Advanced Algorithmic Problem Solving

Le 3 – Strings

Fredrik Heintz
Dept of Computer and Information Science
Linköping University

Outline

- String matching – Knuth-Morris-Pratt (Lab 1.6)
- DP over strings – Edit distance (UVA 11151)
- Trie (UVA 644)
- String multi matching – Aho-Corasick (Lab 1.7)
- Suffix Trie/Tree/Array (Lab 1.8)

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)
    - C++ string::find is $O(nm)$ worst case execution time but doesn't use any extra memory and works well on strings without many partial matches.
  - Knuth-Morris-Pratt ($O(n+m)$ time and $O(m)$ space)
  - Boyer-Moore is 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 exists, as we will see…

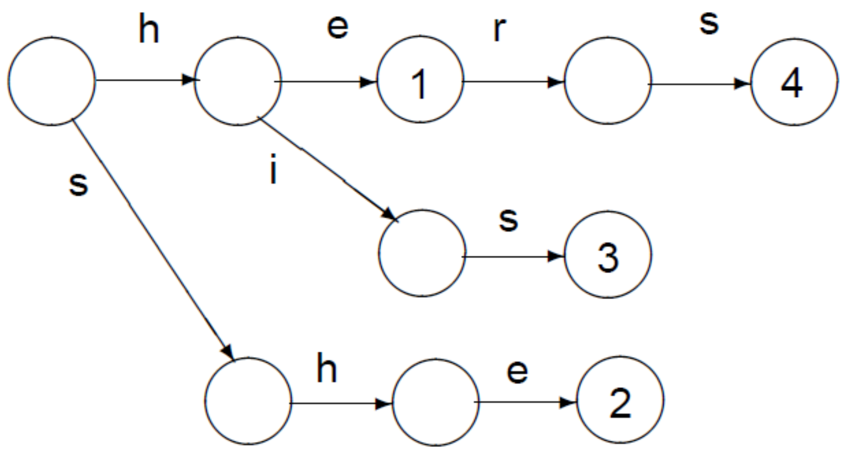
DP on Strings – Edit Distance

- 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.
  - 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[1…j] else 1. DP computation of D(n,m) is in *O(nm)*.
- We can also consider edit operations with *weights* (or *costs* or *scores*): *d* for deletion/insertion, *r* for substitution, and *e* for match. Edit distance is a special case with *d*=*r*=1 and *e*=0.
- The Hamming distance is also a special case.
  - What values of *d*, *r* and *e*? (min, d=oo, r=1, e=0)
- The Longest Common Subsequence is also a special case.
  - What values of *d*, *r* and *e*? (max, d=0, r=-oo, e=1)

UVA 11151

Trie (or keyword tree)

- A **Trie** (or a **keyword tree**) for a set of strings P is a rooted tree K such that
  - each edge of K is labeled by a character
  - any two edges out of a node have different labels
- Define the **label of a node** v as the concatenation of edge labels on the path from the root to v, and denote it by L(v)
  - for each p ∈ P there is a node v with L(v) = p, and
  - the label L(v) of any *leaf* v equals some p ∈ P
- An example trie for P={he, she, his, hers}



String Multi Matching – Aho-Corasick

- Given a text string T and pattern strings P_1, …, P_p, find all occurrences of every pattern P_i in T.
- The **Aho-Corasick algorithm** finds all matches of strings P_1, …, P_p in a string T in $O(n+m+k)$ time and $O(n)$ space, where $n=|T|$, $m=\sum|P_i|$ and k is the total number of matches.

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

Suffix Trie and Suffix Tree

- A **Suffix Trie** is a Trie with suffixes.
- A **Suffix Tree** for S[1…n] is a rooted tree with
  - n leaves numbered 1…n
  - at least two children for each internal node (with the root as a possible exception)
  - each edge labeled by a non-empty *substring* of S
  - no two edges out of a node beginning with the same character
  - Suffix Trees can be generalized to index multiple strings S_1, …, S_k
- Suffix Trees allow linear time algorithms for
  - Exact matching in *O(n+occ)*, where *occ* is the number of matches
  - Longest Repeating Substring in *O(n)*
  - Longest Common Substring in *O(n)*

Suffix Array

- Suffix Trees are space inefficient ($O(nb \log n)$ bits) and hard to implement
- A **Suffix Array** is an array that stores:
  - A permutation of **n** indices of sorted suffixes
  - Each integer takes $O(\log n)$ bits, so a Suffix Array takes $O(n \log n)$ bits
- Suffix Arrays allow efficient algorithms for
  - Exact matching in $O(m \log n)$
  - Longest Repeating Substring in $O(n)$
  - Longest Common Substring in $O(n)$

Summary

- String matching with Knuth-Morris-Pratt (Lab 1.6)
  - Finds all matches of a string P in a string T in $O(n+m)$ time and $O(n)$ space, where $n=|T|$ and $m=|P|$
- String multi matching with Aho-Corasick (Lab 1.7)
  - Finds all matches of strings $P_1, \ldots, P_p$ in a string T in $O(n+m+k)$ time and $O(n)$ space, where $n=|T|$, $m=\sum|P_i|$ and k is the total number of matches
- Suffix Tree
  - Exact matching in $O(n+occ)$, where occ is the number of matches
  - Longest Repeating Substring in $O(n)$
  - Longest Common Substring in $O(n)$
- Suffix Array (Lab 1.8)
  - Exact matching in $O(m \log n)$
  - Longest Repeating Substring in $O(n)$
  - Longest Common Substring in $O(n)$
- DP over strings is common. The Weighted Edit Distance can be computed in $O(nm)$ using DP with Edit Distance, Longest Common Subsequence, Hamming Distance and more as special cases.