An Alternative Approach to Alignments

Matching cDNA to Its Genomic Loci

Frank Lin, Prateek Tandon, Vineet Joshi
fjlin, ptandon1, vineetj
02-714

Abstract—**We investigate the problem of matching cDNA to its genomic loci through the use of approximate string matching in suffix trees. A current method for dealing with this problem is through the use of affine gap penalties and dynamic programming, which takes an average runtime of 3*O(mn). The goal of this project was to develop an efficient algorithm to deal with patterns containing large regions of deletions while maintaining a better average runtime than 3*O(mn). Through the use of the k-mismatch problem, dynamic programming, and suffix trees, we were able to achieve an overall better average runtime of ≤ O(mn) for the purpose of matching cDNA to its genomic loci.**

I. Introduction

Our work focuses on finding an alternative, more efficient method of dealing with alignments. More specifically, we focus on finding methods to deal with patterns containing large regions of deletions to solve the problem of matching cDNA to its genomic loci.

Complementary DNA (cDNA) is DNA that is synthesized from an mRNA through reverse transcription. It is often used in gene cloning, as probes, or in the creation of a cDNA library. cDNA doesn't contain the same sequences as the genomic DNA since most intronic regions and some exonic regions are differentially spliced from the precursor mRNA transcript.

Matching cDNA to its genomic loci when compared to a reference genome is a difficult task due to the alternative splicing of mRNA. Alternative splicing is a regulated process during gene expression in which multiple RNA transcripts can be generated from a single gene. The pre-mRNA is modified in that the introns are removed and exons are joined. Exons can be extended or skipped, or introns can be retained. This allows for the human genome to synthesize many more proteins than genes encoded.

When mRNA is spliced, and cDNA is synthesized, there are still insertions, deletions and substitutions that have to be dealt with in the regions that overlap with reference genome to get the ideal match. The problem within these regions makes use of the approximate string matching problem called k-differences problem, which basically deals with matching 2 strings which differ from each other by at most 'k' edit operations and each edit operation has a unit weight. Aside from biology, this problem has applications in the areas of spam filtering, spell checking, plagiarism detection, etc. The current day solutions solve this problem by either preprocessing the text string (approximate offline string matching) or not preprocessing the text (approximate online string matching). The most frequently used method for solving approximate online string matching employs the use of edit distance dynamic programing. Other methods exist that tend to achieve better theoretical runtime (Landau and Vishkin [7]) but when it comes to actual implementation, their preprocessing steps take O(kn) time rather than O(n) which are not the same when we are talking about genomes since the size of the text so big that constants cannot be ignored.

Our question, however, is a slight extension to this k-difference problem. We do not assume that the entire pattern can be found somewhere in the text as one contiguous fragment. Not only do we want to find if the entire pattern is similar to some overlapping text regions within 'k' edit operations, we also actively search for regions in the reference genome that piecewise overlap with the exons in cDNA, not penalizing for the vast stretch of deletions in the missing regions between two consecutive exons. Our method should not only use k-difference matching in the overlapping exonic regions, but also break off the matching when we pass beyond the exonic region in the text and then subsequently resume the matching algorithm when we again enter the next exonic region.

The most commonly used method for solving this problem is dynamic programming with affine gap penalties. This method has a worst case runtime of 3*O(mn). We use 3*O(mn) instead of using O(mn) to emphasize that it takes 3 times as long as the regular dynamic programming, which can be a tremendous increase in computations given the size of the genome (n).

There is one major feature where there is scope of improvement for runtime speedup. There are a lot of columns in the edit distance matrix that are identical. If we can somehow find out if a column is exactly similar to a previously processed column, then we can simply import its value to our new column. This would save us from repeating additional computations. Ukkonen applied dynamic programming over the suffix tree to solve the problem. He investigated this solution for speed up using suffix trees to pre-process the text in O(n) time [15]**.** Then, Ukkonen stored columns as they were processed, into the nodes of suffix trees. Theoretical results indicate it to have an average case runtime of $O(lm * min\,(n, m^{k+1}|\Sigma|^{k}) + n)$, which is better than the O(mn) dynamic programming algorithm.

Our goal here is to add on the basic concepts of the existing suffix tree method by Ukkonen to fill up the edit distance matrix along with a method that allows us to jump from one relevant exon to next in the text so that we effectively reduce the average runtime to below 3*O(mn). Although the worst case runtime of our method can still approach O(mn), but regarding the context of mapping cDNA to reference genome, where the text and the pattern is not the extreme cases (explained later), the algorithm is proven to be more efficient. Some areas of improvements are still open for exploration in our approach, nevertheless, the current results are appealing to be used in solving the problems at hand.

		a	a	a	a	a	a	a	a	b	b	b	b	b	b	b	b
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
b	2	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
b	3	2	2	2	2	2	2	2	2	1	0	1	1	1	1	1	1
b	4	3	3	3	3	3	3	3	3	2	1	0	1	1	1	1	1

Table 1. Let T = aaaaaaaabbbbbbbb, P = abbb, and k = 1. Assume the unit cost model of edit distance (cost = 1). The table D is shown. The highlighted, identical columns demonstrate that a simple dynamic programming solution for alignments has the drawback that it is explicitly repeated over identical substring of T.

II. Methods

A. Distance Function

1) Levenshstein distance/edit distance [6]

In this function, each edit operation has a unit cost function associated with it (This excludes substitution of a character by itself, which is considered an identity substitution with 0 cost). Thus insertions, deletions and substitutions are weighted equally. It follows the following criterion: For distance between 2 strings x,y: $d(x,y) \rightarrow 0 \leq d(x,y) \leq max(|x|,|y|)$. Literature has it that this edit distance function is used for solving the problem of k-differences.

2) Hamming distance [11]

The Hamming distance function is used as a metric for finding the minimum number of ‘substitutions’ that can transform a string ‘x’ to a string ‘y’. The definition makes it easier to see that it makes sense to use this distance function when we are comparing strings that are equal in length (symmetric hamming distance). Literature survey reveals that this distance function is used in finding solutions to the k-mismatches problem [2].

3) Longest Common Subsequence distance [4]

The name for this function is derived from the similarly named problem and that this function is used when solving for finding LCS between two strings. LCS distance is used to handle the cases of insertions and deletions with unit cost associated. Since insertions are allowed, the maximum length of the alignment possible in this case for 2 strings ‘x’ and ‘y’ is |x|+|y| thereby meaning that $0 \leq d(x,y) \leq |x| + |y|$.

In addition to these methods, there are a wide variety of distance functions that can be used. But, the mentioned methods are the most commonly used when considering the problem of approximate string matching.

B. Suffix Trees

The suffix tree data structure (Weiner [12]) is one of the most important data structures related to strings. A trie is an ordered tree that is based on some set of conditions on the keys of the tree. A suffix trie is one that contains all the possible suffixes of a particular string ‘s’ such that each character occupies a single node in the tree and each path in the data structure corresponds to a unique suffix of the string ‘s’. This is an excellent data structure to be used where runtime speedup is required but it comes with the drawback of space inefficiency. This is where a suffix tree comes in. A suffix tree is a form of a compressed suffix trie with very similar properties in that each path in the suffix tree still corresponds to a suffix of the original string.

For any given string ‘s’, the construction of suffix tree of s, is linear in terms of |s| [15]**.** For the problem at hand, we construct a more sophisticated variant of suffix trees augmented with suffix links that connect each node of the format “$x\alpha$” to a node containing “α”.

Suffix trees are utilized in a wide range of problems related to strings such as:

- String searching
- Finding the longest repeated substring
- Finding the longest common substring
- Finding the longest palindrome in a string

In the words of Dan Gusfield [10], “We know of no other single data structure that allows efficient solutions to such a wide range of complex string problems”

C. Ukkonen’s Algorithm

Ukkonen defines a term called the “viable k-approximate prefix” of P in T. It is the longest substring of T ending at index i of T that is acceptable match to some prefix of P. If two indices in T have the same viable prefixes then their columns must be identical. We store columns into the suffix tree such that a column with viable prefix Q is stored at a node that can be reached along the Q path from the root.

So, the number of columns evaluated by the method will be ≤ n and proportional to q (where q is the total number of different viable prefixes in T *and*

$$q \leq O\left(\min\left((i_2 - i_1), (m - j_{s-1})^{k_s+1}|\Sigma|^{k_s}\right)\right)$$

For a small k, q can be considerably smaller than n.

We can solve the approximate string matching problem using only entries D(i, j) ≤ k of D. These are referred to as essential entries i.e. D(i,j) is an essential entry if D(i,j) ≤ k.

Semi-global alignment is used as it does not penalize gaps in the beginning.

Each such column D(*, i) will be stored with state r_i=g(root,Q_i). States(nodes) are defined as $r_i = g(root, Q_i)$ and $s_i = g(r_{i-1}, t_i) = g(root, Q_{i'})$. Ukkonen proves that states that Q_i is always a suffix of $Q_{i'}$. This enables the traversal to go through states $r_0,s_1,\ldots,r_1,s_2,..,r_2,\ldots r_{n-1},s_n,\ldots,r_n$ which takes a total of 2*n number of suffix tree traversals. (n trie transitions and at most n suffix transitions). Consider the sub-path from r_{j-1} to r_j. The go to (trie) transition g(r_{j-1},t_j)=s_j is taken first. After that there are two cases:

Case 1: If s_j has already been visited during the traversal, then follow the suffix transition path until the first state r is encountered such that d(r) has non-empty value. Then r_j = r.

Case 2: If s_j has not been visited yet, then the algorithm then follows the suffix link path from s_j to the state r whose depth (distance from root) is |Q_j |. Then Then r_j=r and d(r) is calculated using dynamic programming and stored at state r.

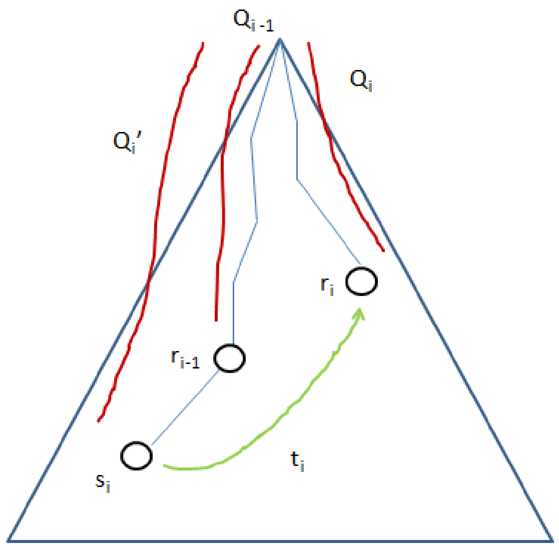

Figure 1. This is a basic suffix tree traversal in Ukkonen's proposed algorithm.

D. Problem Formulation

Let Σ *be the alphabet set such that* $|\Sigma|$ *denotes the size of the alphabet.*
Let $T \in \Sigma^* = t_1t_2 \ldots t_n$ *such that* $|T| = n$
Let $P \in \Sigma^* = p_1p_2 \ldots p_m$ *such that* $|P| = m$
Let $k \in R$ *be the number of differences we are willing to tolerate*
Let d: Σ^* x $\Sigma^* \rightarrow R$ *be the distance function*
Problem: Given T, P, k *and* d *return the set of all the text positions* j *such that there exists* P *aligns with* T *(including deletions in* T*) with a distance score* $\leq k$

E. Motivation

The motivation behind this problem came from the fact that we covered exact pattern matching and k-mismatch using suffix trees in the course. So, we wanted to expand upon what we learned and apply our knowledge to a more biological problem. One such problem that seemed interesting was how we could match cDNA sequences to reference genomes using suffix trees to improve speed-up. Current methods, using dynamic programming and affine gap penalties, require 3*O(mn) time. We wanted to find a method that has a better average time complexity and overall more efficient. We started looking for a method that would not penalize gaps occurring in the intron regions.

We started this project with attempts to come up with ways to directly match cDNA to the suffix tree of the genome. Earlier versions of our implementation involved walking down the suffix tree until we got a long enough match with little errors that we could call exon 1 and then jumping back to the root and start matching the remaining pattern. This, however, raised two problems. Firstly, we would not be able to get a linear match with respect to pattern index for the different exon matches we might find in the tree, which would result in a more complex assembly problem. Second and more importantly, as the human genome frequently has insertions, deletions and substitutions at the different point mutations, using k-mismatch in the suffix trees would not at all be a feasible approach as it would often miss the regions that might be affected by indels.

This made us revisit dynamic programming and look for implementations of the suffix trees that improved its time complexity and use that to come up with a way that could distinguish between intronic/differentially spliced regions.

For our case, we use the Levenshtein edit distance that has a cost 1 for each edit operation. A brute force approach to this would go through all the possible substrings of T and compute the edit distance with P and then pick the minimum out of those. This naive algorithm has a runtime complexity of $O(n^3m)$ [5].

A better runtime of O(mn) is possible using dynamic programming proposed by Sellers [16].

F. Our Approach

1) Definitions Revisited

- Ukkonen defines the viable k-approximate prefix of A in B as "the longest substring of B ending at some index "i" of B that is an acceptable match to some prefix of A.
- We denote a pattern as P = $p_1p_2p_3 \ldots p_m$ and a text as T = $t_1t_2t_3 \ldots t_n$.
- Q denotes any particular viable prefix, q represents the total number of unique viable prefixes in text T.
- We let 'k' be the number of differences we are willing to tolerate while matching the pattern against the text T.
- D is the edit distance matrix that we use.
- We call each entry in the edit distance matrix that has

	t_{i-3}	t_{i-2}	t_{i-1}	t_i	t_{i+1}	t_{i+2}	t_{i+3}	..
p_0	0	0	0	0	0	0	0	0
..	..	..	..	..	..	..	..	..
p_{j-2}	2	2	∞	1	1	∞	..	..
p_{j-1}	∞	3	2	∞	2	1	..	..
p_j	∞	∞	∞	<u>**3**</u> ←	∞	∞	..	..
p_{j+1}	∞	∞	∞	↑ ∞	↖ ∞	∞	..	..
p_{j+2}	∞	∞	∞	∞	∞	∞	..	..

Table 2. The table represents the edit distance matrix computed as our algorithm progresses. The non-essential entries have be changed to infinity.

a score ≤ k as an essential entry. Thus an entry D(i,j) is essential if and only if D(i,j) ≤ k.

- We call an entry in the edit distance table an effective essential entry if

$$D(i,j) \geq \lceil k * j/m \rceil.$$

We proceeded with the offline approximate string matching of the pattern P and text T. Ukkonen states that if two indices in text T have same viable prefixes, then their columns in the edit distance matrix must be identical. (See example in Table 1). Columns of the table were similarly stored in the suffix trees so that we could avoid evaluating the columns whose viable k-approximate prefix has been observed already. A column with viable prefix Q was stored with the state that could be reached along the Q path from the root. This changed the bound on the number of columns evaluated for each exon from 'x' (using dynamic programming) to ≤ 'x' and is equal to some 'q' If there are repetitive elements in the text, this bound helps in speeding up the algorithm to a considerable extent. Again, with a small number of differences allowed i.e. k not too large (which is the case in genomic context), we can expect 'q' << 'n'.

For an edit distance alignment between two strings A and B with k differences, we know that if an alignment of length L has a score of S such that $S \leq k$, then an extension L' = L+Z is a considerable alignment if and only if the extension to the alignment Z has an edit distance $\leq k - S$. Thus, if we already paid for all the k-differences in the alignment L, then we stop with further alignment as soon as we observe another difference in the extension between A and B as that will not be part of any match that can be considered valid. Also, each entry that exists in the edit distance matrix that trace backs to alignment L is an essential entry.

This aspect about edit distance gives a scope of further speed-up in the proposed algorithm. We mark all the 'non-essential entries' in the table i.e. with score > k such that they are not considered while evaluation of the next column, i.e only essential entries give rise to further extension of the matrix. (In practice this can be done by replacing the values with a very high number). We replace the inessential entries of D with ∞ as this will not affect contents of the essential part. (Table 2)

We again use semi-global alignment such that we do not penalize for gaps in the starting. Intuitively too, this makes sense to use since this way, once we break from the current existing alignment, we have to align the remaining pattern from the following index to anywhere in the remaining text, not caring for mismatches/gaps that are in the start of the new alignment. This part of the algorithm helps us break off the matching when we pass beyond the exonic region in the text and then subsequently resume the matching algorithm when we again enter the next exonic region. This way it prevents penalty for the long regions of intron/deletions.

2) Algorithmic Details

We start by filling in the columns starting with t_1, using the Ukkonen method involving suffix tree for T, only evaluating for columns that have a unique Q allowing for a maximum of k-mismatches in the regions that overlap between cDNA and the reference genome.

To ensure that the alignment obtained is significant and not due to random chance, we set a threshold length "L" and continue extending the alignment till it becomes ≥ L. On average there are 8.8 exons and 7.8 introns per gene [13]. The length of the average intron is > 1kb while the length of the average exon is 150bp. Due to this fact; we set the threshold "L" to 100 base pairs. This value can be adjusted accordingly to the data set being viewed, but for our purpose L = 100bp will suffice. Since the size of the human genome is around 3 X 10^9 bp, the probability of seeing 100bp align by chance is statistically unlikely. Therefore, "L" could be said to bear some statistical significance with respect to the problem at hand.

A key assumption that we make is that it is not very likely that the sequencing errors and SNPs that are the primary source of differences, will occur closely. We assume that these errors and mutations will be spread uniformly across the entire length of the pattern. The sequencing errors are random errors that happen by chance during the sequencing process that are equally likely to happen across the total length of the reads. SNPs are also randomly distributed across the genome, about 1 in every 600 base pairs so this assumption should suffice in most cases [9].

With reference to Table 2, let j_i be the highest index in column i that has an "effective essential entry". Similarly, let j_{i+1} be the highest index in column i+1 that has an "effective essential entry".

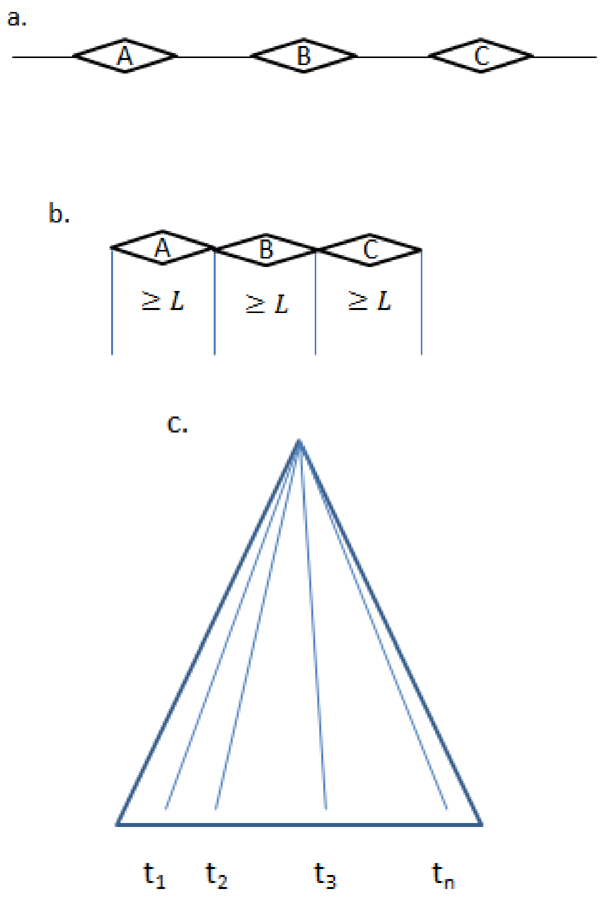

Figure 2. (a,b,c) a is the original genome sequence, b is the cDNA we wish to search, and c is the suffix tree generated for the genome sequence. Note that all b has a length greater than threshold L.

We decide to break with the current matching at column i if $j_i > j_{i+1}$ and also the current matching length "j" in P has exceeded the threshold "L". This is because there is no way we can further extend this matching and maintain it within the effective essential cut-offs as shown in Table 2, in cell (j,i). Since the matching is beyond threshold length 'L', we can be confident that there is some significance behind this matching. (Details given in the supplementary paper).

We continue matching the remaining P(=P') with the remaining T(=T'). So this problem becomes an equivalent problem with k' mismatches between text T' = $t_{i+1}t_{i+2}\ldots t_n$ and pattern P' = $p_{j+1}p_{j+2}\ldots p_m$. Since |P'| < |P| and |T'| < |T| thus the |D| = |P'|*|T'| < |P|*|T|. Thus, for two consecutive iterations 'r' and 'r+1' we have the constraint that the number of columns evaluated in the 'rth' iteration ≤ the number of columns evaluated in the (r+1)th iteration. Here, an iteration corresponds to occurrence of a gene.

In general, after processing the sth exon, we check if the node_length = $(m-j_{s-1})$ then we copy to the edit distance matrix, else we compute it and store it. This can be done relatively easily since we already keep track of the indices at the leaves in Suffix tree (T).

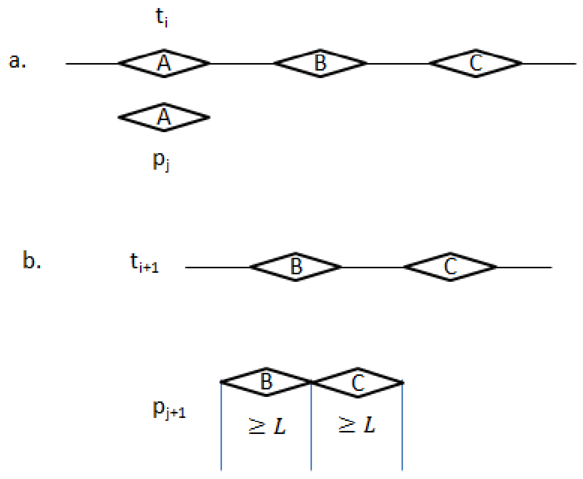

Figure 3. (a,b) a is the original genome sequence matched with the first exon of the cDNA. Now, after A is mapped, the problem is reduced to b. Note that A is no longer considered since it is already mapped. We repeat the procedure for the reaming exons.

We stop if all of pattern P is matched to some location in T and report all such matches. Otherwise, we report all the partial matches if P doesn't match completely by the time we go over the entire text T.

III. RESULTS AND DISCUSSION

The current runtime for the dynamic programming algorithm with affine gap penalties is 3*O(mn). Ukkonen's proposed algorithm has a runtime of O(mq+n). In our implementation, the average case has a runtime ≤ O(mq+n) ≤ O(mn). We are able to achieve this runtime because our approach requires a lot less number of cells to be filled in the edit distance matrix than regular dynamic programming. This, further aided with Ukkonen's method of filling in edit-distance columns reduced the overall total number computations to less than m*n, which results in a runtime, say O(X) which is less than O(mn) for regular edit distance programming ≤ 3*O(mn), with 3 matrices for affine gaps. (A stepwise detailed proof is given in supplementary paper).

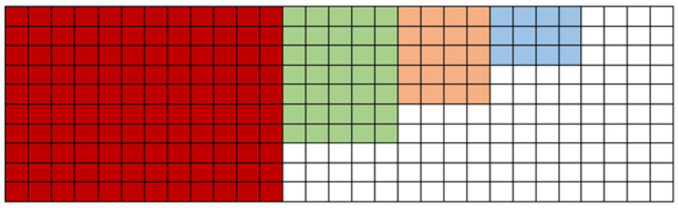

Figure 4. A visualization of the decrease in the number of edit distance matrix columns evaluated with each exon match. The space required decreases significantly from the standard dynamic programming approach.

We have proposed a new algorithm for matching strings with k-differences which gives a better average case runtime than currently known algorithms especially when k is very small and we use a NGS sequencing method where sequencing errors are ~1%. The time bounds are

$$O(m \times i_1) + O\left(\sum_{s=2}^{l} (m - j_{s-1}) \times (i_s - i_{s-1})\right)$$

(*Discussed further in supplementary paper)

This method could with a considerable amount of modifications, be used in the related LCS problem and other related problems solved using simple dynamic programming techniques. Future directions for this algorithm include the investigation of further practical applications of the techniques described to other similar problems, as well as generalizing the results to cover additional edit operations such as swaps..

In addition to matching cDNA to its genomic loci, our implementation can also be used to solve any problems in which there are large deletions in the pattern. Some of these examples include

- Detecting sequence deletions in a mutated gene, when compared to a reference genome,
- Detecting sites of gene fusions.
- Finding locations of concatenation in texts.

IV. FUTURE DIRECTIONS

Though we were able to achieve an average complexity of O(nm), there are still some shortcomings of our project that we can improve on. First, our implementation can only report the best match. If there are other possible matches, our algorithm would not report such cases. Also, in terms of cDNA matching, our implementation might not be as effective on smaller exons. This would require the user to adjust the "L" value depending on their knowledge of the sequences being used.

It is also possible to improve our runtime by getting rid of the dependency on n, the length of the text. Ukkonen suggests the use of more complicated data structures, such as dictionaries, stacks and balanced searched trees, that could serve this purpose.

V. CONCLUSION

Through our greedy implementation of suffix trees and dynamic programming, we were able to demonstrate a method of mapping cDNA to its genomic loci is O(mn) time while taking up less space than traditional methods.

VI. REFERENCES

[1] Chang, Lawler, "Sublinear approximate string matching and biological applications", Algorithmica, vol 12, pp. 327-344, Nov. 1994

[2] Galil, Z., and Giancarlo, R. Improved string matching with k mismatches. SIGACT News 17 (1986), 52-54

[3] Ukkonen, "Approximate String-Matching over Suffix Trees", Proc. CPM 93. Lecture Notes in Computer Science 684, pp. 228-242, Springer 1993

[4] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. "Indexing methods for approximate string matching" IEEE Data Engineering Bulletin, 24(4):19--27, 2001.

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

[6] Levenshtein V, "Binary codes capable of correcting spurious insertions and deletions of ones.", Probl Inf Transmission, 1965, 1:8-17.

[7] Gad M. Landau, Uzi Vishkin, "Fast string matching with k differences, Journal of Computer and System Sciences", Volume 37, Issue 1, August 1988, Pages 63-78, ISSN 0022-0000,

[8] Zvi Galil, Kunsoo Park, "An improved algorithm for approximate string matching

[9] Schmitt et al, "Detection of ultra-rare mutations by next-generation sequencing" PNAS 2012 109 (36) 14508-14513.

[10] Gusfield, D. "Algorithms on Strings, Trees and Sequences" Cambridge Univ. Press, Cambridge 1997.

[11] Hamming, Richard W. (1950), "Error detecting and error correcting codes", Bell System Technical Journal 29 (2): 147–160, MR 0035935

[12] Weiner, P. (1973), "Linear pattern matching algorithms", 14th Annual IEEE Symposium on Switching and Automata Theory, pp. 1–11, doi:10.1109/SWAT.1973.13

[13] Sakharkar MK, Chow VT, Kangueane P. "Distributions of exons and introns in the human genome". In Silico Biol. 2004;4(4):387-93.

[14] Hong X, Scofield DG, Lynch M "Intron size, abundance, and distribution within untranslated regions of genes." Mol. Biol. Evol. 23:2392-404.

[15] E. Ukkonen. "On-line construction of suffix trees." Algorithmica, 14(3):249-260, 1995.

[16] Sellers PH "On the theory and computation of evolutionary distances". SIAM Journal on Applied Mathematics 26 (4): 787–793. doi:10.1137/0126070