Lecture 19,20: String Matching Algorithms / Tries

BT 3051 – Data Structures and Algorithms for Biology

Karthik Raman
Department of Biotechnology
Indian Institute of Technology Madras

Knuth–Morris–Pratt Algorithm

Knuth–Morris–Pratt Algorithm

- What is a major inefficiency of Brute-Force/Boyer-Moore (in the worst cases)?
- For a certain alignment of the pattern, if we find several matching characters but then detect a mismatch, we ignore all the information gained by the successful comparisons after restarting with the next incremental placement of the pattern!
- *What can we do instead?*

Knuth–Morris–Pratt Algorithm

- What is a major inefficiency of Brute-Force/Boyer-Moore (in the worst cases)?
- For a certain alignment of the pattern, if we find several matching characters but then detect a mismatch, we ignore all the information gained by the successful comparisons after restarting with the next incremental placement of the pattern!
- *What can we do instead?*

Knuth–Morris–Pratt Algorithm

- What is a major inefficiency of Brute-Force/Boyer-Moore (in the worst cases)?
- For a certain alignment of the pattern, if we find several matching characters but then detect a mismatch, we ignore all the information gained by the successful comparisons after restarting with the next incremental placement of the pattern!
- *What can we do instead?*

Knuth–Morris–Pratt Algorithm

- Consider a partial match

$$\begin{array}{ccccccc} a_1 & a_2 & \ldots & a_{j-r+1} & \ldots & a_j & a_{j+1} \\ & & & = & = & = & \neq \\ & & & b_1 & \ldots & b_r & b_{r+1} \end{array}$$

- In the next step, we will want to align

$$\begin{array}{ccccccc} a_1 & a_2 & \ldots & a_{j-r+1} & \ldots & a_j & a_{j+1} \\ & & & = & = & = & \neq \\ & & & b_1 & \ldots & b_r & b_{r+1} \\ & & & & ? & ? & ? \\ & & & & b_1 & \ldots b_{r-1} & b_r \end{array}$$

Knuth–Morris–Pratt Algorithm

- Consider a partial match

$$\begin{array}{ccccccc} a_1 & a_2 & \dots & a_{j-r+1} & \dots & a_j & a_{j+1} \\ & & & = & = & = & \neq \\ & & & b_1 & \dots & b_r & b_{r+1} \end{array}$$

- In the next step, we will want to align

$$\begin{array}{ccccccc} a_1 & a_2 & \dots & a_{j-r+1} & \dots & a_j & a_{j+1} \\ & & & = & = & = & \neq \\ & & & b_1 & \dots & b_r & b_{r+1} \\ & & & & ? & ? & ? \\ & & & & b_1 & \dots b_{r-1} & b_r \end{array}$$

Knuth–Morris–Pratt Algorithm

- But, this information comes actually from the pattern itself, rather than the text!
- Can we intelligently pre-process the pattern, so that we can appropriately restart matching?
- We need to figure "what is the longest proper suffix of the matched string, which is also its prefix?"

Knuth–Morris–Pratt Algorithm

- But, this information comes actually from the pattern itself, rather than the text!
- Can we intelligently pre-process the pattern, so that we can appropriately restart matching?
- We need to figure "what is the longest proper suffix of the matched string, which is also its prefix?"

Knuth–Morris–Pratt Algorithm

- But, this information comes actually from the pattern itself, rather than the text!
- Can we intelligently pre-process the pattern, so that we can appropriately restart matching?
- We need to figure "what is the longest proper suffix of the matched string, which is also its prefix?"

Knuth–Morris–Pratt

Complexity

- $O(n)$ for the search phase
- $O(m)$ for the pre-processing of the pattern
- The complexity analysis is non-trivial

Knuth–Morris–Pratt

Complexity

- $O(n)$ for the search phase
- $O(m)$ for the pre-processing of the pattern
- The complexity analysis is non-trivial

Knuth–Morris–Pratt

Complexity

- $O(n)$ for the search phase
- $O(m)$ for the pre-processing of the pattern
- The complexity analysis is non-trivial

String Matching

Algorithm Design Summary (Skiena)

- Is the search pattern/text short? *Naïve matching*
- Is the search pattern/text very long? *Knuth–Morris–Pratt*
- Do we expect to find the pattern or not? *Boyer–Moore*
- Will we perform multiple queries on the same text? *Suffix trees*
- Will we search many texts using the same pattern? *Complex algorithms …*
- What if the input contains a spelling error? *Approximate string matching*

Querying Multiple Patterns

Genome Sequencing

Challenges

- Fragment (read) assembly
  - Inevitable for new species
  - Computationally challenging/expensive
- Read mapping (Fragment alignment)

Why is this important?

- Huge number of genomes being sequenced
- e.g. Personal Genome Project (UK): 10^5 genomes!

Genome Sequencing

Challenges

- Fragment (read) assembly
  - Inevitable for new species
  - Computationally challenging/expensive
- Read mapping (Fragment alignment)

Why is this important?

- Huge number of genomes being sequenced
- e.g. Personal Genome Project (UK): 10^5 genomes!

Genome Sequencing

Challenges

- Fragment (read) assembly
  - Inevitable for new species
  - Computationally challenging/expensive
- Read mapping (Fragment alignment)

Why is this important?

- Huge number of genomes being sequenced
- e.g. Personal Genome Project (UK): 10^5 genomes!

Genome Sequencing

Challenges

- Fragment (read) assembly
  - Inevitable for new species
  - Computationally challenging/expensive
- Read mapping (Fragment alignment)

Why is this important?

- Huge number of genomes being sequenced
- e.g. Personal Genome Project (UK): 10^5 genomes!

Genome Sequencing

Challenges

- Fragment (read) assembly
  - Inevitable for new species
  - Computationally challenging/expensive
- Read mapping (Fragment alignment)

Why is this important?

- Huge number of genomes being sequenced
- e.g. Personal Genome Project (UK): 10^5 genomes!

Genome Sequencing

Challenges

- Fragment (read) assembly
  - Inevitable for new species
  - Computationally challenging/expensive
- Read mapping (Fragment alignment)

Why is this important?

- Huge number of genomes being sequenced
- e.g. Personal Genome Project (UK): 10^5 genomes!

Genome Sequencing

Challenges

- Fragment (read) assembly
  - Inevitable for new species
  - Computationally challenging/expensive
- Read mapping (Fragment alignment)

Why is this important?

- Huge number of genomes being sequenced
- e.g. Personal Genome Project (UK): 10^5 genomes!

How do we assemble individual genomes efficiently using a reference genome?

Genome Sequencing

Challenges

- Fragment (read) assembly
  - Inevitable for new species
  - Computationally challenging/expensive
- Read mapping (Fragment alignment)

Why is this important?

- Huge number of genomes being sequenced
- e.g. Personal Genome Project (UK): 10^5 genomes!

How do we assemble individual genomes efficiently using a reference genome?

or

How do we match several patterns against a single text!?

Exact Pattern Matching: Multiple Patterns

Problem

Where do each of the reads match the reference genome exactly?

- Input: A collection of strings (reads: *Patterns*) and a string (genome: *Text*)
- Output: All positions in the genome *Text* where a string in *Patterns* appears as a substring

Exact Pattern Matching: Multiple Patterns

Problem

Where do each of the reads match the reference genome exactly?

- Input: A collection of strings (reads: *Patterns*) and a string (genome: *Text*)
- Output: All positions in the genome *Text* where a string in *Patterns* appears as a substring

Exact Pattern Matching: Multiple Patterns

Problem

Where do each of the reads match the reference genome exactly?

- Input: A collection of strings (reads: *Patterns*) and a string (genome: *Text*)
- Output: All positions in the genome *Text* where a string in *Patterns* appears as a substring

Can we store the patterns intelligently, in an efficient data structure?

Tries

Trie

- Let S be a set of s strings from alphabet Σ such that *no string in S is a prefix of another string*
- A standard trie[a] for S is an ordered tree T with the following properties:
  - Each node of T, except the root, is labeled with a character of Σ
  - The children of an internal node of T have distinct labels
  - T has s leaves, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from the root to a leaf v of T yields the string of S associated with v
- Thus, a trie T represents the strings of S with paths from the root to the leaves of T
- Note the importance of assuming that no string in S is a prefix of another string
  - this ensures that each string of S is uniquely associated with a leaf of T

[a] from re*trie*val, pronounced '*try*'

Trie

- Let S be a set of s strings from alphabet Σ such that *no string in S is a prefix of another string*
- A standard trie[a] for S is an ordered tree T with the following properties:
  - Each node of T, except the root, is labeled with a character of Σ
  - The children of an internal node of T have distinct labels
  - T has s leaves, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from the root to a leaf v of T yields the string of S associated with v.
- Thus, a trie T represents the strings of S with paths from the root to the leaves of T
- Note the importance of assuming that no string in S is a prefix of another string
  - this ensures that each string of S is uniquely associated with a leaf of T

[a]from re*trie*val, pronounced '*try*'

Trie

- Let S be a set of s strings from alphabet Σ such that *no string in S is a prefix of another string*
- A standard trie[a] for S is an ordered tree T with the following properties:
  - Each node of T, except the root, is labeled with a character of Σ
  - The children of an internal node of T have distinct labels
  - T has s leaves, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from the root to a leaf v of T yields the string of S associated with v.
- Thus, a trie T represents the strings of S with paths from the root to the leaves of T
- Note the importance of assuming that no string in S is a prefix of another string
  - this ensures that each string of S is uniquely associated with a leaf of T

[a]from re*trie*val, pronounced '*try*'

Trie

- Let S be a set of s strings from alphabet Σ such that *no string in S is a prefix of another string*
- A standard trie[a] for S is an ordered tree T with the following properties:
  - Each node of T, except the root, is labeled with a character of Σ
  - The children of an internal node of T have distinct labels
  - T has s leaves, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from the root to a leaf v of T yields the string of S associated with v.
- Thus, a trie T represents the strings of S with paths from the root to the leaves of T
- Note the importance of assuming that no string in S is a prefix of another string
  - this ensures that each string of S is uniquely associated with a leaf of T

[a]from re*trie*val, pronounced '*try*'

Trie

- Let S be a set of s strings from alphabet Σ such that *no string in S is a prefix of another string*
- A standard trie[a] for S is an ordered tree T with the following properties:
    - Each node of T, except the root, is labeled with a character of Σ
    - The children of an internal node of T have distinct labels
    - T has s leaves, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from the root to a leaf v of T yields the string of S associated with v.
- Thus, a trie T represents the strings of S with paths from the root to the leaves of T
- Note the importance of assuming that no string in S is a prefix of another string
    - this ensures that each string of S is uniquely associated with a leaf of T

[a]from re*trie*val, pronounced '*try*'

Trie

- Let S be a set of s strings from alphabet Σ such that *no string in S is a prefix of another string*
- A standard trie[a] for S is an ordered tree T with the following properties:
  - Each node of T, except the root, is labeled with a character of Σ
  - The children of an internal node of T have distinct labels
  - T has s leaves, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from the root to a leaf v of T yields the string of S associated with v.
- Thus, a trie T represents the strings of S with paths from the root to the leaves of T
- Note the importance of assuming that no string in S is a prefix of another string
  - this ensures that each string of S is uniquely associated with a leaf of T

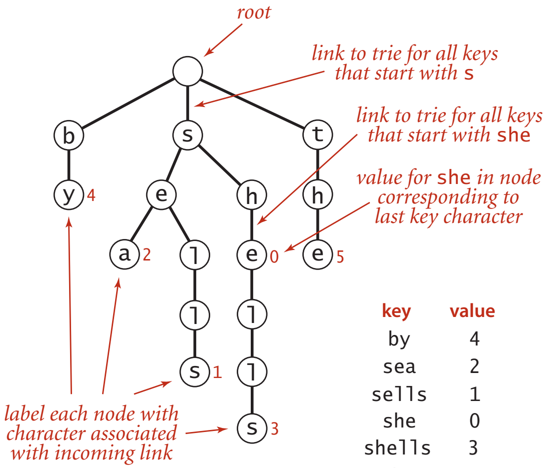
[a]from re*trie*val, pronounced '*try*'

Trie

- Let S be a set of s strings from alphabet Σ such that *no string in S is a prefix of another string*
- A standard trie[a] for S is an ordered tree T with the following properties:
  - Each node of T, except the root, is labeled with a character of Σ
  - The children of an internal node of T have distinct labels
  - T has s leaves, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from the root to a leaf v of T yields the string of S associated with v.
- Thus, a trie T represents the strings of S with paths from the root to the leaves of T
- Note the importance of assuming that no string in S is a prefix of another string
  - this ensures that each string of S is uniquely associated with a leaf of T

[a]from re*trie*val, pronounced '*try*'

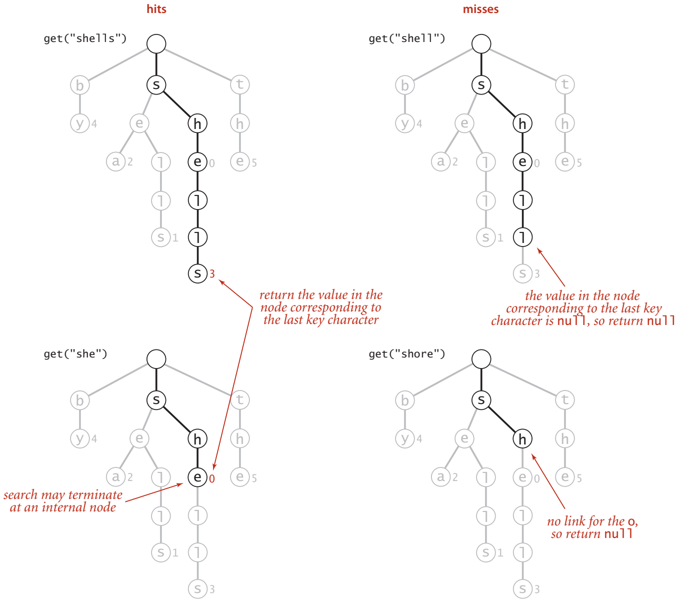
Example Trie
Figure Courtesy: Sedgewick


root
link to trie for all keys that start with s
link to trie for all keys that start with she
value for she in node corresponding to last key character
label each node with character associated with incoming link
b
y 4
s
e
a 2
l
l
s 1
h
e 0
l
l
s 3
t
h
e 5

key	value
by	4
sea	2
sells	1
she	0
shells	3
the	5

Trie Searches

Figure Courtesy: Sedgewick


hits
misses
get("shells")
get("shell")
get("she")
get("shore")
return the value in the node corresponding to the last key character
the value in the node corresponding to the last key character is null, so return null
search may terminate at an internal node
no link for the o, so return null
b
y 4
s
e
a 2
l
l
s 1
h
e 0
l
l
s 3
t
h
e 5

How to match patterns vs. text?

- Slide the trie down the text!
- Search for patterns at every step
- Memory usage is high!

How to match patterns vs. text?

- Slide the trie down the text!
- Search for patterns at every step
- Memory usage is high!

How to match patterns vs. text?

- Slide the trie down the text!
- Search for patterns at every step
- Memory usage is high!

How to match patterns vs. text?

- Slide the trie down the text!
- Search for patterns at every step
- Memory usage is high!

What is the time complexity?