Lecture 19: Text Indexing I

COMP526: Efficient Algorithms

Updated: December 5, 2024

Will Rosenbaum
University of Liverpool

Announcements

1. Quiz 07 on Error Correcting Codes
    - Complete by 11:59pm, Friday 06 November
2. Grading is slow (sorry)
    - Programming assignment 1 grades next week
3. Last lectures:
    - Text indexing (Today and next Tuesday)
    - Final review (next Thursday)
4. Attendance Code:

Meeting Goals

1. Introduce and analyze Parallel MergeSort
2. Introduce the text indexing problem
3. Define the trie data structure
4. Define suffix trees
5. Describe applications of suffix trees

Parallel
MergeSort

Last Time

Parallel Algorithms!

- PRAM model
    - Unlimited parallel processing elements (PEs)
- Brent's Theorem: span T and work W with unlimited PEs

  $\Longrightarrow$ span $O(T + W/p)$ and work $O(W)$ with p PEs
- Parallel string matching with span $T = O(m)$ and work $W = O(n)$
- Sorting networks
    - span $T = O(\log^2 n)$ and work $W = O(n \log^2 n)$
    - limited to specialized hardware and/or small arrays

Parallel Divide & Conquer?

Observation. The Divide & Conquer strategy can lend itself well to parallelism:

1. Divide problem into sub-tasks
2. Solve the subtasks

3. Merge solutions of the subtasks

Parallel Divide & Conquer?

Observation. The Divide & Conquer strategy can lend itself well to parallelism:

1. Divide problem into sub-tasks
2. Solve the subtasks (**independently**)
    - Parallelize these!
3. Merge solutions of the subtasks

Parallel Divide & Conquer?

Observation. The Divide & Conquer strategy can lend itself well to parallelism:

1. Divide problem into sub-tasks
2. Solve the subtasks (**independently**)
    - Parallelize these!
3. Merge solutions of the subtasks (**…?**)
    - How to parallelize this?

Parallel MergeSort?

Revisited: MERGESORT

```
procedure MERGESORT($A, i, k$)
    if $i < k$ then
        $j \leftarrow \lfloor (i+k)/2 \rfloor$
        MERGESORT($A, i, j$)
        MERGESORT($A, j+1, k$)
        $B \leftarrow$ COPY($A, i, j$)
        $C \leftarrow$ COPY($A, j+1, k$)
        MERGE($B, C, A, i$)
    end if
end procedure
```

Parallel MergeSort?

PollEverywhere

What is the span of MergeSort with parallel recursive calls and sequential merges?

pollev.com/comp526

```
procedure MERGESORT($A, i, k$)
    if $i < k$ then
        $j \leftarrow \lfloor (i+k)/2 \rfloor$
        MERGESORT($A, i, j$)
        MERGESORT($A, j+1, k$)
        $B \leftarrow$ COPY($A, i, j$)
        $C \leftarrow$ COPY($A, j+1, k$)
        MERGE($B, C, A, i$)
    end if
end procedure
```

Parallelizing Merges

Question. How can we *parallelize merges?*

Parallelizing Merges

Question. How can we *parallelize merges?*

- For each x, find the final index of x
- How do we find this?

Parallelizing Merges

Question. How can we *parallelize merges?*

- For each x, find the final index of x
- How do we find this?
- index = # elements $\leq x$
    - # in x's sub-array
    - # in other sub-array
- How to compute these?

Parallelizing Merges

Question. How can we *parallelize merges?*

- For each x, find the final index of x
- How do we find this?
- index = # elements $\leq x$
  - # in x's sub-array
  - # in other sub-array
- How to compute these?

Idea.

- In x's own sub-array, just use x's index!

Parallelizing Merges

Question. How can we *parallelize merges?*

- For each x, find the final index of x
- How do we find this?
- index = # elements ≤ x
    - # in x's sub-array
    - # in other sub-array
- How to compute these?

Idea.

- In x's own sub-array, just use x's index!
- For the other sub-array, use binary search!

Parallelizing Merges

Question. How can we *parallelize merges?*

- For each x, find the final index of x
- How do we find this?
- index = # elements ≤ x
  - # in x's sub-array
  - # in other sub-array
- How to compute these?

Idea.

- In x's own sub-array, just use x's index!
- For the other sub-array, use binary search!
- **Parallelize:** do each x in parallel!

Parallel MergeSort in Code

```
procedure ParallelMerge($A[l..m)$, $A[m..r)$, $B$)
    for $i = l, \ldots, m-1$ in parallel do
        $k \leftarrow (i-l) + \textsc{BinarySearch}(A[m..r), A[i])$
        $B[k] \leftarrow A[i]$
    end for
    for $j = m, m+1, \ldots, r-1$ in parallel do
        $k \leftarrow \textsc{BinarySearch}(A[l..m), A[j])$
        $B[k] \leftarrow A[j]$
    end for
end procedure
```

Parallel MergeSort in Code

```
procedure ParallelMerge($A[l..m)$, $A[m..r)$, $B$)
    for $i = l, \ldots, m-1$ in parallel do
        $k \leftarrow (i-l) + \textsc{BinarySearch}(A[m..r), A[i])$
        $B[k] \leftarrow A[i]$
    end for
    for $j = m, m+1, \ldots, r-1$ in parallel do
        $k \leftarrow \textsc{BinarySearch}(A[l..m), A[j])$
        $B[k] \leftarrow A[j]$
    end for
end procedure
```

Questions.

- What is the **span** of ParallelMerge?

Parallel MergeSort in Code

```
procedure ParallelMerge($A[l..m)$, $A[m..r)$, $B$)
    for $i = l, \ldots, m-1$ in parallel do
        $k \leftarrow (i-l) + \textsc{BinarySearch}(A[m..r), A[i])$
        $B[k] \leftarrow A[i]$
    end for
    for $j = m, m+1, \ldots, r-1$ in parallel do
        $k \leftarrow \textsc{BinarySearch}(A[l..m), A[j])$
        $B[k] \leftarrow A[j]$
    end for
end procedure
```

Questions.

- What is the **span** of ParallelMerge?
    - $\Theta(\log n)$
- What is the **work** of ParallelMerge?

Parallel MergeSort in Code

```
procedure ParallelMerge($A[l..m)$, $A[m..r)$, $B$)
    for $i = l, \ldots, m-1$ in parallel do
        $k \leftarrow (i-l) + \text{BinarySearch}(A[m..r), A[i])$
        $B[k] \leftarrow A[i]$
    end for
    for $j = m, m+1, \ldots, r-1$ in parallel do
        $k \leftarrow \text{BinarySearch}(A[l..m), A[j])$
        $B[k] \leftarrow A[j]$
    end for
end procedure
```

Questions.

- What is the **span** of ParallelMerge?
  - $\Theta(\log n)$
- What is the **work** of ParallelMerge?
  - $\Theta(n \log n)$

Parallel MergeSort Analysis

Overall Procedure

1. Split (sub)array in half
2. Parallel recursive MergeSorts
3. PARALLELMERGE sorted halves

Parallel MergeSort Analysis

Overall Procedure

1. Split (sub)array in half
2. Parallel recursive MergeSorts
3. PARALLELMERGE sorted halves

Span Analysis

- Merge has span $\Theta(\log n)$
- Depth of recursion tree is $\Theta(\log n)$
- Total time: $\Theta(\log^2 n)$

Parallel MergeSort Analysis

Overall Procedure

1. Split (sub)array in half
2. Parallel recursive MergeSorts
3. ParallelMerge sorted halves

Span Analysis

- Merge has span $\Theta(\log n)$
- Depth of recursion tree is $\Theta(\log n)$
- Total time: $\Theta(\log^2 n)$

Work Analysis

- Merge has work $\Theta(n \log n)$
- Summing over recursive calls gives $\Theta(n \log^2 n)$

Parallel MergeSort Analysis

Overall Procedure

1. Split (sub)array in half
2. Parallel recursive MergeSorts
3. PARALLELMERGE sorted halves

Span Analysis

- Merge has span $\Theta(\log n)$
- Depth of recursion tree is $\Theta(\log n)$
- Total time: $\Theta(\log^2 n)$

Work Analysis

- Merge has work $\Theta(n \log n)$
- Summing over recursive calls gives $\Theta(n \log^2 n)$

Improvements. Merge can be improved to $\Theta(n)$ work! (but it's complicated)

Concluding Thoughts

Parallelism is Necessary

- Computer hardware is naturally parallel
    - sequential computing is an illusion!

Concluding Thoughts

Parallelism is Necessary

- Computer hardware is naturally parallel
    - sequential computing is an illusion!

Parallelism is Powerful

- Recent explosion in computing power is due to parallelism!

Concluding Thoughts

Parallelism is Necessary

- Computer hardware is naturally parallel
  - sequential computing is an illusion!

Parallelism is Powerful

- Recent explosion in computing power is due to parallelism!

Parallelism is Subtle

- Reasoning about parallel programs is hard
- Writing correct parallel programs is hard
- Idealized models abstract away many challenges
  - no perfect synchrony?
  - tolerate faults?

Text Indexing

Text Indexing

Previously: String Matching.

- Given a text $T[0..n)$ and a pattern $P[0..m)$, determine if/where T contains P
- Focus on *one shot* complexity:
  - how long to search T for a single pattern P

Text Indexing

Previously: String Matching.

- Given a text $T[0..n)$ and a pattern $P[0..m)$, determine if/where T contains P
- Focus on *one shot* complexity:
  - how long to search T for a single pattern P

A Variation. The text T is *fixed*, but we may wish to search T for many different (initially) unknown patterns $P_1, P_2, \ldots$.

- $\Theta(n)$ may be much too much to pay for *each* search
- Applications:
  - web search engines
  - online dictionaries/encyclopedias
  - DNA/RNA databases
  - searching any collection of text documents

Text Indexing

Previously: String Matching.

- Given a text $T[0..n)$ and a pattern $P[0..m)$, determine if/where T contains P
- Focus on *one shot* complexity:
  - how long to search T for a single pattern P

A Variation. The text T is *fixed*, but we may wish to search T for many different (initially) unknown patterns $P_1, P_2, \ldots$.

- $\Theta(n)$ may be much too much to pay for *each* search
- Applications:
  - web search engines
  - online dictionaries/encyclopedias
  - DNA/RNA databases
  - searching any collection of text documents

An Alternative Approach. *Preprocess* the text T to make the searches more efficient

- Pay for preprocessing upfront
- Each query can be *much* more efficient.

Inverted Indices

Example Problem. Given a text T of *words*, implement an *index* of the occurrences of that word.

- Like an index of a textbook
- Only store known words (e.g., whitespace/punctuation separated substrings)

Inverted Indices

Example Problem. Given a text T of *words*, implement an *index* of the occurrences of that word.

- Like an index of a textbook
- Only store known words (e.g., whitespace/punctuation separated substrings)

Goal. Implement an efficient **map** from (possible) keywords P to index of first occurrence (or all occurrences) of P in T (if any)

Inverted Indices

Example Problem. Given a text T of *words*, implement an *index* of the occurrences of that word.

- Like an index of a textbook
- Only store known words (e.g., whitespace/punctuation separated substrings)

Goal. Implement an efficient **map** from (possible) keywords P to index of first occurrence (or all occurrences) of P in T (if any)

- "Easier" than general string matching:
  - Possible (positive) queries are not arbitrary
    - must be a word in the text
  - Keywords are already given (implicitly) in the text

Inverted Indices

Example Problem. Given a text T of *words*, implement an *index* of the occurrences of that word.

- Like an index of a textbook
- Only store known words (e.g., whitespace/punctuation separated substrings)

Goal. Implement an efficient **map** from (possible) keywords P to index of first occurrence (or all occurrences) of P in T (if any)

- "Easier" than general string matching:
  - Possible (positive) queries are not arbitrary
    - must be a word in the text
  - Keywords are already given (implicitly) in the text

Question. How can we implement such a map *efficiently?*

The Trie Data Structure

Idea. Store words in a tree

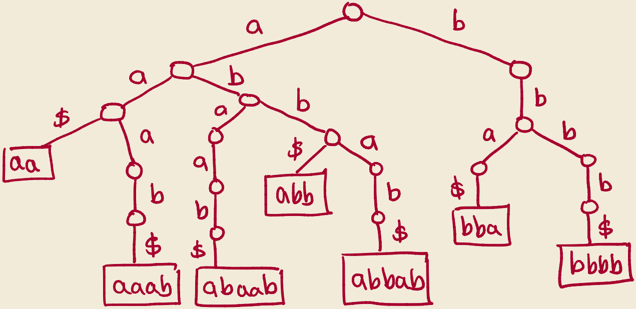
- Each leaf represents a possible word in the text
- Each internal node represents *prefix* of a word in the text
    - path from root to leaf stores letters in the leaf word
- Append a terminating character to each word to make the tree a **prefix tree**

The Trie Data Structure

Idea. Store words in a tree

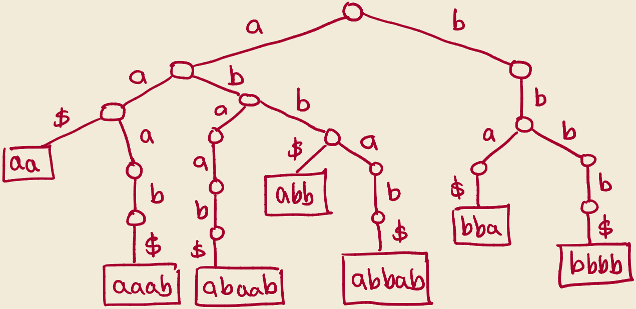
- Each leaf represents a possible word in the text
- Each internal node represents *prefix* of a word in the text
  - path from root to leaf stores letters in the leaf word
- Append a terminating character to each word to make the tree a **prefix tree**

Example: {`aa$, aaab$, abaab$, abb$, abbab$, bba$, bbbb$`}



Searching a Trie

Question. Given a pattern P and a **trie** for the text T, how do we determine if T contains the pattern P?

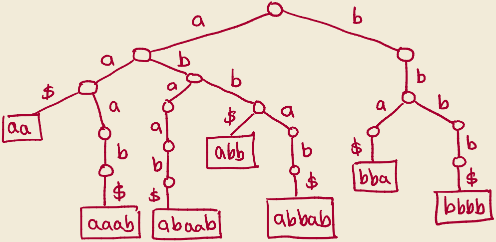


Searching a Trie

Question. Given a pattern P and a **trie** for the text T, how do we determine if T contains the pattern P?

Procedure. Given the pattern $P[0..m)$:

1. Start at the root of the trie
2. Read each character of P, and follow the corresponding edge (if any)
3. If a leaf is reached storing P match is found!
4. If no corresponding edge found or end at an internal node, no mach is found.



Searching a Trie

Question. Given a pattern P and a **trie** for the text T, how do we determine if T contains the pattern P?

Procedure. Given the pattern $P[0..m)$:

1. Start at the root of the trie
2. Read each character of P, and follow the corresponding edge (if any)
3. If a leaf is reached storing P match is found!
4. If no corresponding edge found or end at an internal node, no mach is found.

PollEverywhere

What is the running time of searching a trie?

`pollev.com/comp526`

Searching a Trie

Question. Given a pattern P and a **trie** for the text T, how do we determine if T contains the pattern P?

Procedure. Given the pattern $P[0..m)$:

1. Start at the root of the trie
2. Read each character of P, and follow the corresponding edge (if any)
3. If a leaf is reached storing P match is found!
4. If no corresponding edge found or end at an internal node, no mach is found.

Remarkable fact. The time to search a trie depends only on the length of P, not the size of T!

- Also: the trie can be computed efficiently from T (in $O(n)$ time).

Compact Tries

Observation. Tries are potentially wasteful!

- Can have long paths with no branching
- *Storing* these paths is inefficient

Compact Tries

Observation. Tries are potentially wasteful!

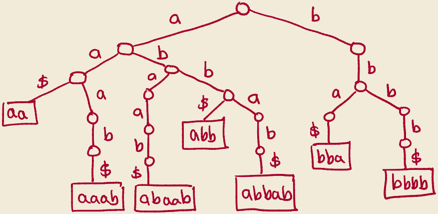
- Can have long paths with no branching
- *Storing* these paths is inefficient

Idea. Compress paths without branches!

- Replace a path of **unary** (single-child) nodes with a single edge
- Label edge with the *first* character of the corresponding path
- Label each *vertex* with the index of the next character

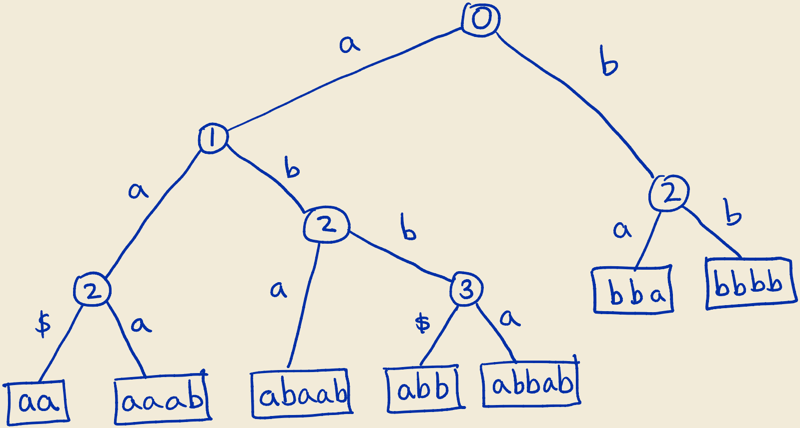
Words, Trie, Compact Trie

Example: {`aa$`, `aaab$`, `abaab$`, `abb$`, `abbab$`, `bba$`, `bbbb$`}



Compact Trie Features

Question. How do we search a *compact* trie?



Compact Trie Features

Question. How do we search a *compact* trie?

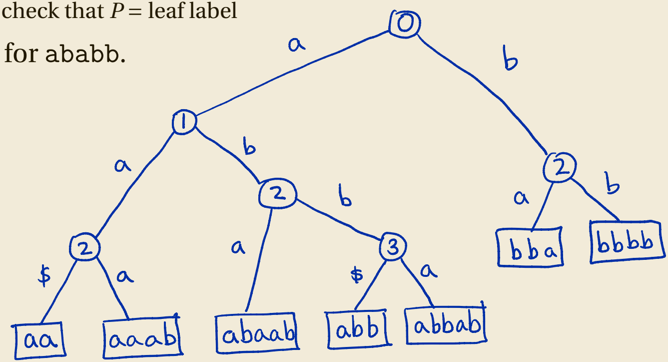
1. Start at the root of the compact trie
2. At node labeled i, follow edge labeled $P[i]$, if any
    - if no such edge exists, search failed
3. If leaf not reached, search failed
4. If leaf reached, check that $P =$ leaf label

Compact Trie Features

Question. How do we search a *compact* trie?

1. Start at the root of the compact trie
2. At node labeled i, follow edge labeled $P[i]$, if any
   - if no such edge exists, search failed
3. If leaf not reached, search failed
4. If leaf reached, check that P = leaf label

Example. Search for `ababb`.



Compact Trie Features

Question. How do we search a *compact* trie?

1. Start at the root of the compact trie
2. At node labeled i, follow edge labeled $P[i]$, if any
    - if no such edge exists, search failed
3. If leaf not reached, search failed
4. If leaf reached, check that $P =$ leaf label

Observation. Searching a compact trie for $P[0..m)$ still takes time $O(m)$.

Compact Trie Features

Question. How do we search a *compact* trie?

1. Start at the root of the compact trie
2. At node labeled i, follow edge labeled $P[i]$, if any
    - if no such edge exists, search failed
3. If leaf not reached, search failed
4. If leaf reached, check that $P =$ leaf label

Observation. Searching a compact trie for $P[0..m)$ still takes time $O(m)$.

Useful feature. If a compact trie stores ℓ words, then it has at most $\ell - 1$ internal nodes as well.

- The size of the trie is proportional to the number of words it stores!
- **Fact** (to prove). If a tree T has ℓ leaves and every internal node has at least two children, then T has at most $2\ell - 1$ vertices.

Trie Discussion

Advantages of tries:

- Simple data structure!
- Space-efficient (compact tries)!
- Fast lookup!

Trie Discussion

Advantages of tries:

- Simple data structure!
- Space-efficient (compact tries)!
- Fast lookup!

Disadvantages:

- Cannot handle more general queries
  - search part of a word
  - search for a phrase (sequence of words)
- Requires the text to be partitioned into words
  - DNA/RNA sequences
  - binary text

Trie Discussion

Advantages of tries:

- Simple data structure!
- Space-efficient (compact tries)!
- Fast lookup!

Disadvantages:

- Cannot handle more general queries
  - search part of a word
  - search for a phrase (sequence of words)
- Requires the text to be partitioned into words
  - DNA/RNA sequences
  - binary text

We need new ideas!!

Suffix Trees

A New Idea

So Far.

- **Goal.** A data structure for efficient pattern matching (and more)
- Compact tries: work for text composed of (designated) words

A New Idea

So Far.

- **Goal.** A data structure for efficient pattern matching (and more)
- Compact tries: work for text composed of (designated) words

Simple Idea. Put every possible word (from T) in a trie!

- For any indices $i < j$, $T[i..j)$ is a possible word
- Add all of them to the trie!

A New Idea

So Far.

- **Goal.** A data structure for efficient pattern matching (and more)
- Compact tries: work for text composed of (designated) words

Simple Idea. Put every possible word (from T) in a trie!

- For any indices $i < j$, $T[i..j)$ is a possible word
- Add all of them to the trie!

The Good.

- Can search for $P[0..m)$ in $O(m)$ time!

A New Idea

So Far.

- **Goal.** A data structure for efficient pattern matching (and more)
- Compact tries: work for text composed of (designated) words

Simple Idea. Put every possible word (from T) in a trie!

- For any indices $i < j$, $T[i..j)$ is a possible word
- Add all of them to the trie!

The Good and the Bad.

- Can search for $P[0..m)$ in $O(m)$ time!
- Must store $\Theta(n^2)$ possible words
- So $\Omega(n^2)$ space, even if a compact trie is used

A New Idea

So Far.

- **Goal.** A data structure for efficient pattern matching (and more)
- Compact tries: work for text composed of (designated) words

Simple Idea. Put every possible word (from T) in a trie!

- For any indices $i < j$, $T[i..j)$ is a possible word
- Add all of them to the trie!

The Good and the Bad.

- Can search for $P[0..m)$ in $O(m)$ time!
- Must store $\Theta(n^2)$ possible words
- So $\Omega(n^2)$ space, even if a compact trie is used

An observation. $P[i, i+1)$, $P[i, i+2)$, $P[i, i+3), \ldots$ can all just be checked against $P[i, n)$

Suffix Trees

Definition. Given a text $T[0..n)$ the **suffix tree** $\mathscr{T}$ of T is formed by:

- take the compact trie of all suffixes of $T\$$ (i.e., all $T_i = T[i..n)\$$)
- **except** replace the leaf label T_i with just the index i
  - must still store T to read from T_i

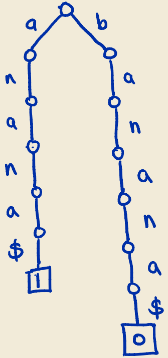
Suffix Trees

Definition. Given a text $T[0..n)$ the **suffix tree** $\mathscr{T}$ of T is formed by:

- take the compact trie of all suffixes of $T\$$ (i.e., all $T_i = T[i..n)\$$)
- **except** replace the leaf label T_i with just the index i
  - must still store T to read from T_i

Example. $T =$ `banana$`

banana$
anana$
nana$
ana$
na$
a$
$



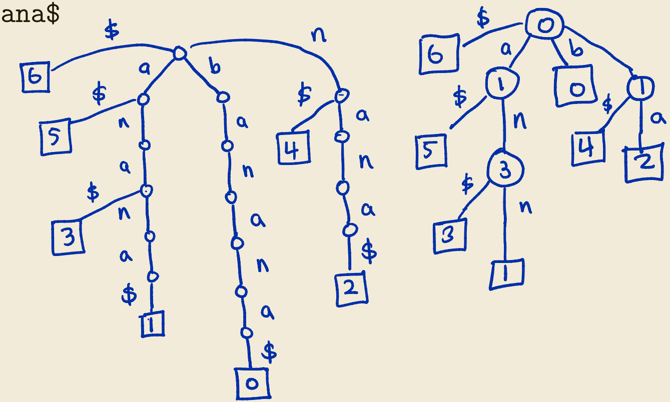
Suffix Trees

Definition. Given a text $T[0..n)$ the **suffix tree** $\mathscr{T}$ of T is formed by:

- take the compact trie of all suffixes of $T\$$ (i.e., all $T_i = T[i..n)\$$)
- **except** replace the leaf label T_i with just the index i
  - must still store T to read from T_i

Example. $T =$ `banana$`

banana\$
anana\$
nana\$
ana\$
na\$
a\$
\$



Suffix Tree Features

Size. Given a text $T[0..n)$, the suffix tree $\mathscr{T}$ has size

Suffix Tree Features

Size. Given a text $T[0..n)$, the suffix tree $\mathscr{T}$ has size

PollEverywhere

Given $T[0..n)$, what is the total size of the associated suffix tree $\mathscr{T}$?

pollev.com/comp526

Suffix Tree Features

Size. Given a text $T[0..n)$, the suffix tree $\mathscr{T}$ has size $\Theta(n)$.

- The size of the suffix tree is only a (small) constant factor larger than T.

Suffix Tree Features

Size. Given a text $T[0..n)$, the suffix tree $\mathscr{T}$ has size $\Theta(n)$.

- The size of the suffix tree is only a (small) constant factor larger than T.

Speed. Given $T[0..n)$, we can compute $\mathscr{T}$ in time

Suffix Tree Features

Size. Given a text $T[0..n)$, the suffix tree $\mathscr{T}$ has size $\Theta(n)$.

- The size of the suffix tree is only a (small) constant factor larger than T.

Speed. Given $T[0..n)$, we can compute $\mathscr{T}$ in time $O(n^2)$ by a “naive” algorithm…

Suffix Tree Features

Size. Given a text $T[0..n)$, the suffix tree $\mathscr{T}$ has size $\Theta(n)$.

- The size of the suffix tree is only a (small) constant factor larger than T.

Speed. Given $T[0..n)$, we can compute $\mathscr{T}$ in time $O(n^2)$ by a "naive" algorithm…

…but $\mathscr{T}$ can be computed in time $O(n)$ by a clever (and practical) algorithm!!!

Suffix Tree Features

Size. Given a text $T[0..n)$, the suffix tree $\mathscr{T}$ has size $\Theta(n)$.

- The size of the suffix tree is only a (small) constant factor larger than T.

Speed. Given $T[0..n)$, we can compute $\mathscr{T}$ in time $O(n^2)$ by a "naive" algorithm…

…but $\mathscr{T}$ can be computed in time $O(n)$ by a clever (and practical) algorithm!!!

- This result is wild, and should be surprising!
- We'll give an overview of the algorithm on Tuesday

Suffix Tree Features

Size. Given a text $T[0..n)$, the suffix tree $\mathscr{T}$ has size $\Theta(n)$.

- The size of the suffix tree is only a (small) constant factor larger than T.

Speed. Given $T[0..n)$, we can compute $\mathscr{T}$ in time $O(n^2)$ by a "naive" algorithm…

…but $\mathscr{T}$ can be computed in time $O(n)$ by a clever (and practical) algorithm!!!

- This result is wild, and should be surprising!
- We'll give an overview of the algorithm on Tuesday

For now. Take it as given that $\mathscr{T}$ can be computed in $O(n)$ time.

Suffix Tree Applications

Application 1: String Matching

Observation. P occurs in $T \iff P$ is a prefix of a suffix of T.

Application 1: String Matching

Observation. P occurs in $T \iff P$ is a prefix of a suffix of T.

- $\mathscr{T}$ stores (references to) all suffixes in T

Application 1: String Matching

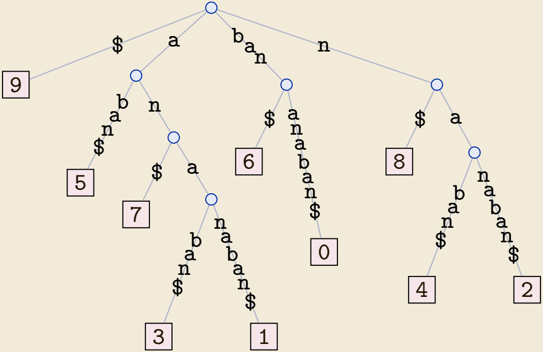
Observation. P occurs in $T \iff P$ is a prefix of a suffix of T.

- $\mathscr{T}$ stores (references to) all suffixes in T
- To search for P, try follow a path with label P until
  1. we get stuck
     - internal node without next character
     - mismatch along an edge
  2. we reach end of pattern P
     - all descendent leaves contain P!
  3. reach a leaf ℓ with part of P left (no match)

String Matching Example

Bananas. $T =$ `b a n a n a b a n $`

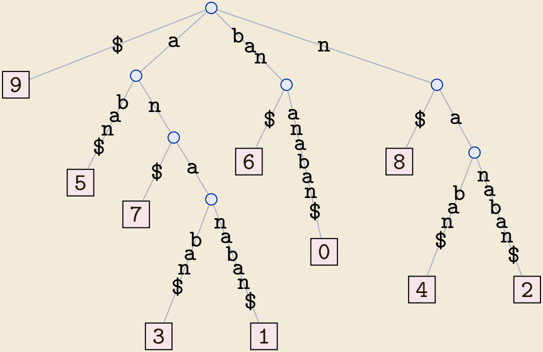
Human readible suffix tree:



String Matching Example

Bananas. $T =$ `b a n a n a b a n $`

Human readible suffix tree:



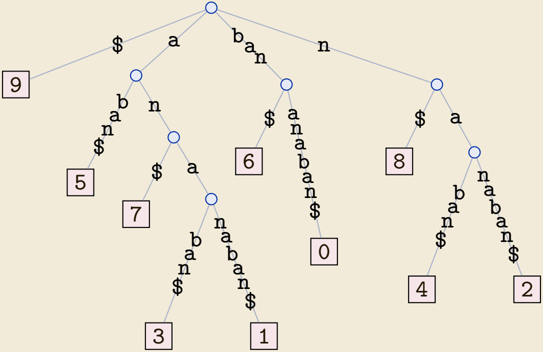
Note. Operations on “human readable” tree can be simulated in true suffix tree.

- each internal node stores pointer to left-most descendant index

String Matching Example

Bananas. $T =$ `b a n a n a b a n $`

Human readible suffix tree:

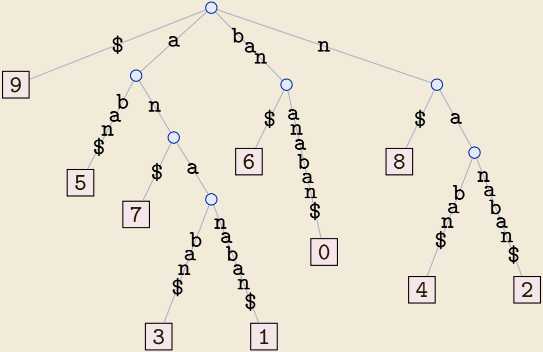


Search $P =$ ann

String Matching Example

Bananas. $T =$ `b a n a n a b a n $`

Human readible suffix tree:

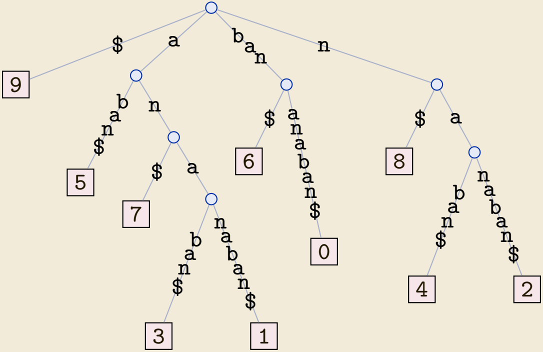


Search $P =$ `baa`

String Matching Example

Bananas. $T =$ `b a n a n a b a n $`

Human readible suffix tree:

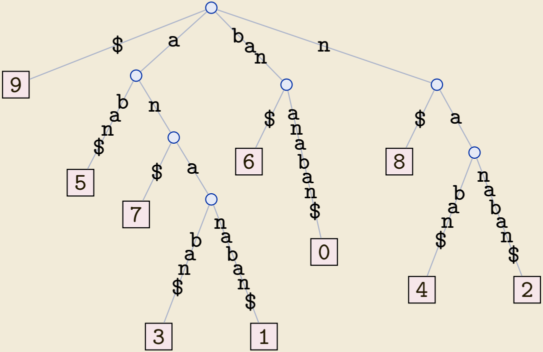


Search $P =$ `ana`

String Matching Example

Bananas. $T =$ `b a n a n a b a n $`

Human readible suffix tree:



Search $P =$ `ba`

String Matching Discussion

Using Suffix Trees

- Pre-process a text $T[0..n)$ in $O(n)$ time *once*
- Search for $P[0..m)$ in time $O(m)$ time

String Matching Discussion

Using Suffix Trees

- Pre-process a text $T[0..n)$ in $O(n)$ time *once*
- Search for $P[0..m)$ in time $O(m)$ time

Compare to KMP.

- Pre-process $P[0..m)$ in $O(m)$ time (per pattern)
- Search in $O(n+m)$ time for each pattern

String Matching Discussion

Using Suffix Trees

- Pre-process a text $T[0..n)$ in $O(n)$ time *once*
- Search for $P[0..m)$ in time $O(m)$ time

Compare to KMP.

- Pre-process $P[0..m)$ in $O(m)$ time (per pattern)
- Search in $O(n+m)$ time for each pattern

Comparison. If T is large and static, and we expect to perform many searches, the suffix tree construction is *much* more efficient!

Application 2: Repeated Substring

Problem. Given T, compute the **longest repeated substring** of T

- find the largest ℓ such that there are distinct indices i and j with $T[i, i+\ell] = T[j, j+\ell]$.

Application 2: Repeated Substring

Problem. Given T, compute the **longest repeated substring** of T

- find the largest ℓ such that there are distinct indices i and j with $T[i, i+\ell] = T[j, j+\ell]$.

Example. $T = \texttt{b a n a n a b a n}$

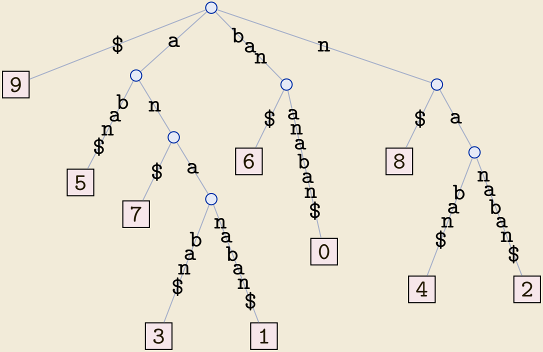
Application 2: Repeated Substring

Problem. Given T, compute the **longest repeated substring** of T

- find the largest ℓ such that there are distinct indices i and j with $T[i, i+\ell] = T[j, j+\ell]$.

Example. $T =$ `b a n a n a b a n`

Repeated substrings in the suffix tree?



Application 2: Repeated Substring

Problem. Given T, compute the **longest repeated substring** of T

- find the largest ℓ such that there are distinct indices i and j with $T[i, i+\ell] = T[j, j+\ell]$.

Example. $T =$ `b a n a n a b a n`

Observation. Repeated substrings correspond to paths of internal nodes in $\mathscr{T}$.

- Longest repeated substring = longest path of internal nodes in $\mathscr{T}$
  - "longest path" includes weight for compressed edges
- Can be computed in $O(n)$ time!
  - use "depth first search" strategy

More Applications

Using suffix trees we can perform the following tasks efficiently:

1. **Longest Common Substring** in time $O(n_1 + n_2 + \cdots + n_k)$
    - Input: texts $T_1, T_2, \ldots, T_k$
    - Output: the longest substring that is contained in all T_i

More Applications

Using suffix trees we can perform the following tasks efficiently:

1. **Longest Common Substring** in time $O(n_1 + n_2 + \cdots + n_k)$
    - Input: texts $T_1, T_2, \ldots, T_k$
    - Output: the longest substring that is contained in all T_i
2. **Longest Common Extension** in time $O(1)$!!
    - Input: text T and indices i, j
    - Output: largest ℓ for which $T[i, i+\ell] = T[j, j+\ell]$

More Applications

Using suffix trees we can perform the following tasks efficiently:

1. **Longest Common Substring** in time $O(n_1 + n_2 + \cdots + n_k)$
    - Input: texts $T_1, T_2, \ldots, T_k$
    - Output: the longest substring that is contained in all T_i
2. **Longest Common Extension** in time $O(1)$!!
    - Input: text T and indices i, j
    - Output: largest ℓ for which $T[i, i+\ell] = T[j, j+\ell]$
3. **Approximate Matching**
    - Input: text $T[0..n)$, pattern $P[0..m)$, parameter $k \in [0..m)$
    - Output: smallest i for which T contains P' with at most k mismatches

More Applications

Using suffix trees we can perform the following tasks efficiently:

1. **Longest Common Substring** in time $O(n_1 + n_2 + \cdots + n_k)$
    - Input: texts $T_1, T_2, \ldots, T_k$
    - Output: the longest substring that is contained in all T_i
2. **Longest Common Extension** in time $O(1)$!!
    - Input: text T and indices i, j
    - Output: largest ℓ for which $T[i, i+\ell] = T[j, j+\ell]$
3. **Approximate Matching**
    - Input: text $T[0..n)$, pattern $P[0..m)$, parameter $k \in [0..m)$
    - Output: smallest i for which T contains P' with at most k mismatches
4. **Matching with Wildcards**
    - Input: text $T[0..n)$, pattern $P[0..m)$ with wildcards
        - wildcard character $*$ matches a substring of any length
    - Output: first appearance of P (with wildcard matches)

Conclusion

Suffix trees are amazing data structures!

- Tons of applications
- Surprising theoretical results

Next time. Constructing suffix trees *efficiently*

Scratch Notes