Algorithmic Problem Solving

Le 7 Strings I

Herman Appelgren
Dept of Computer and Information Science
Linköping University

Outline

- ~~Exercise 7: Graph III~~
    - ~~Hiding Places~~
    - ~~Get Shorty~~
    - ~~XYZZY~~
    - ~~Risk~~
- Introduction to String Processing
- Trie (Prefix Tree)
- The String Matching Problem

Introduction

- Most (all?) modern programming languages have built-in string utilities.
  - Basic tasks (replace, split, find character, …)
  - Creating strings (<sstream>, StringBuilder, str.format)
  - Regular expression (<regex>, Pattern, re)
- Why write your own string algorithms?
  - The built-in features might not support you problem (efficiently).
  - Better worst-case time complexity.
  - Improve efficiency when processing many strings.
  - Process general sequential data.

Introduction

- Built-in hash functions for strings are often very good. Use Hash Tables whenever the internal structure of the strings aren’t required.
  - Example: Replace words with integer IDs, process the IDs, lookup in table when the word is required.
  - Example: Check if a string is found in a dictionary.
- Note that the time complexity must often include the string length.
  - Hash function is in O(S), and so is insertion/lookup in hash table.
  - Comparison of strings are in O(S), so insertion/lookup in a BST is in O(SlogN).

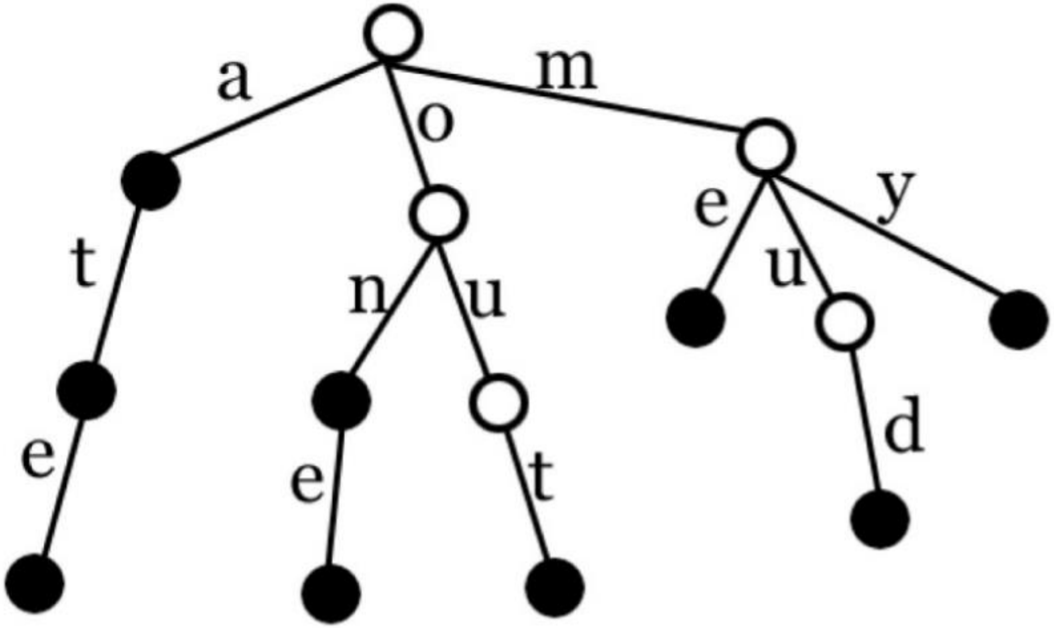
Suffixes and Prefixes

- **Suffixes** are substrings at the end of the string.
  - Example: “banana” (not proper), "anana", "nana", "ana", "na", "a".
- **Prefixes** are substring at the start of the string.
  - Example: “banana” (not proper), "banan", "bana", "ban", "ba", "b".
- Every substring of S is a prefix of some suffix of S.
  - Example: "nan" is a prefix of "nana", which is a suffix of "banana".

Trie (Prefix Tree)

- A Trie is a rooted tree structure used for storing a set of strings and optimize prefix searches.
  - Essentially an 26-ary tree. One node for each prefix. Each node has 26 children, one for each character A-Z.
  - Example: “me”, “are”, “my”, “out”, “one”, “mean”.
  - Time complexity: O(S) where S is the total string length.
  - Memory complexity: O(S), but may include a large constant.
- Can be used to answer many string-related questions. Typically in O(length of the query string).
  - Efficient dictionary representation.
  - Which strings from a set has a given prefix? (auto-complete)
  - Is any string in a set the prefix of another string in the same set? (phone numbers)
  - String Multimatching using the Aho-Corasick Automaton (no longer part of the course)

Trie (Prefix Tree)

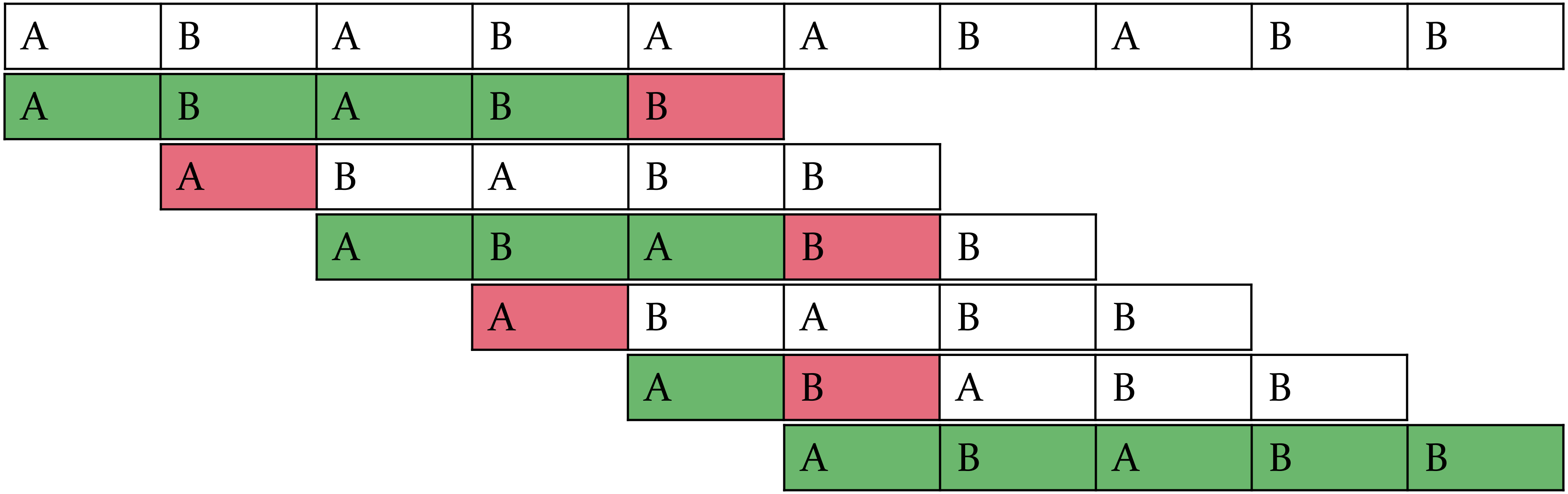


The String Matching Problem

- **Problem:** Find all occurrences of the pattern P in the text S.
- String libraries (e.g. string::find, String.indexOf) often use naïve algorithm:

  -
    ```
    for i in 1..|S|-|P|:
      match := true
      for j in 1..|P|:
        if S[i + j - 1] != P[j]:
          match := false
          break
      if match return i
    ```
  - Works well in practice. However, worst-case time complexity O(SP), e.g. if S = “aaa…” and P = “aaa…ab”. Can we do better?

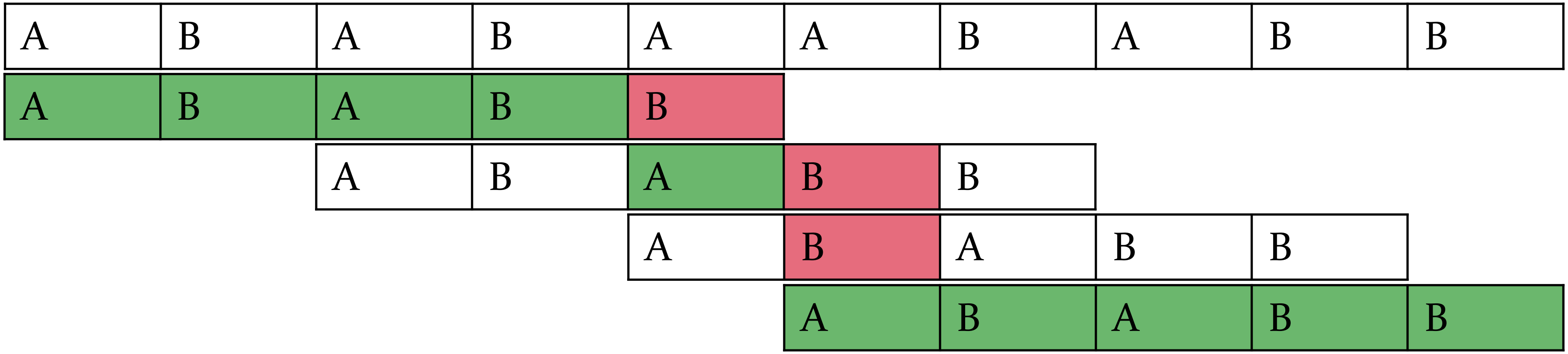
Naive String Matching



A	B	A	B	A	A	B	A	B	B
A	B	A	B	B					
	A	B	A	B	B				
		A	B	A	B	B			
			A	B	A	B	B		
				A	B	A	B	B	
					A	B	A	B	B

- Observation 1: We don’t need to recheck 1..2 in iteration 3.
- Observation 2: Iteration 2 and 4 aren’t needed at all.
- Solution: Whenever there is a mismatch, shift the pattern to the longest prefix of the pattern that is a suffix of the partial match.

Knuth-Morris-Pratt



A	B	A	B	A	A	B	A	B	B
A	B	A	B	B					
		A	B	A	B	B			
				A	B	A	B	B	
					A	B	A	B	B

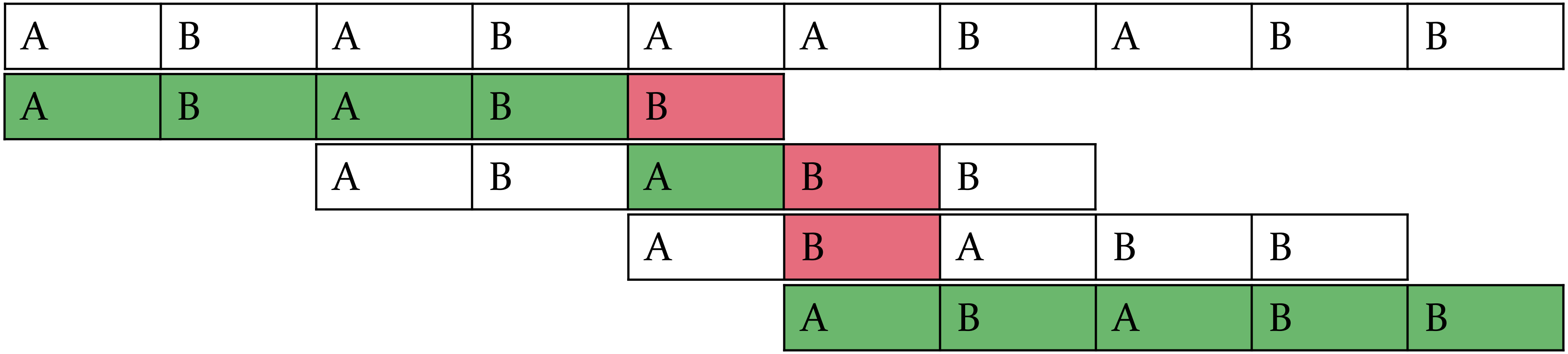
- Observation 3: The shifts only depend on P, not S => Can be precomputed for each position in P.
- This forms the base for the **Knuth-Morris-Pratt** string matching algorithm (Lab 3.1).

Knuth-Morris-Pratt

- Precompute the *prefix function* in O(P).
  - prefix[i] = Length of longest prefix of P[1 .. i] that is also a (proper) suffix of P[1 .. i], i.e. largest j < i such that P[1 .. j] == P[i – j + 1 .. i].
  - Trivially, prefix[1] = 0.
  - For general prefix[i], note that if P[i] == P[prefix[i – 1] + 1] then prefix[i] = prefix[i – 1] + 1. Otherwise repeat with two calls to prefix etc. until a match is found or we have exhausted the pattern.
  - Time complexity: O(P)
- Match against the test.
  - Use the prefix function whenever a mismatch is found, and when you match the entire pattern.
  - Time complexity: O(S)

i	1	2	3	4	5
P[i]	A	B	A	B	B
prefix[i]	0	0	1	2	0

Knuth-Morris-Pratt



A	B	A	B	A	A	B	A	B	B
A	B	A	B	B					
		A	B	A	B	B			
				A	B	A	B	B	
					A	B	A	B	B

i	1	2	3	4	5
P[i]	A	B	A	B	B
prefix[i]	0	0	1	2	0

- Visualization tool: cmps-people.ok.ubc.ca/ylucet/DS/KnuthMorrisPratt.html
- CP-algorithms: https://cp-algorithms.com/string/prefix-function.html

Other String Matching Algorithms

- **Knuth-Morris-Pratt *(lab 3.1)***
  - O(|S|+|P|) time, O(|P|) space.
- **Boyer-Moore**
  - O(|S|+|P|) time, O(|P|) space.
  - More efficient than KMP when the alphabet is large.
- **Aho-Corasick Automaton**
  - String Multimatching, i.e. multiple patterns.
  - O(|S| + |P|) time, O(|P|) space.
  - Generalizes the prefix function to a trie, using the same ideas as KMP.
- **Suffix Array**
  - See next lecture.
  - O(|P|log(|S|)) time assuming a suffix array of S is available. Not worthwhile unless you need the suffix array for another purpose.

Summary

- Exercise 7: Graph III
  - Hiding Places
  - Get Shorty
  - XYZZY
  - Risk
- Introduction to String Processing
- Trie (Prefix Tree)
- The String Matching Problem