# Indexing and Searching

# Introduction

- How to retrieval information?
- A simple alternative is to search the whole text sequentially
- Another option is to build data structures over the text (called *indices*) to speed up the search

# Introduction

- Indexing techniques:
  - Inverted files
  - Suffix arrays
  - Signature files

# Notation

- $n$: the size of the text
- $m$: the length of the pattern
- v: the size of the vocabulary
- $M$: the amount of main memory available

# Inverted Files

- an inverted file is a word-oriented mechanism for indexing a text collection
- an inverted file is composed of vocabulary and the occurrences
- occurrences - set of lists of the text positions where the word appears

# Searching

- The search algorithm on an inverted index follows three steps:
  - Vocabulary search: the words present in the query are searched in the vocabulary
  - Retrieval occurrences: the lists of the occurrences of all words found are retrieved
  - Manipulation of occurrences: the occurrences are processed to solve the query

# Searching

- Searching task on an inverted file always starts in the vocabulary ( It is better to store the vocabulary in a separate file )

- The structures most used to store the vocabulary are *hashing, tries* or *B-trees*

- An alternative is simply storing the words in lexicographical order ( cheaper in space and very competitive with $O(\log v)$ cost )

# Construction

- building and maintaining an inverted index is a low cost task
- an inverted index on a text of n characters can be built in O(n) time
- all the vocabulary is kept in a trie data structure
- for each word, a list of its occurrences (test positions) is stored

# Construction

- each word of is read and searched in the vocabulary

# Other Indices for Text

- Suffix trees and suffix arrays
- Signature files

# Suffix Trees and Suffix Arrays

Inverted indices assumes that the text can be seen as a sequence of words

- Restricts the kinds of queries that can be answered
- Other queries, ex. phrases, are expensive to solve

Problem: the concept of word does not exist in some applications, e.g. genetic database

# Suffix Trees

- A suffix tree is
  - a trie data structure
  - built over all the suffixes of the text

(pointers to the suffixes are stored at the leaf nodes)

  - compacted into a Patricia tree, to improve space utilization
    - The tree has O(n) nodes instead of O($n^2$) nodes of the trie

# Suffix Arrays

- a space efficient implementation of suffix trees
- allow user to answer more complex queries efficiently
- main drawbacks
  - costly construction process
  - text must be available at query time
  - results are not delivered in text position order

# Suffix Arrays

- Can be used to index only words as the inverted index (unless complex queries are important, inverted files performs better)
- a text *suffix* is a position in the text (each suffix is uniquely identified by its position)
- index points, which point to the beginning of the text positions, are selected from the text
- elements which are not indexed are not retrievable

# Suffix Arrays: Structure

Suffix tree problem: space requirement

- Each node of the trie takes 12 to 24 bytes

-> space overhead of 120% to 240% over the text size is produced

Suffix array is an array containing all the pointers to the text suffixes listed in lexicographical order

- requires much less space

# Suffix Arrays: Structure

- are designed to allow binary searches (by comparing the contents of each pointer)
- for large suffix arrays, binary search can perform poorly (due to the number of random disk accesses)
- one solution is to use supra-indices over suffix array

# Suffix Arrays: Searching

- basic pattern (ex. words, prefixes, and phrases) search on a suffix tree can be done in O(m) time by a simple trie search
- binary search on suffix arrays can be done in O(log n) time

# Suffix Arrays: Construction

- a suffix tree for a text of n characters can be built in O(n) time
- if the suffix tree does not fit in main memory, the algorithm performs poorly

# Suffix Arrays: Construction

- an algorithm to build the suffix array in O(n log n) character comparisons
  - suffixes are bucket-sorted in O(n) time according to the first letter only
  - each bucket is bucket-sorted again according to the first two letters

# Suffix Arrays: Construction

- at iteration i the suffixes begin already sorted by their $2^{i-1}$ first letters and end up sorted by their first $2^i$ letters
- each iteration, the total cost of all the bucket sorts is O(n), the total time is O(n log n), and the average is O(n log log n) (O(log n) comparisons are necessary on average to distinguish two suffixes of a text)

# Suffix Arrays: Construction

- large text will not fit in main memory
- split large text into blocks that can be sorted in main memory
- build suffix array in main memory and merge it the the rest of the array already built for the previous text
- compute counters to store information of how many suffixes of the large text lie between each pair of positions of the suffix array

# Signature Files

- Signature files are word-oriented index structures based on hashing
  - low overhead (10%-20% over the text size)
  - forcing a sequential search over the index
  - suitable for not very large texts

Nevertheless, inverted files outperform signature files for most applications

# Signature Files: Structure

- A signature file uses a hash function (or ‘signature’) that maps words to bit masks of B bits
- the text is devided in blocks of b words each
- to each text block of size b, a bit mask of size B is assigned
- this mask is obtained by bitwise ORing the signatures of all the words in the text block

# Signature Files: Structure

- a signature file is the sequence of bit masks of all blocks (plus a pointer to each block)
- the main idea is:
  - if a word is present in a text block, then all the bits set in its signature are also set in the bit mask of the text block
  - false drop: all the corresponding bits are set even though the word is not in the block
- to ensure low probability of a false drop and to keep the signature file as short as possible

# Signature Files: Searching

Searching a single word is done by

1. hashing that word to a bit mask W
2. comparing the bit masks $B_i$ of all the text blocks
3. whenever ($W \& B_i = W$), the text block may contain the word
4. online traversal must be performed on all the candidate text blocks to verify if the word is there

# Signature Files: Searching

This scheme is efficient to search phrases and reasonable proximity queries

# Signature Files: Construction

Constructing a signature file:

1. cut text into blocks
2. generate an entry of the signature file for each block
   - this entry is the bitwise OR of the signatures of all the words in the block

Adding text: adding records to the signature file

Deleting text: deleting the appropriate bit masks

# Signature Files: Construction

Storage proposals:

1. store all the bit masks in sequence
2. make a different file for each bit of the mask

   (ex. one file holding all the first bits, another file for all the second bits)

   result: the reduction in the disk times to search for a query

   (only the files corresponding to the l bits, which are set in the query, have to be traversed)

# Boolean Queries

- the set manipulation algorithms are used when operating on sets of results
- the search proceeds in three phases:
    1. determines which documents classify
    2. determines the relevance of the classifying documents (to present them to the user)
    3. retrieves the exact positions of the matches in those documents (that the user wants to see)

# Sequential Searching

when no data structure has been built on the text

1. brute-force (bf) algorithm
2. Knuth-Morris-Pratt
3. Boyer-Moore Family
4. Shift-Or
5. Suffix Automaton
6. Practical Comparison
7. Phrases and Proximity

# Brute Force (BF)

Trying all possible pattern positions in the text

  - a window of length m is slid over the text
  - if the text in the window is equal to the pattern, report as a match
  - shift the window forward

- worst-case is O(mn)
- average case is O(n)
- does not need any pattern preprocessing

# Knuth-Morris-Pratt (KMP)

- linear worst-case behavior O(2n)
- average case is not much faster than BF
  - slide a window over the text
  - does not try all window positions
  - reuses information from previous checks
- the pattern is preprocessed in O(m) time

# Boyer-Moore Family

- the check inside the window can proceed backwards
- for every pattern position j the next-to-last occurrence of $P_{j..m}$ inside P called *match heuristic*
- *occurrence heuristic*: the text character that produced the mismatch has to be aligned with the same character in the pattern after the shift

# Boyer-Moore Family

- proprocessing time and space of this algorithm: O(m+$\sigma$)
- search time is O(n log (m)/m) on average
- worst case is O(mn)

# Shift-Or

- uses bit-parallelism to simulate the operation of a non-deterministic automaton that searches the pattern in the text
- builds a table B which stores a bit mask $b_m \ldots b_1$ for each character
- update D using the formula

  $$D' \leftarrow (D<<1) \mid B[T_j]$$

- O(n) on average, preprocessing is $O(m+\sigma)$
- $O(\sigma)$ space

# Suffix Automaton

- The Backward DAWG matching (BDM) is based on a suffix automaton
- a suffix automaton on a pattern P is an automaton that recognizes all the suffixes of P
- the BDM algorithm converts a non-deterministic suffix automaton to deterministic

# Suffix Automaton

- the size and construction time is O(m)
- build a suffix automaton $P^r$ (the reversed pattern) to search for pattern P
- a match is found if the complete window is read
- O(mn) time for worst case
- O(n log(m)/m) on average

# Practical Comparison

- Figure 8.20 compares string matching algorithms on:
  - English text from the TREC collection
  - DNA
  - random text uniformly generated over 64 letters
  - patterns were randomly selected from the text, except for random text, patterns are randomly generated

# Phrases and Proximity

- if the sequence of words is searched as appear in text, the problem is similar to search of a single pattern
- to search a phrase element-wise is to search for the element which is less frequent or can be searched faster, then check the neighboring words
- same for proximity query search

# Pattern Matching

Two main groups of techniques to deal with complex patterns:

1. searching allowing errors
2. searching for extended patterns

# Searching Allowing Errors

Approximate string matching problem: given a short pattern P of length m, a long text T of length n, and a maximum allowed number of errors k, find all the text positions where the pattern occurs with at most k errors

The main approaches for solution: dynamic programming, automaton, bit-parallelism and filtering

# Dynamic Programming

- the classical solution to approximate string matching
- fill a matrix C column by column where C[i,j] represents the minimum numbers of errors needed to match $P_{1..i}$ to a suffix of $T_{1..j}$
  - C[0,j] = 0
  - C[i,0] = i
  - C[i,j] = if ($P_i$=$T_j$) then C[i-1,j-1]
    else 1+min(C[i-1,j],C[i,j-1],C[i-1,j-1]

# Automaton

- reduce the problem to a non-deterministic finite automaton (NFA)
- each row of the NFA denotes the number of errors seen
- every column represents matching the pattern up to a giver position

# Automaton

- horizontal arrows represent matching a character
- vertical arrows represent insertions into the pattern
- solid diagonal arrows represent replacements
- dashed diagonal arrows represent deletions in the pattern
- the automaton accepts a text position as the end of a match with k errors whenever the (k+1)-th rightmost state is active

# Bit-Parallelism

- use bit-parallelism to parallelize the computation of the dynamic programming matrix

# Filtering

- reduce the area of the text where dynamic programming needs to be used by filter the text first

# Regular Expressions and Extended Patterns

- build a non-deterministic finite automaton of size O(m)
  - m: the length of the regular expression
- convert this automaton to deterministic form
  - search any regular expression in O(n) time
  - its size and construction time can be exponential in m, i.e. $O(m2^m)$
- bit-parallelism has been proposed to avoid the construction of the deterministic automaton

# Pattern Matching Using Indices

- how to extend the indexing techniques for simple searching for more complex patterns
- inverted files:
  - using a sequential search over the vocabulary and merge their lists of occurrences to retrieve a list of documents and the matching text positions

# Pattern Matching Using Indices

- suffix tree and suffix array:
  - if all text positions are indexed, words, prefixes, suffixes and substrings can be searched with the same search algorithm and cost -> 10 to 20 times text size for index
  - range queries can be done by searching both extremes in the trie and collects all leaves in the middle

# Pattern Matching Using Indices

- regular expression search and unrestricted approximate string matching can be done by sequential searching
- other complex searches that can be done are: find the longest substring in the text that appears more than once, find the most common substring of a fixed size
- suffix array implementation reduce operation cost from C(n) (on suffix tree) to O(C(n) log n) cost

# Structural Queries

The algorithms to search on structured text are dependent on each model

Some considerations are:

- how to store the structural information
  - build and ad hoc index to store the structure (efficient and independent of the text)
  - mark the structure in the text using ‘tags’ (efficient in many cases, integration into an existing text database is simpler)

# Compression

- Sequential searching
- Compressed indices
  - Inverted files
  - Suffix trees and suffix arrays
  - Signature files

# Sequential Searching

- Huffman coding technique allows directly searching compressed text
- Boyer-Moore filtering can be used to speed up the search in Huffman coding trees
- Phrase searching can be accomplished using the i-th bit mask with the i-th element of the phrase query + Shift-Or algorithm (simple and efficient)

# Compressed Indices: Inverted Files

- the lists of occurrences are in increasing order of text position
- the differences between the previous position and the current one can be represented using less space by using techniques that favor small numbers
- the text can also be compressed independently of the index

# Compressed Indices:
# Suffix Trees and Suffix Arrays

- reduction of space requirements -> more expensive searching
- reduced space requirements are similar to those of uncompressed suffix arrays at much smaller performance penalties
- suffix arrays are hard to compress because they represent an almost perfectly random permutation of the pointers to the text

# Compressed Indices:
# Suffix Trees and Suffix Arrays

- building suffix arrays on compressed text:
  - reduce space requirements
  - index construction and querying almost double performance
  - construction is faster because more compressed text fits in the same memory space -> fewer text blocks are needed
- Hu-Tucker coding allows direct binary search over the compressed text

# Compressed Indices: Signature Files

- compression techniques are based on the fact that only a few bits are set in the whole file
- possible to use efficient methods to code the bits which are not set (ex. run-length encoding)
- different considerations arise if the file is stored as a sequence of bit masks or with one file per bit of the mask
- advantages are
  - reduce space and disk times
  - increase B (reduce the false drop probability)

# Trends and Research Issues

The main trends in indexing and searching textual database are

- text collections are becoming huge
- searching is becoming more complex
- compression is becoming a star in the field