Algorithmic Problem Solving

Le 8 – Strings I

Fredrik Heintz
Dept of Computer and Information Science
Linköping University

Outline

- String Matching (lab 3.1)
- String Multi-Matching (lab 3.2)
- DP over Strings
- Trie
- The Substring Problem

String Matching

Given a text string *T* with *n* characters and a pattern string *P* with *m* characters, find all occurrences of *P* in *T*.

- Easiest solution: Use string library (C++ string::find, C strstr, Java String.indexOf))
- Knuth-Morris-Pratt: O(n+m) time and O(m) space (lab 3.1)
- Boyer-Moore: also O(n+m) time and O(m) space, but more efficient when the alphabet is large or the pattern is long since it matches from right to left
- More efficient solutions exist, as we will see…

String Multi-Matching

Given a text string T (with n) characters and pattern strings $P_1,\ldots,P_p$, find all occurrences of every pattern P_i in T.

- The Aho-Corasick algorithm finds all matches of strings $P_1, \ldots, P_p$ in T in $O(n+m+k)$ time and $O(n)$ space, where $m=\sum|P_i|$ and k is the total number of matches (lab 3.2)

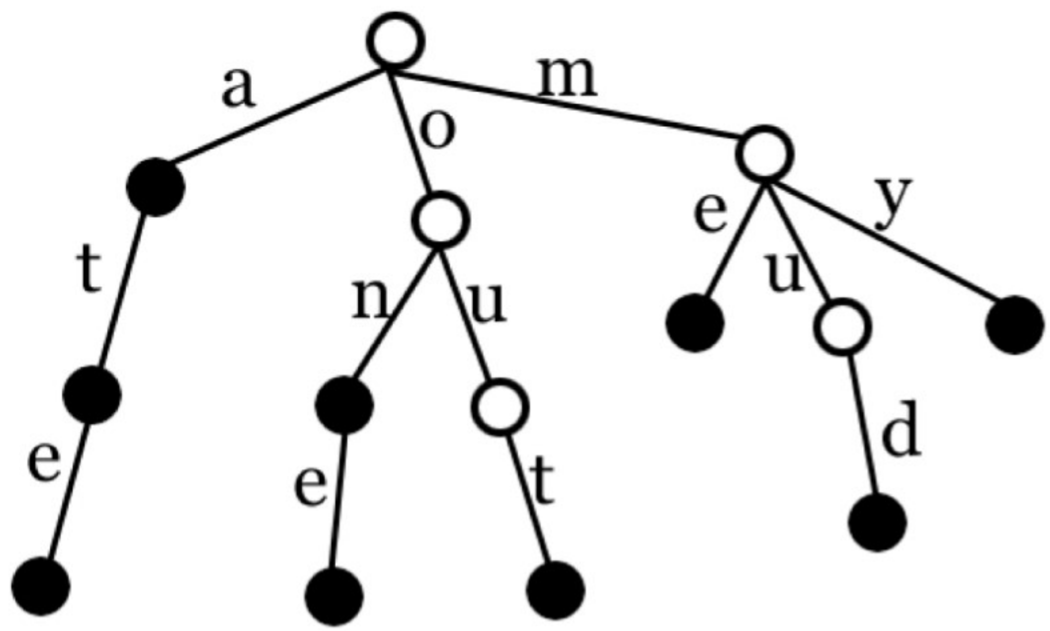
DP over Strings

- The *edit distance* between strings S1 and S2 is the *minimum number* of operations I (insert the next char of S2), D (delete), R (replace by the next char of S2) that transforms S1 into S2 (also known as the Levenshtein distance)
  - Define D(i, j) to be the edit distance of prefixes S1[1…i] and S2[1…j], then D(n, m) is the edit distance of S1 and S2.
  - Define D(i, j) = min(D(i–1, j)+1, D(i, j–1)+1), D(i–1, j–1)+t(i,j)), where t(i,j) = 0 if S1[i] =S2[j] else 1.
  - DP computation of D(n,m) is O(nm).
- We can also consider edit operations with *weights*: *d* for deletion/insertion, *r* for substitution, and *e* for match. Edit distance is then a special case with *d*=*r*=1 and *e*=0.
  - The Hamming distance is also a special case, with *d*=∞, *r*=1, and *e*=0. (Minimization)
  - Longest Common Subsequence is also a special case, with *d*=0, *r*=–∞, and *e*=1. (Maximization)

Trie (Prefix Tree)

Trie: An ordered tree structure used for storing a set of data, usually strings, optimized for doing prefix searches

- Example: Does any word in the set start with the prefix `mart`?
- The idea: use a "26-ary" tree
  - each node has 26 children: one for each letter A-Z
  - add a word to the trie by following the appropriate child pointer



The Substring Problem

The substring problem: For a text S of length n, after $O(n)$ time preprocessing, given any string P either find an occurrence of P in S, or determine that one does not exist in time $O(|P|)$

- Build a trie of all substring of S, $O(n^2)$.
- It is easy to find *prefixes* of string in a trie.
- Each substring $S[i\ldots j]$ is a prefix of the suffix $S[i\ldots n]$ of S.
- Therefore, create a trie of the n non-empty suffixes of S.
- This can be done in $O(n)$ time.

Summary

- String Matching
- String Multi-Matching
- DP over Strings
- Trie
- The Substring Problem