm.

The question is what k-mer size leads to the best expected-time performance of the filter over a random text? Roughly, the number of neighborhood strings is $(p/k)\bar{\aleph}_{\epsilon k}(k)$ and the time spent looking up each is $O(k)$ excluding the time to check hits for each. Thus the lookup phase takes $O(p\bar{\aleph}_{\epsilon k}(k))$ time. The expected number of hits is $(p/k)\bar{\aleph}_{\epsilon k}(k)(n/s^k)$ and thus the expected-time for checking proposed locations is $O((\epsilon p^3/k)\bar{\aleph}_{\epsilon k}(k)(n/s^k))$. Thus the total expected time for the filter is:

$$O(p\bar{\aleph}_{\epsilon k}(k)(1 + \frac{\epsilon p^2 n}{ks^k})) \tag{1}$$

I was unable to produce an analytic formula for the value of k that as a function of n, p, and ϵ gives the minimum time. However, using Stirling's Approximation, I was able to demonstrate (unpublished) that the best value of k is always bigger than $log_s n$ and less than $(1+\alpha)log_s n$ where α becomes increasingly closer to 0 as n/p goes to infinity. For typical values of the parameters, and especially when n is much larger than p, α is quite quite close to zero. Thus one instinctively knows that the k-mer size should be on the order of $log_s n$. We will use this observation later on.

4 BLAST

In May of 1989, I spent 2 weeks at the National Center for Biotechnology Information (NCBI) with David Lipman. I was a smoker at the time and was having a cigarette outside when David came out and showed me an article in *Science* in which Lee Hood was extolling the virtues of a new systolic array chip built by TRW called the Fast Data Finder (FDF) [16]. I proceeded, as a firm believer that special hardware is not the way to solve problems, to explain to David my new ideas for approximate search and how I thought we could do such searches just as well in software rather than spend money on relatively expensive hardware. David had previously developed FASTA with Bill Pearson [5], which at the time was the best heuristic for searching protein databases. David listened carefully and started to think about how the ideas could be used in a heuristic and efficient way to search for significant locally aligned regions of a protein query against a protein database under a general scoring scheme such as the PAM or BLOSSUM metrics. In short order we had the following heuristic adaption of my first filter:

1. Consider the $p-k+1$ *overlapping* k-mers of the query (to increase the chance of not missing a true hit).
2. Generate every string in the τ-neighborhood of the query k-mers under a similarity-based protein metric δ and find all the exact matches to these strings by some means (an index may not be in practice the fastest way to do this).
3. Extend each seed match into a local alignment of significance by some means, and report it if its score is sufficiently high.

Over the next several months a number of versions of codes based on the above template were developed by myself, Webb Miller, and Warren Gish.

Webb tried a simple index for the look up and reported that it was quite slow. In hindsight this was just at the time when the mismatch in speed between memory access and processor speed was becoming severe enough that being aware of cache-coherence was becoming essential for good performance. I still wonder if a better design of an index and the order of lookups within it, e.g. sorting the strings to be looked up, would not lead to a much speedier implementation. The other idea and faster implementation was to generate a finite automaton of the neighborhood strings and in an $O(n)$ scan of the text with the automaton find all potential match locations. Each state of the automaton had an s-array table of transitions. Gish realized that if a Mealy machine [17] was used instead of a Moore machine [18] (i.e. report hits on transitions rather than on states), a factor of s is saved in space. Given that s is 20 for protein alphabets this was a significant space saving.

For the extension step we tried simply extending forward and backward with no indels. I proposed the idea that an extension step stop when the score of the extension dropped too far below the best score seen (the X-factor). I also wrote a version that extended with indels, again observing the X-factor, but Lipman deemed that it was too slow and not worth the additional computer time. He later reversed his position in 1999 with a second release and publication of BLAST [19], albeit with Miller reinventing the gapped extension strategy.

Warren also wrote all the code for practical matters such as low-complexity sequence filtering and he built the initial web server [20]. Altschul, the first author, added the calculation of the significance of each match based on work he and Sam Karlin had published earlier in the year [21]. He also tested the sensitivity and performance of the method and wrote the paper. An unfortunate consequence of this was that the algorithm was inadequately described and led to much confusion about what the the BLAST algorithm was over the ensuing years. However, the use of the match statistics was a great advance and enhanced the popularity of the engine, as previously there had been much optimistic reporting of statistically insignificant matches in the formal molecular biology literature.

5 Doubling Extension of $\log_s n$ Seeds

While BLAST was being developed, I continued to pursue the quest for a provably efficient deterministic algorithm. The simple seed and test strategy hadn't yielded an analytic expected time complexity, but it did suggest that $k = \log_s n$ might be a good seed size. Indeed, I liked immediately that since $s^{log_s n} = n$, a simple $2n$ integer index of all the k-mers in the text permits $O(p + h)$ expected-time lookup of all h exact matches to a string of length p in the text. Later I was further able to give an analytic estimate for the size of neighborhoods of strings of this special size, but at the time I was focused on the extension step, as it was the aspect that was not yielding an analytic bound. Later I would prove it, but at the time I intuited that if ϵ is small enough, then the probability, $Pr(p, \epsilon)$, of a random ϵ-match to a string of length p is less than $1/\alpha^p$ for some fixed $\alpha > 1$ that is a function of ϵ (just as an exact match has probability $1/s^p$). If this is true, then as shown below, the time for an extension strategy based on progressively doubling and checking seed hits telescopes for false hits.

The basic "double and check" idea is as follows. Suppose a k-mer of the query, s_0, ϵ-matches a substring t_0 of the database. The idea of doubling and checking, is to try a $2k$-mer s_1 of the query that spans s_0 and check with the customary zone-based dynamic programming algorithm if there is a string t_1 spanning t_0 that ϵ-matches s_1. If not, then one can, under the right doubling protocol to be given shortly, conclude that an ϵ-match to the query does not exist that spans t_0. Otherwise, a match to the query is still possible, so one proceeds to double s_1 to a substring s_2 of the query of length $4k$ and then check for an ϵ-match to it spanning t_1. And so on, until either there is a failure to match at some doubling stage, or until all of the query is found to match a string spanning the seed hit t_0.

Returning to the complexity claim, if one assumes $Pr(p, \epsilon) < 1/\alpha^p$ for some α, then one starts with $h = (p/k)(n/\alpha^k)$ expected random k-mer seed matches. The idea is to check if these can be extended to ϵ-matches of length $2k$, and then to ϵ-matches of length $4k$, and so on. For a text that is random with respect to the query, the extensions that survive diminish hyper-geometrically. Specifically, there are $n(p/k)/\alpha^{2^{x-1}k}$ surviving hits at doubling stage x and it takes $O(\epsilon(2^x k)^2)$ time to check each implying that the total expected time to eliminate *all* of these random seeds is:

$$n(p/k) \sum_{x=1}^{log_2 p/k} \epsilon(2^x k)^2/\alpha^{2^{x-1}k} = nek/\alpha^k \sum_{x=1}^{log_2 p/k} 4^x/\alpha^{(2^{x-1}-1)k} = O(nek/\alpha^k) \quad (2)$$

But how are the doublings of the seeds arranged? To keep it simple, suppose k divides p, and $p/k = 2^\pi$ is a power of 2. If a query w of length p has an ϵ-match to a substring v of the text, then by the Pigeon Hole principle either the first or second half of w, defined as w_0 and w_1, ϵ-matches a prefix v_0 or suffix v_1 of v, respectively, where $v_0 v_1 = v$. Inductively if w_x has an ϵ-match to a string v_x, then by the Pigeon Hole principle either the first or second half of w_x, defined as w_{x0} and w_{x1}, ϵ-matches a prefix v_{x0} or suffix v_{x1} of v, respectively, where $v_{x0} v_{x1} = v_x$. In other words, if there is an ϵ-match to the query w, then there is at least one binary string α of length π such that w_β has an ϵ-match to a string v_β for all prefixes β of α where it is further true that $v_{\beta x}$ is a prefix or suffix of v_β according to whether x is 0 or 1, respectively. So now reverse the logic and imagine one has found a seed ϵ-match to a piece w_α of the query where $\alpha = a_1 a_2 \ldots a_\pi$. To determine if w has a match involving this seed match, one considers checking for ϵ-matches to the doubling sequence of strings $w_{(a_0 a_1 \ldots a_{\pi-1})}$, $w_{(a_0 a_1 \ldots a_{\pi-2})}$, $\ldots$, $w_{(a_0 a_1)}$, $w_{(a_0)}$, $w_{(\epsilon)} = w$, discovering the prefix and/or suffixes v_β at each level, until either w is confirmed or a check fails. This strategy is deterministic as it never misses a match, yet the expected time spent on a false positive seed is $O(\epsilon k^2)$.

When there is a match present, the time spent confirming the match is:

$$\sum_{x=1}^{\pi} \epsilon(2^x k)^2 = \epsilon k^2 \sum_{x=1}^{\pi} 4^x = \epsilon k^2 \left(4^{\pi+1}/3 - 1\right) < 4/3\epsilon p^2 = O(ep) \tag{3}$$

So in the case that there are exactly h real matches to the query, the extension and reporting phase of an algorithm using this strategy takes expected-time:

$$O(nek/\alpha^k + hep) \tag{4}$$

When in particular k is chosen to be $log_s n$, then $\alpha^k = n^{log_s \alpha}$ and so the extension step takes $O(en^{1-log_s \alpha} logn + hep)$ time. This is exciting because as long as α is greater than 1 (how much so depends on ϵ), then the time is $O(n^c)$ for $c < 1$ and hence truly sublinear in n. In the next section, we will confirm that indeed $Pr(k,\epsilon) < 1/\alpha^k$ for an α that is a function of ϵ, and thus that the complexity of this section holds.

6 Neighborhood Size

I was fairly confident I could come up with a good algorithm for generating neighborhoods that was proportional to the size of the condensed neighborhood, but I was less certain about arriving at an analytic upper bound for the size of a condensed d-neighborhood of a string of length k, $\bar{\aleph}^k_d$. In the introduction I (inaccurately) estimated it as $\binom{k}{d}(2s)^d$ and in the previous section I guessed such a bound would have the form $\alpha(\epsilon)^k$ where α depends on $\epsilon = d/k$. I embarked on developing recurrences for counting the number of sequences of d distinct edits that one could perform on a string of length k. Rather than consider induction over the sequence of edits, I thought about an induction along the characters of the string from left to right. At each character one can either leave it alone, delete it, insert some number of symbols after it, or substitute a different symbol for it and optionally insert symbols after it. Note carefully that redundant possibilities, such as deleting the symbol and then inserting a character after it, or substituting a symbol and then deleting it, need not be counted. While I took some care to produce tight recurrences at the time, I recently noted that I could improve the recurrences but interestingly I could not prove a better complexity bound than with the original recurrence. We will present the new recurrence with the idea that another investigator might prove a better bound.

Suppose one has k symbols left in the query, and needs to introduce d differences into this string of remaining characters where insertions before the first symbol are not allowed. Let $S(k,d)$ be the number of such d-edit scripts. The new lemma is:

Lemma: If $k \leq d$ or $d = 0$ then $S(k,d) = \bar{\aleph}_d(k) = 1$. Otherwise,

$$\begin{aligned} S(k,d) = {} & S(k-1,d) + (s-1)S(k-1,d-1) + (s-1)\textstyle\sum_{j=0}^{d-1} s^j S(k-2,d-1-j) \\ & +(s-1)^2 \textstyle\sum_{j=0}^{d-2} s^j S(k-2,d-2-j) + \sum_{j=0}^{d-1} S(k-2-j,d-1-j) \end{aligned}$$

$$\bar{\aleph}_d(k) \quad \leq S(k,d) + \textstyle\sum_{j=1}^{d} s^j S(k-1,d-j)$$

Proof: The new recurrences begins with the observations that (a) a deletion followed by an insertion is the same as a substitution, (b) a deletion followed by a substitution is the same as a substitution followed by a deletion, (c) an insertion followed by a substitution is the same as a substitution followed by an insertion, and (d) an insertion followed by a deletion is the same as doing nothing. Therefore we need only consider scripts in which deletions can only be followed by deletions or an unchanged character, and in which insertions can only be followed by other insertions or an unchanged character. A substitution or unchanged character can be followed by any edit (or no edit). Furthermore, it is redundant to substitute a character for itself implying there are only $s-1$ choices for a substitution at a given position. Moreover, an insertion following an unchanged or substituted character is redundant if the inserted character is equal to the one behind it, because the net effect is the same as inserting the given character *before* the symbol it follows. So there are only $s-1$ non-redundant characters for the first insert in a sequence of inserts. Finally, we need only produce *condensed* neighborhoods, so when $t \leq d$ the number of scripts is 1 as the null string is in the neighborhood and hence the condensed neighborhood contains only this string. Thus it is clear that $S(k,d) = 1$ when either $k \leq d$ or $d = 0$. For all other values of k and d it follows from the "rules" above that:

$$\begin{aligned} S(k,d) = {} & S(k-1,d)+(s-1)(S(k-1,d-1)+I(k-1,d-1)) \\ & +(s-1)^2 I(k-1,d-2) + D(k-1,d-1) \end{aligned}$$

where $I(k,d)$ is the number of d edit scripts that immediately follow one or more inserts after the $k+1^{st}$ symbol in the query string, and $D(k,d)$ is the number of d edit scripts that immediately follow a deletion of the $k+1^{st}$ symbol in the query string. It follows from the "rules" that:

$$\begin{aligned} I(k,d) &= sI(k,d-1) + S(k-1,d) \\ D(k,d) &= D(k-1,d-1) + S(k-1,d) \end{aligned}$$

Solving the recurrences for I and D in terms of S and substituting these back into the recurrence for S gives the final recurrence of the lemma for S, and the bound for $\bar{\aleph}_d(k)$ simply considers that one can have one or more inserts before the first character of the query. ■

A simple but tedious exercise in induction reveals that for any value $c \geq 1$, $S(k,d) \leq B(k,d,c)$ and $\aleph_d(k) \leq \frac{c}{c-1} B(k,d,c)$ where $B(k,d,c) = (\frac{c+1}{c-1})^k c^d s^d$. It further follows that $B(k,d,c)$ is minimized for $c = c^\star = \epsilon^{-1} + \sqrt{1+\epsilon^{-2}}$ where $\epsilon = d/k$. Given that $\epsilon \in [0,1]$, it follows that $c^\star \in [1+\sqrt{2}, \infty]$ implying $c^\star$ is always larger than 1 and that $\frac{c}{c-1}$ is always less than $1+\sqrt{.5}$. Therefore,

$$S(k,d) \leq B(k,d,c^\star) \text{ and } \aleph_d(k) \leq 1.708 B(k,d,c^\star) \qquad (5)$$

As in the original paper, one can similarly and easily develop recurrences for the probability $Pr(k,\epsilon)$ of an ϵ-match to a string of length k in a uniformly random text and show that the recurrence is bounded by $\frac{c}{c-1}B(k,d,c)/s^k$ where $d = \lceil \epsilon k \rceil$. Therefore:

$$Pr(k,\epsilon) \leq 1.708/\alpha(\epsilon)^k \text{ where } \alpha(\epsilon) = (\frac{c^\star - 1}{c^\star + 1})c^{\star - \epsilon}s^{1-\epsilon} \qquad (6)$$

proving that the bound used in Section 5, and hence also the complexity of the extension step of the algorithm derived in that section.

Now consider the function $pow(\epsilon) = \log_s \frac{c^\star+1}{c^\star-1} + \epsilon \log_s c^\star + \epsilon$. A little algebra and the bounds in Equations (5) and (6) allow us to conclude the following rather striking bounds:

$$\aleph_\epsilon(k) = O((s^{pow(\epsilon)})^k) \text{ and } \alpha(\epsilon) = O(s^{1-pow(\epsilon)}) \qquad (7)$$

The first bound effectively says that one can think of each position in the string as have a certain "flex factor" at a given rate ϵ, namely $s^{pow(\epsilon)}$, so that the neighborhood size is the k^{th} power of the flex factor. The second bound effectively says that the "match specificity" of each position in the set of neighborhood strings is $s^{(1-pow(\epsilon))}$, so that the probability of matching any string in the ϵ-neighborhood of a string of length k is 1 over the k^{th} power of this match specificity.

While quite complex in form, note that $pow(\epsilon)$ is monotone increasing and concave in ϵ and $pow(0) = 0$. The last fact implies $\aleph_0(k) = 1$ and $\alpha(0) = s$ as expected. It rises to the value of 1 before ϵ becomes 1, and does so at a point that depends on the size s of the alphabet. We plot it below in Figure 1 for several value of s. Finally, note that if $k = log_s n$ then $\aleph_\epsilon(k) = O(n^{pow(\epsilon)})$.

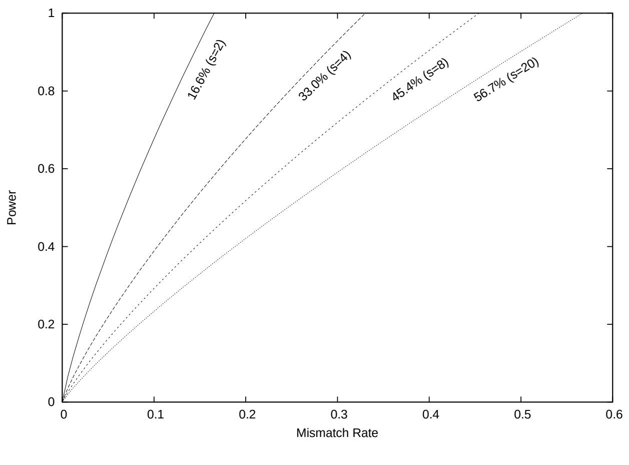

Fig. 1. $Pow(\epsilon)$ plotted for several values of s. For each curve the percentage mismatch at which Pow becomes 1 is given.

7 Generating Condensed Neighborhoods

With the analysis of complexity in hand, the only remaining problem was to efficiently generate all the strings in a condensed d-neighborhood of a string w of length k. The basic idea is to explore the trie of all strings over the underlying alphabet, computing row by row the dynamic programming matrix of w versus the string on the current path in the trie. That is, given the last row, $L_{v,w}$, of the dynamic programming matrix for w versus a string v, one computes, in $O(k)$-time, the last row of the dynamic program matrix for w versus va for every letter a in the alphabet. For the simple Levenshtein measure, the smallest value in a row is monotonically increasing and therefore once a row R has a minimum value, $\min(R)$, greater than d, one can certainly eliminate the current string and all extensions of it as belong to the condensed neighborhood. Conversely, once a row is reached that has d as its last entry, then a string in the condensed neighborhood has been reached and one should report the current string and then backtrack. In pseudo code, one calls *Search*(ϵ,[012 . . . k]), where *Search* is the routine:

```
Search($v$,$R$)
  if $R[k] = d$ then
    Report $v$
  else if $\min(R) \leq d$ then
    for $a \in \Sigma$ do
      Compute $S = L_{va,w}$ from $R$
      Search($va$,$R$)
```

The big problem above is that too much time is taken visiting words that are not in the condensed neighborhood. As soon as $\min(R)$ is d, we know that the only possible words in the condensed neighborhood are those that are extended by the suffix w^x for each x such that $R[x] = d$, where w^x is the suffix of w consisting of its last $k - x$ symbols. This gives us the algorithm:

```
Search($v$,$R$)
1.  if $\min(R) = d$ then
2.    for each x s.t. R[x] = d do
3.      Report $v \cdot w^x$
4.  else # $\min(R) < d$ #
5.    for $a \in \Sigma$ do
6.      Compute $S = L_{va,w}$ from $R$
7.      Search($va$,$S$)
```

Now the number of terminal strings visited (i.e., those for which no further recursion is pursued) is less than the number of words in the condensed neighborhood as at least one member is reported in lines 2 and 3. Moreover, the number of interior strings is also less than the number of terminal strings as the recursion tree is an s-ary complete tree. Thus the total number of calls to *Search* is $O(\bar{\aleph}_d(k))$ and each call takes $O(k)$ time.

Next, immediately note that one need only compute the $2d+1$ values of the dynamic programming matrix that are in the band between diagonals $-d$ and d as one can easily show that any value outside this band must be greater than d. Thus the relevant part of each row is computed in $O(d)$ time. But then how does one look up $v \cdot w^x$ in an index in less than $O(k)$ time given that $|w^x|$ can be on the order of k?

The answer comes from the fact that $k = log_s N$ and thus we can build a very simple index based on directly encoding every k-mer w as an s-ary number, $code(w)$, in the range $[0, s^k - 1] = [0, n-1]$. It is an easy exercise (and shown in the earlier paper [8]) that with two n-vectors of integers, one can build an index that for each k-mer code delivers the positions, in the underlying text (of length n), at which that k-mer occurs. So with such a simple structure, reporting a string in the neighborhood requires only delivering its code. First note that one can incrementally compute $code(va) = code(v) \cdot s + code(a)$ in $O(1)$ time, and second, that one can *precompute* $code(w^z)$ and $power(z) = code(s^z)$ for every z in a relatively minuscule $O(k)$ time before starting the search. So during the generation of neighborhoods, one gets the code for $v \cdot w^x$ in $O(1)$ time by way of the fact that $code(v \cdot x) = code(v) \cdot power(k-x) + code(w^x)$.

The careful reader will note that there is one remaining problem. Namely, that in line 2 of *Search* there could be two or more entries in R, say x and $z > x$ such that $R[x] = R[z] = e$ and it could be that $v \cdot w^x$ is not in the condensed neighborhood because w^z is a prefix of w^x! To me it was fascinating that the answer lies in the failure function, ϕ, of the Knuth-Morris-Pratt (KMP) algorithm for finding matches to a string in a linear time scan of a text. Recall that for a string $v = a_1 a_2 \ldots a_k$ that $\phi(x)$ is the maximum y such that $a_1 a_2 \ldots a_y$ is a suffix of $a_1 a_2 \ldots a_x$. A table of $\phi[x]$ for all x was shown by KMP to be constructible in $O(k)$ time. Building ϕ on the reverse of w gives us exactly the relationships we want. That is, $\phi(x)$ is the maximum y such that w^y is a prefix of w^x. To test in the general case that w^z is a prefix of w^x, we test if $\phi^k(x) = z$ for some number of application k of ϕ. Since only $2e+1$ contiguous suffixes will be considered at the most, using the failure function to test if any one is a prefix of another takes $O(e)$ time with a simple marking strategy.

Thus, we have an $O(d\aleph_d(k))$ algorithm for generating all the words in the condensed d neighborhood of a string w of length k. Given seed size k, there are p/k seeds and so generating the strings in the condensed neighborhoods of all of them takes $O(p/k \cdot \epsilon k \, \aleph_\epsilon(k)) = O(e \, \aleph_\epsilon(k))$ time.

8 Total Running Time

Putting the time complexities of Section 7 for the generation of neighborhoods and Section 5 for the extension of seed matches, we have as a function of seed size k:

$$O(e\,\aleph_\epsilon(k) + nek/\alpha(\epsilon)^k + hep) \tag{8}$$

Using Equation (7) from Section 6, the complexity excluding the $O(hep)$ true positive term is:

$$O(e(s^{pow(\epsilon)k} + nek/s^{(1-pow(\epsilon))k}) = O(e(s^k)^{pow(\epsilon)}(1 + k\frac{n}{s^k})) \tag{9}$$

When one chooses $k = log_s n$ as the seed size, then $s^k = n$ and we formally arrive at the expected time complexity for the entire algorithm given in the following theorem.

Theorem: Using a simple $O(n)$ precomputed index, one can search for ϵ-matches to a query of length p in a text of length n, in:

$$O(en^{pow(\epsilon)} logn + hep) \text{ expected time} \tag{10}$$

where $pow(\epsilon) = \log_s \frac{c^\star+1}{c^\star-1} + \epsilon \log_s c^\star + \epsilon$ and $c^\star = \epsilon^{-1} + \sqrt{1+\epsilon^{-2}}$.

9 Final Remarks and Open Problems

In a hopefully intuitive style, the reader has been introduced to a fairly involved set of theoretical ideas that underlie the BLAST heuristic and that give a deterministic, expected-time algorithm that is provably sublinear in the size of the text for suitably small ϵ. That is the algorithm's running time is $O(n^c)$ for $c < 1$ (and not $O(cn)$ for $c < 1$ as in a "sublinear" method such as the Boyer-Moore exact match algorithm). Interestingly, the author is not aware of any work that builds on this approach or another that has a superior time complexity.

There are at least two interesting questions. The first involves the analysis of neighborhood sizes. Is there a tighter bound to the recurrences formulated here and/or are there better recurrences? Such bounds would give a tighter characterization of the running time of the algorithm. The second question is a bit harder to formulate, but the essence of it is whether or not this algorithm can be shown to be a lower bound on the time required to find all ϵ-matches. In other words, one wonders whether the idea of using a $log_s n$ seed size and then carefully doubling such hits is essential for ruling out false positive locations. That is, must one spend this amount of time eliminating a near miss? If true, it would also explain why a better result of its kind has not been forthcoming in the last 20 years since the first publication of this result.

References

1. T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147(1):195–197, 1981.
2. E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100-119, 1985.
3. E. Myers and W. Miller. Optimal alignments in linear space. *CABIOS*, 4(1):11-17, 1988.
4. G. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239-249, 1986.
5. W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci USA*, 85:2444–2448, 1988.
6. E. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251-266, 1986.
7. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215(3):403–410, 1990.
8. E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, 1994.
9. P. Weiner. Linear pattern matching algorithm. *14th Annual IEEE Symposium on Switching and Automata Theory*:111, 1973.
10. U. Manber and E. Myers. Suffix arrays: a new method for on-line searches. *Proc. 1st ACM-SIAM Symp. on Discrete Algorithms*:319-327, 1990.
11. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
12. H. Li, J. Ruan, and R. Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research*, 18(11):18511858, 2008.
13. P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. of MFCS'91*, volume 520 of *LNCS*, pages 240–248, 1991.
14. E. Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theor. Comput. Sci.*, 92(1):191-211, 1992.
15. D. Harel and R.E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Computing*, 13(2): 338-355, 1984.
16. L. Roberts. New chip may speed genome analysis. *Science*, 244:655-656, 1989.
17. G. Mealy. A Method for Synthesizing Sequential Circuits. *Bell Systems Technical Journal*, 34:10451079, 1955.
18. E. Moore. Gedanken-experiments on Sequential Machines. *Automata Studies, Annals of Mathematical Studies* Princeton University Press, 34:129153, 1956.
19. S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, 25(17):3389–3402, 1997.
20. `http://blast.ncbi.nlm.nih.gov/Blast.cgi`.
21. S. Karlin, S. Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proc. Natl. Acad. Sci. USA* 87:2264-2268, 1990.