Note: Any solution to an algorithm design question MUST contain the following

four sections:

(1) Problem statement. A clear unambiguous statement of the problem to

be solved, which includes the input, the output, and the object function

with the constraints.

(2) Algorithm description. A clear, unambiguous description of the algo-

rithm.

(3) Correctness proof. A convincing mathematical argument that the algo-

rithm described solves the computational problem described.

(4) Time analysis. A time analysis of the algorithm, up to order, in terms of

all relevant parameters.

You may use any algorithms and data structures from class.

1. Ukkonen’s algorithm

(I) Formalize the pseudocode for the Ukkonen’s algorithm for constructing the

suffix tree of a given string in linear time. (II) Draw the implicit suffix tree and

show the list of rules used for each phase (i + 1) and each extension (j) to construct

the suffix tree for string “xabxababxba” by using the Ukkonen’s algorithm. (50%)

Sources:

<https://www.youtube.com/watch?v=1A6zrh7mfzg&ab_channel=Keida>

https://www.youtube.com/watch?v=aPRqocoBsFQ&t=3068s

D. Gusfield. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1997 (Dan)

Three rules to building the suffix tree:

1) add character to the end of existing path

2) create a path if it doesn’t already and append the character to it

3) if the path already exists, do nothing

Ukkonen’s algorithm tricks:

1) skip/count and edge label compression

2) rule 3 extension is a show stopper (stop the current phase and start the next phase)

3) global end for all leaves

active points and rule three extension:

if you’re moving on the same edge then just increment your active length by 1

if you’re jumping the internal node then reset your active edge, active node, and active length variables

When stuck at a phase, increment the active edge by 1 and decrement the active length by 1 IFF active node is root. Otherwise we follow the suffix link of the current active node.

Ukkonen’s algorithm is an effective algorithm for building an implicit suffix tree in linear time complexity. The algorithm is composed of phases such that for a string of length m, there are m phases. In each phase, we would take into account the three rules one by one, if applicable. Rule 1 would apply to every existing branch since we can use a global end reference that is updated with every phase. Rule 2 applies when the path from the active point doesn’t exist, so it needs to be created. Rule 3 applies when the path already exists from the active point, and in that case we would need to update the active length and the active edge variables. However, if we jump and internal suffix node doing so, then we would update the active node to that internal node and the active length and active edge variables accordingly.

high-level pseudocode:

Make tree T

For i from 1 to m-1:

# begin phase i+1

increment remainingSuffixes and

# extend j times (rule 1)

find the end of every path from root to leaf nodes and append the current character.

# rule 2

If it is the first encounter of this character, then create a new path of this character to the active node, and decrement the remainingSuffixes variable

# rule 3

If the character can be found along the path, then update the active point to that character, stop the current phase and start the next

implicit suffix tree:

Each row shows what the variables are at the end of phase i.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Phase | Remaining  Suffixes | End | Active  node | Active edge | Active length | Implicit suffix tree at the end of current phase |
| 0 | 0 | 0 | root | -1 | 0 |  |
| 1 | 0 | 1 | root | -1 | 0 |  |
| 2 | 0 | 2 | root | -1 | 0 |  |
| 3 | 1 | 3 | root | 0 | 1 |  |
| 4 | 2 | 4 | root | 0 | 2 |  |
| 5 | 3 | 5 | root | 0 | 3 |  |
| 6 | 1 | 6 | Root | 1 | 1 |  |
| 7 | 2 | 7 | Root | 1 | 2 |  |
| 8 | 3 | 8 | B | 3 | 1 |  |
| 9 | 4 | 9 | B | 3 | 2 |  |
| 10 | 5 | 10 | B | 3 | 3 |  |

2. Suffix tree for large alphabet

When introducing the Ukkonen’s algorithm for suffix tree constructing, we as-

sume a constant size of the alphabet. If we assume the alphabet size |a| is compa-

rable to the length of the input string n, there is a trivial low bound O(n log n) for

applying Ukkonen’s algorithm. Describe a simple algorithm to achieve this lower

bound. (25%)

Since Ukkonnen’s algorithm takes linear time to build a suffix tree from a given string, we can query search on the alphabet size |a| in the suffix tree, which would take O(log n) time since it’s a tree-like structure. Together we would have a lower bound of O(n log n).

3. Peptide vaccine design

The activation of helper T-cells is essential to initiate a protective immune re-

sponse. To mimic pathogen invasion, biologists synthesize peptide vaccines, i.e.

small peptides of the essential proteins from a pathogen (bacterium or virus) that

can be recognized by the major histocompatibility complex (MHC) and presented

to the helper T-cells. A simple version of the peptide vaccine design problem can

be formulated as the shortest unique substring problem, which attempts to find the

shortest peptide in the proteins of the pathogen (called pathogen proteins) that are

not a part of any protein from the host (human) (called host proteins). (25%)

Given the fact that every substring is always a prefix of some suffix of the string. We could use Ukkonen’s algorithm to build a suffix tree from the host proteins and then generate every substring from the pathogen proteins from shortest to longest in length and check if that substring is in the suffix tree. We can stop checking prematurely if the current substring not found. The worst case runtime would arise when we go thru all the subtrings; in other words, a scenario where the pathogen proteins exactly match that of the host.

High level pseudo-code:

host strings h

pathogen strings p

Make suffix tree T out of host strings h

Make list of every possible substrings L out of pathogen strings p from shortest to longest in length

for each substring in L:

if substring is not in the suffix tree T:

return substring

# since all substrings were in the tree, so the answer is undefined

return null

We can also perform the same algorithm but this time make the tree out of pathogen strings and check whether any of the substrings of the host strings are in the pathogen suffix tree.

Sources:

<https://www.youtube.com/watch?v=1A6zrh7mfzg&ab_channel=Keida>