A *De Nuevo* Algorithm for Genome Wide Searching of LTR Retrotransposon

Jeong-Hyeon Choi, Mina Rho, Sun Kim, and Haxiu Tang

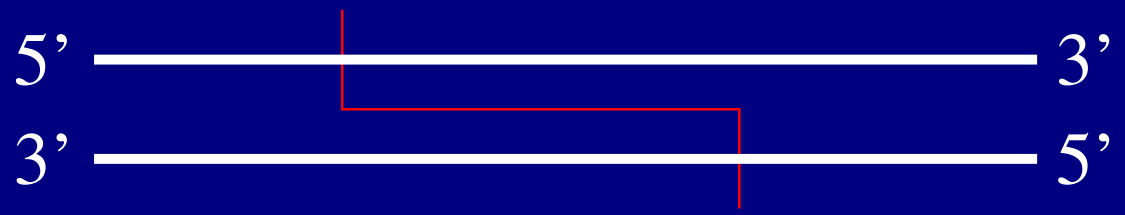
School of Informatics,
Indiana University, USA
jeochoi@indiana.edu

Outline

The Structure of LTR Retrotransposon

- LTR retrotransposon = LTR – inter-portion – LTR
- LTR: length 100 $\sim$ 5,000 bps, $\geq$80% identity
- inter-portion = *gag* – *pol* – *env*, length 1,000 $\sim$ 40,000 bps

The Structure of LTR Retrotransposon

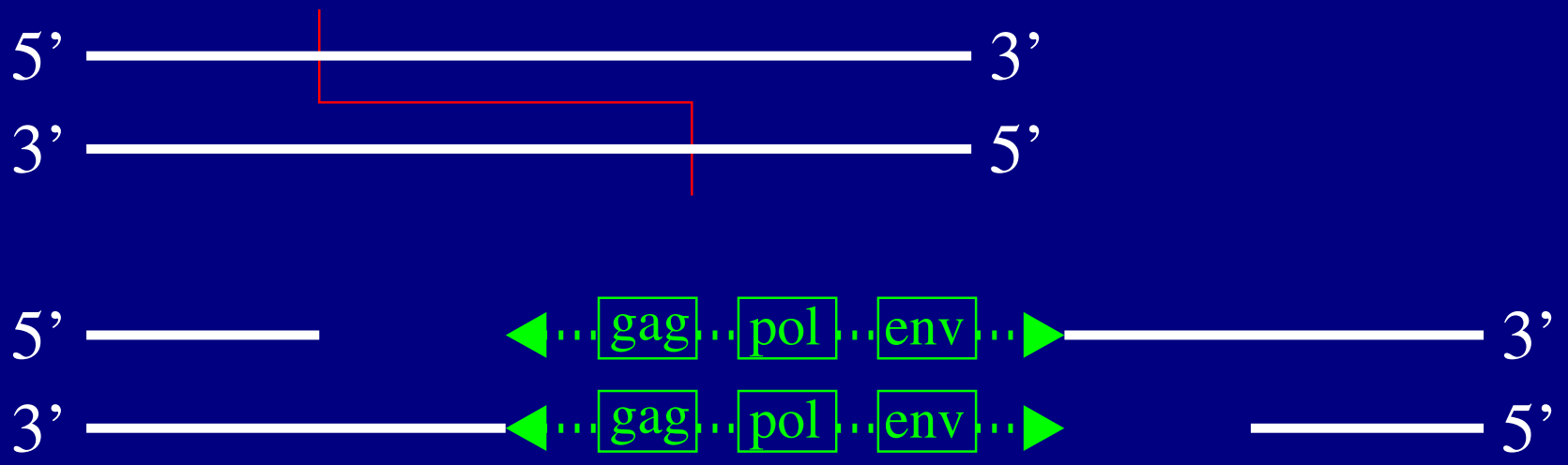


- LTR retrotransposon = LTR – inter-portion – LTR
- LTR: length 100 $\sim$ 5,000 bps, $\geq$80% identity
- inter-portion = *gag* – *pol* – *env*, length 1,000 $\sim$ 40,000 bps

The Structure of LTR Retrotransposon

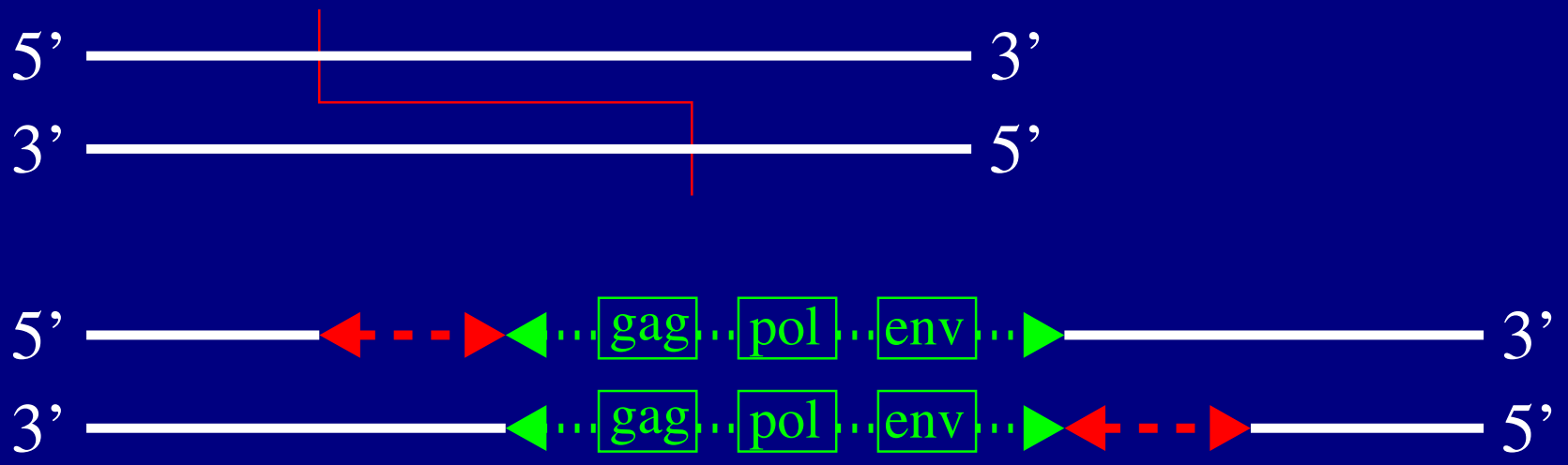
- LTR retrotransposon = LTR – inter-portion – LTR
- LTR: length 100 $\sim$ 5,000 bps, $\geq$80% identity
- inter-portion = *gag* – *pol* – *env*, length 1,000 $\sim$ 40,000 bps

The Structure of LTR Retrotransposon



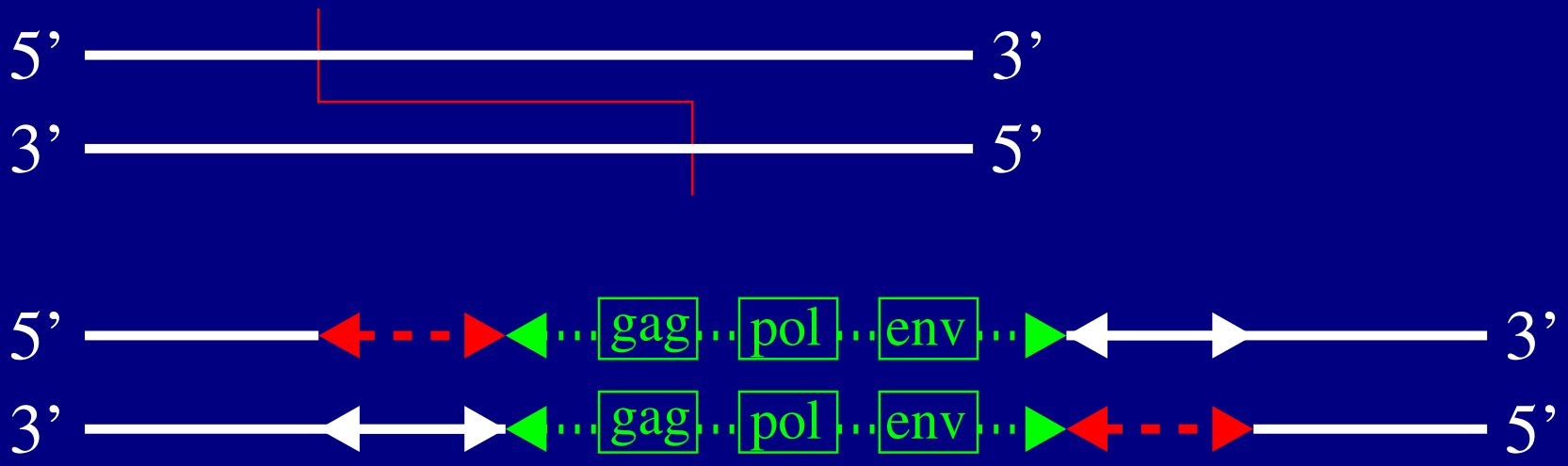
- LTR retrotransposon = LTR – inter-portion – LTR
- LTR: length 100 $\sim$ 5,000 bps, $\geq$80% identity
- inter-portion = *gag* – *pol* – *env*, length 1,000 $\sim$ 40,000 bps

The Structure of LTR Retrotransposon



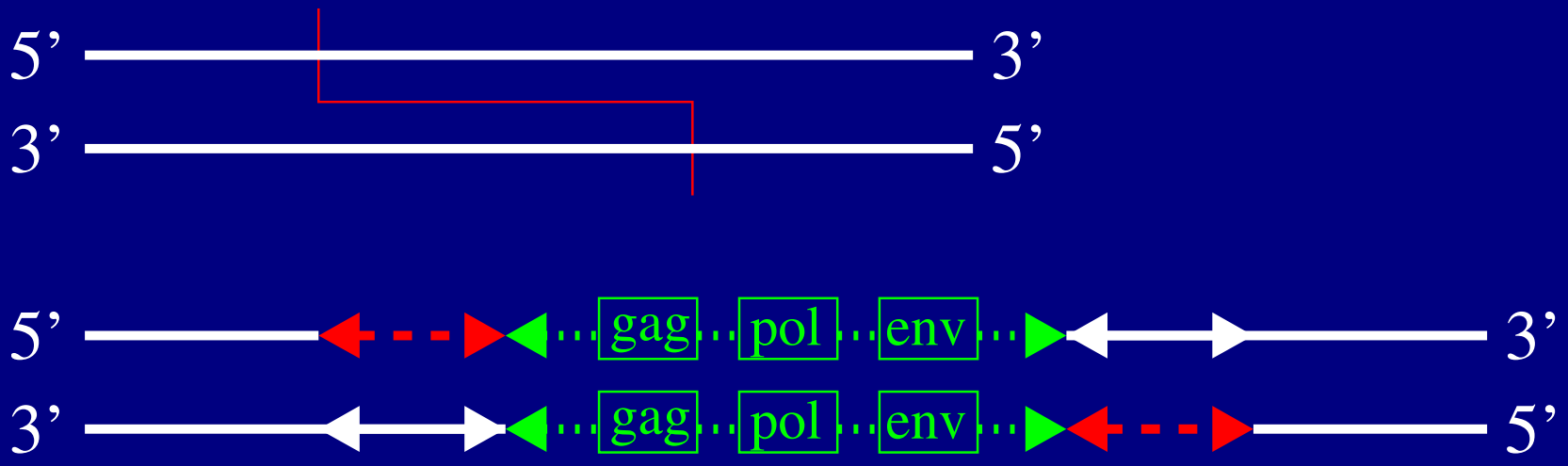
- LTR retrotransposon = LTR – inter-portion – LTR
- LTR: length 100 $\sim$ 5,000 bps, $\geq$80% identity
- inter-portion = *gag* – *pol* – *env*, length 1,000 $\sim$ 40,000 bps

The Structure of LTR Retrotransposon



- LTR retrotransposon = LTR – inter-portion – LTR
- LTR: length 100 $\sim$ 5,000 bps, $\geq$80% identity
- inter-portion = *gag* – *pol* – *env*, length 1,000 $\sim$ 40,000 bps

The Structure of LTR Retrotransposon



- LTR retrotransposon = LTR – inter-portion – LTR
- LTR: length 100 $\sim$ 5,000 bps, $\geq$80% identity
- inter-portion = *gag* – *pol* – *env*, length 1,000 $\sim$ 40,000 bps

Strategy for Finding LTR Retrotransposon

1. Finding all pairs of LTRs.
   - searching maximal exact pairs using suffix array and discarding the maximal exact pair such that substrings don't locate within the length of inter-portion
   - combining maximal exact pairs where the left substrings and right substrings of each maximal exact pair are simultaneously within the length of LTR
2. Verifying inter-portion of each pair of LTRs.
   - domain search using `Pfam` + `hmmsearch`.

LTR — *gag* — *pol* — *env* — LTR

Strategy for Finding LTR Retrotransposon

1. Finding all pairs of LTRs.
   - searching maximal exact pairs using suffix array and discarding the maximal exact pair such that substrings don't locate within the length of inter-portion
   - combining maximal exact pairs where the left substrings and right substrings of each maximal exact pair are simultaneously within the length of LTR
2. Verifying inter-portion of each pair of LTRs.
   - domain search using `Pfam` + `hmmsearch`.

LTR — *gag* — *pol* — *env* — LTR

Strategy for Finding LTR Retrotransposon

1. Finding all pairs of LTRs.
   - searching maximal exact pairs using suffix array and discarding the maximal exact pair such that substrings don't locate within the length of inter-portion
   - combining maximal exact pairs where the left substrings and right substrings of each maximal exact pair are simultaneously within the length of LTR
2. Verifying inter-portion of each pair of LTRs.
   - domain search using `Pfam` + `hmmsearch`.

LTR — *gag* — *pol* — *env* — LTR

Strategy for Finding LTR Retrotransposon

1. Finding all pairs of LTRs.
   - searching maximal exact pairs using suffix array and discarding the maximal exact pair such that substrings don't locate within the length of inter-portion
   - combining maximal exact pairs where the left substrings and right substrings of each maximal exact pair are simultaneously within the length of LTR
2. Verifying inter-portion of each pair of LTRs.
   - domain search using `Pfam` + `hmmsearch`.

LTR — *gag* — *pol* — *env* — LTR

Maximal Exact Pair

Definition

A maximal exact pair (MEP) in a sequence S, denoted by (p_1, p_2, ℓ) is a pair of identical substrings α and β in S such that the character to the immediate left (right) of α is different than the character to the immediate left (right) of β.

Example

TACTATCACTCATGCTA

1234567890123456 7

For the above sequence, MEPs of length 3 or more are

ACT: (2, 8) AT: (5, 12)

CTA: (3, 15) CT: (9, 15)

TA: (1, 4), (1, 16)

Maximal Exact Pair

Definition

A maximal exact pair (MEP) in a sequence S, denoted by (p_1, p_2, ℓ) is a pair of identical substrings α and β in S such that the character to the immediate left (right) of α is different than the character to the immediate left (right) of β.

Example

TACTATCACTCATGCTA

1234567890123456 7

For the above sequence, MEPs of length 3 or more are

ACT: (2, 8) AT: (5, 12)

CTA: (3, 15) CT: (9, 15)

TA: (1, 4), (1, 16)

Text Indexing (Pattern Retrieval)

Idea

- Preprocess the text : $O(m)$ time
- Searching : $O(n + occ)$ time

Index data structures

Exact match

- Suffix Trie
- Suffix Tree
- Suffix Array
- String B-tree

Approximate match

- Suffix Tree
- Suffix Array
- Q-grams
- Q-samples

Suffix

Definition

Suffix S_i is a substring of S that starts at position i and ends at position $|S|$.

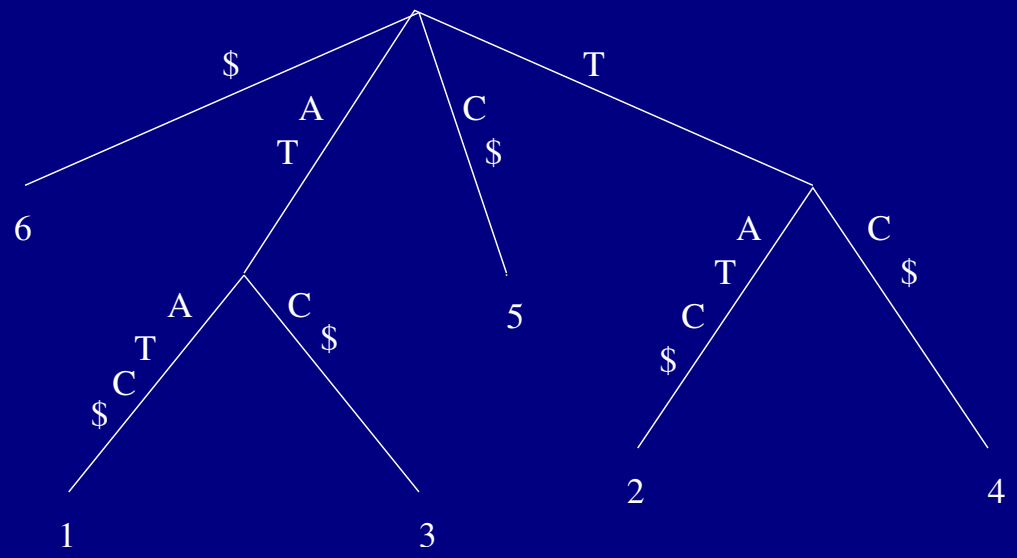
Example

Given $S = \texttt{ATATC}$,

$$
\begin{aligned}
S_1 &= \texttt{ATATC\$} \\
S_2 &= \texttt{TATC\$} \\
S_3 &= \texttt{ATC\$} \\
S_4 &= \texttt{TC\$} \\
S_5 &= \texttt{C\$} \\
S_6 &= \texttt{\$}
\end{aligned}
$$

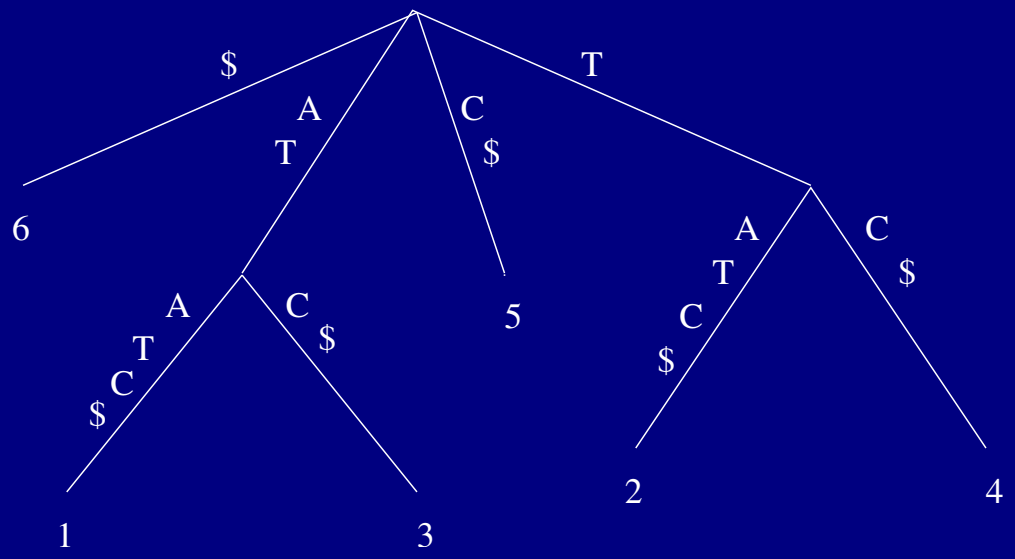
Suffix Tree

The compacted trie of all suffixes of a string, e.g., `ATATC`



Suffix Tree

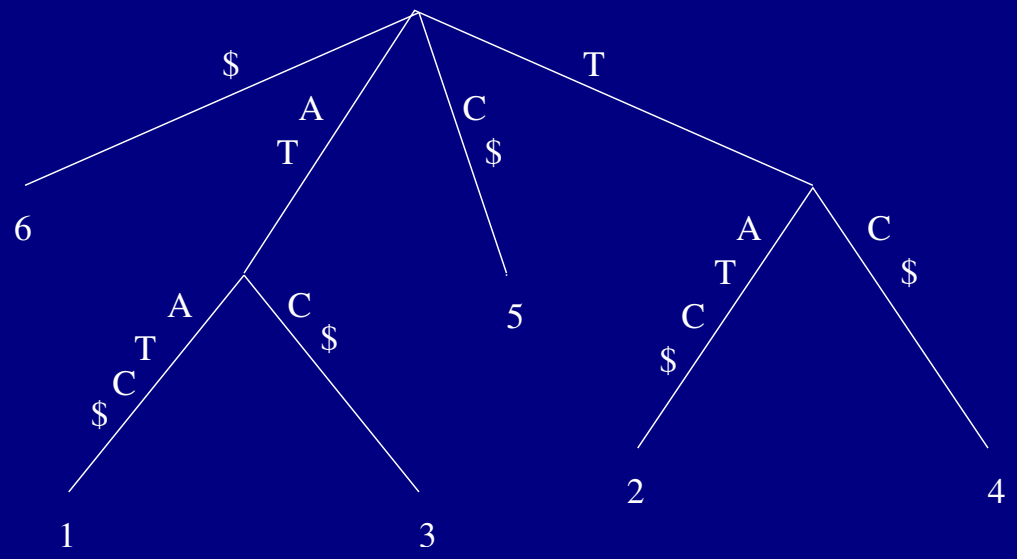
The compacted trie of all suffixes of a string, e.g., `ATATC`



- Each leaf node is numbered to suffix number.

Suffix Tree

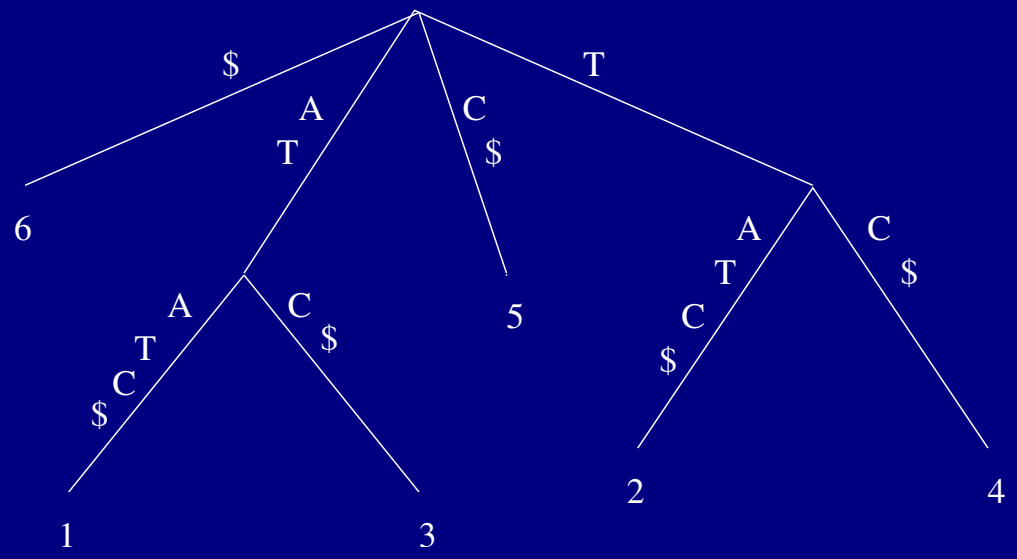
The compacted trie of all suffixes of a string, e.g., `ATATC`



- The concatenation of the edge-labels on the path from the root to leaf i exactly spell out the suffix of S that starts at position i.

Suffix Tree

The compacted trie of all suffixes of a string, e.g., `ATATC`



- Each internal node except the root has at least two children and each edge is labeled with a nonempty substring of S.

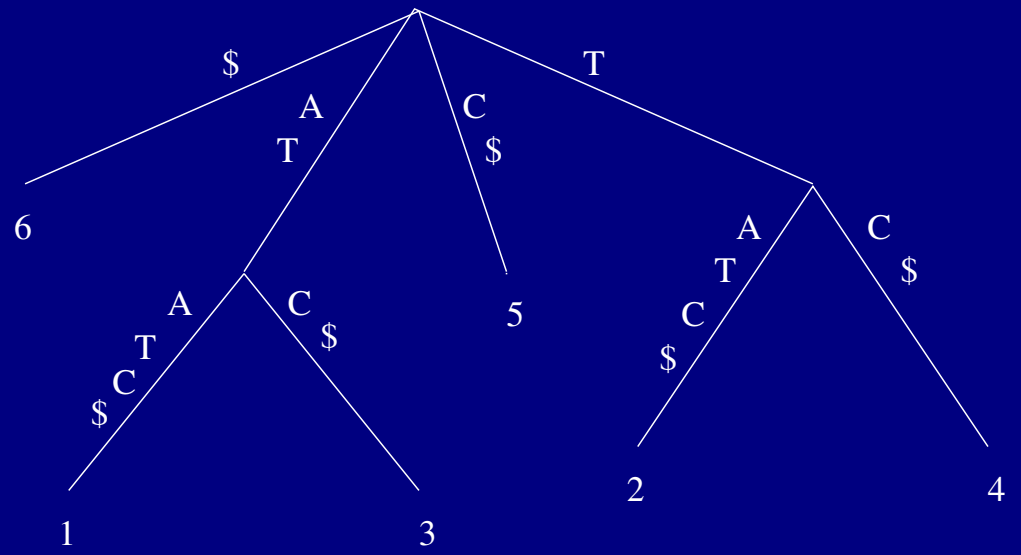
Pattern Retrieval Problem

Definition

Given a pattern P of length n and a text T of length m, find the positions of all occurrences of P in T

Example

$T =$ `ATATC`



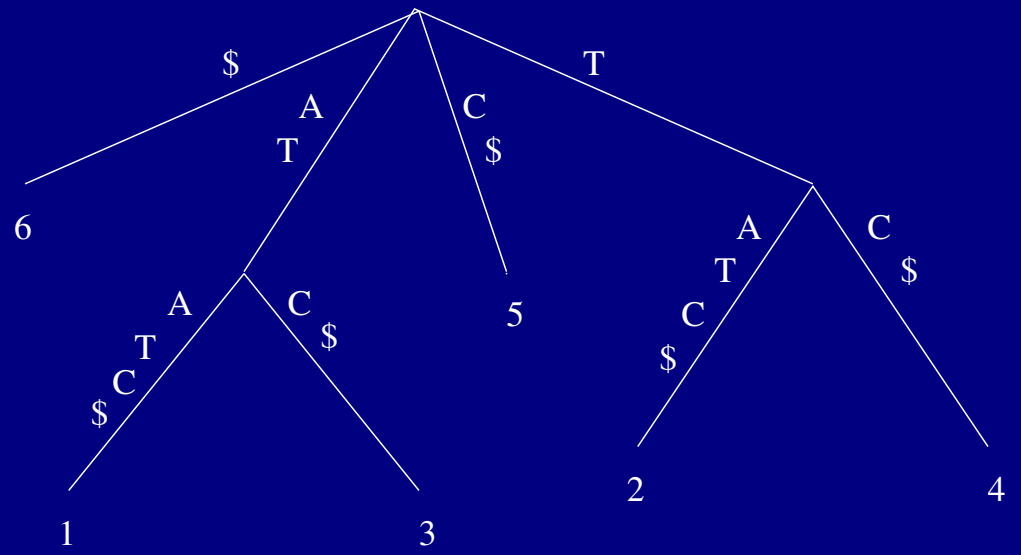
Pattern Retrieval Problem

Definition

Given a pattern P of length n and a text T of length m, find the positions of all occurrences of P in T

Example

$T =$ `ATATC`
$P =$ `ATA`



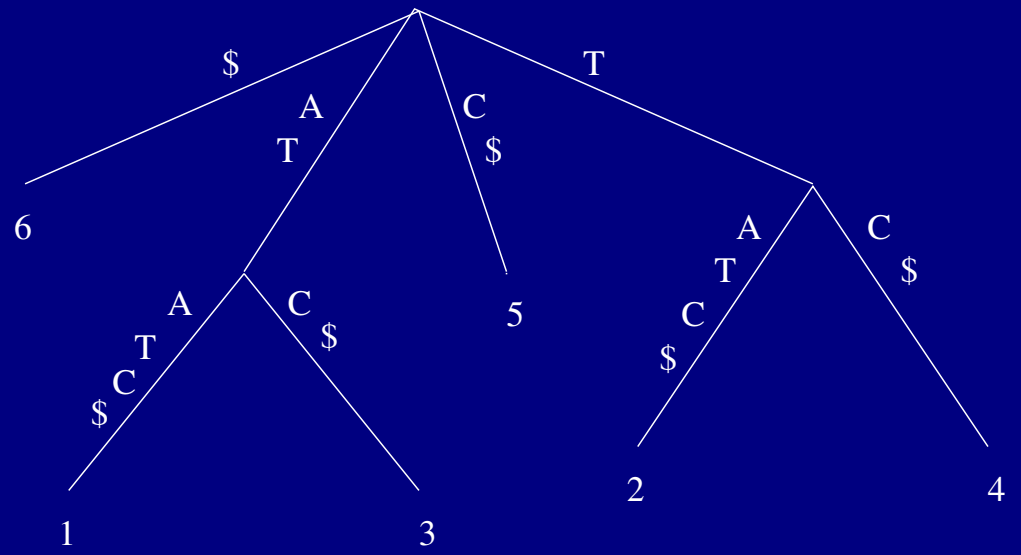
Pattern Retrieval Problem

Definition

Given a pattern P of length n and a text T of length m, find the positions of all occurrences of P in T

Example

$T = \texttt{ATATC}$
$P = \texttt{AT}$



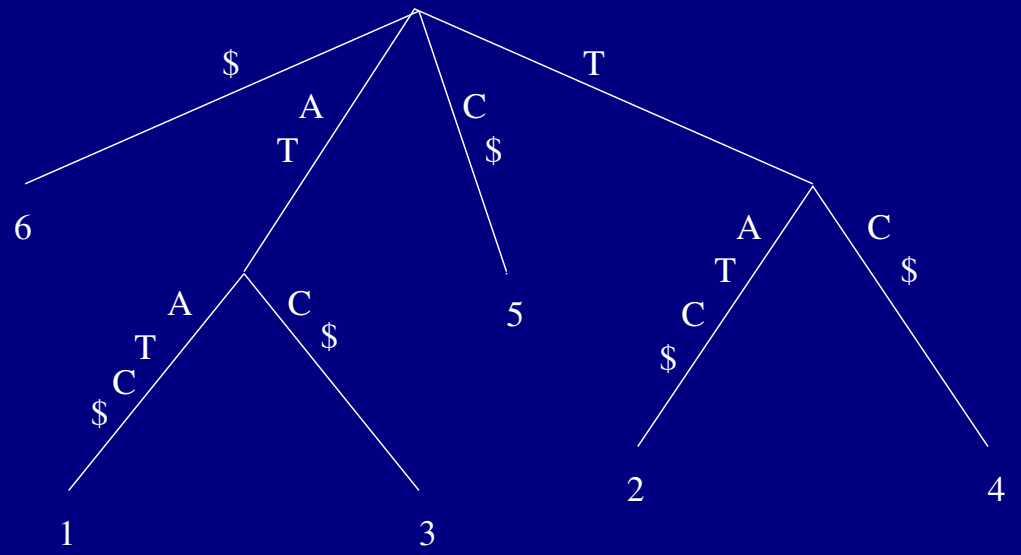
Pattern Retrieval Problem

Definition

Given a pattern P of length n and a text T of length m, find the positions of all occurrences of P in T

Example

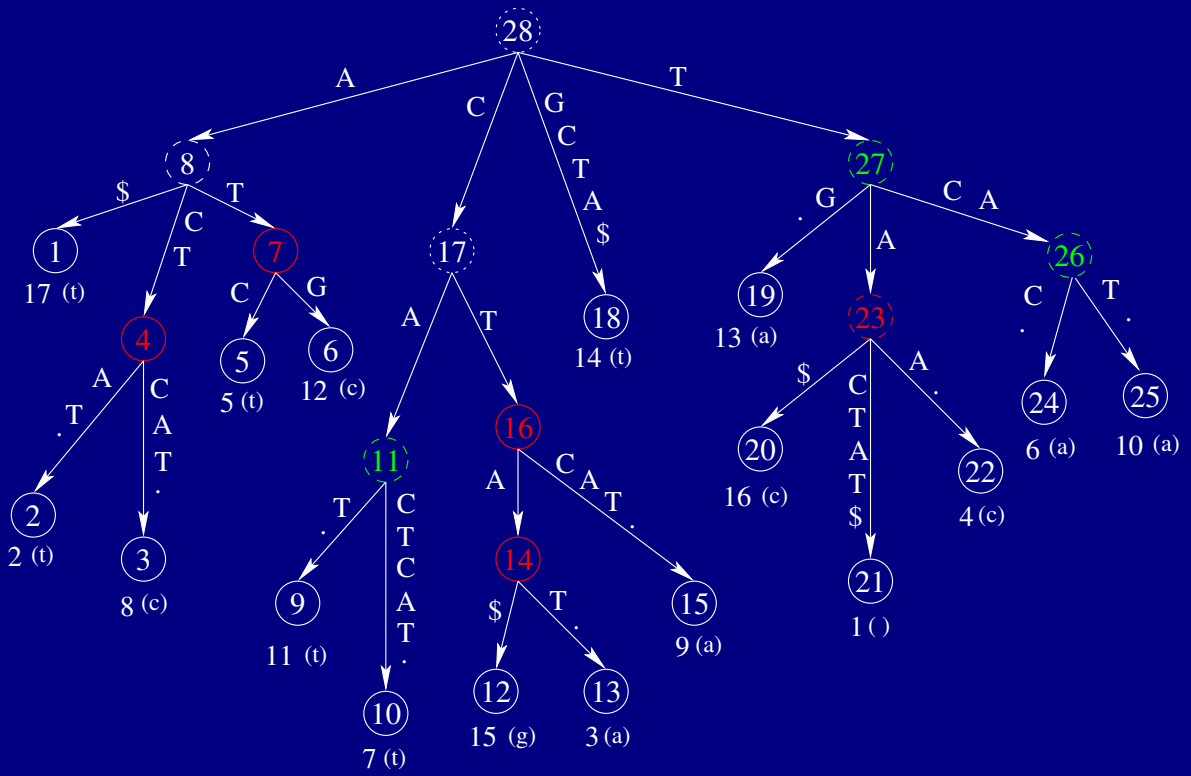
$T =$ `ATATC`
$P =$ `TAC`



Complexity of Suffix Tree

1. Preprocessing
   - Constant alphabet size : $O(m)\log|\Sigma|$ time and $O(m)$ space
     - Weiner (1973) [10]
     - McCreight (1976) [8]
     - Ukkonen (1995) [9]
   - Integer alphabet : $O(m)$ time
     - Farach (1997) [1]
2. Searching: $O(n)\log|\Sigma| + occ$ time

Finding All MEPs



Left	Pos list
v_{12}	G 15
v_{13}	A 3
v_{14}	A 3 G 15

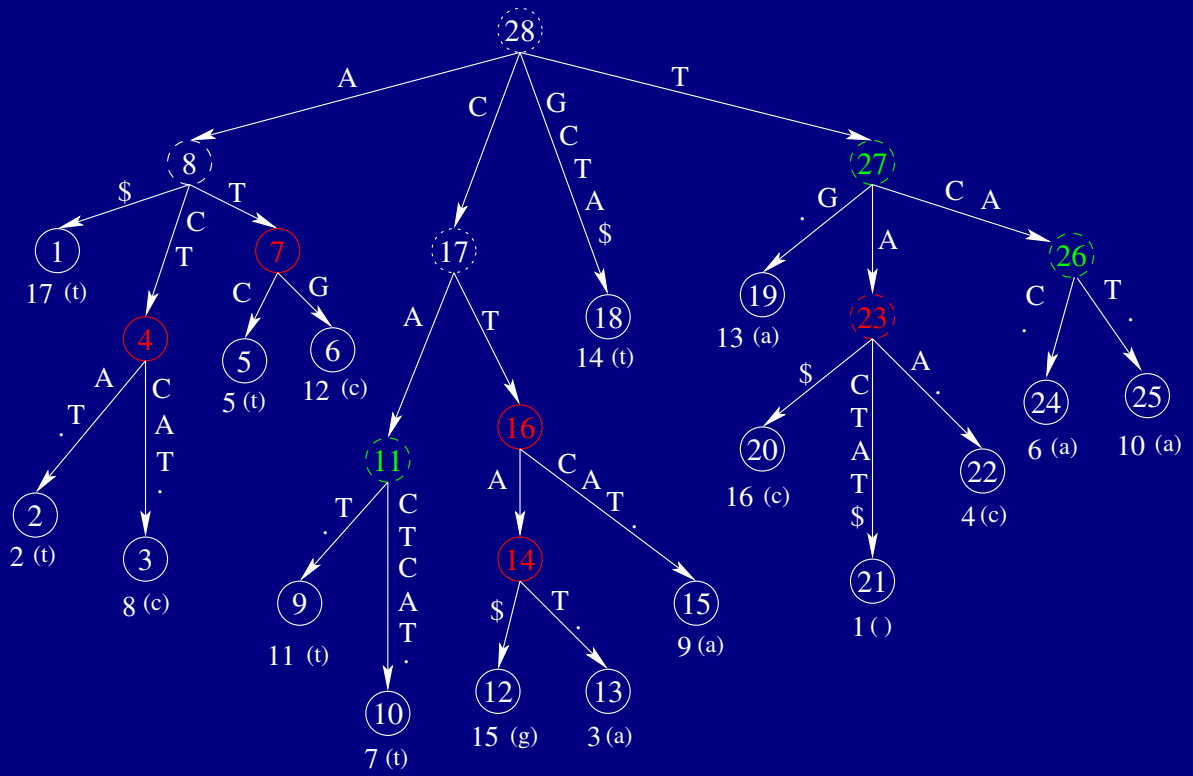
MEP: (3,15,3)

v_{15}	A 9
v_{16}	A 3,9 G 15

MEP: (9,15,2)

Finding All MEPs

TACTATCACTCATGCTA



Left	Pos list

v_{12}

G	15

v_{13}

A	3

v_{14}

A	3
G	15

MEP: (3,15,3)

v_{15}

A	9

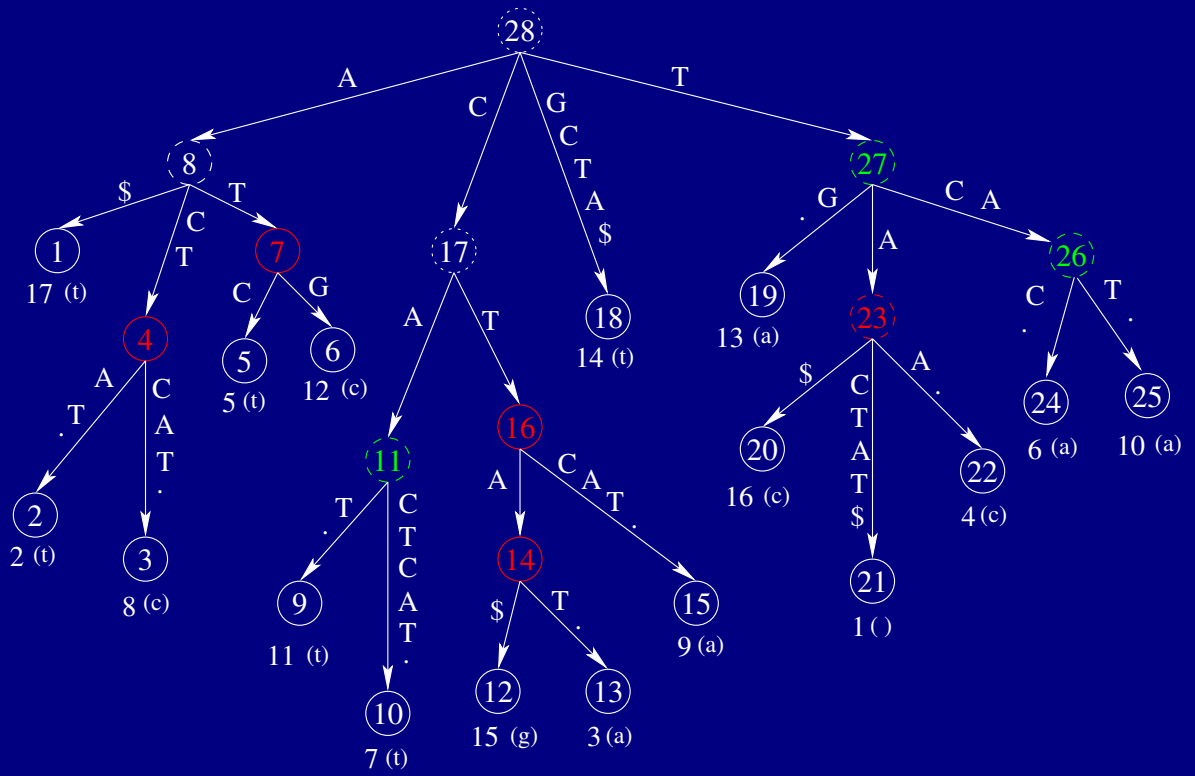
v_{16}

A	3,9
G	15

MEP: (9,15,2)

Finding All MEPs

TACTATCACTCATGCTA



Left	Pos list

v_{12}

G	15

v_{13}

A	3

v_{14}

A	3
G	15

MEP: (3,15,3)

v_{15}

A	9

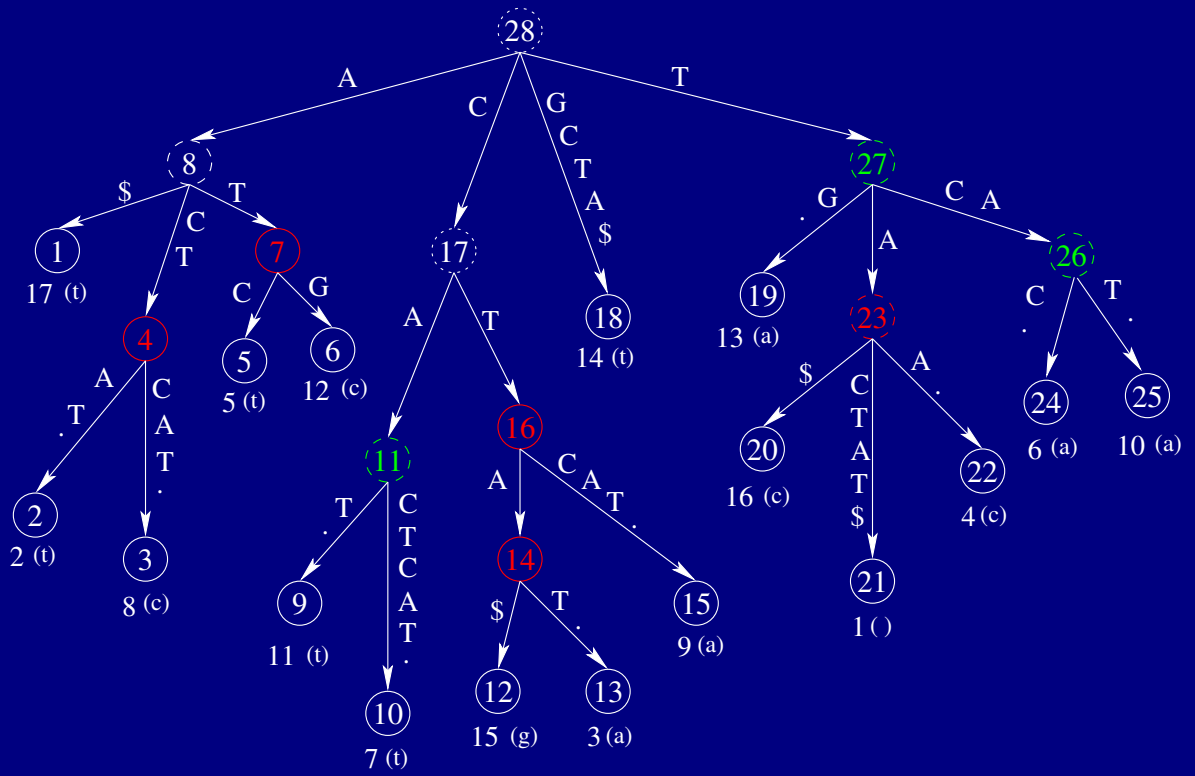
v_{16}

A	3,9
G	15

MEP: (9,15,2)

Finding All MEPs

TACTATCACTCATGCTA



Left	Pos list

v_{12}

G	15

v_{13}

A	3

v_{14}

A	3
G	15

MEP: (3,15,3)

v_{15}

A	9

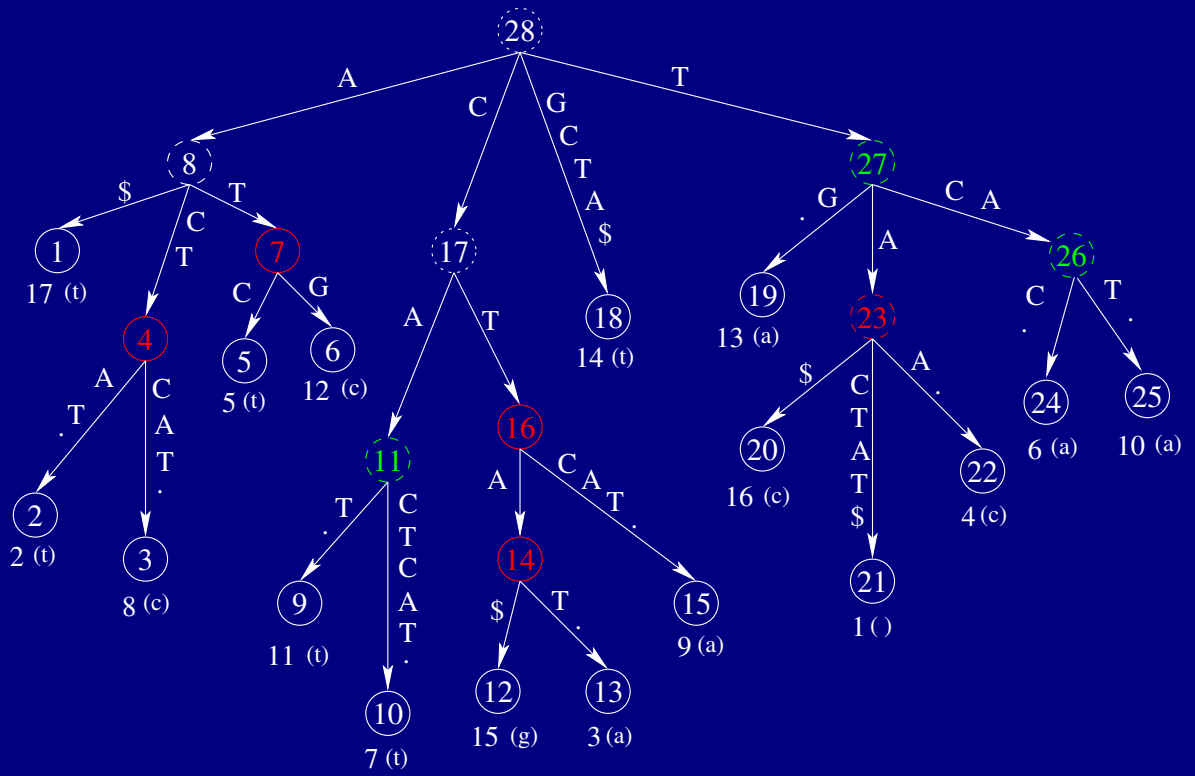
v_{16}

A	3,9
G	15

MEP: (9,15,2)

Finding All MEPs

TACTATCACTCATGCTA



Left	Pos list

v_{12}

G	15

v_{13}

A	3

v_{14}

A	3
G	15

MEP: (3,15,3)

v_{15}

A	9

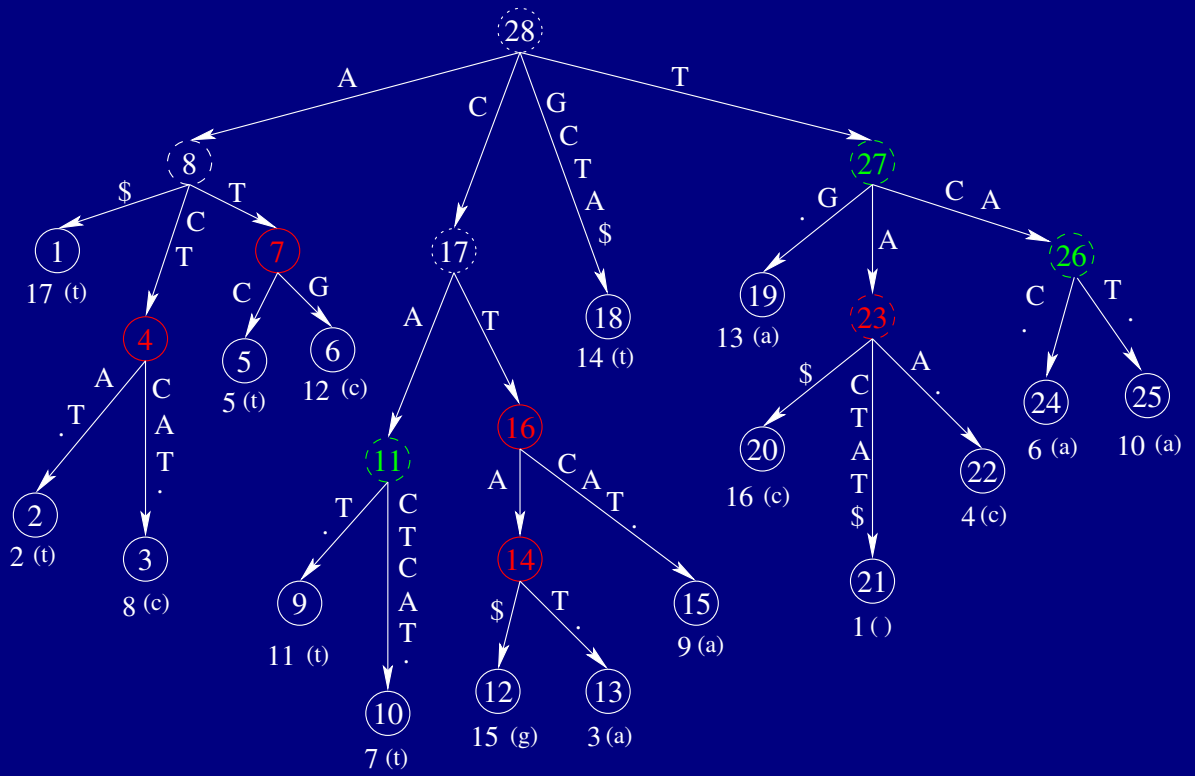
v_{16}

A	3,9
G	15

MEP: (9,15,2)

Finding All MEPs

TACTATCACTCATGCTA



Left	Pos list

v_{12}

G	15

v_{13}

A	3

v_{14}

A	3
G	15

MEP: (3,15,3)

v_{15}

A	9

v_{16}

A	3,9
G	15

MEP: (9,15,2)

Finding All MEPs

- Build a suffix tree for S
- $L_x(v)$: the list of leaf numbers in the subtree of node v with a left character x.
  e.g., $L_G(12) = \{15\}, L_A(16) = \{3, 9\}$.
- Do bottom up traversal of tree.
  For a node v, create a linked lists $L_x(v)$ indexed by all left character x.
  - Leaf node: create $L_x(v) = \{i\}$ where x is a left character and i is a suffix number for v.
  - Internal node:
    - $C(v)$: a set of children of v.
    - $D(v)$: a set of left characters of leaves in v's subtree.
    - For each $x \in D(v)$ and each $w \in C(v)$, do Cartesian product of $L_x(w)$ with $L_{x'}(w')$ for $x' \neq x$ and $w' \neq w$.

Finding All MEPs (cont.)

- Any pair in the list gives the starting positions of a maximal pair.
- Finally, for each x, compute $L_x(v)$ by union of $L_x(w)$ for every w.

- Let k' be the number of maximal pairs from Cartesian product, then Cartesian product takes $O(k')$ time.
- union can be done in $O(|\Sigma|)$ time using linked list.

- Totally $O(m + k)$ time and $O(m)$ space where k is the number of maximal pairs.

Implementation

1. Construct a suffix array for a given sequence of length n
   1. Larsson and Sadakane [7]: $O(n \log n)$
   2. Kim *et. al* [5], Ko and Aluru [6], Kärkkäinen and Sanders [3]: $O(n)$
2. Compute longest common prefix of adjacent sorted suffixes
   1. Kasai *et. al* [4]: $O(n)$
3. Traverse suffix array from bottom-up
   1. Gusfield [2]: $O(n)$
   2. leaf node: make a linked list for each pair of genome and left character
   3. branch node: Cartesian product and union children's linked lists if the length of its path label is above a threshold T_m

Strategy for Finding LTR Retrotransposon

1. Finding all pairs of LTRs.
   - searching maximal exact pairs using suffix array and discarding the maximal exact pair such that substrings don't locate within the length of inter-portion
   - combining maximal exact pairs where the left substrings and right substrings of each maximal exact pair are simultaneously within the length of LTR
2. Verifying inter-portion of each pair of LTRs.
   - domain search using `Pfam` + `hmmsearch`.

LTR — *gag* — *pol* — *env* — LTR

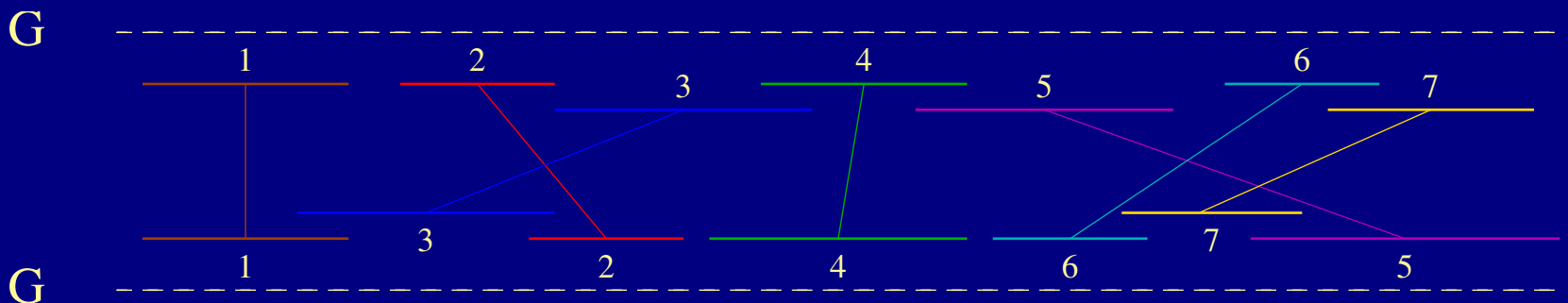
Chaining Problem

Definition

Given n MEPs $M_1, \cdots, M_n$, find the chain C of colinear non-overlapping MEPs such that its total score is maximum over all other chains. The total score of a chain is defined as

$$Score(C) = \sum_i (f(M_i) - g(M_{i+1}, M_i))$$

where $f(M_i)$ is the weight of M_i and $g(M_{i+1}, M_i)$ is the gap cost of connecting M_i to M_{i+1}.



Previous Work

- Graph based algorithm takes $O(n^2)$ time
- Geometric based algorithm is subquadratic (sparse dynamic programming)
  - Zhang et al. (1994) used space division with a kd-tree (no complexity analysis was given).
  - Myers and Miller (1995) used orthogonal range search with a range tree yielding a complexity of $O(nlog^k n)$ time and $O(nlog^{k-1} n)$ space.
  - Abouelhoda et al. (2003) used a range tree supported by fractional cascading and enhanced with priority queues and its complexity is $O(n \log^{k-2} n \log \log n)$ time and $O(n \log^{k-2} n)$ space.

Sparse Dynamic Programming

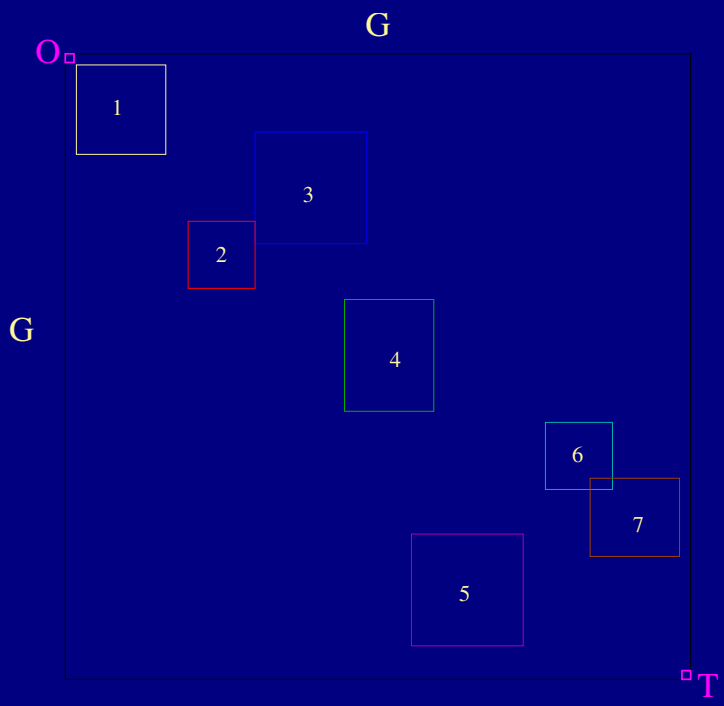
Definition

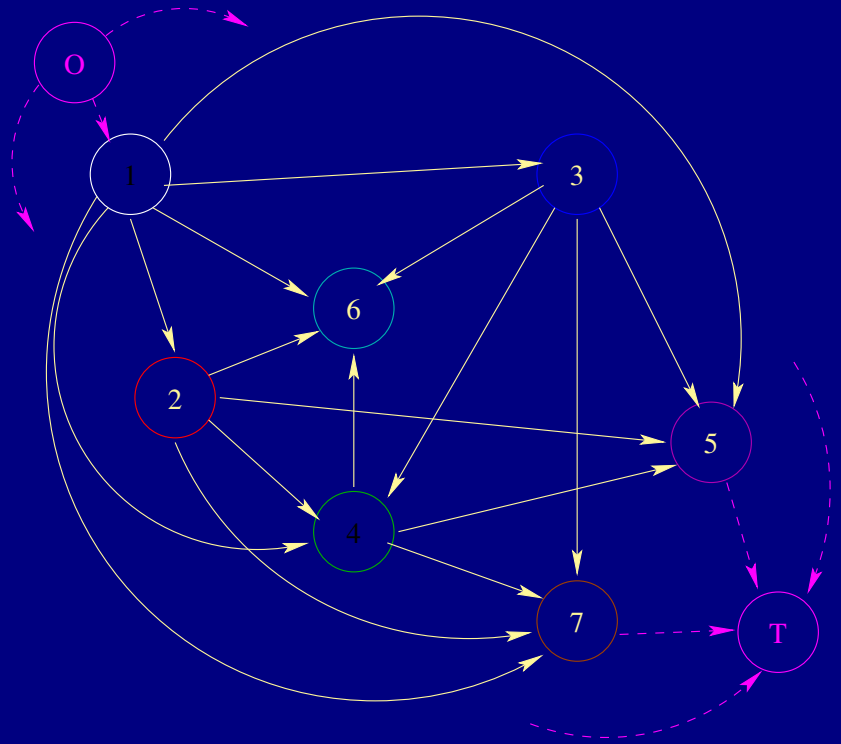
The maximum score can be computed by the recurrence

$$Score(M_j) = f(M_j) + \max_i \{0, Score(M_i) - g(M_i, M_j) : M_i \ll M_j\}$$

where $M_i \ll M_j$ means $end(M_i).p_r < start(M_j).p_r$ for all $r \in \{1, 2\}$ and $g(M_i, M_j)$ is the gap cost of connecting M_i to M_j.

Graph Based Algorithm



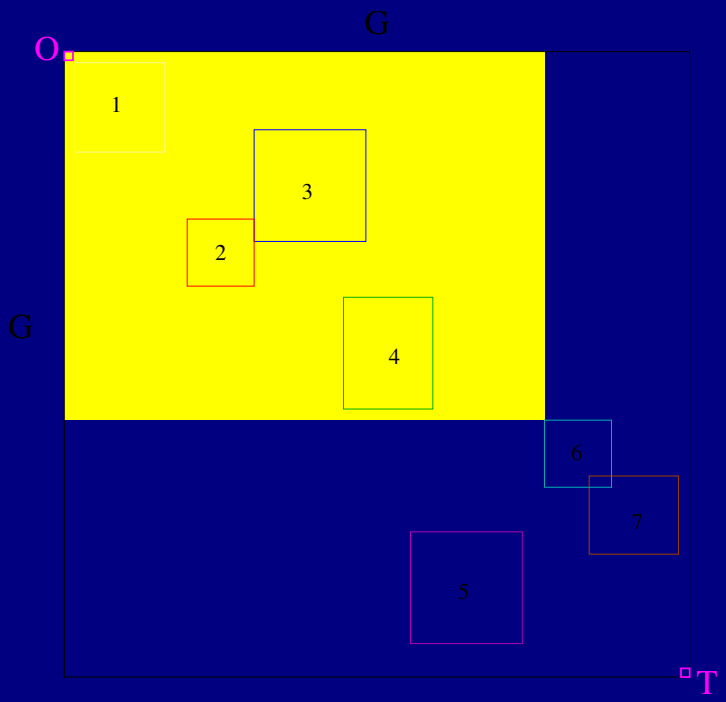


- Vertex : MEP with weight $f(M_i)$
- Edge : gap cost $g(M_i, M_j)$
- Chaining problem is converted to maximal weighted chain problem for a weighted graph and solved by Dynamic Programming.

Geometric Based Algorithm

Definition (RMQ (Range Maximum Query))

Retrieves the MEP M_i whose end point lies in the hyper-rectangle bounded by $start(M_j)$ and O such that $Score(M_i) - g(M_i, M_j)$ is maximum.



Geometric Based Algorithm

Definition (RMQ (Range Maximum Query))

Retrieves the MEP M_i whose end point lies in the hyper-rectangle bounded by $start(M_j)$ and O such that $Score(M_i) - g(M_i, M_j)$ is maximum.

The recurrence

$$Score(M_j) = f(M_j) + \max_i 0, Score(M_i) - g(M_i, M_j) : M_i \ll M_j$$

can be written as

$$Score(M_j) = f(M_j) + RMQ(O, start(M_j)$$

where O is a imaginary MEM with weight zero.

Gap Cost in L

In L_1 metric,

$$g_1(M_i, M_j) = \sum_{i \in \{1,2\}} (start(M_j).p_i - end(M_i).p_i).$$

In L_∞ metric,

$$g_\infty(M_i, M_j) = \max_{i \in \{1,2\}} (start(M_j).p_i - end(M_i).p_i)$$

However, any metric doesn't represent the real gap cost.

Implementation

1. Select two adjacent anchors that are apart no more than T_d.
2. Compute locally optimal chains by dynamic programming:
   1. Align each region between two adjacent anchors by Needleman-Wunsch algorithm.
   2. If its alignment score is below a preset threshold T_{nw}, then two anchors are not chained.

Strategy for Finding LTR Retrotransposon

1. Finding all pairs of LTRs.
   - searching maximal exact pairs using suffix array and discarding the maximal exact pair such that substrings don't locate within the length of inter-portion
   - combining maximal exact pairs where the left substrings and right substrings of each maximal exact pair are simultaneously within the length of LTR
2. Verifying inter-portion of each pair of LTRs.
   - domain search using `Pfam` + `hmmsearch`.

LTR — *gag* — *pol* — *env* — LTR

References

M. Farach.

Optimal suffix tree construction with large alphabets.

In *Proceedings of the 38th IEEE Annual Symposium on Foundations of Computer Science*, pages 137–143, Miami Beach, FL, 1997.

D. Gusfield.

Algorithms on strings, trees and sequences: Computer science and computational biology.

Cambridge University Press, Cambridge, 1997.

J. Kärkkäinen and P. Sanders.

Simple linear work suffix array construction.

In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, number 2719 in Lecture Notes in Computer Science, pages 943–955, Eindhoven, The Netherlands, 2003. Springer-Verlag, Berlin.

References (cont.)

Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park.

Linear-time longest-common-prefix computation in suffix arrays and its applications.

In A. Amir and G. M. Landau, editors, *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, number 2089 in Lecture Notes in Computer Science, pages 181–192, Jerusalem, Israel, 2002. Springer-Verlag, Berlin.

Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park.

Linear-time construction of suffix arrays.

In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, number 2676 in Lecture Notes in Computer Science, pages 186–199, Morelia, Michocán, Mexico, 2003. Springer-Verlag, Berlin.

References (cont.)

Pang Ko and Srinivas Aluru.

Space efficient linear time construction of suffix arrays.

In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, number 2676 in Lecture Notes in Computer Science, pages 200–210, Morelia, Michocán, Mexico, 2003. Springer-Verlag, Berlin.

N. J. Larsson and K. Sadakane.

Faster suffix sorting.

Report LU-CS-TR:99-214, Lund University, 1999.

E. M. McCreight.

A space-economical suffix tree construction algorithm.

J. Algorithms, 23(2):262–272, 1976.

References (cont.)

E. Ukkonen.

On-line construction of suffix trees.

Algorithmica, 14(3):249–260, 1995.

P. Weiner.

Linear pattern matching algorithm.

In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.