String algorithms II
Prof. Manolis Kellis

Lecture 12

String algorithms

• Last time: Exact string matching
  – Naïve algorithm
  – Fundamental pre-processing
    • Knuth-Morris-Pratt / Boyer-Moore / Z-algorithm
  – Semi-numerical string matching
    • Rabin-Karp algorithm
• Today: String matching II
  – Suffix-trees
  – Linear time construction
  – Applications
• Recitation:
  – More on Suffix Trees
  – Finite State Machines
  – Regular Expression Matching

Where have we gotten so far?

• Last time
  – Fundamental preprocessing in linear time
  – Searching for pattern p in linear time: O(Text)
• Today’s challenge: Can we do better?
  – Searching for any pattern p in linear time O(pattern)
  – After pre-processing the text once

More involved pre-processing step

• Fundamental pre-processing only searched for:
  – Common prefix / suffix at any position
  – Redundancy with beginning/end of string
• Suffix trees
  – Redundancy across all substrings
    • starting at every position
    • over the remainder of the list
• Example:
  – Suffix tree of xabxac

Suffix tree definition

• Definition: Suffix tree T for string S (of length n)
  – Rooted, directed tree T, n leaves, numbered 1..n
  – Path to leaf i spells out the suffix S[i..], by concatenating edge labels
  – Common prefixes share common paths, diverge to form internal nodes
  – Effectively exhibit common prefixes of every suffix
  – Explores full substring redundancy structure of S

Exact string matching with suffix trees

• Given the suffix tree for text T
• Search pattern P in O(pattern) time
  – For every character in P, traverse the appropriate path of the tree, reading one character each time
  – If P is not found in a path, P does not occur in T
  – If P is found in its entirety, then all occurrences of P in T are exactly the children of that node
  – Every child corresponds to exactly one occurrence
  – Simply list each of the leaf indices

Suffix Tree Construction

- Naïve algorithm

```
xa

1 x

ab

2 a

xa

3 b

xb

4 a

ac

5 c

6 c
```

- Running time: $O(n^2)$

Rules for suffix tree extension

- Extension rule #1: add a character to the end of an edge label
- Extension rule #2: add a new branch and internal node
- Extension rule #3: do nothing

High-level description

- UKKONEN-ALGORITHM
  - Construct $I_1$
  - for $i$ in $[1..m-1]$
    - [begin phase $i+1$]
      - for $j$ in $[1..i+1]$
        - [begin extension $j+1$]
          - Find the end of the path labeled $S[j..i]$ in current tree (call it $\beta$)
          - if [rule 1]: $\beta$ ends in a leaf: [rule #1]
            - then add $S(i+1)$ to that edge label
          - if [rule2]: $\beta$ ends in middle of edge
            - then add an internal node and new branch
          - if [rule3]: $S(i+1)$ already exists
            - then do nothing
    - [end extension $j+1$]
  - [end phase $i+1$]
  - [end for $i$]

- Running time: $O(n^3)$

Can we do better?

Speedup the search for $\beta$: Use suffix links

- Ukkonen’s algorithm (with suffix links)

```
xa

1 x

ab

2 a

xa

3 b

xb

4 a

ac

5 c

6 c
```

- Running time: $O(n^2)$?

Ensure we only back up by one node at each iteration

- Not yet, but almost there!

Link between internal nodes. From $c' + \alpha \rightarrow \alpha$

- How does this help?
  - We ensure we only backtrack by 1 node, before traversing $(x,s(v))$
  - Node-depth decreases by another 1 node, during traversal
    - since $c' \alpha$ is a substring of $\alpha$
    - all branches are identical
  - Thus, over an entire phase, at most $O(n)$ increases
  - max node-depth is $O(n)$, bounded by the number of characters
  - at most $2^n$ decrements $\rightarrow$ at most $3^n$ increments

- Must ensure constant traversal time per node!
Constant-time node traversal

- Fixed alphabet allows constant-time lookup for edge
  - Translate character to index
- Keeping track of label length L on each edge
  - Jump to next node, and skip over L characters in γ

Summary: Linear time for each phase

- O(n) node traversals over the entire phase
  - No more than 2n decreases in node-depth
    - By using v → s(v) links, avoid backtracking to root
  - Longest depth is n (length of string)
- O(n) node traversals per phase
  - Constant time per node traversal
    - Fixed alphabet
    - Indexing of branches to follow at every node
    - Known length to traverse for every edge
    - No need for explicit character comparisons
- O(n) time over the entire phase
  - O(n^2) time over the entire construction

Can we do better?

Three more ideas

(a) Edge representation
  - Use (start,end) indexing to S → space reduction

(b) Once a skip, always a skip: rule #3 terminates entire phase
  - If prefix is already included → entire extension already included
  - When rule #3 applies once, it applies for entire phase

(c) Once a leaf, always a leaf: rule #1 work done centrally
  - Special symbol for end of S → centralized work
  - Series of extensions in start of each phase → constant work

Reducing space usage: Edge-label compression

- (1,3) (2,3)
- (4,6) (10,12)

Constant space for every node. O(n) space for entire tree

Reducing the work for rule #3

- Extension rule #1: add a character to the end of an edge label
- Extension rule #2: add a new branch and internal node
- Extension rule #3: do nothing (path already exists)

If rule #3 applies to extension j, then it applies to all j' > j,
   since the entire suffix S[j..i] is already included in the tree
   → Rule #3 terminates an entire phase

Example:

Phase j=5

<table>
<thead>
<tr>
<th>j</th>
<th>Phase</th>
<th>Rule #3 applies</th>
<th>do work</th>
<th>if 'a' already exists:</th>
<th>'a' already exists</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>do work</td>
<td>'b' already exists</td>
<td>'b' already exists</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>do work</td>
<td>'b' already exists</td>
<td>'b' already exists</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>3</td>
<td>do work</td>
<td>'b' already exists</td>
<td>'b' already exists</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>3</td>
<td>rule #3 applies</td>
<td>b a x a x a</td>
<td>b a x a x a</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>3</td>
<td>skip</td>
<td>b a x a x a</td>
<td>b a x a x a</td>
</tr>
</tbody>
</table>

Reducing the work for rule #1

- Extension rule #1: add a character to the end of an edge label

- Once a leaf always a leaf:
  - If rule #1 applies for character j at some phase, it applies for j in all subsequent phases
  - Initial stretch of rule #1 applications at beginning of each phase
  - Centralize this work
    - Create "current end" variable e, updated centrally at start of each phase, extending all leaves
    - Start explicit work after this initial stretch (at "j" where rule #3 first applied in previous phase)

- Implicit work
  - Implicit work: a, a, a, a (already includes a)
  - Implicit work: a, a, a, a (already includes a)
  - Implicit work: a, a, a, a (already includes a)
  - Implicit work: a, a, a, a (already includes a)
  - Start explicit work here: x a x a (already includes a)
  - End explicit work here: x a x a (already includes a)

- Skipped (after rule #3) implicit extensions

Can we do better?
Amortized constant cost per phase!

Phase 1
- 1 2 3 4 5 6 7 8
- Phase 1 completed

Phase 2
- 9 10 11
- Centralized rule #1 extension
- Phase 2 completed

Phase 3
- 11 12 13 14 15 16
- Centralized rule #1 extension
- Phase 3 completed

Phase 4
- 16 17
- Centralized rule #1 extension

- Summary: three smart ideas
  - (a) Constant space per edge (start,end) variables
  - (b) Once a skip, always a skip
  - (c) Once a leaf, always a leaf

⇒ Constant amortized cost per phase
⇒ Linear cost for entire algorithm!

Putting it all together

- **UKKONEN-ALGORITHM**
  - Construct $I_1$
  - for $i$ in $[1..m-1]$
    - Increment end variable $e$ [centralized rule #1] (a)
    - for $j$ in $[i+1..m]$
      - [begin explicit work where last rule #3 was applied (c)]
        - Use link $s(v)\rightarrow s(v)$ to find $S[j..i]$ in current tree (call it $\beta$) (suffix-links)
        - if [rule1]: $\beta$ ends on a leaf
          - then do nothing [rule #1 is now centralized] (a)
        - if [rule2]: $\beta$ ends in middle of edge
          - then add an internal node and new branch
        - if [rule3]: $S(i+1)$ already exists
          - then end entire phase, set $j^*$ (b)

Applications of suffix trees

- Exact string matching
- Dictionary lookup for one word
- Search of multiple strings for one pattern
- Longest common substring problem
- Common substrings of more than two strings
- Longest common extension
- Inexact string matching