DFAs

DFA stands for deterministic finite automaton. Basically, it is a directed graph where each node has edges coming out of it labeled with the letters from a fixed finite alphabet Σ. One node is designated as the *start node*, and a set of nodes are designated as *final nodes*. If $|\Sigma| = k$, then each node has k distinctly labeled outgoing edges, one for each letter in Σ. The strings accepted by a DFA are the ones corresponding to walks from the start state to a final state following the correctly labeled edges.

DFA Exercises

1. Give a DFA for $\Sigma = \{0, 1\}$ and strings that have an odd number of 1's and any number of 0's.

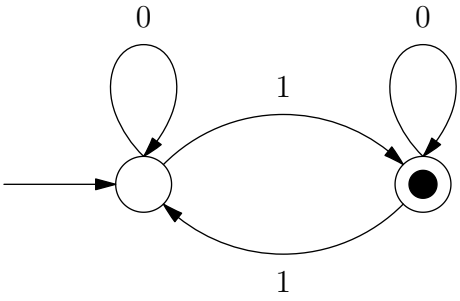
2. Give a DFA for $\Sigma = \{a, b, c\}$ that accepts any string with aab as a substring.

3. Give a DFA for $\Sigma = \{a, b\}$ that accepts any string with $aababb$ as a substring.

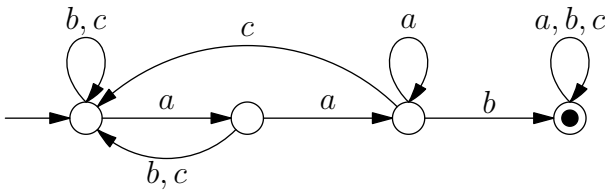
4. Suppose you are given many texts (strings) $T_1, \ldots, T_n$ and one pattern string P. You want to determine which texts have the pattern P as a substring. What is the total runtime of doing this using DFAs?

Solutions

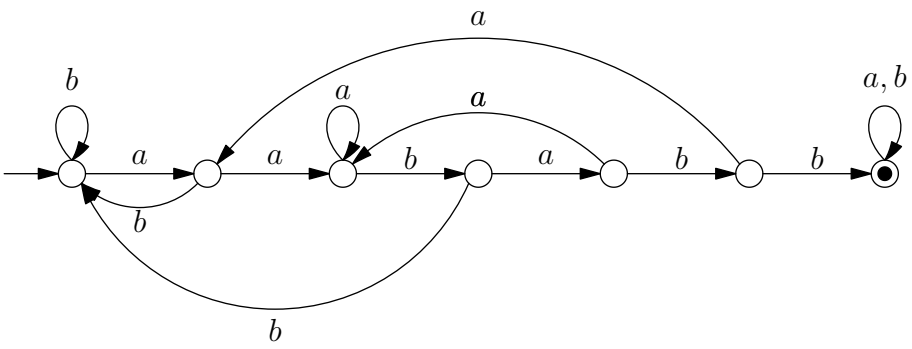
1.



2.



3.



4. Once the DFA is built, the total runtime is $O(\sum T_i)$. The question is how long does it take to construct the DFA? We will see a particularly efficient way in the next section giving an $O(|P|)$ runtime for DFA construction.

DFAs and Substring Matching

As we saw above, once we have a DFA, we can quickly determine whether a given pattern P is a substring of a text T. The issue is building the DFA. We now present an algorithm (Knuth-Morris-Pratt) for quickly building and compactly storing the required DFA, and then an algorithm (actually the same algorithm) for simulating our compactly stored DFA.

The idea is to build an array whose entries correspond to each entry of our DFA. Each array entry points back to an earlier index in the array. Recall from the above exercises that each state of the DFA corresponds to how much progress we have made toward matching the pattern P. When we get a letter that doesn't make further progress, we must determine which state we land on. Going back to our array, assume each array entry corresponds to some prefix z of our pattern string (i.e., the prefix of the pattern we matched thus far). Then the array entry will refer to the index of the longest pattern prefix which is a proper suffix of z. Using DP we will build this array in $|P|$ time. Using this array we can simulate the DFA on a text T in $O(|T|)$ time. This gives us an $O(|T|)$ substring detection algorithm with $O(|P|)$ preprocessing time.

KMP Exercises

1. For the last DFA (*aababb*) in the previous group of exercises, construct the corresponding KMP table.
2. Build the corresponding KMP table for the string *abcaabc*.
3. Show how to build the above array in $O(|P|)$ time.
4. Show how to simulate the corresponding DFA in $O(|T|)$ time.
5. Show how to find all occurrences of a pattern P in a text T using the KMP algorithm.

6. Given a string z find the shortest string x such that $z = x^k$ for some $k > 0$. That is, z is x repeated k times.

Solutions

1. The table is

index	0	1	2	3	4	5	6
value	0	0	1	0	1	0	0

2. The table is

index	0	1	2	3	4	5	6	7
value	0	0	0	0	1	1	2	3

3.

```
public static int[] buildKMPTable(String pattern)
{
  int[] table = new int[pattern.length()+1];
  for (int i = 2; i < table.length; ++i)
  {
    int j = table[i-1];
    do
    {
      if (pattern.charAt(j) == pattern.charAt(i-1))
      { table[i] = j+1; break;}
      else j = table[j];
    } while (j != 0);
  }
  return table;
}
```

4.

```
public static int simulate(int[] table, String pattern, String
    text)
{
  int state = 0;
  for (int i = 0; i < pattern.length(); ++i)
  {
    while (true)
    {
      if (pattern.charAt(i) == text.charAt(state))
      {
        state++;
```

```
            break;
          }
          else if (state == 0) break;
          state = table[state];
        }
      }
      return state;
    }
```

5. Look for the pattern $P\$$ in the string T and determine how many times you arrive at the next-to-last state.

6. Using KMP you can find the suffixes of P that are also prefixes. Suppose $P = vw$ and $P = wx$ where v, w, x are strings. That is, w is both a suffix and a prefix of P. If $|v|$ divides $|P|$ then $P = v^{|P|/|v|}$. Looping through the KMP table, find the shortest such v.

Tries, Suffix Tries, and Suffix Trees

A trie is a tree data structure for holding strings. You can sort of think of it as a particular type of DFA for a set of strings, with certain edges omitted. To find a word, you traverse the edges corresponding to it. The word is in the trie if you can traverse the edges corresponding to the word and end on a final state. Each node of the trie correponds to a prefix of one of the stored words.

A suffix trie for a string x is a trie made from all possible suffixes of x. Usually, we want every leaf to correspond to a different suffix, so we append an extra character \$ to the end that is not used in the string. Each node of the suffix trie corresponds to a prefix of a suffix, i.e., a substring.

A suffix trie is a nice data structure, but it is too big ($O(|x|^2)$). The problem is that many nodes in the tree may have 1 child. To deal with this, we compress the suffix trie into a suffix tree by allowing edges to have strings as labels instead of just letters. The label of each edge is denoted by an interval of indices referring to the original string. We use this to create a tree where every internal (non-leaf) node has at least 2 children.

As we will see, a suffix tree is an incredibly useful data structure allowing us to solve many useful (particularly in bioinformatics) problems in linear time. The real challenge is constructing a suffix tree. Knuth conjectured that constructing a suffix tree in linear time would be impossible as it would allow the longest common substring problem to be solved in linear time. He was later proved incorrect by Weiner who gave the first linear suffix tree algorithm in 1973. The code I will be giving out uses Ukkonen's algorithm.

It is also possible to build a data structure called a generalized suffix tree for a set of strings $\{x_1, \ldots, x_n\}$. To do this we build a suffix tree for the concatenated string $x_1\$_1x_2\$_2 \cdots x_n\$_n$, where the $\$_i$ are all distinct and do not occur in any of the x_i. The prefix corresponding to any internal node of the suffix tree cannot have a $\$_i$ in it, as a $\$_i$ uniquely determines

your position in the string, and hence could not lead to a node with at least 2 children. To determine what suffix a leaf node corresponds to, simply find the first $\$_i$ that occurs in the edge that leads to it, and ignore all characters following it. The resulting tree has size $O(\sum |x_i|)$.

Suffix Tree Exercises

1. Suppose you have a tree T where every non-leaf node has at least 2 children. If there are n leaves give an upper bound on the total size of the tree.

2. Draw the suffix tree for the string "aababb$".

3. Given a suffix tree for the text $T\$$ and a pattern string P, show how to find all occurrences of P in $O(|P| + \text{occ})$ time.

4. Given a suffix tree for the string $x\$$ and an integer k, show how to find all maximum length substrings s that occur at least k times in x

   (a) If overlapping is allowed.

   (b) ($\star$) If overlapping is not allowed.

5. Given a suffix tree for the string $x\$$, find the smallest string z such that $x = z^k$ for some $k \geq 1$.

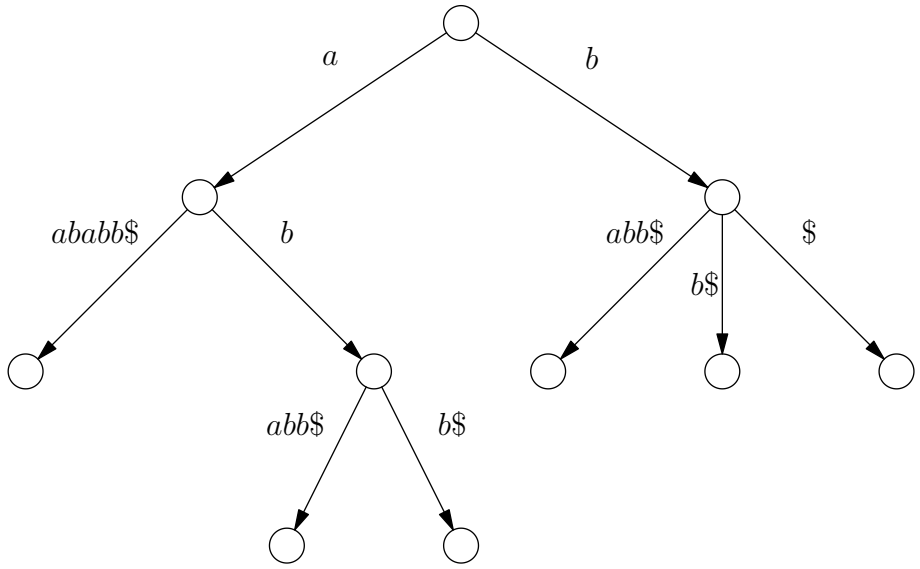
6. Given a generalized suffix tree for the strings $x_1, \ldots, x_n$, find all substrings of maximum length that are common to every string.

7. Given a list of strings $x_1, \ldots, x_n$, compute for every k the length of the longest string common to k of the x_i.

Solutions

1. There are $O(|n|)$ nodes in the tree (a complete binary tree is the worst case).

2.



3. Travel edges according to the letters in P. You will either not be able to (0 occurrences), you will end at a node, or you will end in the middle of an edge. The number of leaves below your position in the tree is the number of occurrences. This can be counted with a DFS.

4. (a) First do a DFS and compute the length of the maximum length substring that occurs at least k times. To do this, consider all nodes with at least k leaves below them, and take the maximum length of all prefixes that lead to them. Then do a second DFS, and output all of the prefixes that have this computed length.

   (b) Do a DFS. At each node, you then have access to all of the leaves that lie below it. Each leaf gives you an index range of the corresponding suffix. This allows you to compute the index ranges of all occurrences of the substring x corresponding to the node. Sort these index ranges, and greedily choose a maximum disjoint set of intervals. These give you the number of non-overlapping substrings. Use this to find the answer, as in the previous part.

5. As before, this reduces to finding all suffixes that are prefixes. To do this, simply traverse the path of the suffix tree corresponding to the string itself. If at any point you reach a node that has a \$ edge coming out of it, that marks the end of a suffix. Hence you have found a prefix that is a suffix.

6. Build the generalized suffix tree. For each node, determine how many leaves below it correspond to distinct x_is. Consider every node that has n distinct leaves below it.

7. The same as above, but only look for k distinct leaves.