PAT Tree and PAT Array

Presented by Huiqin Körkel-Qu
Institute for Computer Linguistics, Heidelberg University

March 14, 2011
Outline

- Trie, patricia tree
- From semi-infinite strings to a PAT tree
- Algorithms on PAT tree
- Structures modified from a PAT tree
Trie and Patricia tree

- **trie**: originates from the word ‘retrieval’
  every path from root to a leave represents a string.

- **patricia tree**: space optimized trie
  nodes with only one child will be merged

- **example**: given strings 100, 010, 110
Patricia tree:

- every internal node has two children
- each internal node has an indication of branching
  1. bit position to branch
  2. ‘zero’ bit to left subtree
  3. ‘one’ bit to right subtree
- only useful (real) branching is produced!!
Given a text (character array), a **sistring** (semi-infinite string) of this text is:

- a subsequence of this text
- starting from some point (position) of the text
- going until end of the text

**Eg:**

*Text*: this is an example of sistring....

*sistring 1*: this is an example of sistring....

*sistring 2*: his is an example of sistring...

*sistring 9*: an example of sistring...

*sistring 13*: xample of sistring...

**order of the sistrings**: 9 < 2 < 1 < 13
Why sistring?

Humans can easily grasp all substrings of a text easily.

Eg:

text: ‘HEID’,
              ‘ID’, ‘I’, ‘D’

saving only $n$ sistrings, we can get all $n(n + 1)/2$ substrings easily by prefix searching.
A PAT tree is a patricia tree over all sistrings of a text.

**internal nodes:** branching position
  pointers to subtrees

**external nodes:** sistrings

Text: 01011011000111.....
Position: 123456789.....

sistring1: 01011011000111...
sistring2: 1011011000111...
sistring3: 011011000111...
sistring4: 11011000111...
sistring5: 1011000111...
sistring6: 011000111...
....... 

text size: \( n \)
tree size: \( O(n) \)  
tree height: \( O(\log n) \) to \( O(n) \)
Algorithms on PAT tree

- prefix searching
- proximity searching
- range searching
- longest repetition searching
- most frequent searching
- regular expression searching
- the longest palindrome searching
Prefix searching

**Eg:** searching for the prefix 110

![Diagram of a PAT Tree with nodes labeled 1, 2, 3, 4, 5, 6, 7, 8, and 9. The search path from the root to the node 9 is highlighted, showing the searching time: proportional to the query length and no more than the height of the tree.]

searching time: proportional to the query length

no more than the height of the tree
Proximity searching

Find all places where two strings (substrings in the text) are not too ‘far away’

  two strings: $s_1$ and $s_2$
  distance: $b$, $b \in \mathbb{N}$ (number of symbols, words, etc)

Eg:
$s_1 = \text{‘cat’}$, $s_2 = \text{‘mouse’}$, $b = 2$ (number of words)

  \textbf{‘cat catches mouse’} \in s_1 bs_2
  \textbf{‘a cat has caught a mouse’} \not\in s_1 bs_2
Proximity searching

Proximity searching algorithm based on PAT tree:

1. search for \( s_1 \) and \( s_2 \)
2. assume that the answer set sizes are \( m_1 \) and \( m_2 \) respectively and \( m_1 \leq m_2 \)
3. sort the answer of \( s_1 \) whose size is \( m_1 \)
4. check every answer in the answer set of \( s_2 \) to see if it satisfies the distance condition

Complexity: sort + check = \( m_1 \log m_1 + m_2 \log m_1 \)
Proximity searching

\[ s_1 = 011, s_2 = 110, b = 2 \]

The final answer (no constraint on order): \( \{(3, 4), (6, 7), (6, 4)\} \)
**Range searching**

**Eg:** Searching in the lexicographical range ‘ab’ .... ‘ad’

- ‘abc’ ∈ range(‘ab’, ‘ad’)  
- ‘aea’ ∉ range(‘ab’, ‘ad’)  

searching in the range 011 and 10
Longest repetition searching

Find the longest match between two different positions in a text. (the ‘biggest’ internal node)

**Example:** 01011011000111

6 is the 'biggest' internal node
10110 is the longest repetition
Most frequent searching

Find the string that appears most frequently in the text.

**Eg:** find the most frequent substring of length 2
(the biggest subtree with distance 2 to root)

The most frequent 2-grams are 01 and 10 
both appear 3 times
regular expression $\Rightarrow$ binary DFA (Deterministic Finite Automaton with input alphabet $\{0, 1\}$) without final state outgoing transition

simulate the binary DFA on **binary trie**
- initial state $\Rightarrow$ root
- for transition $i \rightarrow_{0} j$
  - state $j \Rightarrow$ internal node (associated with state $i$)'s left child
- for transition $i \rightarrow_{1} j$
  - state $j \Rightarrow$ internal node (associated with state $i$)'s right child
- if final state $\Rightarrow$ internal node, accept the whole subtree
- if final state $\Rightarrow$ external node, run DFA continue.
This figure is from Gonnet, Baeza-Yates and Snider (1992).
Bucketing: replace subtrees (size limitation $b$) with buckets

bucketing with $b=3$

tree after bucketing

PAT Tree and PAT Array
Some properties of bucketing

Bucketing: tradeoff between time and space

- not every bucket is full
- every bucket saves up to \( b - 1 \) internal nodes
- on average there are \( b \ln 2 \) keys per bucket
- for random text after bucketing there are \( \frac{n}{b \ln 2} \) internal nodes in the tree left
- the searching time increases up to \( b \)
Supernodes—mapping tree on disk

Idea: big tree is stored in many disk pages
one page has only one entry

supernode ➔ diskpage
one entry point

internal node points to diskpage
or node in its page

entry point

PAT Tree and PAT Array
From PAT tree to PAT array

**idea:**
- bucket the whole tree
- keep the order of external nodes

**construction:**
- quicksort

**advantage:**
- keep most information of PAT tree
- save space

**disadvantage:**
- time complexity may increase
- not all searching described above can be done
Construction of PAT array for large text

If a text is small, its PAT array can be built in memory. what if the text is too big?

- cut the text into small pieces
- construct a PAT array for every piece in memory
- merge the PAT arrays

two merging cases

1. merge small array with large array
2. merge large arrays
Merge small with large arrays

What is stored in memory? (small and fast medium)

1. the small text
2. the array for small text
3. a **counter**

What is stored on hard disk? (large but slow storage medium)

1. the large text
2. the array for the large text

What is the counter in memory?

- the counter contains an item for every sistring in the small array
- Item $i$ in the counter indicates how many sistrings in the **large array** are between sistrings $(i - 1)$ and $i$ in the **small array**
The large text is read sequentially to create the **counter**.

The sistrings in the small array are inserted into the large array according to the counter.
Idea:

- reduce random access to hard disk
  read a block of pointers in PAT array instead of one by one

**Eg.** If block size is $m$, and the text length is $n$, reading block by block needs $\lceil n/m \rceil$ times hard disk access with $\lceil n/m \rceil$ times text scan.

- sort sistrings of every block respectively
- put the results above into temporary disk space
- merge the PAT arrays by comparing sistrings (from two texts)
Merge large texts

**PAT Tree and PAT Array**
PAT tree and PAT array are the data structures which
- preprocess text
- allow many different ways of searching
- fit for large text
- with high efficiency in space and time


