More Pattern Matching Techniques

Giulia Bernardini
giulia.bernardini@units.it

Fundamentals of algorithms
a.y. 2021/2022

Suffix trees are not always practical

Recall that the representation of the suffix tree is, in general, a tradeoff between space usage and time required for searching a pattern. In particular, we can guarantee that searching a pattern P can be done in O(|P|) time inly if Θ(|Σ||T|) space is used to represent the tree (using arrays of size |Σ| to store the branches).

Depending on the size of the alphabet of the indexed text, the size of the suffix tree may be thus too large in practice, even for constant-size alphabets.

The suffix array is a data structure for text indexing that has been introduced to address this issue.

The Suffix Array

Reference: Chapter 7.14 of: Gusfield, D. *Algorithms on Strings, Trees and Sequences.*

The suffix array

Given a text T of length n, the suffix array of T is an array of length n, whose elements are exactly the integers in [1,n].

The suffix array specifies the lexicographic order of the suffixes of T: its i-th element is the starting position of the i-th lexicographically smallest suffix of T.

The suffix array requires only n machine words of space (assuming a word size of at least log(n) bits).

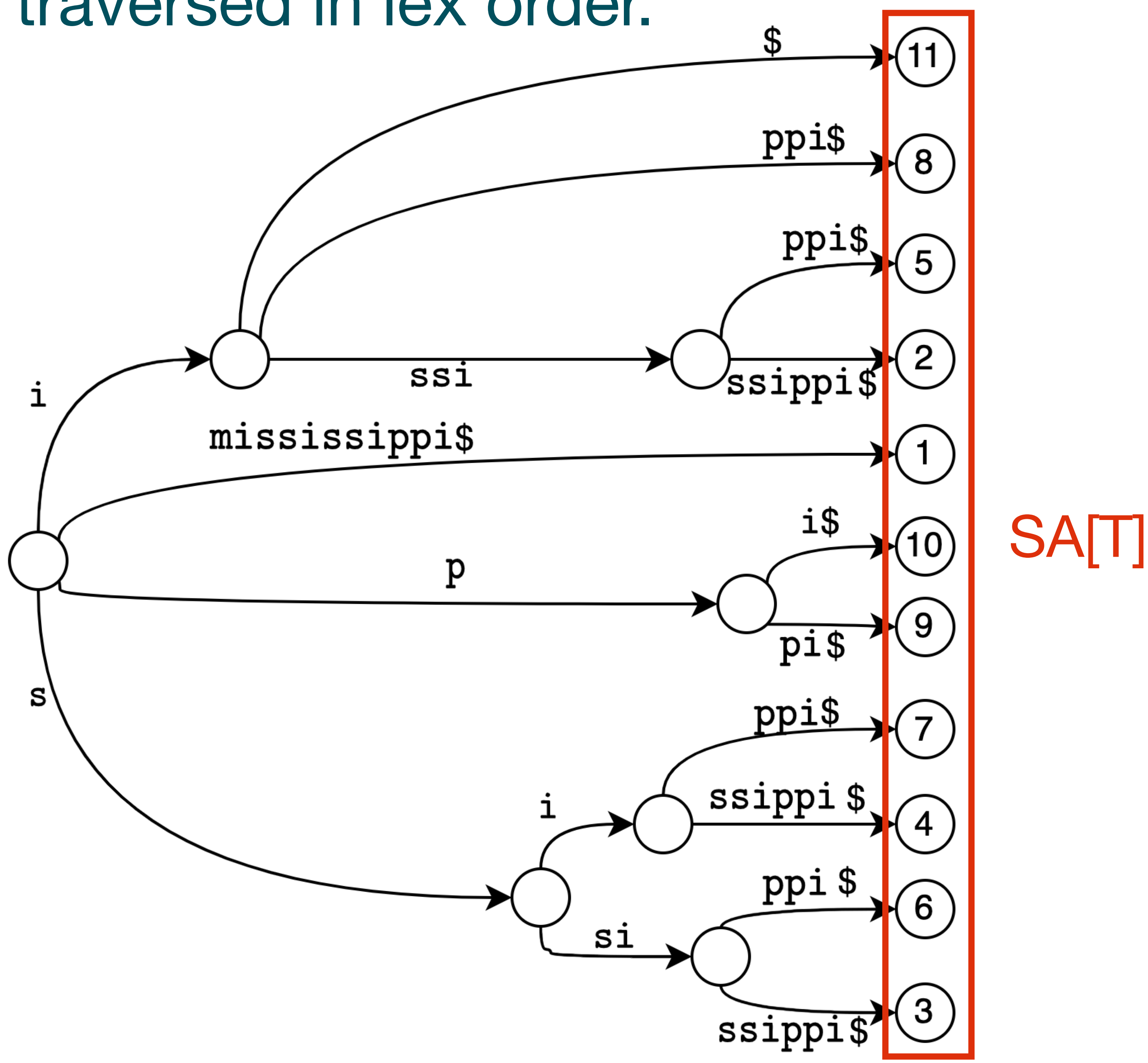
Consider string T=`mississippi`.

SA[6]=10 because T_{10} is the 6th lexicographically smallest suffix of T

SA(T)	Lex order of suffixes
11:	i
8:	ippi
5:	issippi
2:	ississippi
1:	mississippi
10:	pi
9:	ppi
7:	sippi
4:	sisippi
6:	ssippi
3:	ssissippi

Construction of the suffix array

SA[T] can be constructed in O(n) time by constructing a suffix tree of T such that the edges at each node are lexicographically ordered, and then performing a “lexical” DFS of T, in which the edges are traversed in lex order.



Pattern matching with suffix arrays

Key observation: since the suffixes of T are ordered, all the suffixes that start with an occurrence of P are are consecutive in the suffix array.

For example: “issi” occurs in T at positions 5 and 2, that are consecutive in the suffix array.

Pattern matching can thus be done by binary searching: if P is lex smaller than SA[n/2], then search it in the first half of SA; otherwise, search it in the second half…

Each iteration costs O(|P|) time. There are at most log(|T|) iterations. Thus this requires O(|P|log(|T|)) time in the worst case.

SA(T)	Lex order of suffixes
11:	i
8:	ippi
5:	issippi
2:	ississippi
1:	mississippi
10:	pi
9:	ppi
7:	sippi
4:	sisippi
6:	ssippi
3:	ssissippi

Pattern matching with suffix arrays

To avoid reading the same characters of P over and over, we can make the following observation.

Let L and R be the left and right boundaries of an interval considered during binary search. During the search, we can keep track of the length l of the longest prefix of SA(L) and the length r of the longest prefix of SA(R) that match a prefix of P. Let mlr=min{l,r}: for any index i between L and R, there is a common prefix of T_i and P of length at least mlr.

Then, when comparing P with T_M, where $M=\lceil (R-L)/2 \rceil$, we can start comparing the characters from position mlr+1 of both P and T_M. The worst-case time bound for this method is still O(|P|log(|T|)), but in practice it runs in time O(|P|+log(|T|)).

SA(T)	Lex order of suffixes
11:	i
8:	ippi
5:	issippi
2:	ississippi
1:	mississippi
10:	pi
9:	ppi
7:	sippi
4:	sisippi
6:	ssippi
3:	ssissippi

Pattern matching with suffix arrays

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
T:	a	b	b	a	b	b	a	b	b	a	b	b	a	b	a	a	a	b	a	b	a	b	b	a	b	b	b	a	b	b	a	#
SA :	15	16	31	13	17	19	28	10	7	4	1	21	24	32	14	30	12	18	27	9	6	3	20	23	29	11	26	8	5	2	22	25

1	15	aaabababbabbbabba#
2	16	aababab babbbabba#
3	31	a#
4	13	abaaababab babbbabba#
5	17	abababbabbbabba#
6	19	ababbabbbabba#
7	28	abba#
8	10	abbabaaabababbabbbabba#
9	7	abbabbabaaabababbabbbabba#
10	4	abbabbabbabaaabababbabbbabba#
11	1	abbabbabbabbabaaabababbabbbabba#
12	21	abbabbbabba#
13	24	abbbabba#
14	32	#
15	14	baaabababbabbbabba#
16	30	ba#
17	12	babaaabababbabbbabba#
18	18	bababbabbbabba#
19	27	babba#
20	9	babbabaaabababbabbbabba#
21	6	babbabbabaaabababbabbbabba#
22	3	babbabbabbabaaabababbabbbabba#
23	20	babbabbbabba#
24	23	babbbabba#
⋮	⋮	⋮
32	25	bbbabba#

Example from: R. Grossi and J.S. Vitter,
COMPRESSED SUFFIX ARRAYS AND SUFFIX TREES WITH
APPLICATIONS TO TEXT INDEXING AND STRING MATCHING

Seminumerical String Matching

Reference: Chapter 4 of: Gusfield, D. *Algorithms on Strings, Trees and Sequences.*

The shift-and method

All the methods we have seen so far are comparison-based: the main primitive operation they do is the comparison of two characters. Not all existing strategies are of this type.

The shift-and method is based on bit-level operations, and is extremely fast for relatively short patterns (e.g., words in a natural language) that fit into a machine word.

Let M be a |P|x|T| binary matrix, such that M[i,j]=1 if and only if the first i characters of P match exactly the i characters of T ending at position j. Otherwise M[i,j]=0.

For example, for T=california and P=for, M[1,5]=M[2,6]=M[3,7]=1; all the other entries are 0.

In other words, the 1-entries of row i of M encode all the positions in T where an occurrence of P[1..i] ends, and the 1-entries of column j encode all the prefixes of P that end at position j of T.

Then M[|P|,j]=1 if and only if an occurrence of P ends at position j.

The shift-and method

The goal is thus to compute the last row of M. To do so, we use $|\Sigma|$ auxiliary binary arrays of length $|P|$.

For each character x of Σ, U(x)[i]=1 if and only if P[i]=x: e.g., if P=abcdaba, U(a)=1000101.

Let bit-shift(j) the operation consisting of shifting column j of M down by one position, putting a 1 in the first position. E.g, a column 0010010100 would become 1001001010.

The shift-and method: computing M

To construct M, we start from the first column, that is initialised to all zeros if P[1]≠T[1]; and otherwise the first entry is a 1 and the others are all zeros. Then we construct M column-by-column.

For j>1, column j is obtained from column j-1 and U(T[j]) by doing the bit-level operation U(T[j]) AND bit-shift(j-1).

This is correct because, for any i>1, M[i,j] should be 1 if and only if the first i-1 chars of P match the i-1 chars of T ending at j-1, and additionally P[i]=T[j]. The first condition is true when M[i-1,j-1] is a 1; the second condition is true when U(T[j])[i]=1. Thus shifting column j-1 allows to compare entries M[i-1,j-1] with the entry i of U(T[j]): and column j has a 1 in position i only if they are both 1.

The shift-and method: complexity

In the worst case, the shift-and algorithm requires $\Theta(|P||T|)$ bit operations. Nevertheless, if $|P|$ is within one machine word, the bit-shift operation and the AND operation for the columns are single-word operations, that are very fast (constant-time). This is true also if $|P|$ fits within a small number of machine words. In these cases, the shift-and method requires $\Theta(|T|)$ time in practice.

As for the space, there is no need to keep the whole matrix M in memory: at iteration j, it suffices to keep in memory only columns j-1 and j, that consist of $|P|$ bits each.

The shift-and method with errors

The shift-and method can be easily extended to solve the k-difference pattern matching problem with $O(k|P||T|)$ bit operations. This algorithm is known as “agrep” and is extremely efficient, in practice, again for relatively short patterns.

Let us first see how it can be extended to solve the k-mismatches pattern matching problem. We generalise matrix M to encode partial occurrences of P in T with up to k mismatches.

For any h from 1 to k, M^h is a $|P| \times |T|$ binary matrix, such that $M^h[i,j]=1$ if and only if the first i characters of P match the i characters of T ending at position j with at most h mismatches. Otherwise $M^h[i,j]=0$.

If $M^k[|P|,j]=1$ then there is an occurrence of P ending at position j in T with up to k mismatches.

The shift-and method with errors

We compute all matrixes M^h column-by-column, for h increasing from 1 to k. We compute column j of all the matrixes before computing column j+1 in any of them.

After initialising the first column of all matrixes to all zeros or to a one followed by all zeros, depending on h and on whether P[1]=T[1], we compute the j-th column of M^h from the (j-1)-th column of M^h and M^{h-1} and from the j-th column of M^{h-1} as follows.

$$M^h[:,j] = \underbrace{M^{h-1}[:,j]}_{\substack{\text{P[1..i] matches i} \\ \text{chars up to T[j] with} \\ \text{up to h-1 mismatches}}} \text{ OR } \underbrace{\{\text{bit-shift}(M^h[:,j-1]) \text{ AND } U(T[j])\}}_{\substack{\text{P[1..i-1] matches i-1 chars} \\ \text{up to T[j-1] with up to h} \\ \text{mismatches and P[i]=T[j]}}} \text{ OR } \underbrace{M^{h-1}[:,j-1]}_{\substack{\text{P[1..i-1] matches i-1} \\ \text{chars up to T[j-1] with} \\ \text{up to h-1 mismatches}}}$$