Maximal Unique Matches (MUMs)

Veli Mäkinen

Maximal Unique Matches

- Given a collection C containing d strings from a finite alphabet ∑.
  - Let n be the *total length* of the input C.
- Frequency freq(P, C) is the number of strings containing substring P.
- Occurrence count Occ(P, C) is the number of occurrences of P in C.
- Problem: Given C, report all maximal substrings P having freq(P, C) = Occ(P, C) = d.

Optimal-time algorithm

- Integration of
  - Kasai et al. [Kas01] algorithm to visit all *branching substrings* of a text,
  - Hui's [Hui92] *color set size* technique.
  - Backward search [FM00].

- Toolbox:
  - Suffix array,
  - LCP-array,
  - Range Minimum Queries (RMQ).
  - FM-index

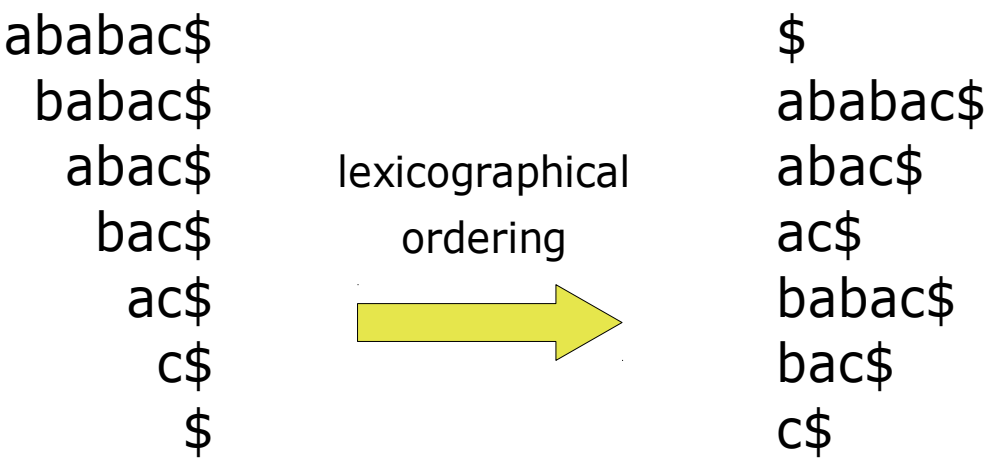
Visiting all Branching Substrings

- Substring α of T is *branching*, if there exists $a, b \in \Sigma$ $(a \neq b)$ such that αa and αb are both also substrings of T.
  - At most O(n) branching substrings.
  - No need to go through all $O(n^2)$ substrings!

- Branching substrings correspond to internal-nodes of a *suffix tree*.
  - Simulate traversal through the in-nodes by using only the Suffix array and LCP [Kas01].

Suffix Array

- Enumerate all suffixes of string ababac$:
  - Special end-marker $ is lexicographically smallest.



Suffix Array

- Suffix array SA contains the lexicographically ordered set of suffixes:
  - SA[i] gives the starting position of the i'th suffix.

1234567
ababac$

i	SA[i]	Suffix
1	7	$
2	1	ababac$
3	3	abac$
4	5	ac$
5	2	babac$
6	4	bac$
7	6	c$

LCP-array

- LCP[i] = lcp($T_{SA[i]..n}$, $T_{SA[i-1]..n}$)
  - lcp is the length of the *longest common prefix*.

i	SA[i]	LCP[i]	Suffix
1	7	0	$
2	1	0	ababac$
3	3	3	abac$
4	5	1	ac$
5	2	0	babac$
6	4	2	bac$
7	6	0	c$

  - SA and LCP can be constructed in O(n) time!

FM-index: backward step

- FM-index [FM00] is a compressed representation of suffix array that allows to compute interval [sp(cP),ep(cP)] of suffix array matching cP, given interval [sp(P),ep(P)] of suffix array matching P, in O(log |Σ|) time.

2	1	0	ababac$
3	3	3	abac$
4	5	1	ac$
5	2	0	babac$
6	4	2	bac$
7	6	0	c$

Visiting all Branching Substrings

- Example collection:

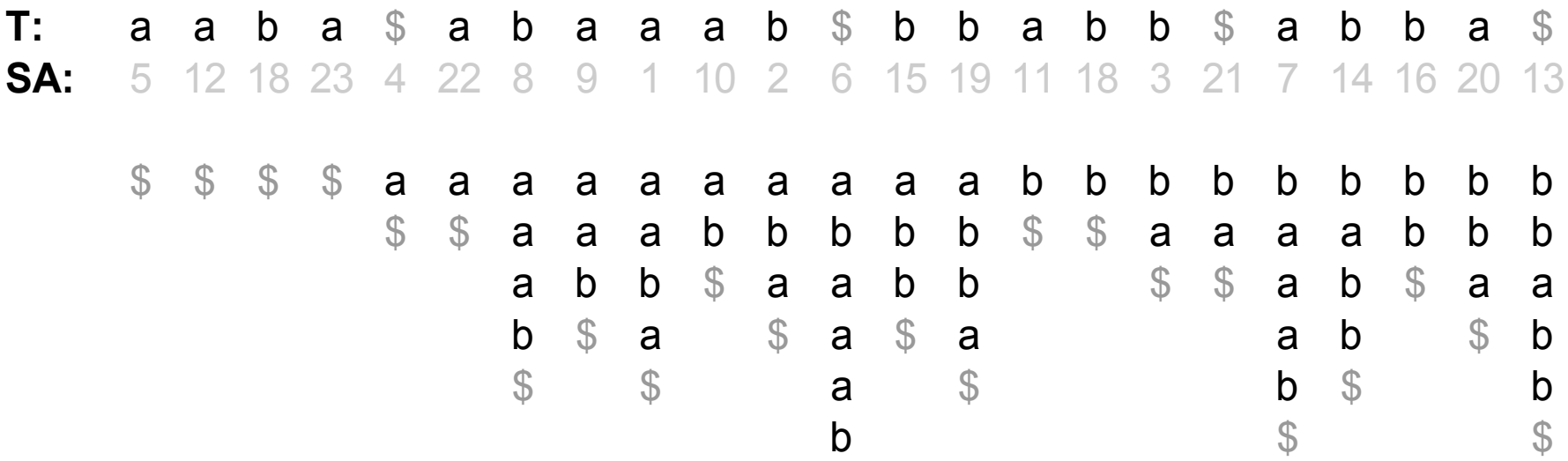
  C = {aaba, abaaab, bbabb, abba}

T: a a b a $ a b a a a b $ b b a b b $ a b b a $

Visiting all Branching Substrings

- Suffix array of

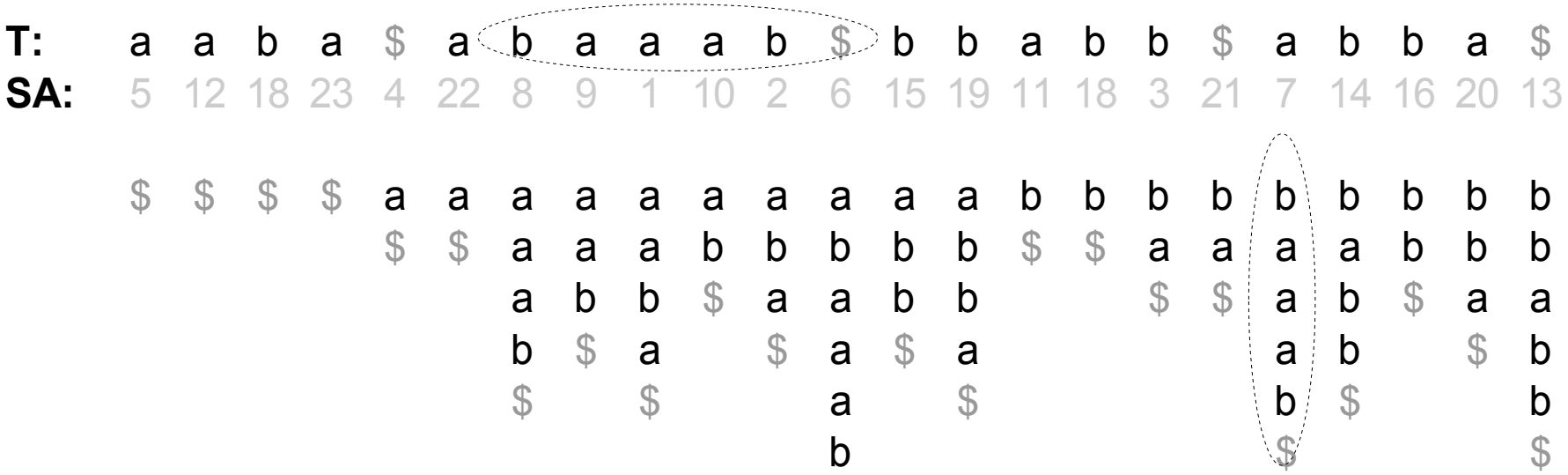
  C = {aaba, abaaab, bbabb, abba}.



Visiting all Branching Substrings

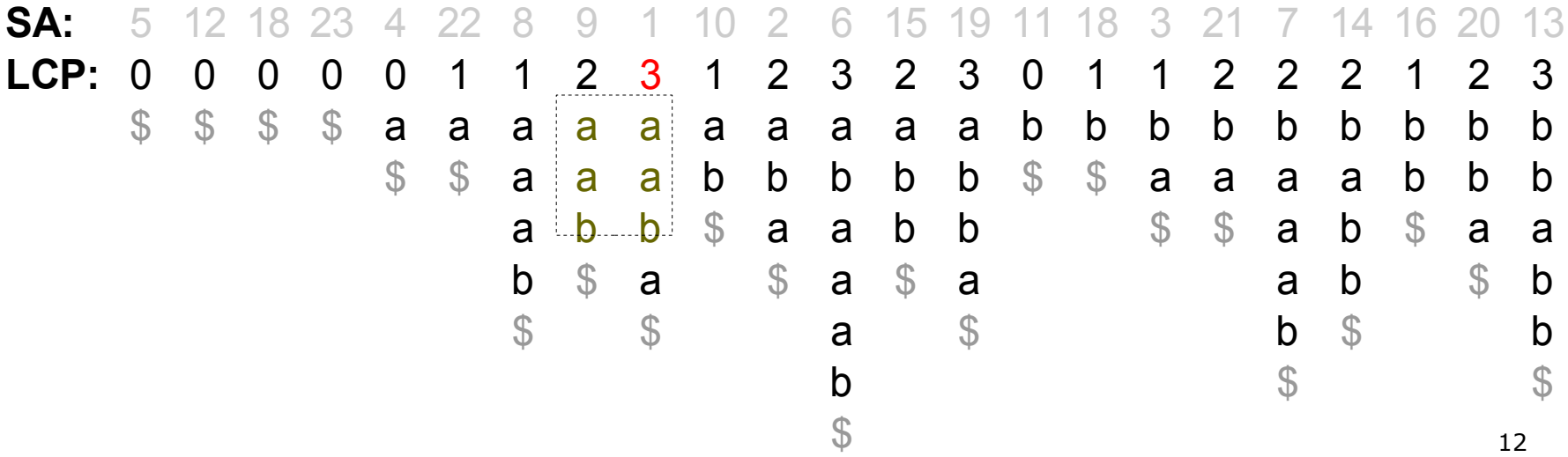
- Suffix array of

  C = {aaba, abaaab, bbabb, abba}.

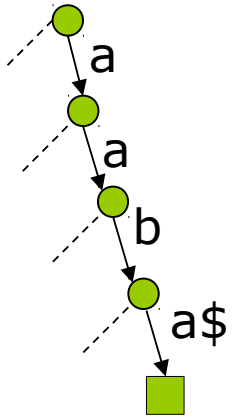


Visiting all Branching Substrings

- LCP-array:
  - LCP[i] = lcp($T_{SA[i]..n}$, $T_{SA[i-1]..n}$)



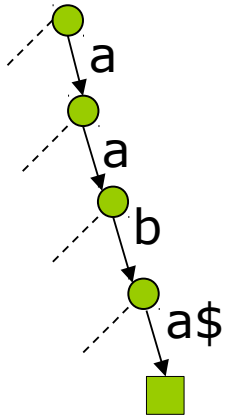
Visiting all Branching Substrings



- Proceed from left to right.
- At each step i, maintain a stack of nodes corresponding to the suffix SA[i].
- "Right-most path"

SA:	5	12	18	23	4	22	8	9	1	10	2	6	15	19	11	18	3	21	7	14	16	20	13
LCP:	0	0	0	0	0	1	1	2	3	1	2	3	2	3	0	1	1	2	2	2	1	2	3
	$	$	$	$	a$	a$	aab$	ab$	aba$	b$	ba$	baab$	bab$	bba$	b$	b$	ba$	ba$	baab$	bab$	bb$	ba$	bbab$

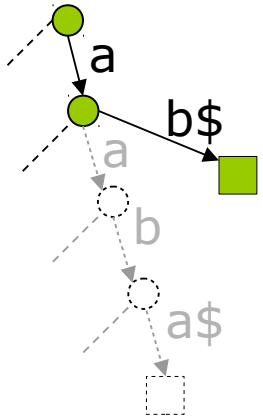
Updating the Stack



- To proceed to column i+1, first remove nodes with string-depth > LCP[i+1].
- In the next column LCP[10] = 1.

SA:	5	12	18	23	4	22	8	9	1	10	2	6	15	19	11	18	3	21	7	14	16	20	13
LCP:	0	0	0	0	0	1	1	2	3	1	2	3	2	3	0	1	1	2	2	2	1	2	3
	$	$	$	$	a$	a$	aab$	ab$	ab$	ab$	ba$	baab$	bab$	bba$	b$	b$	ba$	ba$	baab$	bab$	bb$	ba$	bbab$

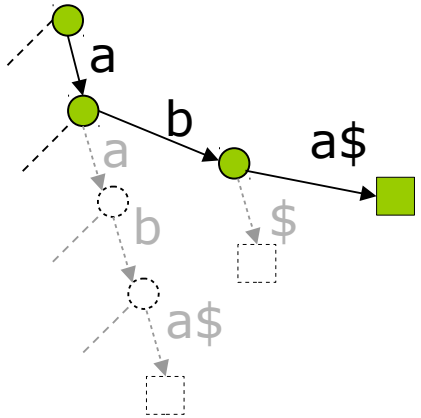
Updating the Stack



- Add a new leaf-node corresponding to suffix SA[i].

SA:	5	12	18	23	4	22	8	9	1	10	2	6	15	19	11	18	3	21	7	14	16	20	13
LCP:	0	0	0	0	0	1	1	2	3	1	2	3	2	3	0	1	1	2	2	2	1	2	3
	$	$	$	$	a$	a$	aab$	ab$	a$	ab$	abaab$	abaab$	abb$	abba$	b$	b$	ba$	ba$	baab$	bab$	bb$	bba$	bbab$

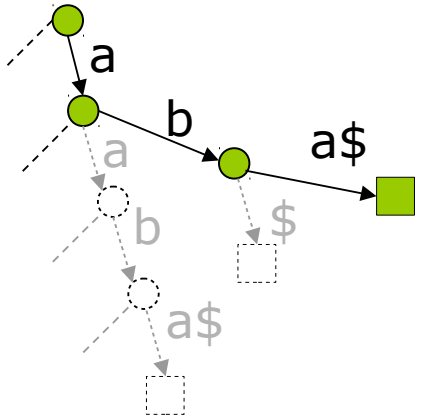
Updating the Stack



- In the next column LCP[11] = 2.
- Special case: No node with string-depth LCP[11] = 2.
- Add a new internal-node with string-depth 2.

SA:	5	12	18	23	4	22	8	9	1	10	2	6	15	19	11	18	3	21	7	14	16	20	13
LCP:	0	0	0	0	0	1	1	2	3	1	2	3	2	3	0	1	1	2	2	2	1	2	3
	$	$	$	$	a	a	a	a	a	a	a	a	a	a	b	b	b	b	b	b	b	b	b
					$	$	a	b	b	b	b	a	b	b	$	$	a	a	a	a	b	b	b
							a	$	a	$	a	a	b	b			$	$	b	b	$	a	a
							b		$		$	a	$	a					$	$		$	b
							$					b		$					$				$
												$											

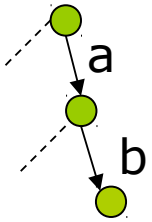
Calculating Frequency of Branching Substrings



- Branching substrings correspond to the internal-nodes.
- E.g. substring ab is branching.

- How can we calculate the *frequency* of all branching substrings?

Hui's algorithm



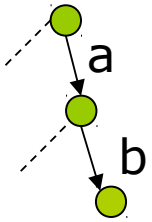
S[v] = 5
C[v] = 1

- Store for every node v in the stack values:
  - S[v]: number of occurrences of the substring ending at node v, and
  - C[v]: number of *duplicate* occurrences of the substring ending at node v.

C = {aaba, abaaab, bbabb, abba}

- Total S[v] = 5 occurrences of substring ab.
- Occurs twice in the second string, thus C[v] = 1.

Hui's algorithm



S[v] = 5
C[v] = 1

When v is removed from stack, S[v] - C[v] tells the number of *different* strings containing the substring ending at node v.

S[v] - C[v] is the *frequency* of the substring!

- Store for every node v in the stack values:
  - S[v]: number of occurrences of the substring ending at node v, and
  - C[v]: number of *duplicate* occurrences of the substring ending at node v.
- Updating S[v] is easy:
  - When popping, propagate the value to the parent.
- Updating C[v] is more tricky:
  - Memorize the previous suffix SA[i'] of the same document.
  - C[v']=C[v']+1 for v' being at string depth min LCP[i'+1..i].
  - When popping, update like S[v]

From frequency to MUMs

- Branching node v corresponds to is right-maximal substring P.
- When C[v]=0 and S[v]=d, P is unique.
- Enough to check left-maximality.
- With FM-index of the collection, we can calculate the backward step from interval [sp(P),ep(P)]=[i-d+1,i] to [sp(cP),sp(cP)] with all c. If there are two non-empty intervals, say [sp(cP),sp(cP)] and [sp(c'P),sp(c'P)], then v is left-maximal.
- If all above conditions hold, P is maximal unique match.

Space-Efficient MUMs

- Optimal-time algorithm requires $O(n)$ time, and $O(n \log n)$ bits.
  - Space is actually $\geq 3\, n \log n$ bits.

- Space-efficient MUMs:
  - Based on same algorithm, but using space-efficient data structures.

$O(n \log n)$ time, and
$O(n \log |\Sigma| + d \log n)$ bits of space.

Space-Efficient Data Structures

- Right-most path is kept in a special stack:
  - Relative *string-depths* and S[v] values are stored using Elias codes.
  - Takes O(n) bits.
  - Allows constant time pop/push.

- C[v] values are encoded using a *dynamic searchable partial sums* structure.
  - Unary coding in a dynamic bit vector.
  - O(n) bits with O(log n) time updates.

Space-Efficient Data Structures

- Compressed suffix array [HSS03]:
  - Construction in $O(n \log \log |\Sigma|)$ time, and $O(n \log |\Sigma|)$ bits.
  - Computing $SA[i]$ value takes $O(\log^{\varepsilon} n)$ time, for $\varepsilon \leq 1$.

- LCP and RMQ structures require $2n+o(n)$ bits [HS02,FH07].

References

[FHK06] Johannes Fischer, Volker Heun, Stefan Kramer: Optimal String Mining under Frequency Constraints, *Proc. PKDD'06*, LNAI 4213, pages 139-150, 2006.

[FH07] Johannes Fischer, Volker Heun: A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In *Proc. ESCAPE'07*, LNCS 4614, pages 459-470, 2007.

[HS02] Wing-Kai Hon, Kunihiko Sadakane: Space-Economical Algorithms for Finding Maximal Unique Matches. In *Proc. CPM 2002*, LNCS 2373, pages 144-152, 2002.

[HSS03] W.-K. Hon, K. Sadakane, and W.-K. Sung: Breaking a time-and-space barrier in constructing full-text indices. In *Proc. FOCS*, pages 251-260. IEEE Computer Society, 2003.

[Hui92] Lucas Hui: Color Set Size Problem with Application to String Matching. In *Proc. CPM 1992*, LNCS 644, pages 230-243, 1992.

[Kas01] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, Kunsoo Park: Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Proc. CPM 2001*, LNCS 2089, pages 181-192, 2001.