Computing Matching Statistics on Repetitive Texts

Younan Gao

Faculty of Computer Science, Dalhousie University, Canada

Matching Statistics

The matching statistics MS of a pattern $P[1..m]$ with respect to a text $T[1..n]$ is an array of integers $MS[1..m]$ such that the i-th entry $MS[i]$ stores the length of the longest prefix of $P[i..m]$ that occurs in T.

For example, given that $T[1..8] =$"*aaabbbcc*" and $P[1..5] =$"ccabb", the matching statistics $MS[1..5] = \{2, 1, 3, 2, 1\}$.

Related Work

Space	Time	Reference
$O(n)$	$O(m \lg \sigma)$	Textbook
$(n \lg \sigma + o(n \lg \sigma))$ bits	$O(m \lg \sigma)$	Enno et al.
$O(r + S(n))$	$O(m \cdot f(n))$	Bannai et al.
$O(z + S(n))$	$O(m^2 \lg^{\epsilon} z + m \cdot f(n))$	New
$O(z \lg z + S(n))$	$O(m^2 + m \lg z \lg \lg z + m \cdot f(n))$	New
$O(z \lg z + \frac{z}{\log_{\sigma} n} \lg^{2\epsilon+1} z + S(n))$	$O(m^2 \lg \lg \sigma + m \cdot f(n))$	New
$O(z \lg z + S(n))$	$O(m^2 + m \cdot f(n))$	σ is constant

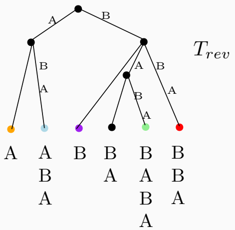
- z is the num of phrases in the Lempel-Ziv Parsing, while r is the num of runs in BWT.
- Assume that there is a data structure of $S(n)$ words of space to support retrieving any substring $T[i..i+\ell]$ in $O(f(n) + \ell)$ time.
- $r = O(z \lg^2 n)$.

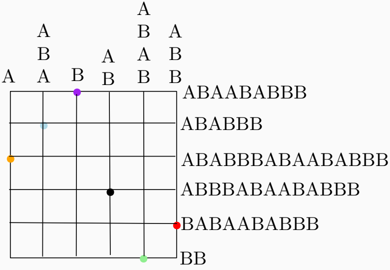
Preliminaries

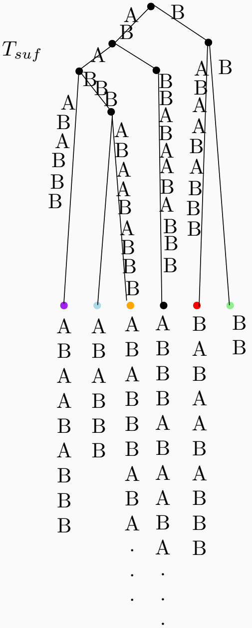
$Text[1..16] = A|AB|ABB|B|ABA|ABAB|BB$

Phrases	Phrases_{rev}	Suffixes
A	A	$\|AB\|ABB\|B\|ABA\|ABAB\|BB$
AB	BA	$\|ABB\|B\|ABA\|ABAB\|BB$
ABB	BBA	$\|B\|ABA\|ABAB\|BB$
B	B	$\|ABA\|ABAB\|BB$
ABA	ABA	$\|ABAB\|BB$
$ABAB$	BABA	$\|BB$

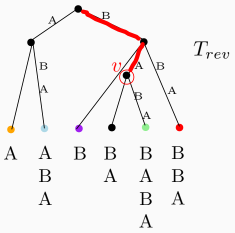
Preliminaries

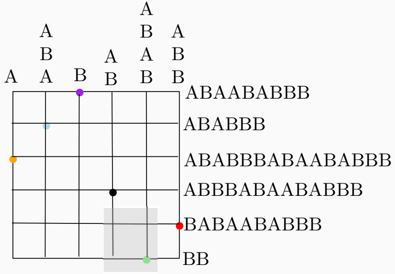


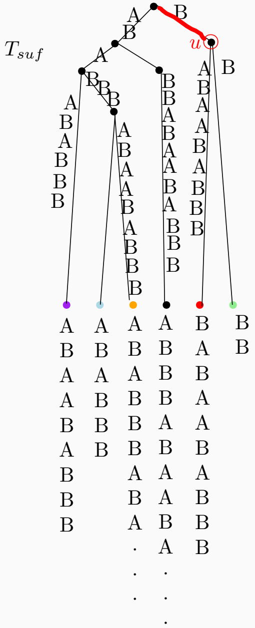




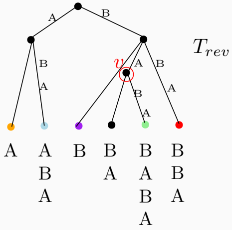
Preliminaries: Induced Relationship

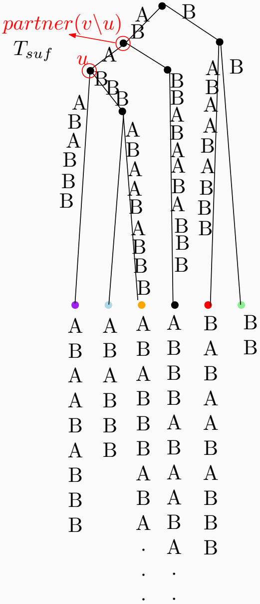






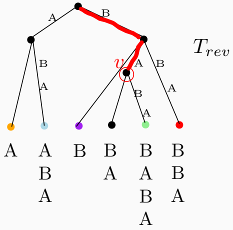
Preliminaries: Partner Finding

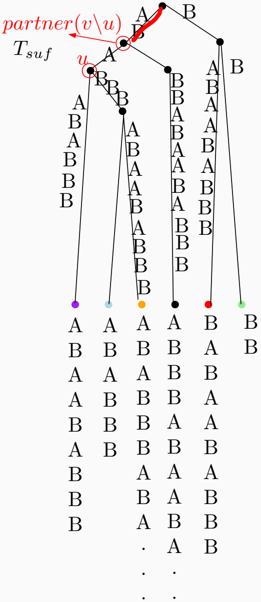




- Operation *partner*$(v \backslash u)$ can be implemented by 2D orthogonal range succ/prec queries.
- String ABAB appears in the text, AABABBBABAABABBB, but string ABABA does not;

Preliminaries: Partner Finding

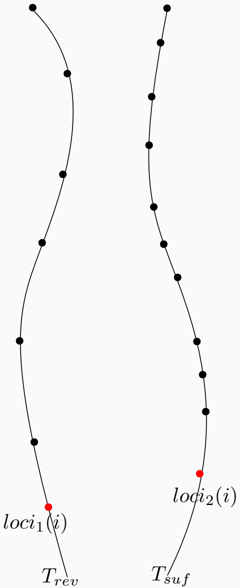




- Operation *partner*$(v\backslash u)$ can be implemented by 2D orthogonal range succ/prec queries.
- String ABAB appears in the text, AABABBBABAABABBB, but string ABABA does not;

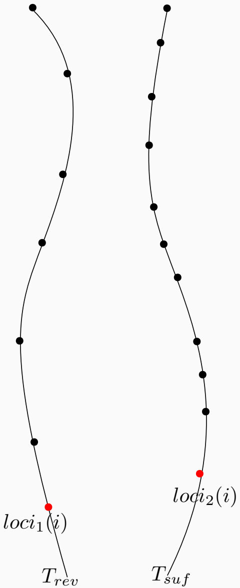
Naive Method for Computing $MS[1..m]$

- Query pattern $P[1..m]$ is split into $m-1$ pairs of prefix and suffix pairs.
- Consider the i-th pair, $P[1..i]$ and $P[i+1,m]$;
- For each ancestor, u, of $loci_1(i)$, find $partner(u \backslash loci_2(i))$
- Overall, we have $\sum_{i=1}^{m-1} i = O(m^2)$ partner (or LCP) queries.
- Query Time: $O(m^2 + mf(n) + m^2 \lg^{\epsilon} z)$.
- Space Cost: $O(z + S(n))$ words of space.



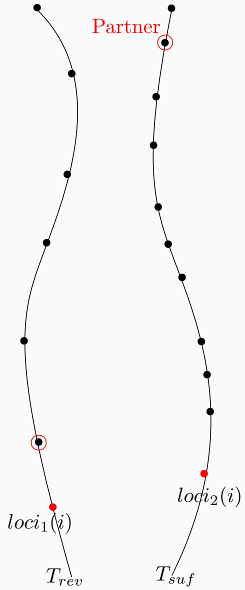
Naive Method for Computing $MS[1..m]$

- Query pattern $P[1..m]$ is split into $m-1$ pairs of prefix and suffix pairs.
- Consider the i-th pair, $P[1..i]$ and $P[i+1,m]$;
- For each ancestor, u, of $loci_1(i)$, find $partner(u \backslash loci_2(i))$
- Overall, we have $\sum_{i=1}^{m-1} i = O(m^2)$ partner (or LCP) queries.
- Query Time: $O(m^2 + mf(n) + m^2 \lg^{\epsilon} z)$.
- Space Cost: $O(z + S(n))$ words of space.



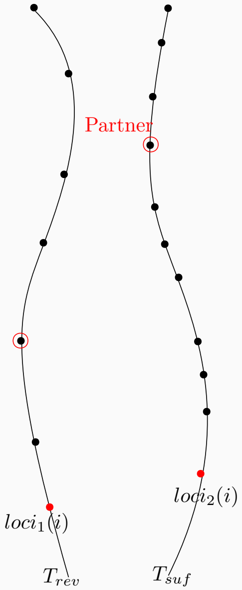
Naive Method for Computing $MS[1..m]$

- Query pattern $P[1..m]$ is split into $m-1$ pairs of prefix and suffix pairs.
- Consider the i-th pair, $P[1..i]$ and $P[i+1,m]$;
- For each ancestor, u, of $loci_1(i)$, find $partner(u \backslash loci_2(i))$
- Overall, we have $\sum_{i=1}^{m-1} i = O(m^2)$ partner (or LCP) queries.
- Query Time: $O(m^2 + mf(n) + m^2 \lg^{\epsilon} z)$.
- Space Cost: $O(z + S(n))$ words of space.



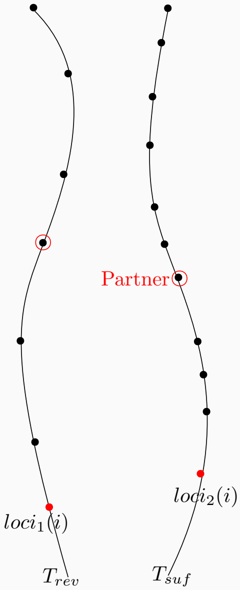
Naive Method for Computing $MS[1..m]$

- Query pattern $P[1..m]$ is split into $m-1$ pairs of prefix and suffix pairs.
- Consider the i-th pair, $P[1..i]$ and $P[i+1,m]$;
- For each ancestor, u, of $loci_1(i)$, find $partner(u \backslash loci_2(i))$
- Overall, we have $\sum_{i=1}^{m-1} i = O(m^2)$ partner (or LCP) queries.
- Query Time: $O(m^2 + mf(n) + m^2 \lg^{\epsilon} z)$.
- Space Cost: $O(z + S(n))$ words of space.



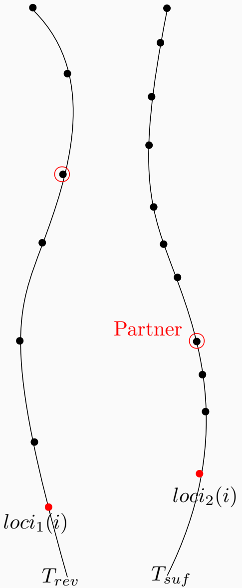
Naive Method for Computing $MS[1..m]$

- Query pattern $P[1..m]$ is split into $m-1$ pairs of prefix and suffix pairs.
- Consider the i-th pair, $P[1..i]$ and $P[i+1, m]$;
- For each ancestor, u, of $loci_1(i)$, find $partner(u \backslash loci_2(i))$
- Overall, we have $\sum_{i=1}^{m-1} i = O(m^2)$ partner (or LCP) queries.
- Query Time: $O(m^2 + mf(n) + m^2 \lg^{\epsilon} z)$.
- Space Cost: $O(z + S(n))$ words of space.



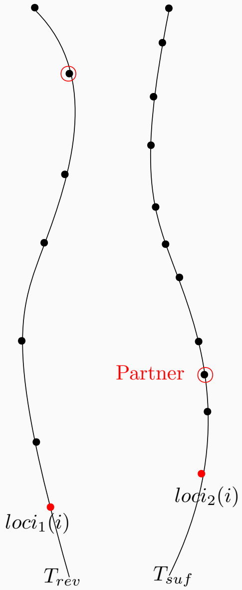
Naive Method for Computing $MS[1..m]$

- Query pattern $P[1..m]$ is split into $m-1$ pairs of prefix and suffix pairs.
- Consider the i-th pair, $P[1..i]$ and $P[i+1, m]$;
- For each ancestor, u, of $loci_1(i)$, find $partner(u \backslash loci_2(i))$
- Overall, we have $\sum_{i=1}^{m-1} i = O(m^2)$ partner (or LCP) queries.
- Query Time: $O(m^2 + mf(n) + m^2 \lg^{\epsilon} z)$.
- Space Cost: $O(z + S(n))$ words of space.



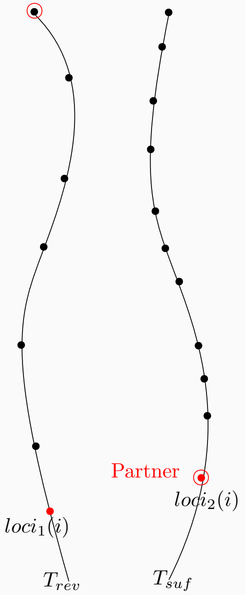
Naive Method for Computing $MS[1..m]$

- Query pattern $P[1..m]$ is split into $m-1$ pairs of prefix and suffix pairs.
- Consider the i-th pair, $P[1..i]$ and $P[i+1, m]$;
- For each ancestor, u, of $loci_1(i)$, find $partner(u \backslash loci_2(i))$
- Overall, we have $\sum_{i=1}^{m-1} i = O(m^2)$ partner (or LCP) queries.
- Query Time: $O(m^2 + mf(n) + m^2 \lg^{\epsilon} z)$.
- Space Cost: $O(z + S(n))$ words of space.



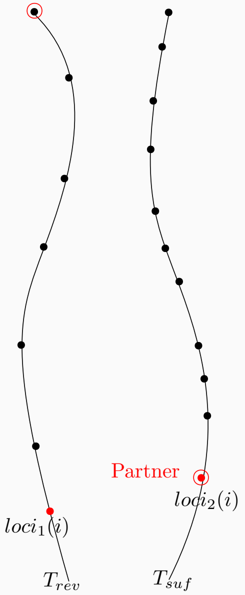
Naive Method for Computing $MS[1..m]$

- Query pattern $P[1..m]$ is split into $m-1$ pairs of prefix and suffix pairs.
- Consider the i-th pair, $P[1..i]$ and $P[i+1, m]$;
- For each ancestor, u, of $loci_1(i)$, find $partner(u \backslash loci_2(i))$
- Overall, we have $\sum_{i=1}^{m-1} i = O(m^2)$ partner (or LCP) queries.
- Query Time: $O(m^2 + mf(n) + m^2 \lg^{\epsilon} z)$.
- Space Cost: $O(z + S(n))$ words of space.



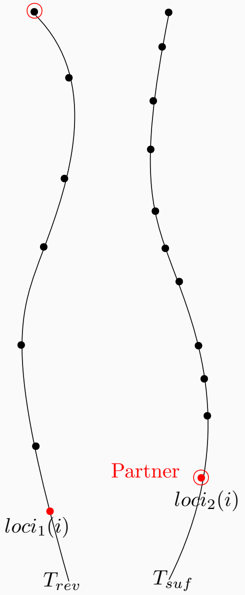
Naive Method for Computing $MS[1..m]$

- Query pattern $P[1..m]$ is split into $m-1$ pairs of prefix and suffix pairs.
- Consider the i-th pair, $P[1..i]$ and $P[i+1, m]$;
- For each ancestor, u, of $loci_1(i)$, find $partner(u \backslash loci_2(i))$
- Overall, we have $\sum_{i=1}^{m-1} i = O(m^2)$ partner (or LCP) queries.
- Query Time: $O(m^2 + mf(n) + m^2 \lg^{\epsilon} z)$.
- Space Cost: $O(z + S(n))$ words of space.



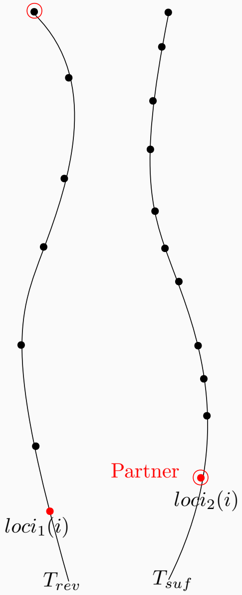
Naive Method for Computing $MS[1..m]$

- Query pattern $P[1..m]$ is split into $m-1$ pairs of prefix and suffix pairs.
- Consider the i-th pair, $P[1..i]$ and $P[i+1, m]$;
- For each ancestor, u, of $loci_1(i)$, find $partner(u \backslash loci_2(i))$
- Overall, we have $\sum_{i=1}^{m-1} i = O(m^2)$ partner (or LCP) queries.
- Query Time: $O(m^2 + mf(n) + m^2 \lg^{\epsilon} z)$.
- Space Cost: $O(z + S(n))$ words of space.



Naive Method for Computing $MS[1..m]$

- Query pattern $P[1..m]$ is split into $m-1$ pairs of prefix and suffix pairs.
- Consider the i-th pair, $P[1..i]$ and $P[i+1,m]$;
- For each ancestor, u, of $loci_1(i)$, find $partner(u \backslash loci_2(i))$
- Overall, we have $\sum_{i=1}^{m-1} i = O(m^2)$ partner (or LCP) queries.
- Query Time: $O(m^2 + mf(n) + m^2 \lg^{\epsilon} z)$.
- Space Cost: $O(z + S(n))$ words of space.



Second Method: LPMEM

$P[1..i] = a\boxed{bwwz}$ $\quad \downarrow loci_1(i)$

$P[i+1..m] = \boxed{xyyc}d\cdots$ $\quad \downarrow loci_2(i)$

Second Method: LPMEM

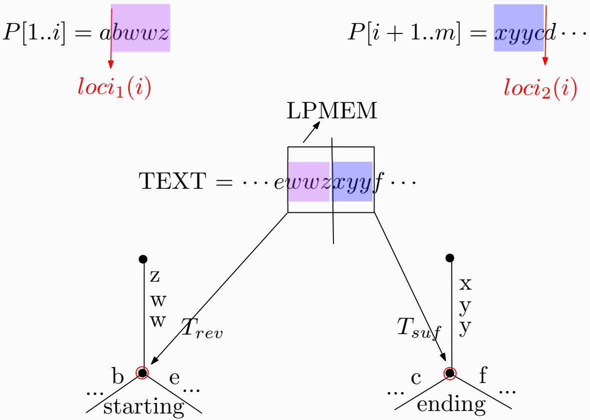
$P[1..i] = abwwz$ $\qquad$ $P[i+1..m] = xyycd\cdots$

$loci_1(i)$ $\qquad$ $loci_2(i)$

LPMEM

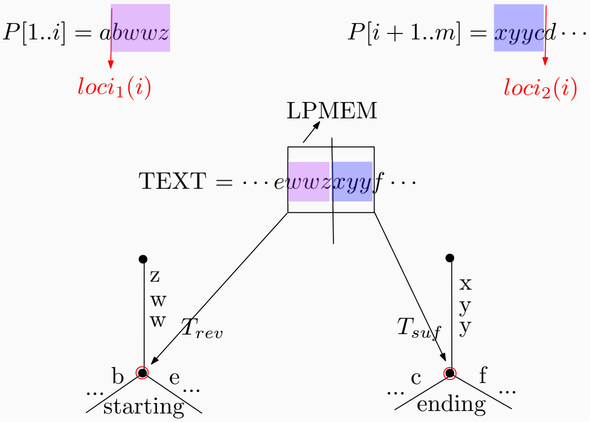
$$\text{TEXT} = \cdots e\boxed{wwzxyy}f\cdots$$

Second Method: LPMEM



- The number, *occ*, of LPMEMs for $P[1..m]$ is at most $\binom{m}{2}$.

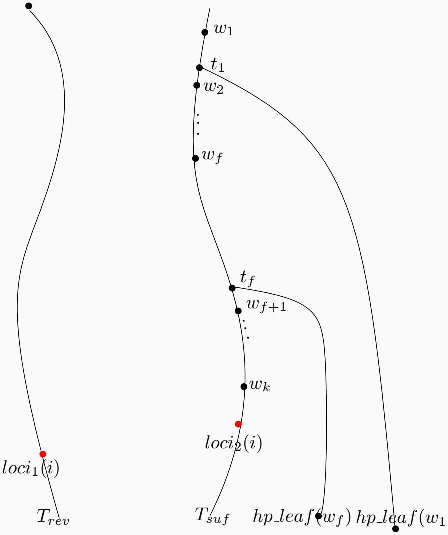
Second Method: LPMEM



- The number, *occ*, of LPMEMs for $P[1..m]$ is at most $\binom{m}{2}$.
- MS can be computed in $O(m + occ)$ time —Algorithm 2.

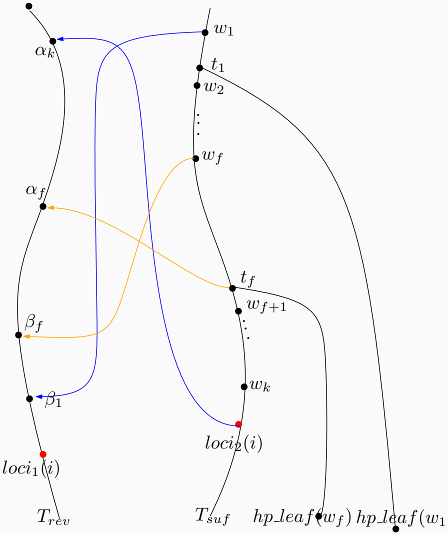
Second Method: Query Algorithms

- Apply heavy path decomposition on T_{suf}.
- For each $1 \leq f \leq k = O(\lg z)$, set $\alpha_f = partner(t_f \backslash loci_1(i))$ and $\beta_f = partner(w_f \backslash loci_1(i))$;
- α_f and $partner(\alpha_f \backslash loci_2(i))$ induce a LPMEM; so do β_f and $partner(\beta_f \backslash loci_2(i))$: **Lemma 6**;
- If u and v are induced together, and if v stays between w_f and t_f, then u stays between α_f and β_f: **Lemma 7**.



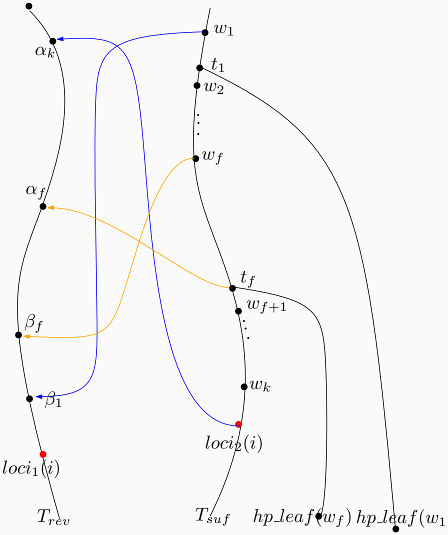
Second Method: Query Algorithms

- Apply heavy path decomposition on T_{suf}.
- For each $1 \le f \le k = O(\lg z)$, set $\alpha_f = partner(t_f \backslash loci_1(i))$ and $\beta_f = partner(w_f \backslash loci_1(i))$;
- α_f and $partner(\alpha_f \backslash loci_2(i))$ induce a LPMEM; so do β_f and $partner(\beta_f \backslash loci_2(i))$: **Lemma 6**;
- If u and v are induced together, and if v stays between w_f and t_f, then u stays between α_f and β_f: **Lemma 7**.



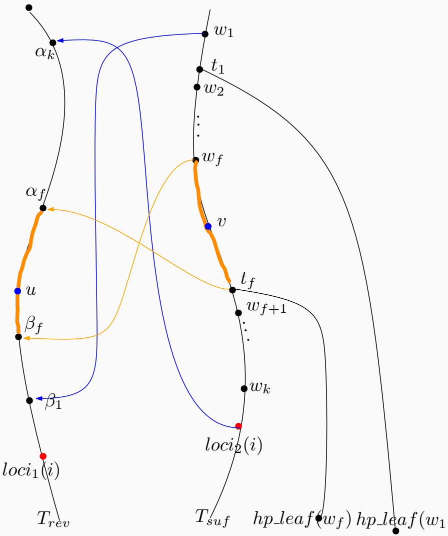
Second Method: Query Algorithms

- Apply heavy path decomposition on T_{suf}.
- For each $1 \leq f \leq k = O(\lg z)$, set $\alpha_f = \mathit{partner}(t_f \backslash \mathit{loci}_1(i))$ and $\beta_f = \mathit{partner}(w_f \backslash \mathit{loci}_1(i))$;
- α_f and $\mathit{partner}(\alpha_f \backslash \mathit{loci}_2(i))$ induce a LPMEM; so do β_f and $\mathit{partner}(\beta_f \backslash \mathit{loci}_2(i))$: **Lemma 6**;
- If u and v are induced together, and if v stays between w_f and t_f, then u stays between α_f and β_f: **Lemma 7**.



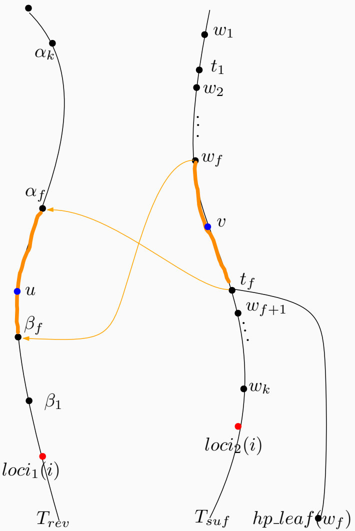
Second Method: Query Algorithms

- Apply heavy path decomposition on T_{suf}.
- For each $1 \le f \le k = O(\lg z)$, set $\alpha_f = partner(t_f \backslash loci_1(i))$ and $\beta_f = partner(w_f \backslash loci_1(i))$;
- α_f and $partner(\alpha_f \backslash loci_2(i))$ induce a LPMEM; so do β_f and $partner(\beta_f \backslash loci_2(i))$: **Lemma 6**;
- If u and v are induced together, and if v stays between w_f and t_f, then u stays between α_f and β_f: **Lemma 7**.



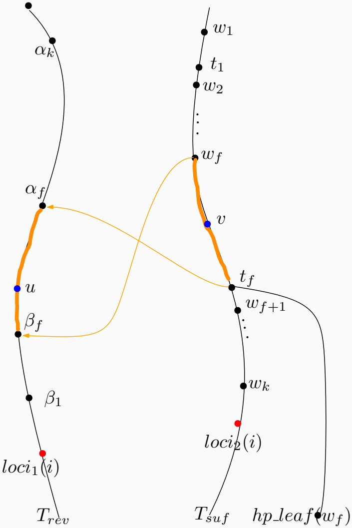
Second Method: Preprocessing

- w_f and $hp\_leaf(w_f)$ are known during the preprocessing stage;
- t_f and $loci_2(i)$ are unknown, but $partner(u \backslash loci_2(i)) = partner(u \backslash hp\_leaf(w_f)) = partner(u \backslash t_f)$: **Lemma 8**;
- α_f and β_f are unknown, but we can use the induced subtree $T_{rev}(w_f)$.



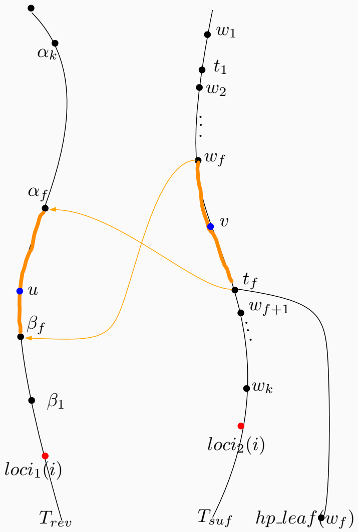
Second Method: Preprocessing

- w_f and $hp\_leaf(w_f)$ are known during the preprocessing stage;
- t_f and $loci_2(i)$ are unknown, but $partner(u \backslash loci_2(i)) = partner(u \backslash hp\_leaf(w_f)) = partner(u \backslash t_f)$: **Lemma 8**;
- α_f and β_f are unknown, but we can use the induced subtree $T_{rev}(w_f)$.

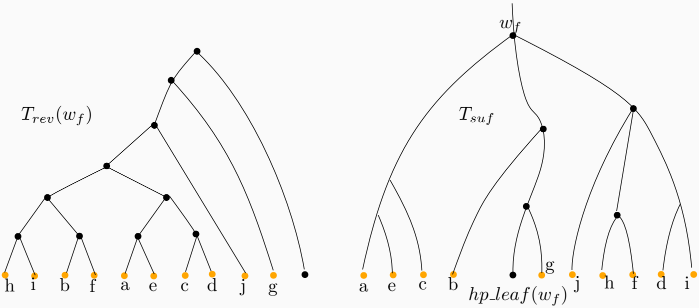


Second Method: Preprocessing

- w_f and $hp\_leaf(w_f)$ are known during the preprocessing stage;
- t_f and $loci_2(i)$ are unknown, but $partner(u \backslash loci_2(i)) = partner(u \backslash hp\_leaf(w_f)) = partner(u \backslash t_f)$: **Lemma 8**;
- α_f and β_f are unknown, but we can use the induced subtree $T_{rev}(w_f)$.

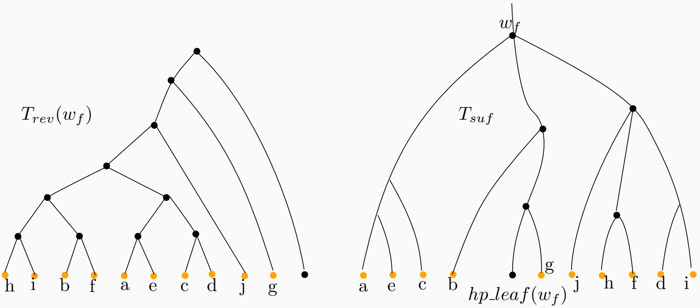


Second Method: Induced Subtree $T_{ref}(w_f)$



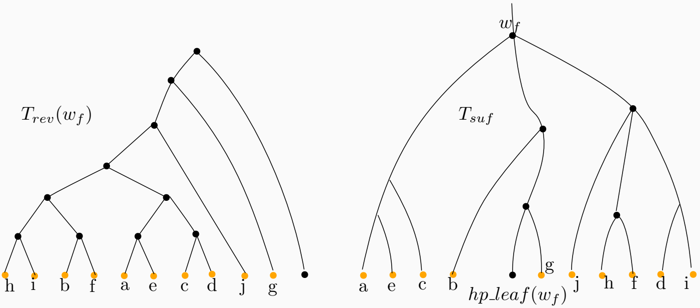
- Leaves in T_{rev} that are induced with w_f are called special leaves.
- $T_{rev}(w_f)$ contains the special leaves and their LCAs (special nodes).
- $\sum_{w_f} |special(w_f)| = O(z \lg z)$.
- The left endpoint of a LPMEM always stays at a special internal node: **Lemma 9**.

Second Method: Induced Subtree $T_{ref}(w_f)$



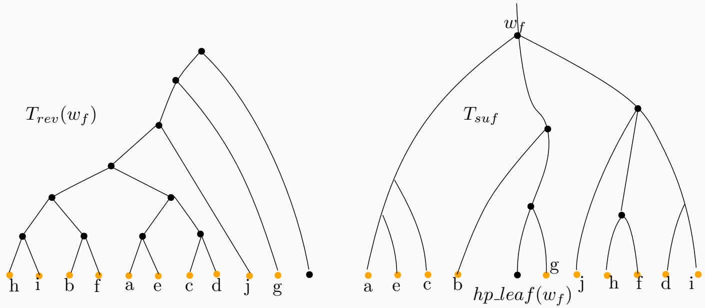
- Leaves in T_{rev} that are induced with w_f are called special leaves.
- $T_{rev}(w_f)$ contains the special leaves and their LCAs (special nodes).
- $\sum_{w_f} |special(w_f)| = O(z \lg z)$.
- The left endpoint of a LPMEM always stays at a special internal node: **Lemma 9**.

Second Method: Induced Subtree $T_{ref}(w_f)$



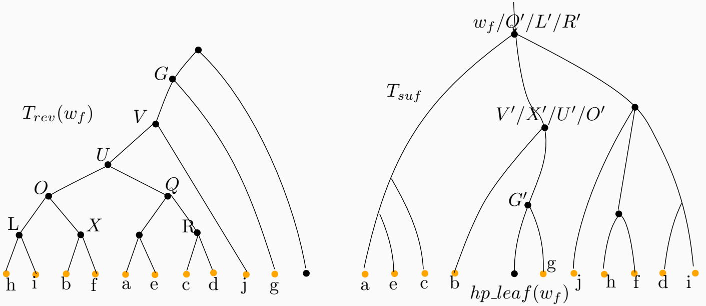
- Leaves in T_{rev} that are induced with w_f are called special leaves.
- $T_{rev}(w_f)$ contains the special leaves and their LCAs (special nodes).
- $\sum_{w_f} |special(w_f)| = O(z \lg z)$.
- The left endpoint of a LPMEM always stays at a special internal node: **Lemma 9**.

Second Method: Induced Subtree $T_{ref}(w_f)$



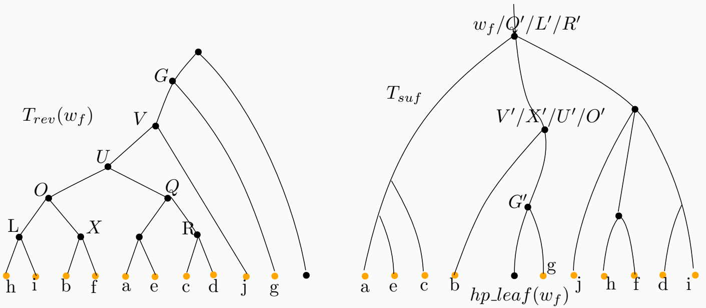
- Leaves in T_{rev} that are induced with w_f are called special leaves.
- $T_{rev}(w_f)$ contains the special leaves and their LCAs (special nodes).
- $\sum_{w_f} |special(w_f)| = O(z \lg z)$.
- The left endpoint of a LPMEM always stays at a special internal node: **Lemma 9**.

Second Method: Induced Subtree $T_{ref}(w_f)$

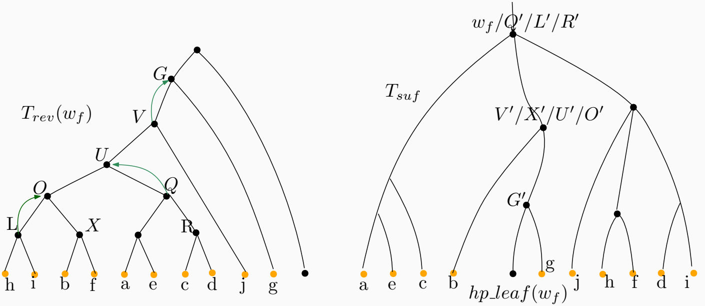


- Each special internal node U in $T_{rev}(w_f)$ stores a pointer, e_2, pointing to $U' = partner(U/hp\_leaf(w_f))$ in T_{suf}.
- Each special internal node U stores another pointer, e_0:
  - Given a pair of parent and child nodes, O and L, if L does not induce with O', then pointer e_0 of L points to O.
  - Otherwise, its e_0 points to the same node as of its lowest ancestor that has a valid e_0.

Second Method: Induced Subtree $T_{ref}(w_f)$

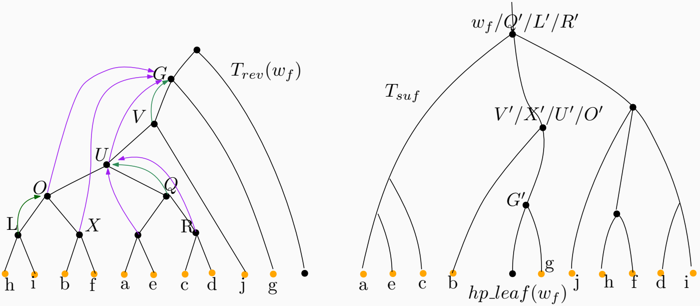


- Each special internal node U in $T_{rev}(w_f)$ stores a pointer, e_2, pointing to $U' = partner(U/hp\_leaf(w_f))$ in T_{suf}.
- Each special internal node U stores another pointer, e_0:
  - Given a pair of parent and child nodes, O and L, if L does not induce with O', then pointer e_0 of L points to O.
  - Otherwise, its e_0 points to the same node as of its lowest ancestor that has a valid e_0.



- Each special internal node U in $T_{rev}(w_f)$ stores a pointer, e_2, pointing to $U' = partner(U/hp\_leaf(w_f))$ in T_{suf}.
- Each special internal node U stores another pointer, e_0:
  - Given a pair of parent and child nodes, O and L, if L does not induce with O', then pointer e_0 of L points to O.
  - Otherwise, its e_0 points to the same node as of its lowest ancestor that has a valid e_0.

Second Method: Induced Subtree $T_{ref}(w_f)$



- Each special internal node U in $T_{rev}(w_f)$ stores a pointer, e_2, pointing to $U' = partner(U/hp\_leaf(w_f))$ in T_{suf}.
- Each special internal node U stores another pointer, e_0:
  - Given a pair of parent and child nodes, O and L, if L does not induce with O', then pointer e_0 of L points to O.
  - Otherwise, its e_0 points to the same node as of its lowest ancestor that has a valid e_0.

Second Method & Open Problems

- The second method:
  - Query Time: $O(m^2 + mf(n) + m \lg z \lg \lg z)$.
  - Space Cost: $O(z \lg z + S(n))$ words of space.
- Open Problems:
  - Query Time: $O(m \cdot f(n))$
  - Space: $O(z \lg z + S(n))$ or even $O(z + S(n))$?