# An Alternative Approach to Alignments

**Vineet Joshi, Prateek Tandon, Frank Lin**

# Project Goal

To find an alternative, more efficient method to deal with patterns containing long regions of deletions.

# Problems Solved

- Matching cDNA to its genomic loci (matching exons to genomic loci while ignoring introns).
- Detecting sequence deletions in a mutated gene, when compared to a reference genome.

# Motivation

- In this course, we covered exact pattern matching using suffix trees.
  - We wanted to expand upon what we learned and apply our knowledge to a more biological problem.
- Ukkonen's paper "Approximate String-Matching Over Suffix Trees".
  - Try to use suffix trees to solve this problem.

# Motivation

- Current methods, using dynamic programming and affine gap penalties, require **O(mn)** time.
  - We wanted to find a method that has a better average time complexity.
- We wanted to look for a method that doesn't penalize the gaps occurring in the intron regions.
  - To better estimate of the overall alignments.

# Motivation

- The simple dynamic programming solution for alignments has the drawback that it is explicitly repeated over identical repeated substrings of T.

**Example.** Let $T = \texttt{aaaaaaaabbbbbbbb}$, $P = \texttt{abbb}$, and $k = 1$. Assume the unit cost model of the edit distance (each edit operation has cost $= 1$). Then table $D$ is

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

# Approach

We used a combination of theories from:

- K-difference problem
- Suffix trees (with suffix links)
- Dynamic programming
- Ukkonen's algorithm for K differences

# Overview of Ukkonen's Algorithm

- The number of columns evaluated by the method is $\leq$ n and proportional to q (where q is the total number of different viable prefixes in T).

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

- **Essential entries :**
  - The approximate string matching problem can be solved using only entries $D(i, j) \leq k$ of D.
  - Call each entry $D(i,j) \leq k$ an essential entry.
  - Replace the inessential entries of D with $\infty$ as this will not affect contents of the essential part.

- Ukkonen uses Semi-global alignment, which does not penalize gaps in the beginning.

# Our Implementation

- We start 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.

- We allow for k-mismatches in the regions that overlap between cDNA and gene.

- We will extend the matching beyond a certain threshold length “L” that is long enough to ensure that the current matches are not occurring by chance alone.

# A Greedy Approach

- We define the **effective essential entry** as

$$D(j,i) \leq \left\lceil k * \frac{j}{m} \right\rceil$$

- **Assumption:** It is not very likely that the sequencing errors/mutations will occur closely but will be spread across uniformly.

- 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".

- We decide to break with the current matching at column $i$ if $j_i > j_{i+1}$ and also the matching length "j" in P has exceeded the threshold "L".

| | ti-3 | ti-2 | ti-1 | ti | ti+1 | ti+2 | ti+3 | .. |
|---|---|---|---|---|---|---|---|---|
| p0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| .. | .. | .. | .. | .. | .. | .. | .. | .. |
| pj-2 | 2 | 2 | ∞ | 1 | 1 | ∞ | .. | .. |
| pj-1 | ∞ | 3 | 2 | ∞ | 2 | 1 | .. | .. |
| pj | ∞ | ∞ | ∞ | **3** | ∞ | ∞ | .. | .. |
| pj+1 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | .. | .. |
| pj+2 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | .. | .. |

Also, j >= ’L’

# A Greedy Approach

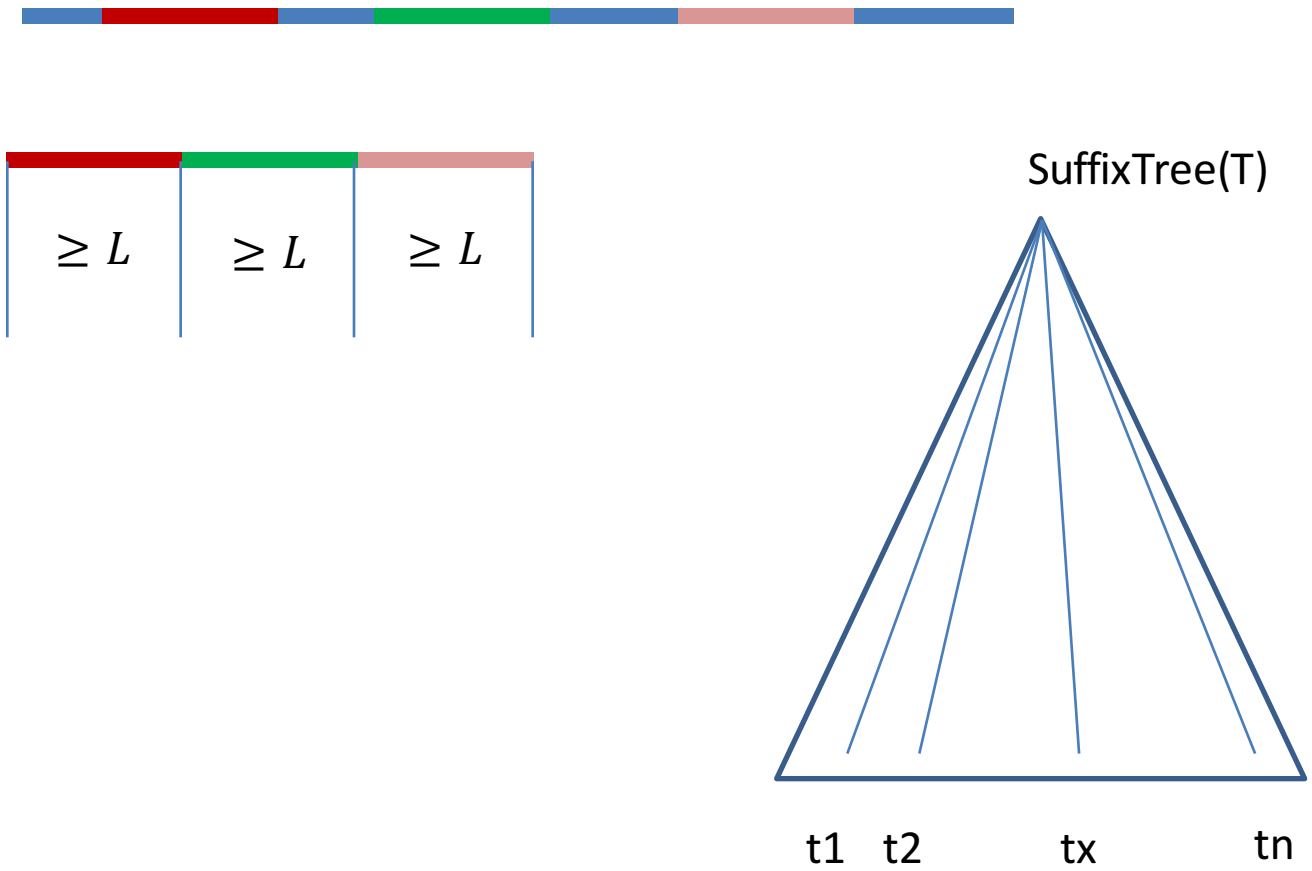
- We continue matching the remaining P with the remaining T and we will now have shorter columns in our table.

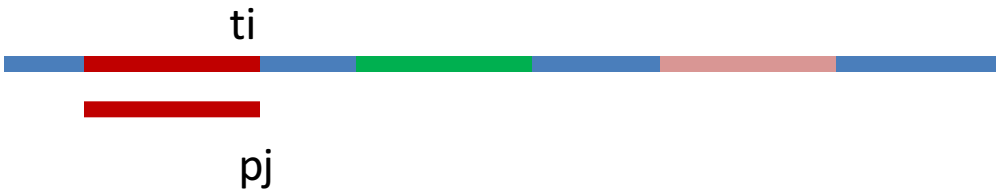
- So this problem becomes an equivalent problem with k' mismatches between text T' = $t_{i+1}$, $t_{i+2}$, …, $t_n$ and pattern P' = $p_{j+1}$, $p_{j+2}$, …, $p_m$

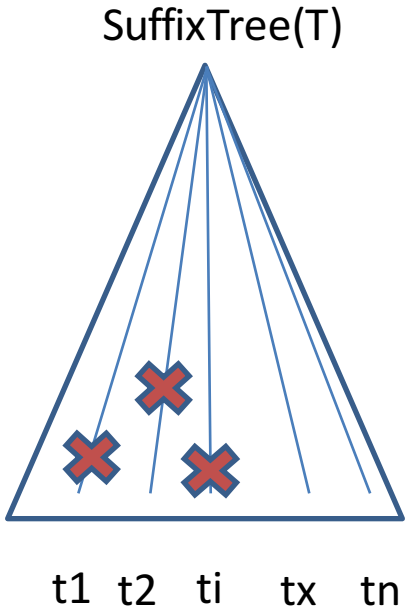
- In general, after processing the $s^{th}$ exon, we check if the node_length = $(m-j_{s-1})$ then we copy to the edit distance matrix, else we compute it and store it.

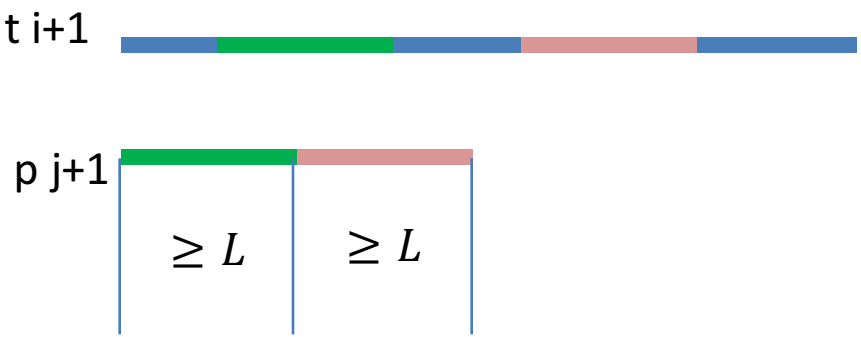
- This can be done relatively easily since we already keep track of the indices at the leaves in Suffix tree (T).

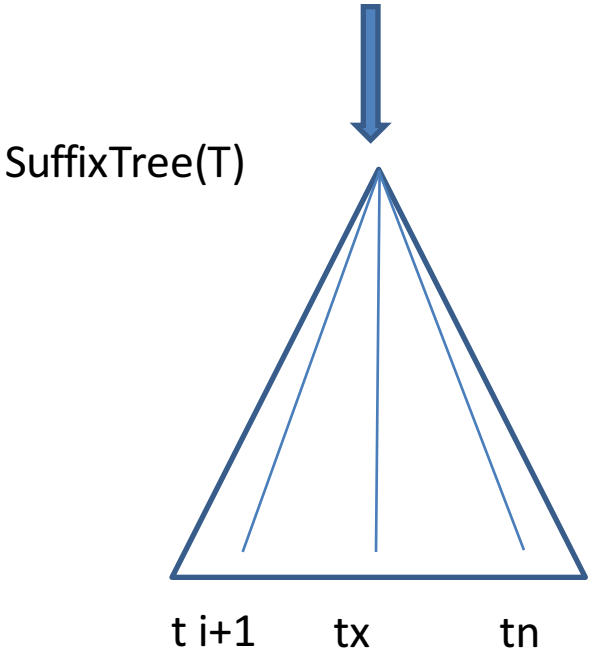
# Example




ti
pj


SuffixTree(T)
t1
t2
ti
tx
tn


t i+1
p j+1
$\geq L$
$\geq L$


SuffixTree(T)
t i+1
tx
tn

# A Greedy Approach

- 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.

# Runtime Analysis

- The basic Ukkonen method runs in time $O(mq + n)$
- This method will take time

$$O(mq + n) + O(m'q' + n') + O(m''q'' + n'') + \cdots$$

$\leq l * O(mq + n)$, where $l$ is the number of exons in the cDNA.

- Also,

$$q \leq \min\left(n, \frac{12}{5}(m+1)^{k+1}(|\Sigma|+1)^k\right)$$

$$q \leq \mathrm{O}(\min\left(n, m^{k+1}|\Sigma|^k\right))$$

- Therefore, this method will have a runtime of

$$\leq \mathrm{O}(l * m * \min\left(n, m^{k+1}|\Sigma|^k\right) + n)$$

- This will have a worst case running time of $\mathbf{O}(\boldsymbol{lmn})$ but will have an average case runtime which will be always be better than $\mathbf{O}(\boldsymbol{lmn})$.

# Further Runtime Improvements

- Ukkonen suggests the use of more complicated data structures to get rid of the dependency on n.
  - Possibly dictionaries and stacks
  - Possibly balanced search trees

- For example, the runtime can be reduced to

$$O(mq * logq + output\ size)$$

- Which would also reduce our runtime to

$$l * O(mq * logq + output\ size)$$

# Plausible?

- Average length of exons is around **150** $\boldsymbol{bp}$
- We set a threshold "L" of **100** $\boldsymbol{bp}$
- Size of Human genome $\sim \mathbf{3\times10^9}\ \boldsymbol{bp}$
- Therefore, the probability of seeing 100 $bp$ length aligning by chance is $\sim \frac{3\times10^9}{4^{100}}$

$\sim$ **1.8669046e-51**

# Plausible?

- **On average, there are 8.8 exons and 7.8 introns per gene**
  - In Silico Biol. 2004;4(4):387-93.
- **Distributions of exons and introns in the human genome.**
  - Sakharkar MK, Chow VT, Kangueane P.
- **Average intron length is > 1 kb and**
- **Average exon length is only 150 bp**
  - http://people.ibest.uidaho.edu/~bree/courses/17_EBR_ME_genome.pdf

# Pros and Cons of Our Implementation

<u>Pros</u>

- Average time complexity < O(mn)
- Overall less space required
- Can align cDNA to genomic loci

<u>Cons</u>

- Current implementation can not give multiple matches (only first best match)
- Might not be efficient for very small exon lengths
  - Possibly change “L”
- Further improvements needed

Questions?

# More On Ukkonen's Algorithm

- Length $L(i,j)$of the shortest substring of T ending at $t_j$ whose edit distance from $P_i$ equals $\mathrm{P}(i,j)$

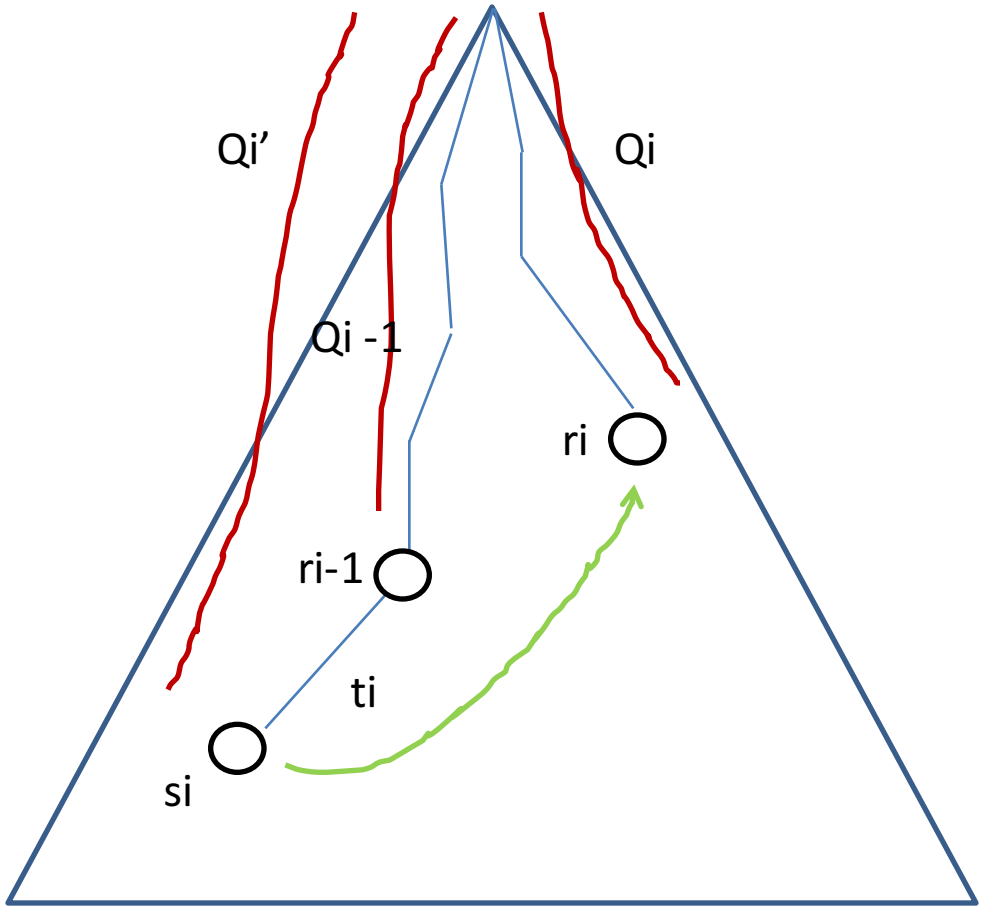
- Each such column D(*, i) together with column L (*, i) will be stored with state $r_i = g(root, Q_i)$ as $d(r_i) = D(*, i)$ and $l(r_i) = L(*, i)$

- $r_i = g(root, Q_i)$ and $s_i = g(r_{i-1}, t_i) = g(root, Q_i{}')$

- **Theorm : $\boldsymbol{Q_i}$ is always a suffix of $\boldsymbol{Q_i}'$**

- The traversal goes through states $r_0, s_1, \dots, r_1, s_2, \dots., r_2, \dots r_{n-1}, s_n, \dots, r_n$

# More On Ukkonen's Algorithm



# More On Ukkonen's Algorithm

- Consider the sub-path from $r_{j-1}$ to $r_j$. The go to 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) and l(r) have non-empty values. Then $r_j = r$.

# More On Ukkonen's Algorithm

**Case 2:**

- If $s_j$ has not been visited yet, then evaluate a pair $(d, l)$ of columns as
- $(d, l) = dp\big(d(r_{j-1}), l(r_{j-1}), t_j\big)$
- This gives $(d, l) = (D(*, j), L(*, j))$
- Where $|Q_j|$ is the maximum entry in column $l$
- The algorithm then follows the suffix link path from $s_j$ to the state $r$ whose depth (distance from root) is $|Q_j|$.
- Then $r_j = r$ and the algorithm saves columns $(d, l)$ as $d\left(\mathrm{r_j}\right) = d$ and $l\left(\mathrm{r_j}\right) = l$