Homework 2

Due: Thursday, Nov 8, 8:30pm

Problem 1: Probability (15 points)

Consider that you have a box of two types of dice: 90% are type I and 10% are type II. Both types of dice are loaded. Type I dice have 10% of chance of rolling a six, and type II dice have 50% chance of rolling a six. You randomly pick a die from the box and roll once. Then you put the die back into the box and randomly pick one again.

(1) What is the probablity that you will see a six in your first roll?

According to the theorem of total probablity,

$$\begin{aligned} P(six) &= P(six|I) \times P(I) + P(six|II) \times P(II) \\ &= 0.1 \times 0.9 + 0.5 \times 0.1 \\ &= 0.14 \end{aligned}$$

(2) If you observed a six, what is the probability that the die is of type I?

According to Bayes theorem,

$$\begin{aligned} P(I|six) &= \frac{P(six|I) \times P(I)}{P(six)} \\ &= \frac{0.1 \times 0.9}{0.14} \\ &= 0.643 \end{aligned}$$

(3) What is the probability that you will see two six in a row?

Because the two events are indepdent,

$$\begin{aligned} P(six, six) &= P(six) \times P(six) \\ &= 0.14^2 \\ &= 0.0196 \end{aligned}$$

(In fact, when I was asking this question, I was thinking about the theorem of total probability:

$$P(six, six) = \sum_{a,b \in (I,II)} P(six, six|a, b) \times P(a, b),$$

which is related to problem 2(2) below.)

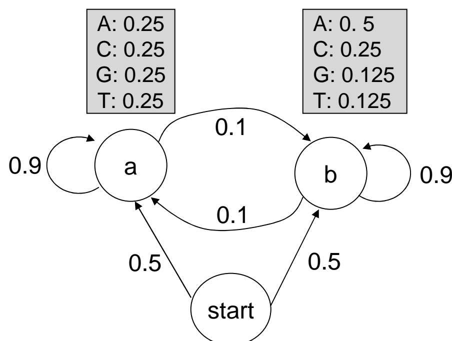
(4) If you observed two six in a row, what is the probability that both dice are of type I?

According to Bayes theorem,

$$\begin{aligned} P(I, I|six, six) &= \frac{P(six, six|I, I) \times P(I, I)}{P(six, six)} \\ &= \frac{P(six|I)^2 \times P(I)^2}{P(six, six)} \\ &= \frac{0.1^2 \times 0.9^2}{0.0196} \\ &= 0.413 \end{aligned}$$

Problem 2: Hidden Markov Model (15 points)

Consider the hidden Markov model below with given transition and emission parameters.



(1) What is the most probable state path for a sequence AACT? What is the probability of that path?

(2) What is the probability of the sequence AACT (considering all possible paths)?

Hint: It's best to use Viterbi and Forward algorithms to answer the two questions, although not absolutely necessary (it would be necessary if I give you a sequence of length 10).

Solution sketch: There are two ways to solve this problem. Each symbol can be in one of the two states, a or b. Therefore, there are only $2^4 = 16$ possible state paths, i.e., aaaa, aaab, ..., bbbb. It is easy to compute the probablity of each state path by multiplying the transition and emission probabilities along the path. To answer question (1), you can pick the path with the largest probability. The best path is bbbb, and the corresponding probablity is

$$\begin{aligned} P(AACT, bbbb) &= P(start->b) \times P(b->b)^3 \times P(A|b) \times P(A|b) \times P(C|b) \times P(T|b) \\ &= 0.5 \times 0.9^3 \times 0.5^2 \times 0.25 \times 0.125 \\ &= 0.0028 \end{aligned}$$

Similarly, you can compute P(AACT, aaaa), P(AACT, aaab), etc. To answer question (2), you can simply add the prabilities of all 16 paths together. The answer is P(AACT) = 0.0063.

Of course, this method is not very efficient, because there are 2^n number of possible state paths, given a string of length n. Using the Viterbi and Forward algorithms, the you only need $O(2n)$ instead of $O(2^n)$ computations for each question.

Problem 3: Boyer-Moore (20 points)

(1) Implement the extended bad character rule for the Boyer-Moore algorithm. Assume that your pattern and text strings contain only upper-case letters (so you can directly compute indices for each char from its ascii code subtracted by 64. In ascii code A is 65, B is 66, etc.).

- The syntax of your program should look like this:
  ./BM ⟨patternString⟩ ⟨textFileName⟩ .
- If your input text file contains multiple lines, concatenate them into a single string before you do the matching.
- Output the starting position of each occurrence of patternString in your text file.
- Count separately the number of character comparisons and the number of steps needed to find the next matching character using the bad character rule.

(2) Implement the naive string matching algorithm. Input and output formats should be similar to the one above.

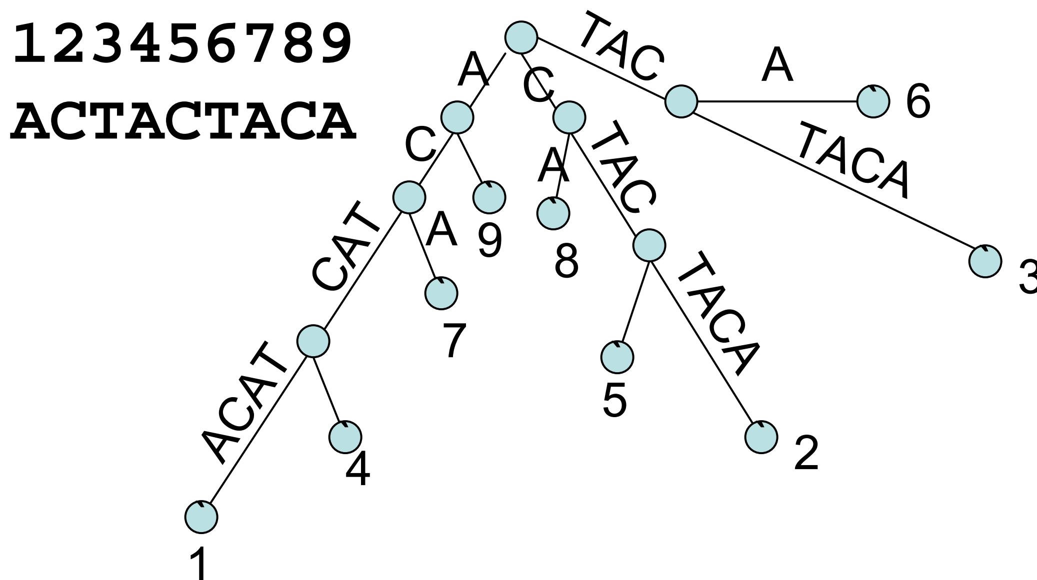
(3) Download two text files from the course website. One file contains a famous novel, "Moby Dick", with spaces and punctuation marks removed. The other file contains 1M random DNA sequences. Create several patterns of your choice and find them in the files with your programs. Does BM actually show some advantage? For what kind of patterns BM worked the best and for what kind of patterns BM did not do so well? Why?

(4) Bonus (10 points) for implementing the good suffix rule. You can use a naive algorithm to do the preprocessing. (May not worth the points. But I give you an opportunity if you desire to implement it anyway. I just did and it took me about 3 hours.). Similar to last time, if you have done a perfect job for the other problems in this assignment, the bonus points will be wasted.

Note: Turn in your source code, experimental results on a few (less than 10) patterns, and some discussion. Also email me your source code. Don't print out patterns that appear more than 10 times in the text. It would be a waste of paper.

Problem 4: Suffix Tree (20 points)

(1) Draw a suffix tree for a string ACTACTACA. Label the edges and terminal nodes.

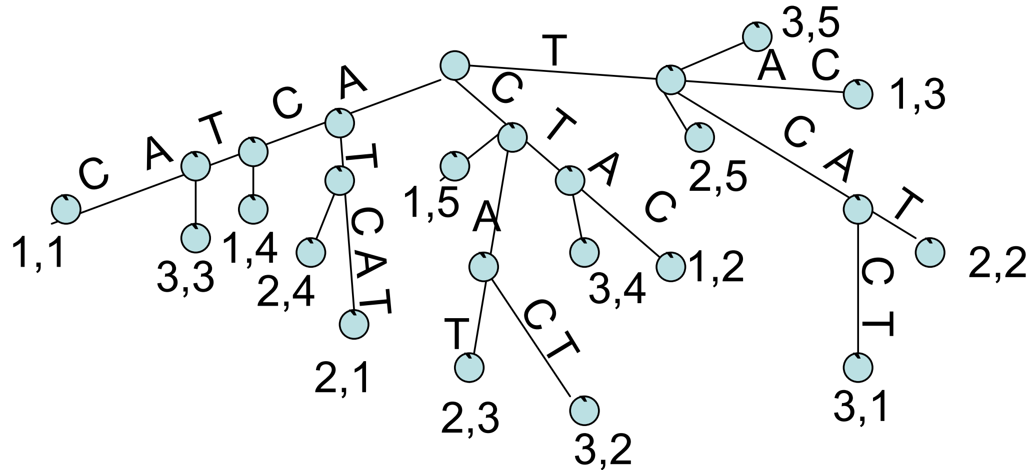


(2) Draw a joint suffix tree for three strings ACTAC, ATCAT, TCACT. Label the edges and terminal nodes.

12345 12345 12345

ACTAC ATCAT TCACT

1 2 3



(3) Design an efficient algorithm for finding the shortest nonrepeated string in a text, that is, a shortest string that appears in the text only once.

This problem can be solved with the help of a suffix tree data structure. There are two key observations: (1) each internal node i encodes a substring s_i, and the number of leaf nodes descending from i is equal to the number of times s_i appears in the string; (2) each non-empty leaf node (i.e., one that is connected to an internal node by a non-empty edge) encodes a suffix that appears only once in the string.

From observation (1), we know that the substrings encoded by any internal node is not unique, since each internal node has at least two children. From observation (2), we know that each suffix is potentially unique, as long as it is not contained entirely within another suffix. Furthermore, a substring formed by removing some trailing characters of the suffix may still be unique, as long as the shortened substring is not contained within another suffix.

Finding the shortest non-repreated substring in a string can be achieved with the following procedure:

(a) Construct a suffix tree for the string.

(b) Label each internal node with the length of the substring it encods.

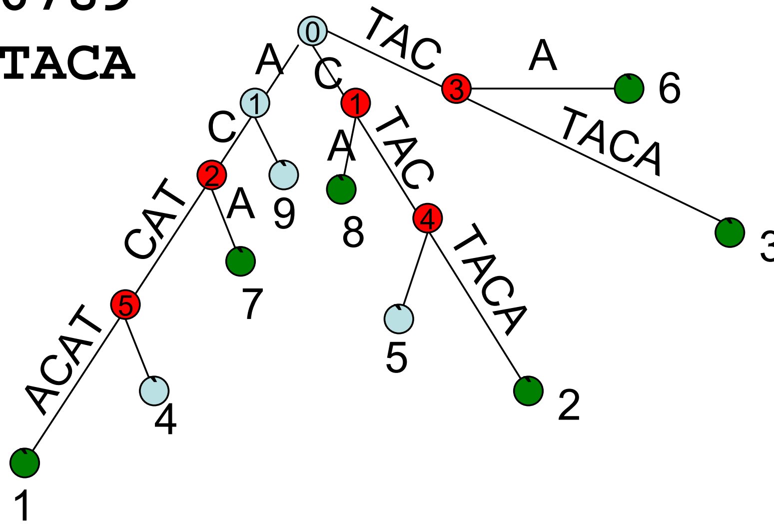
(c) Find all internal nodes with at least one non-empty leaf node. In the figure below, those internal nodes and their non-empty leaf nodes are colored in red and green, respectively.

(d) Among all the red nodes, find the one labeled with the samllest integer (computed above in (b)).

(e) Return the substring encoded by the red node found in (d), plus the first character extending into its green child node.

123456789

ACTACTACA



The algorithm runs in $O(n)$ time for a string of length n, since it takes linear time to construct the suffix tree, and there are at most $2n$ nodes in the suffix tree.

The above procedure can be made more efficient. In steps (b) and (c), instead of labeling all nodes, we only need to label the first red / green nodes encounted on each path during a tree traversal, since a node that is further down the path must have longer labels. Also a breadth-first or best-first tree traversal might work better than a depth-first tree traversal. However, those improvement will not change the asympotic linear running time.

(4) Design an efficient algorithm to find the minimum l for a set of strings $T_1, T_2, \ldots, T_k$, such that there exist a unique "signature" substring of length l for each string. For example, if T_1 = ACGACGTA, T_2 = ACTATGAC, and T_3 = GATAGTA, the smallest $l = 2$, since a signature of length 2 can be found for each string: CG only appears in T_1, CT only in T_2 and AG only in T_3.

The algorithm in its essence is similar to the algorthm above, except that we need to construct a joint suffix tree instead of a suffix tree. Some crucial points are:

(a) We want a unique substring from each string.

(b) A substring that occurred two or more times within one string but never in the other strings are considered "unique".

(c) We would like the "signature" sequences for different strings to have equal lengths.

To take (b) into consideration, we need to modify the above algorithm a little bit. In the previous discussion, we have mentioned that each non-empty leaf node in a suffix tree encodes to a unique substring. This observation is still valid in a joint suffix tree. Furthermore, an internal node may encode a substring that is unique to one string, if the node's descendant leaf nodes all come from the same string. Therefore, the first step in the algorithm is to mark those unique substrings. This can be done easily with a post-order tree traversal. As shown in the figure below, we colored the unique substrings from strings 1, 2 and 3 with three different colors (green, orange, and blue, respectively). Note the color for the internal node encoding "AT".

For (a), we can find the shortest unique substring for each string separately. Or equivalently, during tree traversal, we can use a vector to remember the currently shortest unique substring for each string. For example, in the joint suffix tree below, the shortest unique substring for the three strings are "TA", "AT", and "CAC", respectively, which can be found by considering the green, orange, and blue nodes separately.

For (c), we note that if a substring is unique, extending the substring to its left or right by any number of characters will result in another unique substring. Therefore, the simplest solution is to compute the

maximum length after finding the shortest unique substrings for all the strings, and then extend the shorter ones to the maximum length. This may involve determining the exact location of a substring in the original string, and figuring out to which direction the substring is extendable. Alternatively, when visiting a tree node, we may remember the shortest unique substring as well as the longest one, and compute an intersection of the ranges when comparing results from different branches.

12345 12345 12345

ACTAC ATCAT TCACT

1 2 3

Bonus (5 points)

How much time did you spent? Who did you discuss with and what was the discussion about? Any comments about the course and homeworks? What topic would you like to learn more about? What topic you didn't enjoy so much?