Modern Information Retrieval

Chapter 9

Indexing and Searching

with Gonzalo Navarro

Introduction

- Although **efficiency** might seem a secondary issue compared to **effectiveness**, it can rarely be neglected in the design of an IR system
- **Efficiency in IR systems**: to process user queries with minimal requirements of computational resources
- As we move to larger-scale applications, efficiency becomes more and more important
  - For example, in Web search engines that index terabytes of data and serve hundreds or thousands of queries per second

Introduction

- **Index**: a data structure built from the text to speed up the searches
- In the context of an IR system that uses an index, the efficiency of the system can be measured by:
  - **Indexing time:** Time needed to build the index
  - **Indexing space:** Space used during the generation of the index
  - **Index storage:** Space required to store the index
  - **Query latency:** Time interval between the arrival of the query and the generation of the answer
  - **Query throughput:** Average number of queries processed per second

Introduction

- When a text is updated, any index built on it must be updated as well
- Current indexing technology is not well prepared to support very frequent changes to the text collection
- **Semi-static collections:** collections which are updated at reasonable regular intervals (say, daily)
- Most real text collections, including the Web, are indeed semi-static
  - For example, although the Web changes very fast, the crawls of a search engine are relatively slow
- For maintaining freshness, incremental indexing is used

Inverted Indexes

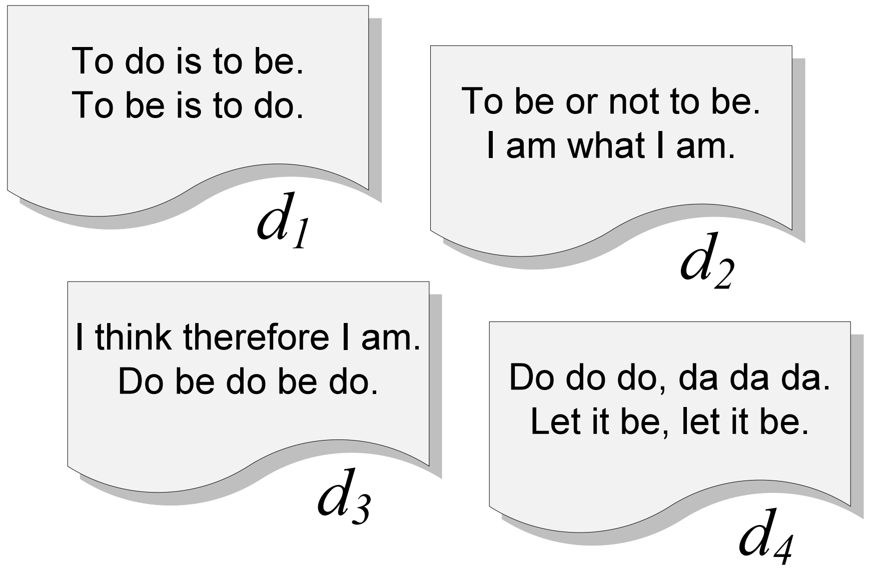
Basic Concepts

- **Inverted index**: a word-oriented mechanism for indexing a text collection to speed up the searching task
- The inverted index structure is composed of two elements: the **vocabulary** and the **occurrences**
- The vocabulary is the set of all different words in the text
- For each word in the vocabulary the index stores the documents which contain that word (inverted index)

Basic Concepts

- **Term-document matrix**: the simplest way to represent the documents that contain each word of the vocabulary

Vocabulary	n_i	d_1	d_2	d_3	d_4
to	2	4	2	-	-
do	3	2	-	3	3
is	1	2	-	-	-
be	4	2	2	2	2
or	1	-	1	-	-
not	1	-	1	-	-
I	2	-	2	2	-
am	2	-	2	1	-
what	1	-	1	-	-
think	1	-	-	1	-
therefore	1	-	-	1	-
da	1	-	-	-	3
let	1	-	-	-	2
it	1	-	-	-	2



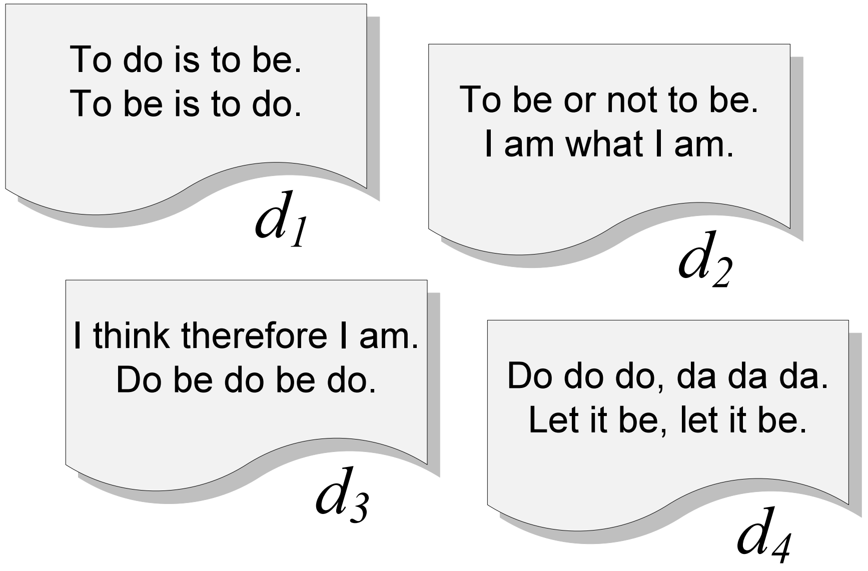
Basic Concepts

- The main problem of this simple solution is that it requires too much space
- As this is a sparse matrix, the solution is to associate a list of documents with each word
- The set of all those lists is called the **occurrences**

Basic Concepts

- Basic inverted index

Vocabulary	n_i	Occurrences as inverted lists
to	2	[1,4],[2,2]
do	3	[1,2],[3,3],[4,3]
is	1	[1,2]
be	4	[1,2],[2,2],[3,2],[4,2]
or	1	[2,1]
not	1	[2,1]
I	2	[2,2],[3,2]
am	2	[2,2],[3,1]
what	1	[2,1]
think	1	[3,1]
therefore	1	[3,1]
da	1	[4,3]
let	1	[4,2]
it	1	[4,2]

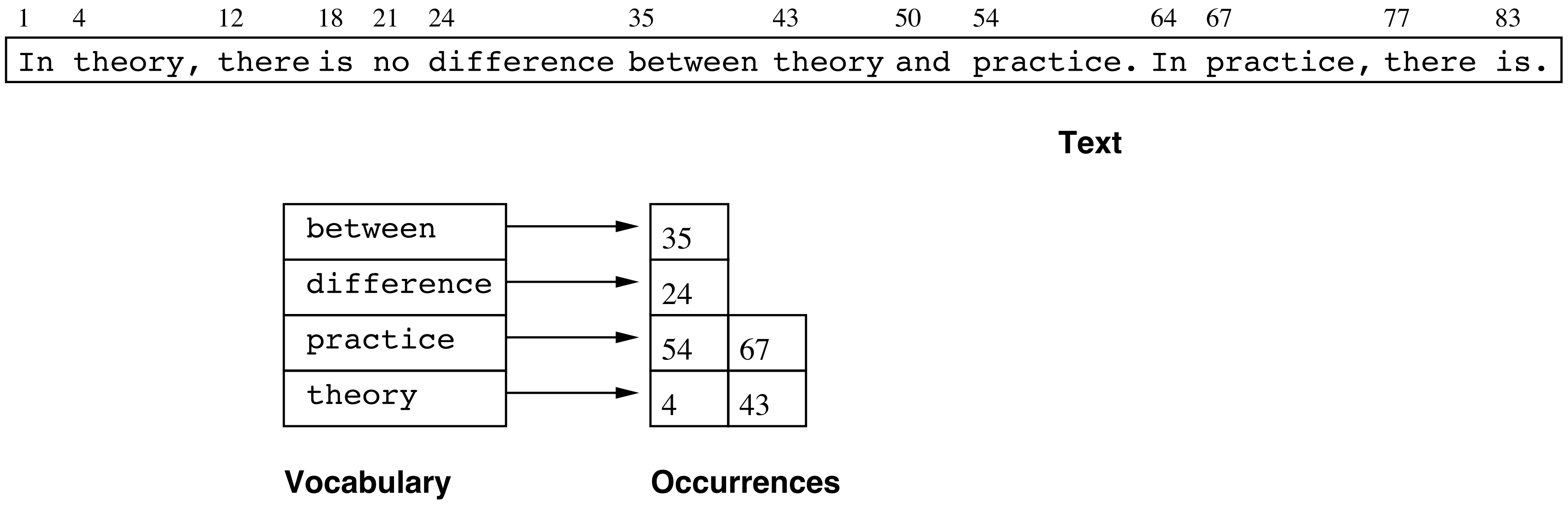


Inverted Indexes

Full Inverted Indexes

Full Inverted Indexes

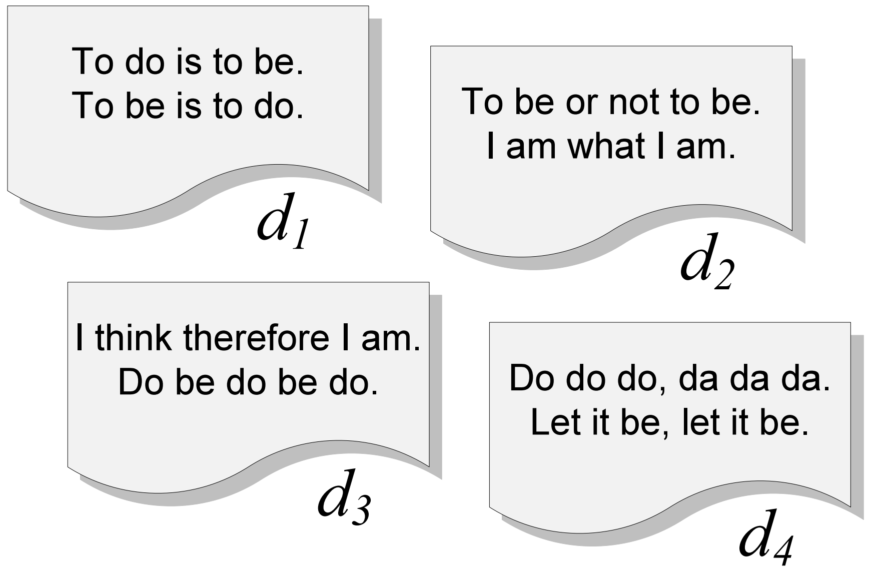
- The basic index is not suitable for answering phrase or proximity queries
- Hence, we need to add the positions of each word in each document to the index (full inverted index)



Full Inverted Indexes

- In the case of multiple documents, we need to store one occurrence list per term-document pair

Vocabulary	n_i	Occurrences as full inverted lists
to	2	[1,4,[1,4,6,9]],[2,2,[1,5]]
do	3	[1,2,[2,10]],[3,3,[6,8,10]],[4,3,[1,2,3]]
is	1	[1,2,[3,8]]
be	4	[1,2,[5,7]],[2,2,[2,6]],[3,2,[7,9]],[4,2,[9,12]]
or	1	[2,1,[3]]
not	1	[2,1,[4]]
I	2	[2,2,[7,10]],[3,2,[1,4]]
am	2	[2,2,[8,11]],[3,1,[5]]
what	1	[2,1,[9]]
think	1	[3,1,[2]]
therefore	1	[3,1,[3]]
da	1	[4,3,[4,5,6]]
let	1	[4,2,[7,10]]
it	1	[4,2,[8,11]]



To do is to be.
To be is to do.
d_1

To be or not to be.
I am what I am.
d_2

I think therefore I am.
Do be do be do.
d_3

Do do do, da da da.
Let it be, let it be.
d_4

Full Inverted Indexes

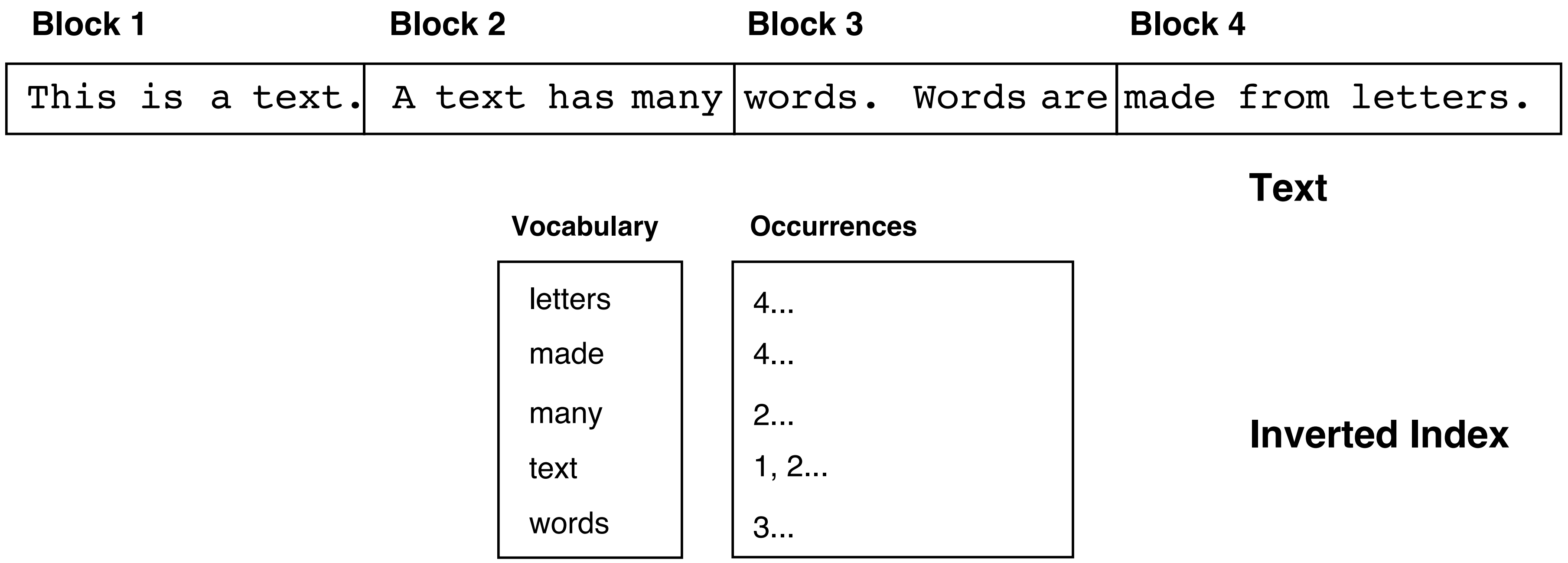
- The space required for the vocabulary is rather small
- Heaps' law: the vocabulary grows as $O(n^\beta)$, where
  - n is the collection size
  - β is a collection-dependent constant between 0.4 and 0.6
- For instance, in the TREC-3 collection, the vocabulary of 1 gigabyte of text occupies only 5 megabytes
- This may be further reduced by stemming and other normalization techniques

Full Inverted Indexes

- The occurrences demand much more space
- The extra space will be $O(n)$ and is around
  - 40% of the text size if stopwords are omitted
  - 80% when stopwords are indexed
- Document-addressing indexes are smaller, because only one occurrence per file must be recorded, for a given word
- Depending on the document (file) size, document-addressing indexes typically require 20% to 40% of the text size

Full Inverted Indexes

- To reduce space requirements, a technique called **block addressing** is used
- The documents are divided into blocks, and the occurrences point to the blocks where the word appears



Full Inverted Indexes

- The Table below presents the projected space taken by inverted indexes for texts of different sizes

Index granularity	Single document (1 MB)		Small collection (200 MB)		Medium collection (2 GB)	
Addressing words	45%	73%	36%	64%	35%	63%
Addressing documents	19%	26%	18%	32%	26%	47%
Addressing 64K blocks	27%	41%	18%	32%	5%	9%
Addressing 256 blocks	18%	25%	1.7%	2.4%	0.5%	0.7%

Full Inverted Indexes

- The blocks can be of fixed size or they can be defined using the division of the text collection into documents
- The division into blocks of fixed size improves efficiency at retrieval time
  - This is because larger blocks match queries more frequently and are more expensive to traverse
- This technique also profits from *locality of reference*
  - That is, the same word will be used many times in the same context and all the references to that word will be collapsed in just one reference

Single Word Queries

- The simplest type of search is that for the occurrences of a single word
- The vocabulary search can be carried out using any suitable data structure
  - Ex: hashing, tries, or B-trees
- The first two provide $O(m)$ search cost, where m is the length of the query
- We note that the vocabulary is in most cases sufficiently small so as to stay in main memory
- The occurrence lists, on the other hand, are usually fetched from disk

Multiple Word Queries

- If the query has more than one word, we have to consider two cases:
  - conjunctive (AND operator) queries
  - disjunctive (OR operator) queries
- **Conjunctive queries** imply to search for all the words in the query, obtaining one inverted list for each word
- Following, we have to **intersect** all the inverted lists to obtain the documents that contain all these words
- For **disjunctive queries** the lists must be **merged**
- The first case is popular in the Web due to the size of the document collection

List Intersection

- The most time-demanding operation on inverted indexes is the merging of the lists of occurrences
  - Thus, it is important to optimize it
- Consider one pair of lists of sizes m and n respectively, stored in consecutive memory, that needs to be intersected
- If m is much smaller than n, it is better to do m binary searches in the larger list to do the intersection
- If m and n are comparable, Baeza-Yates devised a double binary search algorithm
  - It is $O(\log n)$ if the intersection is trivially empty
  - It requires less than $m + n$ comparisons on average

List Intersection

- When there are more than two lists, there are several possible heuristics depending on the list sizes
- If intersecting the two shortest lists gives a very small answer, might be better to intersect that to the next shortest list, and so on
- The algorithms are more complicated if lists are stored non-contiguously and/or compressed

Phrase and Proximity Queries

- Context queries are more difficult to solve with inverted indexes
- The lists of all elements must be traversed to find places where
  - all the words appear in sequence (for a phrase), or
  - appear close enough (for proximity)
  - these algorithms are similar to a list intersection algorithm
- Another solution for phrase queries is based on indexing two-word phrases and using similar algorithms over pairs of words
  - however the index will be much larger as the number of word pairs is not linear

More Complex Queries

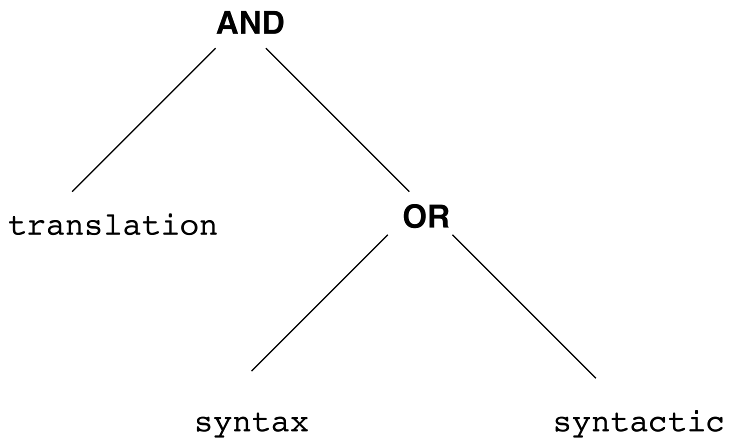
- Prefix and range queries are basically (larger) disjunctive queries
- In these queries there are usually several words that match the pattern
    - Thus, we end up again with several inverted lists and we can use the algorithms for list intersection

More Complex Queries

- To search for regular expressions the data structures built over the vocabulary are rarely useful
- The solution is then to **sequentially** traverse the vocabulary, to spot all the words that match the pattern
- Such a sequential traversal is not prohibitively costly because it is carried out only on the vocabulary

Boolean Queries

- In boolean queries, a **query syntax** tree is naturally defined



- Normally, for boolean queries, the search proceeds in three phases:
  - the first phase determines which documents to match
  - the second determines the likelihood of relevance of the documents matched
  - the final phase retrieves the exact positions of the matches to allow highlighting them during browsing, if required

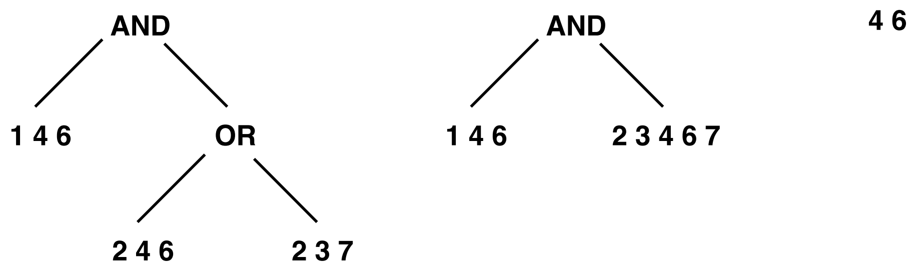
Boolean Queries

- Once the leaves of the query syntax tree find the classifying sets of documents, these sets are further operated by the internal nodes of the tree
- Under this scheme, it is possible to evaluate the syntax tree in **full** or **lazy** form
  - In the full evaluation form, both operands are first completely obtained and then the complete result is generated
  - In lazy evaluation, the partial results from operands are delivered only when required, and then the final result is recursively generated

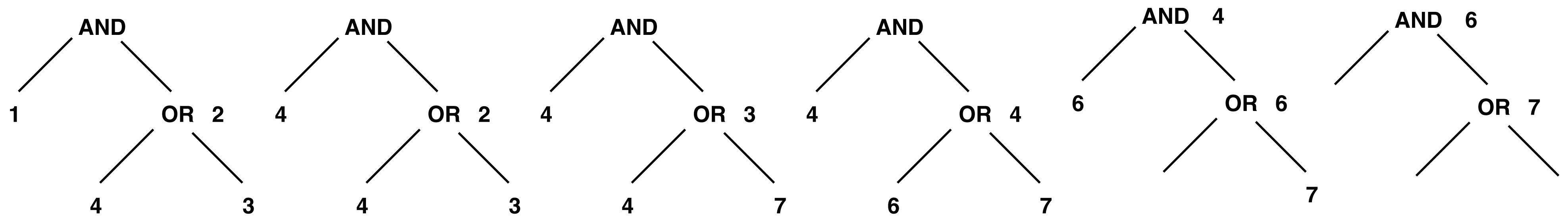
Boolean Queries

- Processing the internal nodes of the query syntax tree
  - In (a) full evaluation is used
  - In (b) we show lazy evaluation in more detail

a)



b)



Inverted Indexes

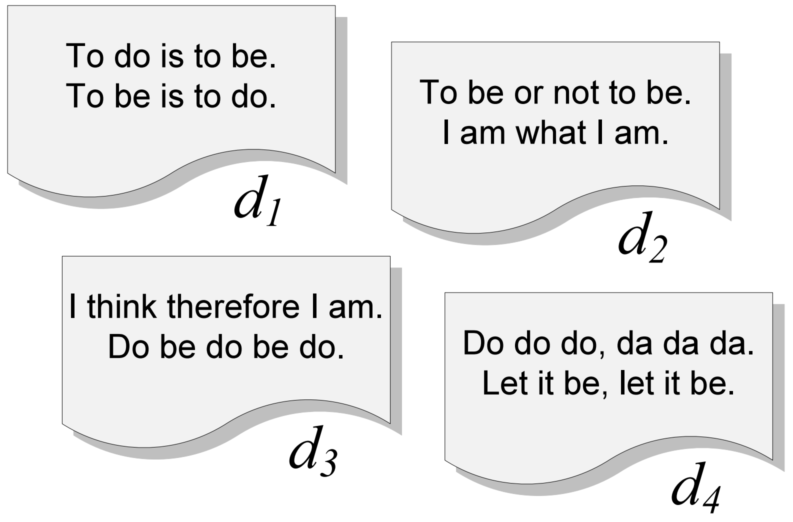
Searching

Ranking

- How to find the top-k documents and return them to the user when we have weight-sorted inverted lists?
- If we have a single word query, the answer is trivial as the list can be already sorted by the desired ranking
- For other queries, we need to merge the lists

Ranking

- Suppose that we are searching the disjunctive query "`to do`" on the collection below



- As our collection is very small, let us assume that we are interested in the top-2 ranked documents
- We can use the following heuristic:
  - we process terms in idf order (shorter lists first), and
  - each term is processed in tf order (simple ranking order)

Ranking

```
Ranking-in-the-vector-model( query terms $t$ )
01 Create $P$ as $C$-candidate similarities initialized to $(P_d, P_w) = (0, 0)$
02 Sort the query terms $t$ by decreasing weight
03 $c \leftarrow 1$
04 for each sorted term $t$ in the query do
05   Compute the value of the threshold $t_{add}$
06   Retrieve the inverted list for $t$, $L_t$
07   for each document $d$ in $L_t$ do
08     if $w_{d,t} < t_{add}$ then break
09     $psim \leftarrow w_{d,t} \times w_{q,t}/W_d$
10     if $d \in P_d(i)$ then
11       $P_w(i) \leftarrow P_w(i) + psim$
11     elif $psim > min_j(P_w(j))$ then
11       $n \leftarrow min_j(P_w(j))$
12     elif $c \leq C$ then
13       $n \leftarrow c$
14       $c \leftarrow c + 1$
15     if $n \leq C$ then $P(n) \leftarrow (d, psim)$
16 return the top-$k$ documents according to $P_w$
```

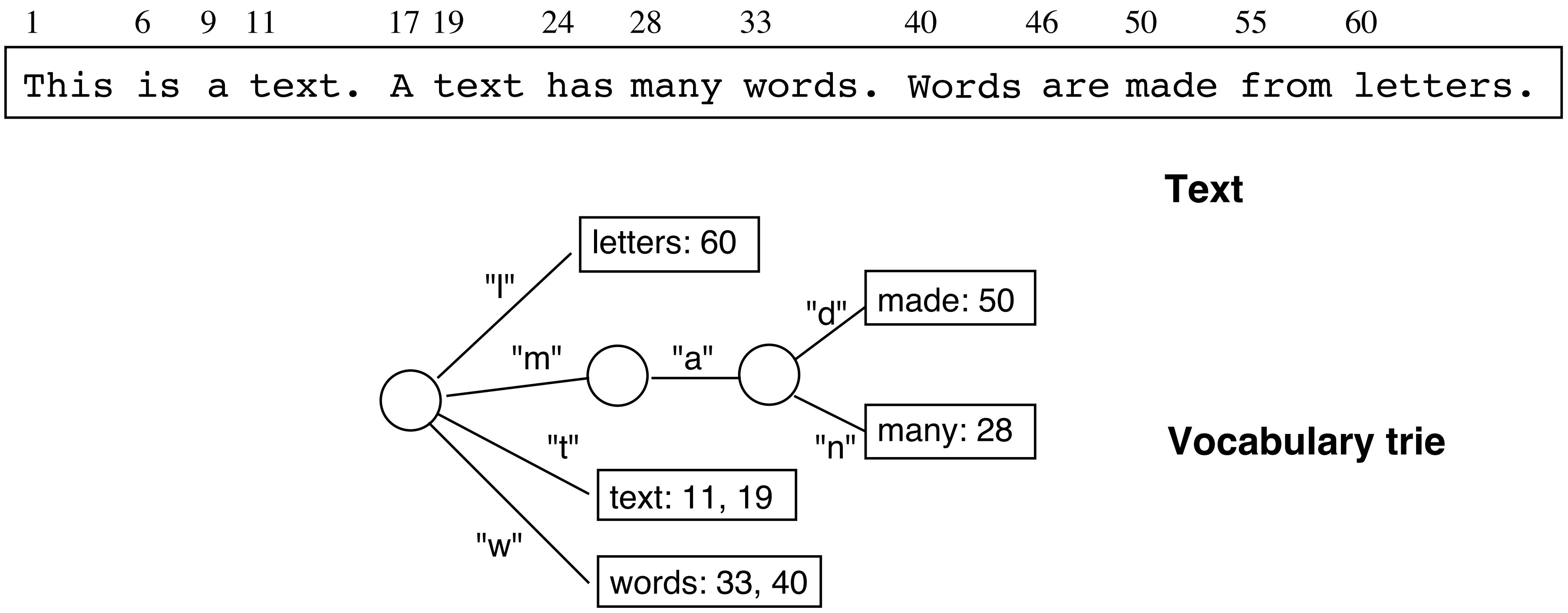
- This is a variant of Persin's algorithm
- We use a priority queue P of C document candidates where we will compute partial similarities

Internal Algorithms

- Building an index in internal memory is a relatively simple and low-cost task
- A dynamic data structure to hold the vocabulary (B-tree, hash table, etc.) is created empty
- Then, the text is scanned and each consecutive word is searched for in the vocabulary
- If it is a new word, it is inserted in the vocabulary before proceeding

Internal Algorithms

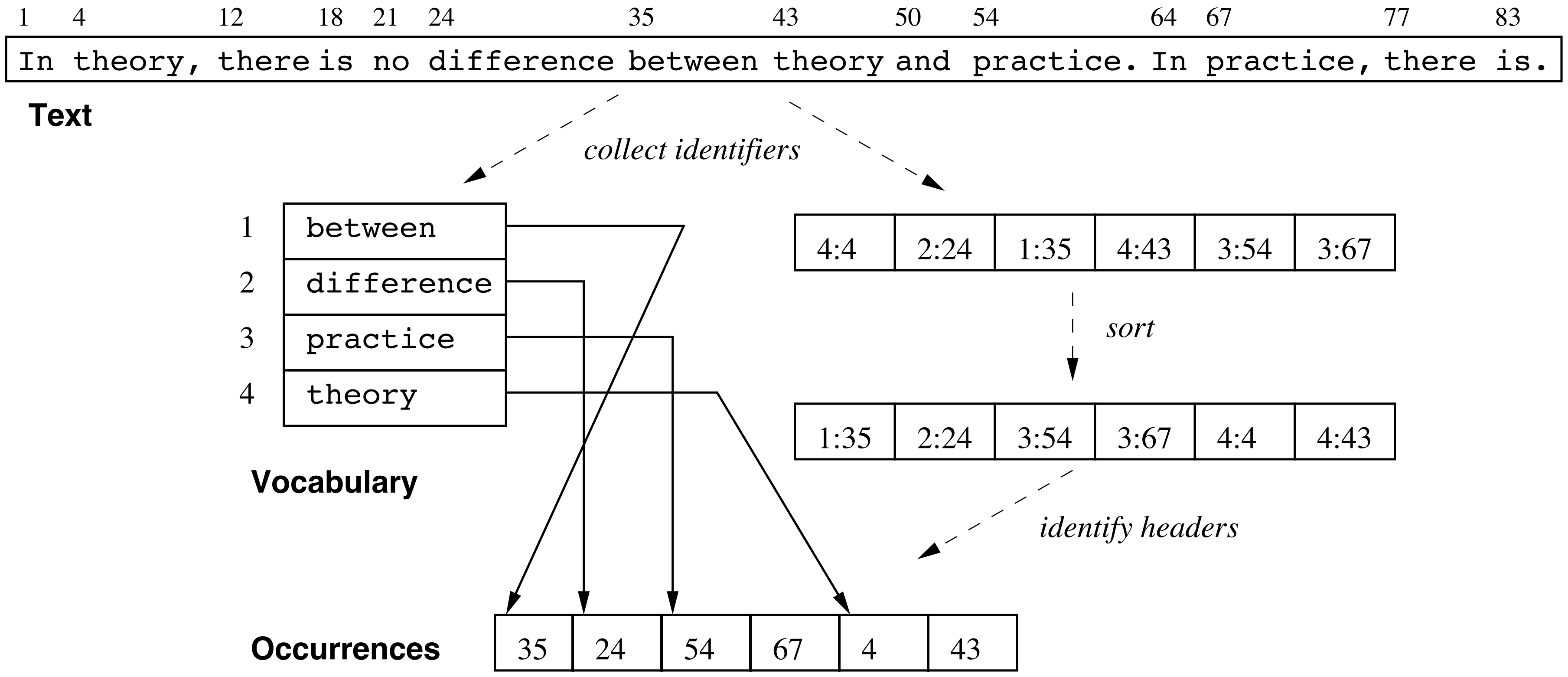
- A large array is allocated where the identifier of each consecutive text word is stored
- A full-text inverted index for a sample text with the incremental algorithm:



```
1      6   9  11      17 19    24   28    33       40     46    50     55     60
This is a text. A text has many words. Words are made from letters.
```

Internal Algorithms

- A full-text inverted index for a sample text with a sorting algorithm:



Internal Algorithms

- An alternative to avoid this sorting is to separate the lists from the beginning
  - In this case, each vocabulary word will hold a pointer to its own array (list) of occurrences, initially empty
- A non trivial issue is how the memory for the many lists of occurrences should be allocated
  - A classical list in which each element is allocated individually wastes too much space
  - Instead, a scheme where a list of blocks is allocated, each block holding several entries, is preferable

Internal Algorithms

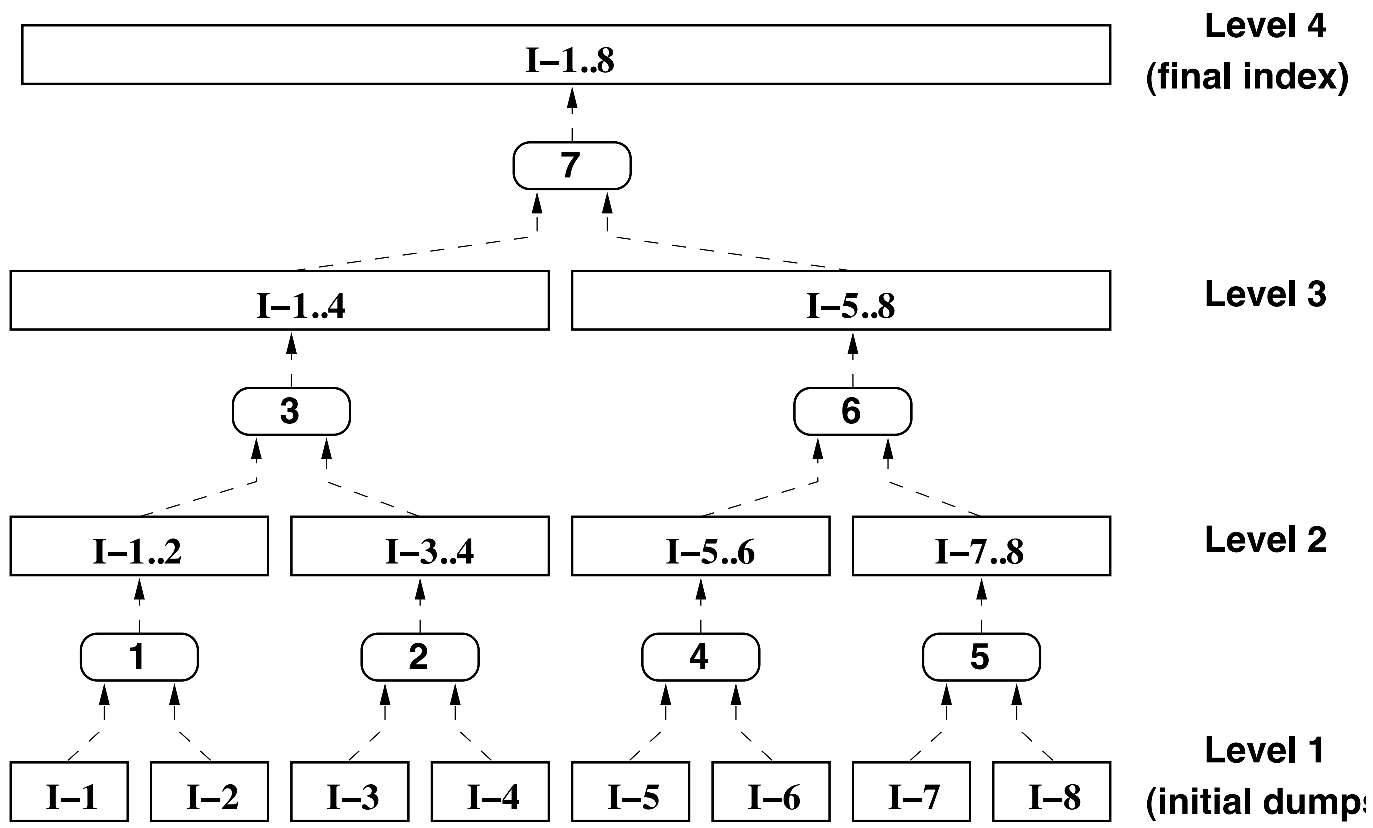
- Once the process is completed, the vocabulary and the lists of occurrences are written on two distinct disk files
- The vocabulary contains, for each word, a pointer to the position of the inverted list of the word
- This allows the vocabulary to be kept in main memory at search time in most cases

External Algorithms

- All the previous algorithms can be extended by using them until the main memory is exhausted
- At this point, the **partial** index I_i obtained up to now is written to disk and erased from main memory
- These indexes are then **merged** in a hierarchical fashion

External Algorithms

- Merging the partial indexes in a binary fashion
  - Rectangles represent partial indexes, while rounded rectangles represent merging operations



External Algorithms

- In general, maintaining an inverted index can be done in three different ways:
  - **Rebuild**
    - If the text is not that large, rebuilding the index is the simplest solution
  - **Incremental updates**
    - We can amortize the cost of updates while we search
    - That is, we only modify an inverted list when needed
  - **Intermittent merge**
    - New documents are indexed and the resultant partial index is merged with the large index
    - This in general is the best solution

Inverted Indexes

Compressed Inverted Indexes

Compressed Inverted Indexes

- It is possible to combine index compression and text compression without any complication
  - In fact, in all the construction algorithms mentioned, compression can be added as a final step
- In a full-text inverted index, the lists of text positions or file identifiers are in ascending order
- Therefore, they can be represented as sequences of **gaps** between consecutive numbers
  - Notice that these gaps are small for frequent words and large for infrequent words
  - Thus, compression can be obtained by encoding small values with shorter codes

Compressed Inverted Indexes

- A coding scheme for this case is the **unary code**
  - In this method, each integer $x > 0$ is coded as $(x - 1)$ 1-bits followed by a 0-bit
- A better scheme is the Elias-γ code, which represents a number $x > 0$ by a concatenation of two parts:
  1. a unary code for $1 + \lfloor \log_2 x \rfloor$
  2. a code of $\lfloor \log_2 x \rfloor$ bits that represents the number $x - 2^{\lfloor \log_2 x \rfloor}$ in binary
- Another coding scheme is the Elias-δ code
- Elias-δ concatenates parts (1) and (2) as above, yet part (1) is not represented in unary but using Elias-γ instead

Compressed Inverted Indexes

- Example codes for integers

Gap x	Unary	Elias-γ	Elias-δ	Golomb ($b = 3$)
1	0	0	0	00
2	10	100	1000	010
3	110	101	1001	011
4	1110	11000	10100	100
5	11110	11001	10101	1010
6	111110	11010	10110	1011
7	1111110	11011	10111	1100
8	11111110	1110000	11000000	11010
9	111111110	1110001	11000001	11011
10	1111111110	1110010	11000010	11100

Note: Golomb codes will be explained later

Compressed Inverted Indexes

- In general,
  - Elias-γ for an arbitrary integer $x > 0$ requires $1 + 2\lfloor \log_2 x \rfloor$ bits
  - Elias-δ requires $1 + 2\lfloor \log_2 \log_2 2x \rfloor + \lfloor \log_2 x \rfloor$ bits
- For small values of x Elias-γ codes are shorter than Elias-δ codes, and the situation is reversed as x grows
- Thus the choice depends on which values we expect to encode

Compressed Inverted Indexes

- Golomb presented another coding method that can be parametrized to fit smaller or larger gaps
- For some parameter b, let q and r be the quotient and remainder, respectively, of dividing $x - 1$ by b
  - I.e., $q = \lfloor (x-1)/b \rfloor$ and $r = (x-1) - q \cdot b$
- Then x is coded by concatenating
  - the unary representation of $q + 1$
  - the binary representation of r, using either $\lfloor \log_2 b \rfloor$ or $\lceil \log_2 b \rceil$ bits

Compressed Inverted Indexes

- If $r < 2^{\lfloor \log_2 b \rfloor - 1}$ then r uses $\lfloor \log_2 b \rfloor$ bits, and the representation always starts with a 0-bit
- Otherwise it uses $\lceil \log_2 b \rceil$ bits where the first bit is 1 and the remaining bits encode the value $r - 2^{\lfloor \log_2 b \rfloor - 1}$ in $\lfloor \log_2 b \rfloor$ binary digits
- For example,
  - For $b = 3$ there are three possible remainders, and those are coded as 0, 10, and 11, for $r = 0$, $r = 1$, and $r = 2$, respectively
  - For $b = 5$ there are five possible remainders r, 0 through 4, and these are assigned the codes 00, 01, 100, 101, and 110

Compressed Inverted Indexes

- To encode the lists of occurrences using Golomb codes, we must define the parameter b for each list
- Golomb codes usually give better compression than either Elias-γ or Elias-δ
  - However they need two passes to be generated as well as information on terms statistics over the whole document collection
- For example, in the TREC-3 collection, the average number of bits per list entry for each method is
  - Golomb = 5.73
  - Elias-δ = 6.19
  - Elias-γ = 6.43
- This represents a five-fold reduction in space compared to a plain inverted index representation

Compressed Inverted Indexes

- Let us now consider inverted indexes for ranked search
  - In this case the documents are sorted by decreasing frequency of the term or other similar type of weight
- Documents that share the same frequency can be sorted in increasing order of identifiers
- This will permit the use of gap encoding to compress most of each list

Inverted Indexes

Structural Queries

Structural Queries

- Let us assume that the structure is marked in the text using **tags**
- The idea is to make the index take the tags as if they were words
- After this process, the inverted index contains all the information to answer structural queries

Structural Queries

- Consider the query:
  - select structural elements of type **A** that contain a structure of type **B**
- The query can be translated into finding `<A>` followed by `<B>` without `</A>` in between
- The positions of those tags are obtained with the full-text index
- Many queries can be translated into a search for tags plus validation of the sequence of occurrences
- In many cases this technique is efficient and its integration into an existing text database is simpler

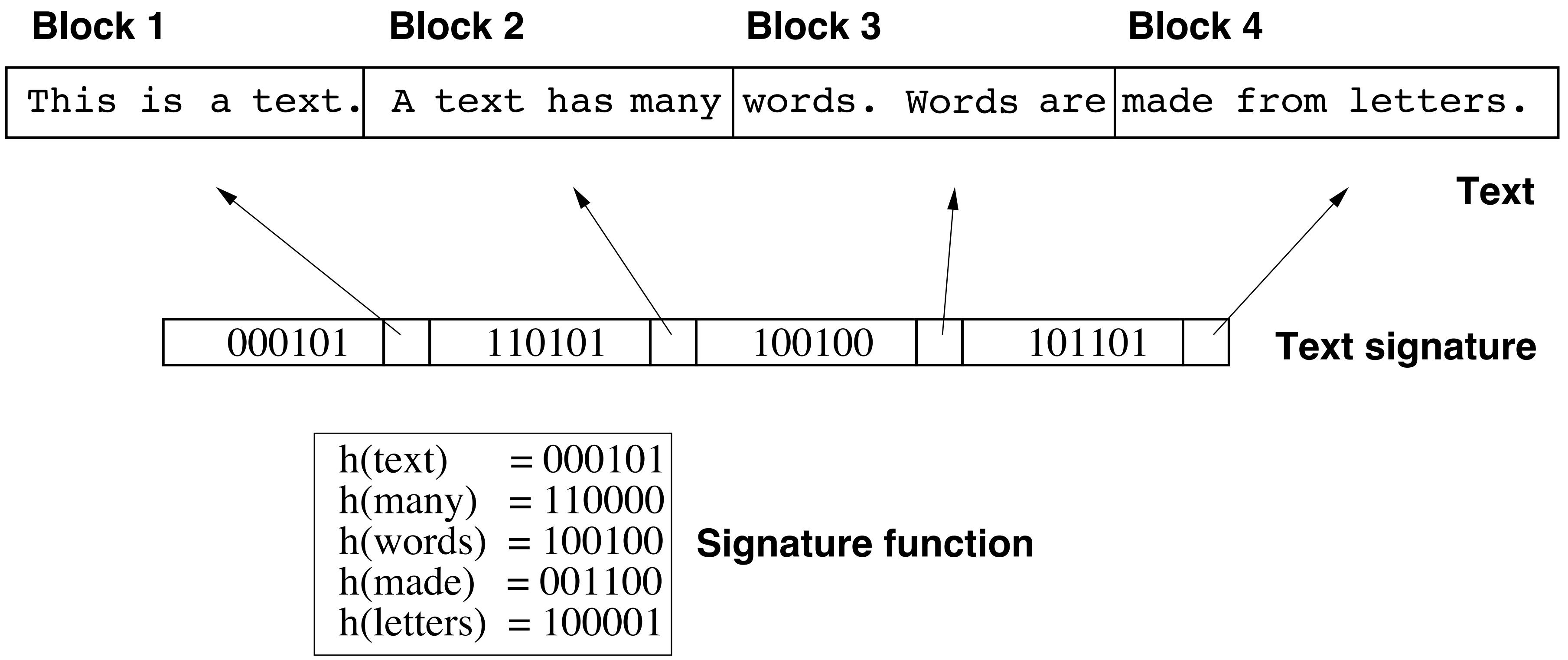
Signature Files

Signature Files

- Signature files are word-oriented index structures based on hashing
- They pose a low overhead, at the cost of forcing a sequential search over the index
- Since their search complexity is linear, it is suitable only for not very large texts
- Nevertheless, inverted indexes outperform signature files for most applications

Structure

- A signature divides the text in blocks of b words each, and maps words to bit masks of B bits
- This mask is obtained by bit-wise ORing the signatures of all the words in the text block



Structure

- If a word is present in a text block, then its signature is also set in the bit mask of the text block
- Hence, if a query signature is not in the mask of the text block, then the word is not present in the text block
- However, it is possible that all the corresponding bits are set even though the word is not there
    - This is called a **false drop**
- A delicate part of the design of a signature file is:
    - to ensure the probability of a false drop is low, and
    - to keep the signature file as short as possible

Structure

- The hash function is forced to deliver bit masks which have at least ℓ bits set
- A good model assumes that ℓ bits are randomly set in the mask (with possible repetition)

Inverted Indexes

Searching

Searching

- Searching a single word is made by comparing its bit mask W with the bit masks B_i of all the text blocks
- Whenever $(W \ \& \ B_i = W)$, where $\&$ is the bit-wise AND, the text block **may** contain the word
- Hence, an online traversal must be performed to verify if the word is actually there
- This traversal cannot be avoided as in inverted indexes (except if the risk of a false match is accepted)

Searching

- This scheme is more efficient to search phrases and reasonable proximity queries
  - This is because **all** the words must be present in a block in order for that block to hold the phrase or the proximity query
- Hence, the bit-wise OR of all the query masks is searched, so that **all** their bits must be present
  - This reduces the probability of false drops
- Some care has to be exercised at block boundaries, to avoid missing a phrase which crosses a block limit
- To search phrases of j words or proximities of up to j words, consecutive blocks must overlap in $j-1$ words
- This is the only indexing scheme which improves in phrase searching

Inverted Indexes

Construction

Construction

- The construction of a signature file is rather easy:
    - The text is simply cut in blocks, and for each block an entry of the signature file is generated
- Adding text is also easy, since it is only necessary to keep adding records to the signature file
- Text deletion is carried out by deleting the appropriate bit masks

Compression

- There are many alternative ways to compress signature files
- All of them are based on the fact that only a few bits are set in the whole file
- Compression ratios near 70% are reported

Suffix Trees and Suffix Arrays

Suffix Trees and Suffix Arrays

- Inverted indexes are by far the preferred choice to implement IR systems
- However, they work if the vocabulary is not too large, otherwise their efficiency would drastically drop
- This condition holds in many languages (particularly Western languages), but not in all
  - For example, Finnish and German are languages that concatenate short particles to form long words
- Usually, there is no point in querying for those long words, but by the particles that form them

Suffix Trees and Suffix Arrays

- Suffix trees and suffix arrays enable indexed searching for any text substring matching a query string
- These indexes regard the text as one long string, and each position in the text is considered as a text **suffix**
- For example, if the text is `missing mississippi`, the suffixes are

```
missing mississippi
issing mississippi
ssing mississippi
..
ppi
pi
i
```

Suffix Trees and Suffix Arrays

Structure

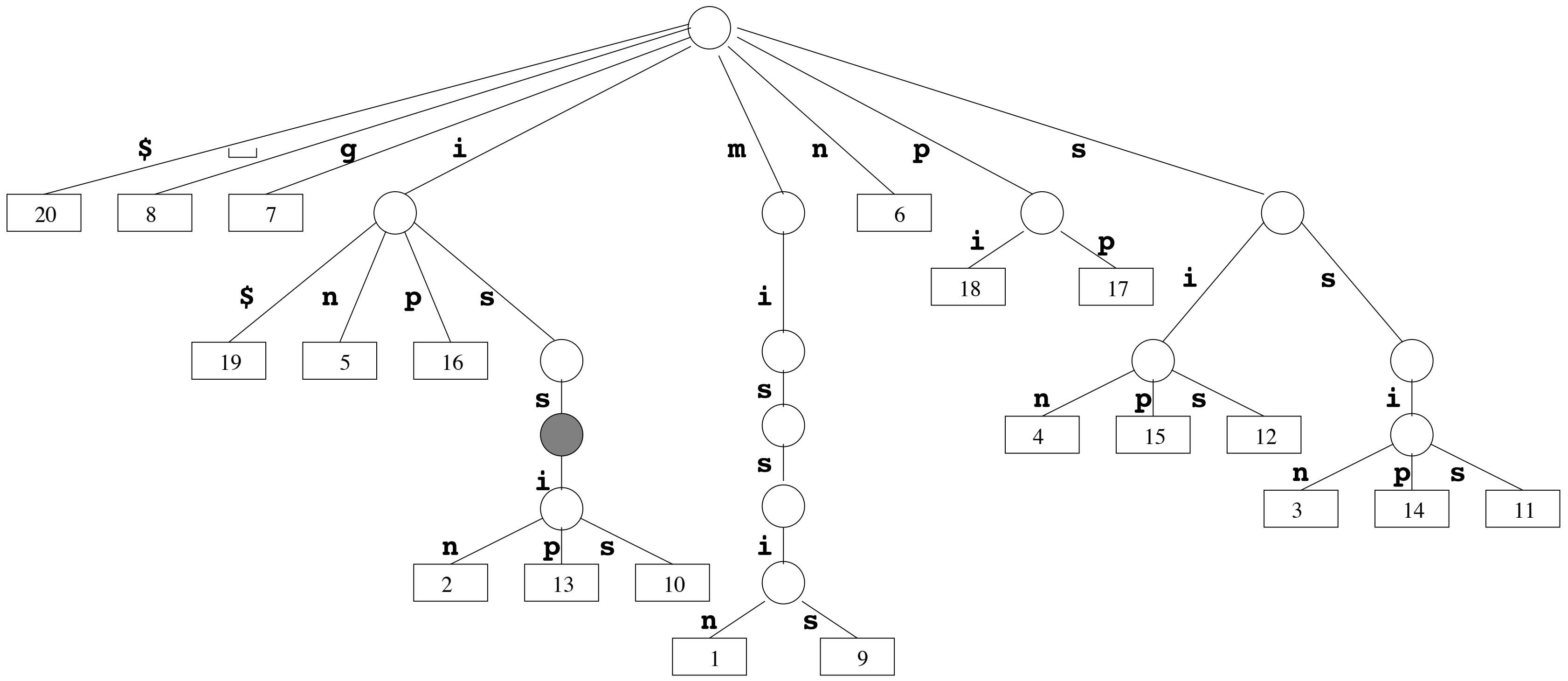
Structure

- A trie over a set of strings $\mathcal{P} = \{P_1, \ldots, P_r\}$ is a tree-shaped DFA that recognizes $P_1 \mid \ldots \mid P_r$
- Hence looking for a string in $\mathcal{P}$ is equivalent to determining whether the DFA recognizes the string
- A **suffix trie** is, in essence, a trie data structure built over all the suffixes of a text $T = t_1 t_2 \ldots t_n$, t_n, '$'
- The pointers to the suffixes $t_i \ldots t_n$ are stored at the final states

Structure

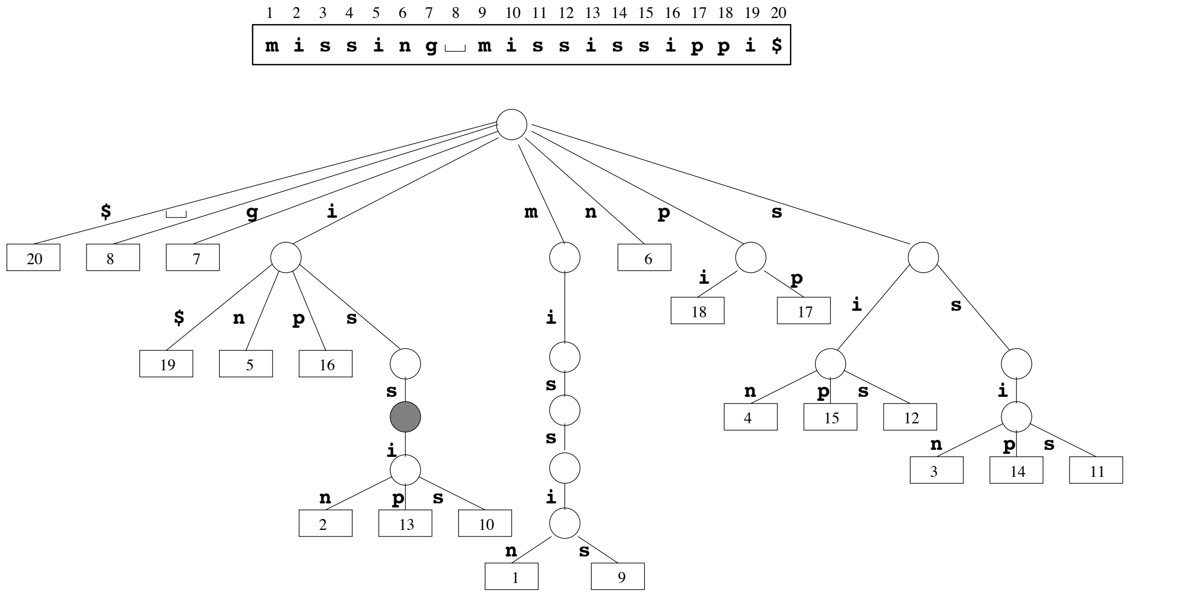
- To reduce the number of nodes in such a trie, the suffix trie removes all unary paths that finish at a leaf.
- The suffix trie for the text `missing mississippi` is

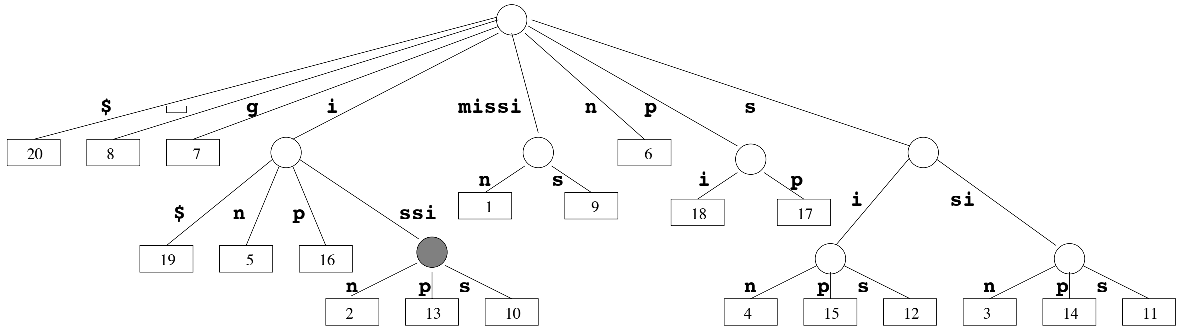
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
m	i	s	s	i	n	g	␣	m	i	s	s	i	s	s	i	p	p	i	$



Structure

- **Suffix tree:** to further reduce the space requirement, all the remaining unary paths can be compressed



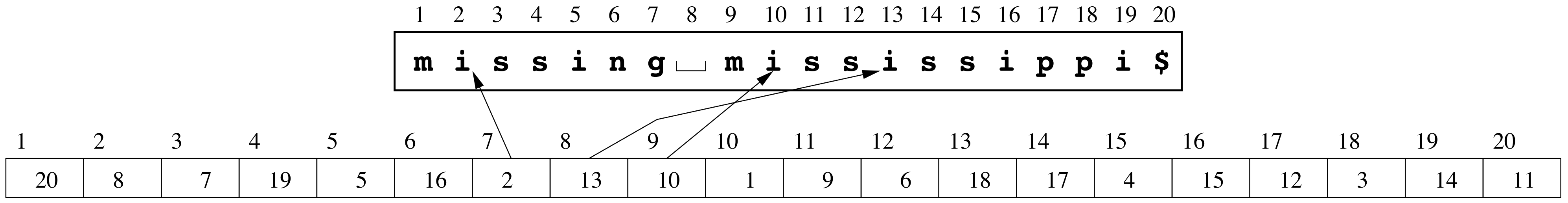


Structure

- The problem with suffix trees is their space
- Depending on the implementation, a suffix tree takes 10 to 20 times the space of the text itself
    - For example, the suffix tree of a text of 1 gigabyte would need at least 10 gigabytes of space
- In addition, suffix trees do not perform well in secondary memory

Structure

- **Suffix arrays** provide essentially the same functionality of suffix trees with much lower space requirements
- A suffix array of T is defined as an array pointing to all the suffixes of T, where suffixes have been lexicographically sorted (that is, the leaves of the suffix trie from left to right)
- The suffix array for the text `missing mississippi`:



Structure

- A suffix array takes typically 4 times the text size, which makes it appealing for longer texts
- In exchange, suffix arrays are slightly slower than suffix trees
- In some papers suffix trees and arrays are called PAT trees and arrays

Suffix Trees and Suffix Arrays

Searching for Simple Strings

Searching for Simple Strings

- The main property that permits finding substrings equal to a given pattern string $P = p_1 p_2 \ldots p_m$ is as follows:
  - **Every text substring is a prefix of a text suffix**
- The main idea is then to descend in the trie by following the characters of P
- There are three possible outcomes:
  - There might be no path in the trie spelling out P: then P does not occur in T
  - We find P before reaching a leaf: then P appears in all the positions stored in the leaves under that path
  - Maybe we arrive at a leaf before reading P completely: in this case we have to keep comparing in the text pointed by the leaf to know if P is in the text or not

Searching for Simple Strings

- If the search is carried out on a suffix tree instead, then the edges are labeled by strings
- Yet, all the strings labeling edges that depart from a given node differ in their first character
- Therefore, at each node, there is at most one edge to follow

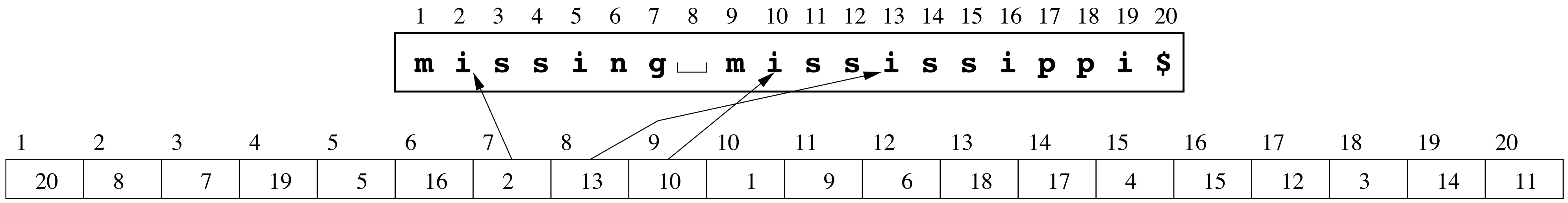
Searching for Simple Strings

- Pseudocode for a suffix tree search

```
Suffix-Tree-Search $(S,\ P = p_1 p_2 \ldots p_m)$
 (1)  $i \leftarrow 1$
 (2)  while $true$ do
 (3)    if $S$ is a leaf pointing to $j$ then
 (4)      if $p_i \ldots p_m = t_{j+i-1} \ldots t_{j+m-1}$
 (5)        then return $S$
 (6)        else return $null$
 (7)    if there is an edge $S \xrightarrow{p'_1 \ldots p'_s} S' \wedge p'_1 = p_i$ then
 (8)      $j \leftarrow 0$
 (9)      while $j < s \wedge i + j \le m \wedge p'_{j+1} = p_{i+j}$ do $j \leftarrow j + 1$
(10)      $i \leftarrow i + j$
(11)      if $i > m$ then return $S'$
(12)      if $j < s$ then return $null$
(13)      $S \leftarrow S'$
(14)    else return $null$
```

Searching for Simple Strings

- Searching in a suffix array is slightly different
- We perform a binary search with indirect comparisons



- Note that each step in this binary search requires comparing P against a text suffix

Suffix Trees and Suffix Arrays

Searching for Complex Patterns

Searching for Complex Patterns

- Searching for a complex pattern using a suffix trie it is not a trivial task
  - Assume for example we wish to search for a certain regular expression
  - We build the corresponding non-deterministic finite automaton *without* adding the initial self-loop
  - We will detect all the text suffixes that start with a string matching the regular expression

Searching for Complex Patterns

- For this sake, the algorithm begins at the trie root
- For each child of a node c, the automaton is fed with c and the algorithm recursively enters the subtree
  - When the recursion returns from the subtree, the original automaton state before feeding it with c is restored
- The process is repeated for each children of c
- The search stops in three possible forms:
  - The automaton runs out of active states
  - The automaton arrives at a final state
  - We arrive at a trie leaf and we keep searching in the suffix referenced there

Searching for Complex Patterns

- Indexed approximate string matching with tolerance k is also possible using the same idea
- Approximate searching on a trie cannot exceed depth $m + k$, and thus the time is independent on the text size for short enough patterns
  - For longer patterns the exponential dependence on m becomes apparent in the search time
- Suffix trees are able to perform other complex searches that we have not considered
- Some examples are:
  - Find the longest substring in the text that appears more than once
  - Find the most common substring of a fixed length

Suffix Trees and Suffix Arrays

Construction

Construction

- A simple way to generate a suffix array is to lexicographically sort all the pointed suffixes
- To compare two suffix array entries in this sorting, the corresponding text positions must be accessed
- There are several much stronger sorting algorithms for suffix arrays
  - **The main idea:** if we know that $t_{i+1} \ldots t_n < t_{j+1} \ldots t_n$ and $t_i = t_j$, then we directly infer that $t_i \ldots t_n < t_j \ldots t_n$
- Different algorithms build on this idea in different ways

Suffix Trees and Suffix Arrays

Construction for Large Texts

Construction for Large Texts

- When the data is not in main memory, algorithms for secondary memory construction are required
- We present an algorithm that splits the text into blocks that can be sorted in main memory
- For each block, it builds the suffix array of the block in main memory, and merges it with the rest of the suffix array already built for the preceding text:

  (1) build the suffix array for block 1
  (2) build the suffix array for block 2
  (3) merge the suffix array for block 2 with that of block 1
  (4) build the suffix array for block 3
  (5) merge the suffix array for block 3 with that of block 1+2
  (6) .... etc

Construction for Large Texts

- How to merge a large suffix array LA for blocks $1, 2, \ldots, i-1$ with the small suffix array SA for block i?
- The solution is to determine how many elements of LA are to be placed between the elements in SA
  - The information is stored in a **counter** array C: $C[j]$ tells how many suffixes of LA lie between $SA[j]$ and $SA[j+1]$
- Once C is computed, LA and SA are easily merged:

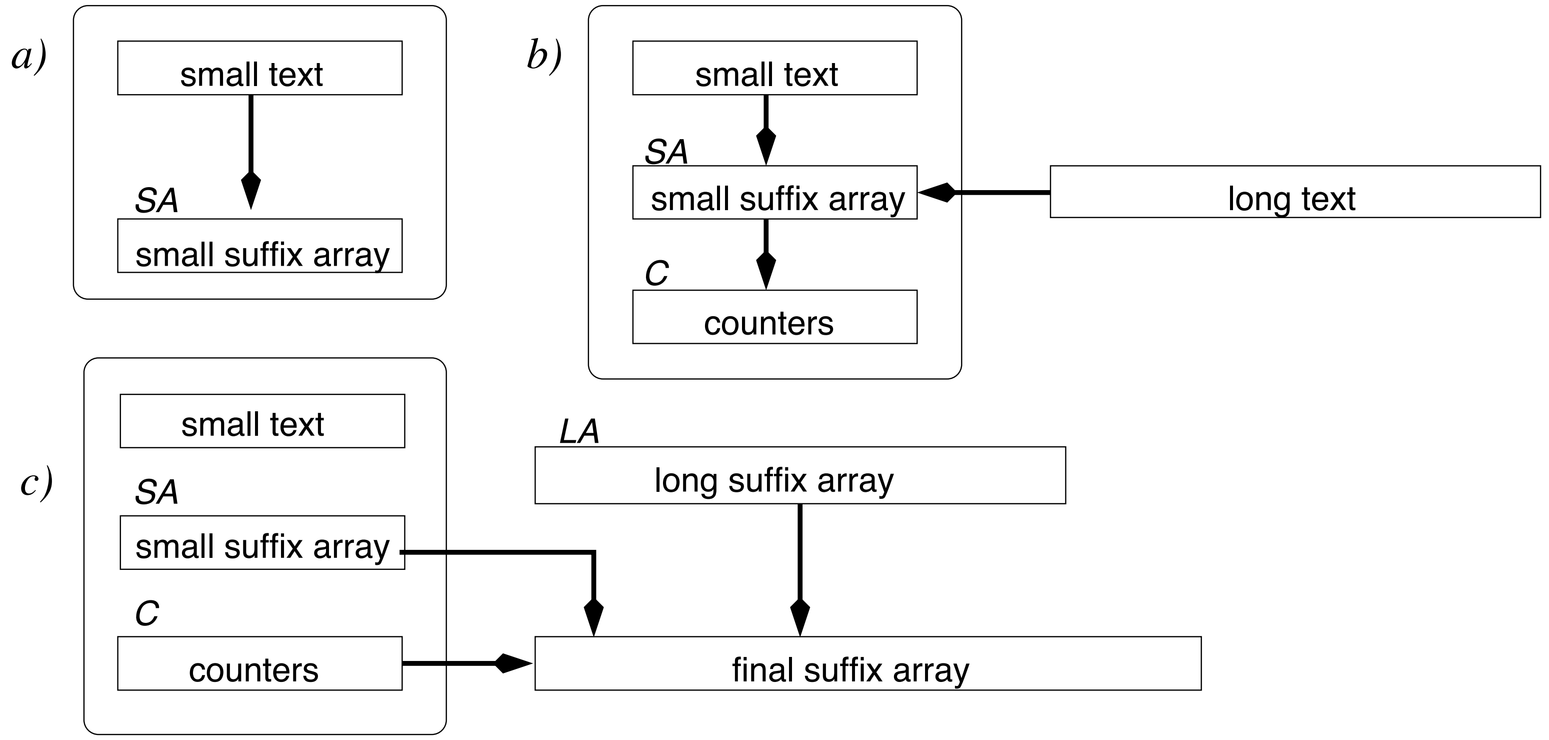
  (1) append the first C[0] elements of LA
  (2) append SA[1]
  (3) append the next C[1] elements of LA
  (4) append SA[2]
  (5) append the next C[2] elements of LA
  (6) append SA[3]
  (7) .... etc

Construction for Large Texts

- The remaining point is how to compute the counter array C
  - This is done **without** accessing LA: the **text** corresponding to LA is sequentially read into main memory
- Each suffix of that text is **searched for** in SA (in main memory)
- Once we determine that the text suffix lies between $SA[j]$ and $SA[j+1]$, we increment $C[j]$
- Notice that this same algorithm can be used for index maintenance

Construction for Large Texts

- A step of the suffix array construction for large texts:
  (a) the local suffix array SA is built
  (b) the counters C are computed
  (c) suffix arrays SA and LA are merged

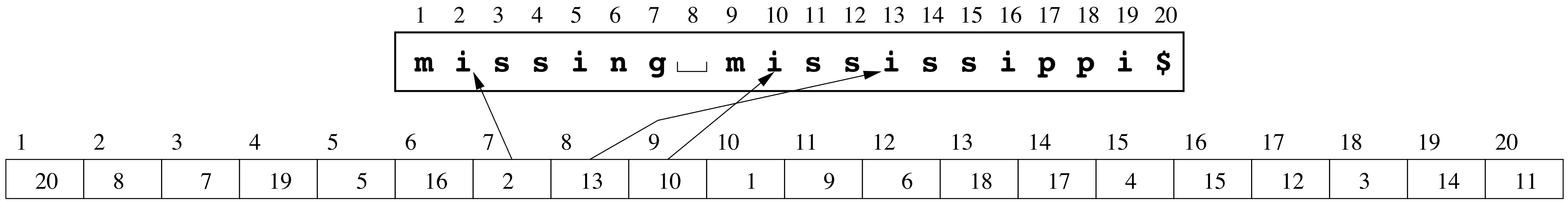


Suffix Trees and Suffix Arrays

Compressed Suffix Arrays

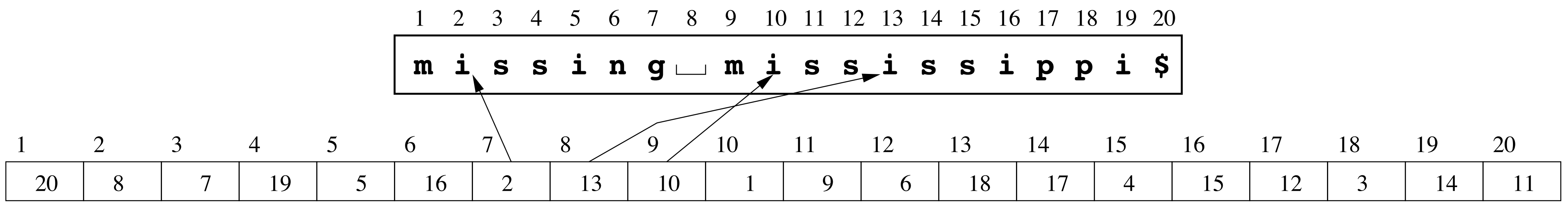
Compressed Suffix Arrays

- An important problem of suffix arrays is their high space requirement
- Consider again the suffix array of the Figure below, and call it $A[1, n]$



- The values at $A[15..17]$ are 4, 15, 12
- The same sequence is found, displaced by one value, at $A[18..20]$, and further displaced at $A[7..9]$

Compressed Suffix Arrays



- A compressor can realize that every time it has seen `si`, the next character it will see is `s` and then `i`
    - This is related to k-th order compression
- By exploiting these regularities, the suffix array of a compressible text can also be compressed
- Manipulating a suffix array in compressed form is indeed slower than using the uncompressed suffix array
- There is not much development on using suffix array indexes (compressed or not) on disk

Using Function Ψ

- One way to exhibit the suffix array regularities is by means of a function called Ψ and defined so that

$$A[\Psi(i)] \quad = \quad A[i] + 1,$$

  except when $A[i] = n$, in which case $A[\Psi(i)] = 1$

- That is, $\Psi(i)$ tells where in A is the value that follows the current one

Using Function Ψ

- Function Ψ computed for the example suffix array, and *diff*$(i) = \Psi(i) - \Psi(i-1)$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	m	i	s	s	i	n	g	␣	m	i	s	s	i	s	s	i	p	p	i	$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	20	8	7	19	5	16	2	13	10	1	9	6	18	17	4	15	12	3	14	11
Ψ	10	11	2	1	12	14	18	19	20	7	9	3	4	13	5	6	8	15	16	17
diff	10	1	– 9	– 1	11	2	4	1	1	– 13	2	– 6	1	9	– 8	1	2	7	1	1
	$	␣	g				i			m		n	p					s		

- We indicate the areas of the arrays where the suffixes start with the same character

Using Function Ψ

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	m	i	s	s	i	n	g	␣	m	i	s	s	i	s	s	i	p	p	i	$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	20	8	7	19	5	16	2	13	10	1	9	6	18	17	4	15	12	3	14	11
Ψ	10	11	2	1	12	14	18	19	20	7	9	3	4	13	5	6	8	15	16	17
diff	10	1	– 9	– 1	11	2	4	1	1	– 13	2	– 6	1	9	– 8	1	2	7	1	1
	$	␣	g	i						m		n	p		s					

- Several properties of Ψ are apparent from the figure and not difficult to prove:
  - First, Ψ is increasing within the areas where the suffixes start with the same character
  - Second, in the areas where the regularities we pointed out before arise, it holds $\Psi(i) - \Psi(i-1) = 1$

Using Function Ψ

- What is most interesting is that the characters of T can be obtained without accessing T
- Therefore, T can be actually **deleted** and any substring of it can be obtained just from Ψ
- However, this is rarely sufficient, as one usually wants to know the text positions where the pattern P occurs, not the interval in A
- Yet, we do not have A in order to display the positions of the occurrences, $A[i]$ for $sp \leq i \leq ep$

Using Function Ψ

- To be able to locate those occurrences in T, we **sample** A at regular intervals of T:
  - Every s-th character in T, record the suffix array position pointing to that text position
- That is, for each text position of the form $1 + j \cdot s$, let $A[i] = 1 + j \cdot s$
- Then we store the pair $(i, A[i])$ in a dictionary searchable by its first component
- Finally, we should also be able to display any text substring, as we plan to discard T
- Note that we already know how to obtain the characters of T starting at text position $A[i]$, given i

The Burrows-Wheeler Transform

- A radically different method for compressing is by means of the Burrows-Wheeler Transform (BWT)
- The BWT of T can be obtained by just concatenating the characters that **precede** each suffix in A
  - That is, $t_{A[i]-1}$ or t_n if $A[i] = 1$
- For example, the BWT of $T =$ `missing mississippi$` is $T^{bwt} =$ `ignpssmsm$ ipisssiii`
- It turns out that the BWT tends to group equal characters into runs
- Further, there are large zones where few different characters appear

Sequential Searching

Sequential Searching

- In general the sequential search problem is:
  - Given a text $T = t_1 t_2 \ldots t_n$ and a pattern denoting a set of strings $\mathcal{P}$, find all the occurrences of the strings of $\mathcal{P}$ in T
- **Exact string matching:** the simplest case, where the pattern denotes just a single string $P = p_1 p_2 \ldots p_m$
- This problem subsumes many of the basic queries, such as word, prefix, suffix, and substring search
- We assume that the strings are sequences of characters drawn from an alphabet Σ of size σ

Sequential Searching

Simple Strings

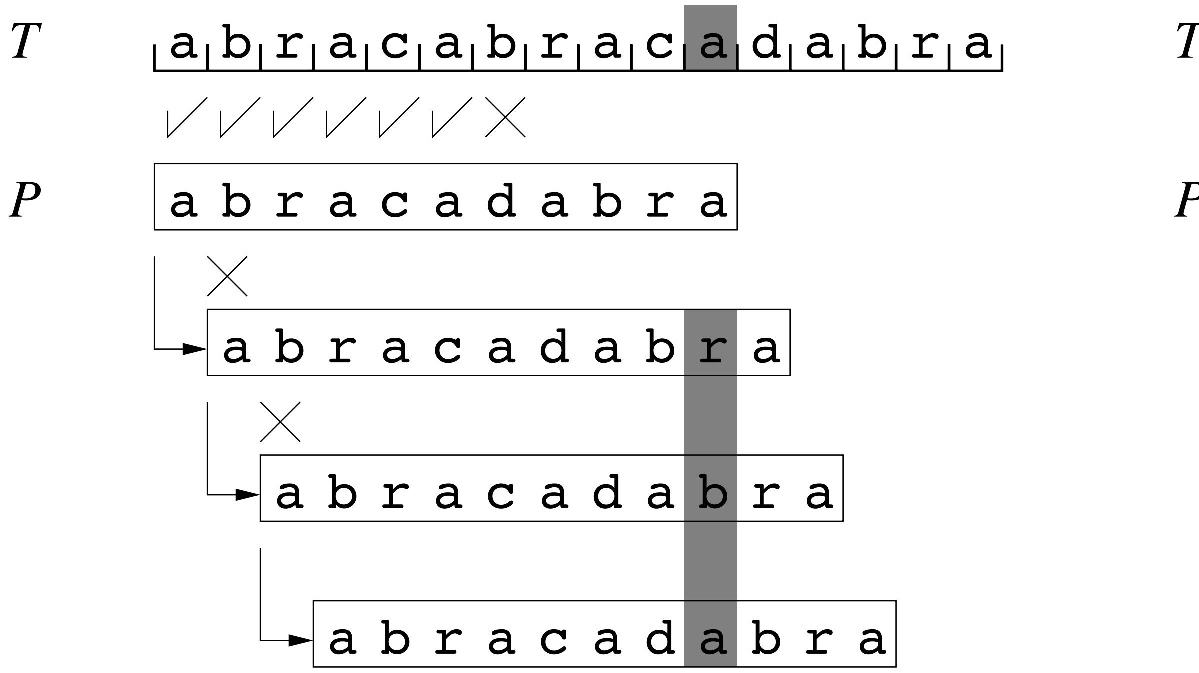
Simple Strings: Brute Force

- The **brute force** algorithm:
  - Try out all the possible pattern positions in the text and checks them one by one
- More precisely, the algorithm slides a **window** of length m across the text, $t_{i+1}t_{i+2}\ldots t_{i+m}$ for $0 \leq i \leq n-m$
- Each window denotes a potential pattern occurrence that must be verified
- Once verified, the algorithm slides the window to the next position

Simple Strings: Brute Force

- A sample text and pattern searched for using brute force

  - The first text window is `abracabraca`
  - After verifying that it does not match P, the window is shifted by one position

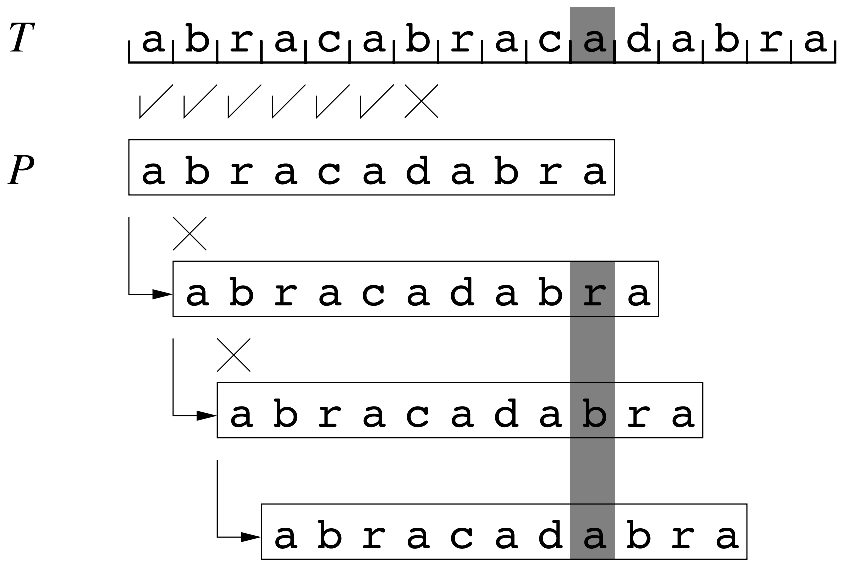


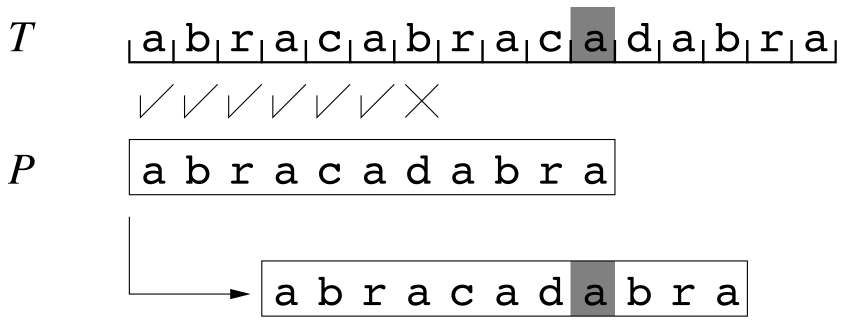
Simple Strings: Horspool

- **Horspool's algorithm** is in the fortunate position of being very simple to understand and program
- It is the fastest algorithm in many situations, especially when searching natural language texts
- Horspool's algorithm uses the previous idea to shift the window in a smarter way
- A table d indexed by the characters of the alphabet is precomputed:
    - $d[c]$ tells how many positions can the window be shifted if the final character of the window is c
- In other words, $d[c]$ is the distance from the end of the pattern to the last occurrence of c in P, excluding the occurrence of p_m

Simple Strings: Horspool

- The Figure repeats the previous example, now also applying Horspool's shift





Simple Strings: Horspool

- Pseudocode for Horspool's string matching algorithm

```
**Horspool** $(T = t_1 t_2 \ldots t_n,\ P = p_1 p_2 \ldots p_m)$
(1)  **for** $c \in \Sigma$ **do** $d[c] \leftarrow m$
(2)  **for** $j \leftarrow 1 \ldots m-1$ **do** $d[p_j] \leftarrow m - j$
(3)  $i \leftarrow 0$
(4)  **while** $i \leq n - m$ **do**
(5)      $j \leftarrow 1$
(6)      **while** $j \leq m \wedge t_{i+j} = p_j$ **do** $j \leftarrow j + 1$
(7)      **if** $j > m$ **then** report an occurrence at text position $i + 1$
(8)      $i \leftarrow i + d[t_{i+m}]$
```

Small alphabets and long patterns

- When searching for long patterns over small alphabets Horspool’s algorithm does not perform well
  - Imagine a computational biology application where strings of 300 nucleotides over the four-letter alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ are sought
- This problem can be alleviated by considering consecutive pairs of characters to shift the window
  - On other words, we can align the pattern with the last pair of window characters, $t_{i+m-1}t_{i+m}$
- In the previous example, we would shift by $4^2 = 16$ positions on average

Small alphabets and long patterns

- In general we can shift using q characters at the end of the window: which is the best value for q?
  - We cannot shift by more than m, and thus $\sigma^q \leq m$ seems to be a natural limit
  - If we set $q = \log_\sigma m$, the average search time will be $O(n \log_\sigma(m)/m)$
- Actually, this average complexity is optimal, and the choice for q we derived is close to correct
- It can be analytically shown that, by choosing $q = 2\log_\sigma m$, the average search time achieves the optimal $O(n \log_\sigma(m)/m)$

Small alphabets and long patterns

- This technique is used in the **agrep** software
- A hash function is chosen to map q-grams (strings of length q) onto an integer range
- Then the distance from each q-gram of P to the end of P is recorded in the hash table
- For the q-grams that do not exist in P, distance $m - q + 1$ is used

Small alphabets and long patterns

- Pseudocode for the **agrep**'s algorithm to match long patterns over small alphabets (simplified)

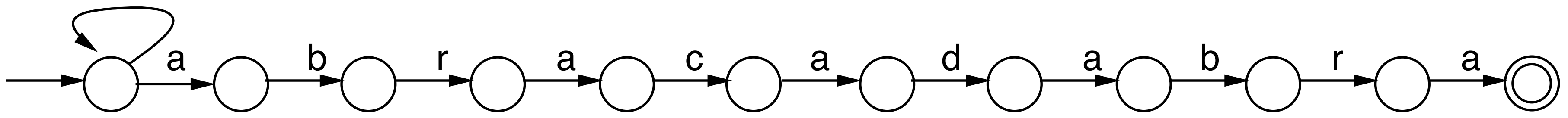
```
**Agrep** $(T = t_1t_2\ldots t_n,\ P = p_1p_2\ldots p_m,\ q,\ h(\ ),\ N)$
(1)  **for** $i \in [1, N]$ **do** $d[i] \leftarrow m - q + 1$
(2)  **for** $j \leftarrow 0 \ldots m - q$ **do** $d[h(p_{j+1}p_{j+2}\ldots p_{j+q})] \leftarrow m - q - j$
(3)  $i \leftarrow 0$
(4)  **while** $i \leq n - m$ **do**
(5)      $s \leftarrow d[h(t_{i+m-q+1}t_{i+m-q+2}\ldots t_{i+m})]$
(6)      **if** $s > 0$ **then** $i \leftarrow i + s$
(7)      **else**
(8)          $j \leftarrow 1$
(9)          **while** $j \leq m \wedge t_{i+j} = p_j$ **do** $j \leftarrow j + 1$
(10)         **if** $j > m$ **then** report an occurrence at text position $i + 1$
(11)         $i \leftarrow i + 1$
```

Automata and Bit-Parallelism

- Horspool’s algorithm, as well as most classical algorithms, does not adapt well to complex patterns
- We now show how **automata** and **bit-parallelism** permit handling many complex patterns

Automata

- Figure below shows, on top, a NFA to search for the pattern $P =$ `abracadabra`
  - The initial self-loop matches any character
  - Each table column corresponds to an edge of the automaton



B[a] =	0	1	1	0	1	0	1	0	1	1	0
B[b] =	1	0	1	1	1	1	1	1	0	1	1
B[r] =	1	1	0	1	1	1	1	1	1	0	1
B[c] =	1	1	1	1	0	1	1	1	1	1	1
B[d] =	1	1	1	1	1	1	0	1	1	1	1
B[*] =	1	1	1	1	1	1	1	1	1	1	1

Automata

- It can be seen that the NFA in the previous Figure accepts any string that finishes with $P =$ `‘abracadabra’`
- The initial state is always active because of the self-loop that can be traversed by any character
- Note that several states can be simultaneously active
  - For example, after reading `‘abra’`, NFA states 0, 1, and 4 will be active

Bit-parallelism and Shift-And

- **Bit-parallelism** takes advantage of the intrinsic parallelism of bit operations
- Bit masks are read right to left, so that the first bit of $b_m \ldots b_1$ is b_1
- Bit masks are handled with operations like:
  - $|$ to denote the bit-wise **or**
  - $\&$ to denote the bit-wise **and**, and
  - $\hat{}$ to denote the bit-wise **xor**
- Unary operation '$\sim$' complements all the bits

Bit-parallelism and Shift-And

- In addition:
  - $mask << i$ means shifting all the bits in $mask$ by i positions to the left, entering zero bits from the right
  - $mask >> i$ is analogous
- Finally, it is possible to operate bit masks as numbers, for example adding or subtracting them

Bit-parallelism and Shift-And

- The simplest bit-parallel algorithm permits matching single strings, and it is called **Shift-And**
- The algorithm builds a table B which, for each character, stores a bit mask $b_m \ldots b_1$
  - The mask in $B[c]$ has the i-th bit set if and only if $p_i = c$
- The state of the search is kept in a machine word $D = d_m \ldots d_1$, where d_i is set if the state i is active
  - Therefore, a match is reported whenever $d_m = 1$
- Note that state number zero is not represented in D because it is always active and then can be left implicit

Bit-parallelism and Shift-And

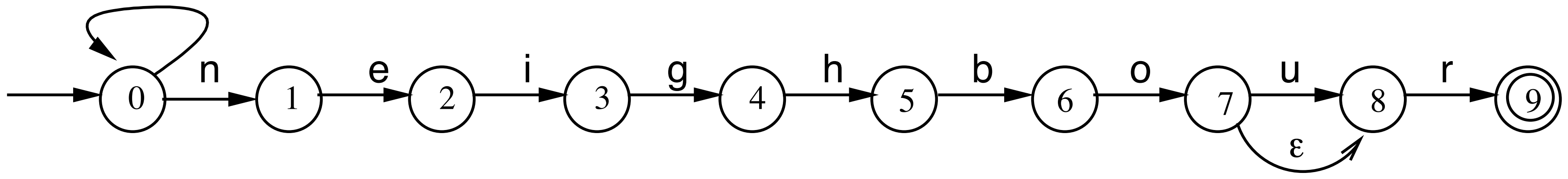
- Pseudocode for the Shift-And algorithm

```
**Shift-And** $(T = t_1 t_2 \ldots t_n,\ P = p_1 p_2 \ldots p_m)$

(1)  for $c \in \Sigma$ do $B[c] \leftarrow 0$
(2)  for $j \leftarrow 1 \ldots m$ do $B[p_j] \leftarrow B[p_j] \mid (1 << (j-1))$
(3)  $D \leftarrow 0$
(4)  for $i \leftarrow 1 \ldots n$ do
(5)     $D \leftarrow ((D << 1) \mid 1) \ \& \ B[t_i]$
(6)     if $D \ \& \ (1 << (m-1)) \neq 0$
(7)     then report an occurrence at text position $i - m + 1$
```

- There must be sufficient bits in the computer word to store one bit per pattern position
  - For longer patterns, in practice we can search for $p_1 p_2 \ldots p_w$, and directly check the occurrences of this prefix for the complete P

Extending Shift-And

- Shift-And can deal with much more complex patterns than Horspool
- The simplest case is that of **classes of characters**:
  - This is the case, for example, when one wishes to search in case-insensitive fashion, or one wishes to look for a whole word
- Let us now consider a more complicated pattern
  - Imagine that we search for `neighbour`, but we wish the `u` to be optional (accepting both English and American style)
- The Figure below shows an NFA that does the task using an ε-transition

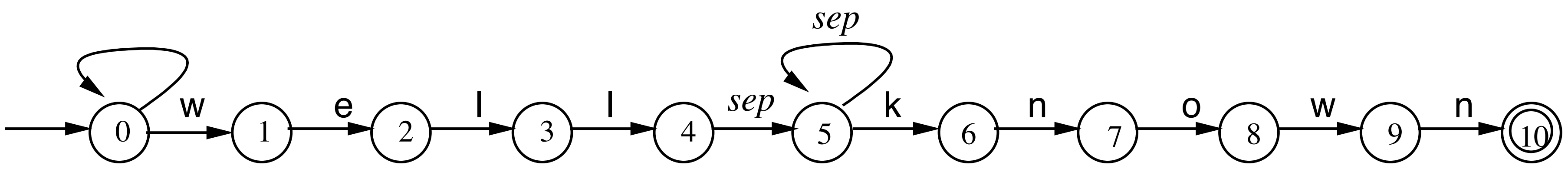


Extending Shift-And

- Another feature in complex patterns is the use of **wild cards**, or more generally repeatable characters
    - Those are pattern positions that can appear once or more times, consecutively, in the text
- For example, we might want to catch all the transfer records in a banking log

Extending Shift-And

- As another example, we might look for `well known`, yet there might be a hyphen or one or more spaces
  - For instance `‘well known’`, `‘well  known’`, `‘well-known’`, `‘well - known’`, `‘well` $\backslash n$ `known’`, and so on



Extending Shift-And

- Figure below shows pseudocode for a Shift-And extension that handles all these cases

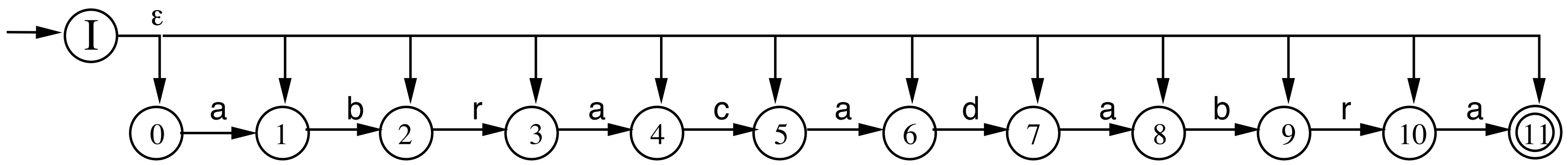
```
Shift-And-Extended $(T = t_1 t_2 \ldots t_n,\ m,\ B[\,],\ A,\ S)$

(1) $I \leftarrow (A >> 1)\ \&\ (A\,\hat{}\,(A >> 1))$
(2) $F \leftarrow A\ \&\ (A\,\hat{}\,(A >> 1))$
(3) $D \leftarrow 0$
(4) for $i \leftarrow 1 \ldots n$ do
(5)     $D \leftarrow (((D << 1) \mid 1) \mid (D\ \&\ S))\ \&\ B[t_i]$
(6)     $Df \leftarrow D \mid F$
(7)     $D \leftarrow D \mid (A\ \&\ ((\sim (Df - I))\,\hat{}\,Df))$
(8)     if $D\ \&\ (1 << (m-1)) \neq 0$
(9)     then report an occurrence at text position $i - m + 1$
```

Faster Bit-Parallel Algorithms

- There exist some algorithms that can handle complex patterns and still skip text characters (like Horspool)
  - For instance, Suffix Automata and Interlaced Shift-And algorithms
- Those algorithms run progressively slower as the pattern gets more complex

Suffix Automata

- The **suffix automaton** of a pattern P is an automaton that recognizes all the suffixes of P
- Below we present a non-deterministic suffix automaton for $P =$ `'abracadabra'`



Suffix Automata

- To search for pattern P, the suffix automaton of P^{rev} (the reversed pattern) is built
- The algorithm scans the text window backwards and feeds the characters into the suffix automaton of P^{rev}
- If the automaton runs out of active states after scanning $t_{i+m}t_{i+m-1}\ldots t_{i+j}$, this means that $t_{i+j}t_{i+j+1}\ldots t_{i+m}$ is not a substring of P
- Thus, no occurrence of P can contain this substring, and the window can be safely shifted past t_{i+j}
- If, instead, we reach the beginning of the window and the automaton still has active states, this means that the window is equal to the pattern

Suffix Automata

- The need to implement the suffix automaton and make it deterministic makes the algorithm more complex
- An attractive variant, called BNDM, implements the suffix automaton using bit-parallelism
- It achieves improved performance when the pattern is not very long
    - say, at most twice the number of bits in the computer word

Suffix Automata

- Pseudocode for BNDM algorithm:

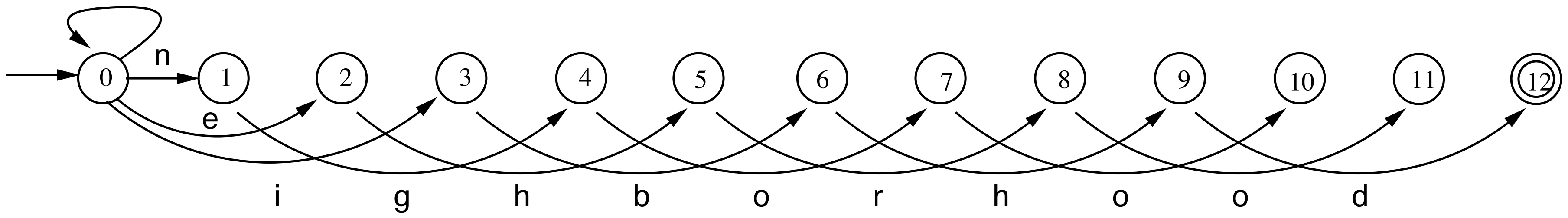
```
**BNDM** $(T = t_1t_2 \ldots t_n,\ P = p_1p_2 \ldots p_m)$

(1)  for $c \in \Sigma$ do $B[c] \leftarrow 0$
(2)  for $j \leftarrow 1 \ldots m$ do $B[p_j] \leftarrow B[p_j] \mid (1 << (m - j))$
(3)  $i \leftarrow 0$
(4)  while $i \leq n - m$ do
(5)     $j \leftarrow m - 1$
(6)     $D \leftarrow B[t_{i+m}]$
(7)     while $j > 0 \ \wedge \ D \neq 0$ do
(8)        $D \leftarrow (D << 1)\ \&\ B[t_{i+j}]$
(9)        $j \leftarrow j - 1$
(10)    if $D \neq 0$ then report an occurrence at text position $i + 1$
(11)    $i \leftarrow i + j + 1$
```

Interlaced Shift-And

- Another idea to achieve optimal average search time is to read one text character out of q
- To fix ideas, assume $P = \texttt{neighborhood}$ and $q = 3$
- If we read one text position out of 3, and P occurs at some text window $t_{i+1}t_{i+2}\dots t_{i+m}$ then we will read either `‘ngoo’`, `‘ehro’`, or `‘ibhd’` at the window
- Therefore, it is sufficient to search simultaneously for the three subsequences of P

Interlaced Shift-And

- Now the initial state can activate the first q positions of P, and the bit-parallel shifts are by q positions
- A non-deterministic suffix automaton for interlaced searching of $P =$ `‘neighborhood’` with $q = 3$ is:



Interlaced Shift-And

- Pseudocode for Interlaced Shift-And algorithm with sampling step q (simplified):

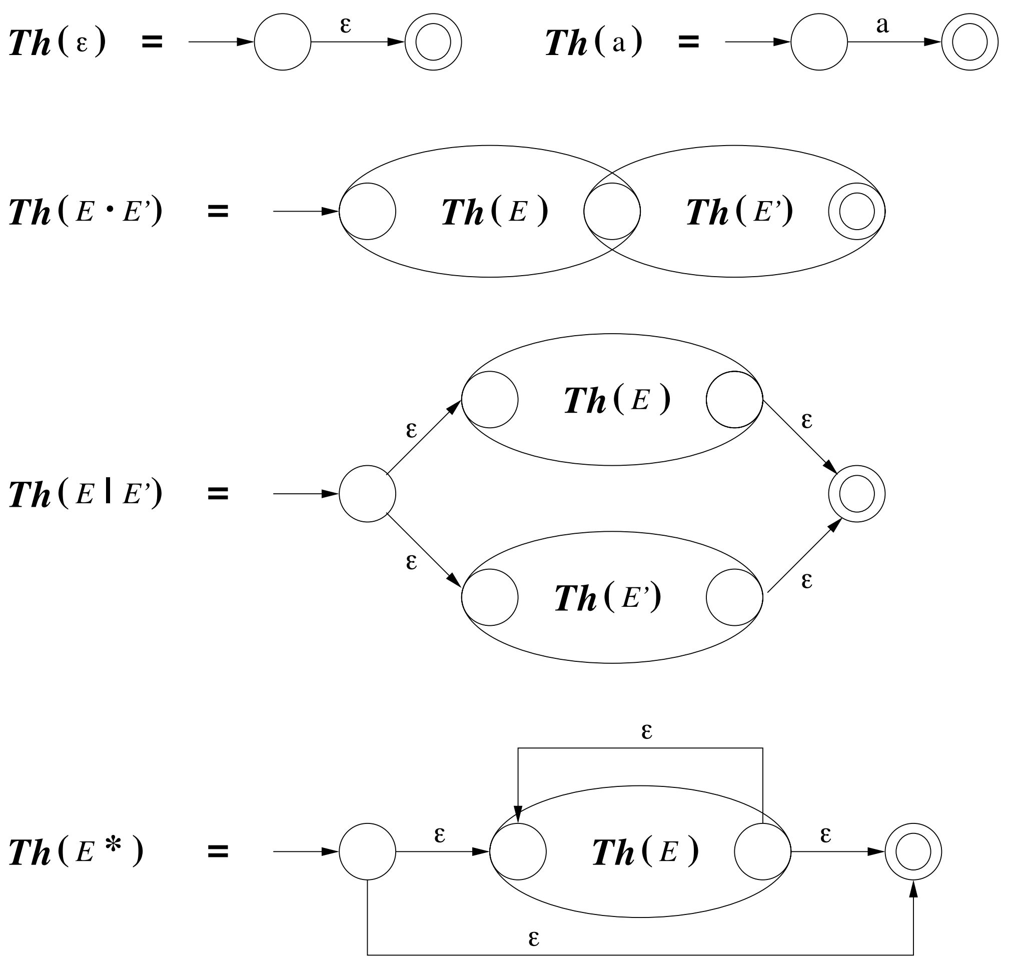
```
Interlaced-Shift-And $(T = t_1 t_2 \ldots t_n,\ P = p_1 p_2 \ldots p_m,\ q)$

(1)  for $c \in \Sigma$ do $B[c] \leftarrow 0$
(2)  for $j \leftarrow 1 \ldots m$ do $B[p_j] \leftarrow B[p_j] \mid (1 << (j-1))$
(3)  $S \leftarrow (1 << q) - 1$
(4)  $D \leftarrow 0$
(5)  for $i \leftarrow 1 \ldots \lfloor n/q \rfloor$ do
(6)     $D \leftarrow ((D << q) \mid S)\ \&\ B[t_{q \cdot i}]$
(7)     if $D\ \&\ (S << (\lfloor m/q \rfloor \cdot q - q)) \neq 0$
(8)     then run Shift-And over $t_{q \cdot i - m + 1} \ldots t_{q \cdot i + q - 1}$
```

Regular Expressions

- The first part in processing a regular expression is to build an NFA from it
    - There are different NFA construction methods
- We present the more traditional Thompson’s technique as it is simpler to explain

Regular Expressions

- Recursive Thompson’s construction of an NFA from a regular expression



Regular Expressions

- Once the NFA is built we add a self-loop (traversable by any character) at the initial state
- Another alternative is to make the NFA deterministic, converting it into a DFA
- However the number of states can grow non linearly, even exponentially in the worst case

Multiple Patterns

- Several of the algorithms for single string matching can be extended to handle multiple strings $\mathcal{P} = \{P_1,\ P_2,\ \ldots,\ P_r\}$
  - For example, we can extend Horspool so that $d[c]$ is the minimum over the $d_i[c]$ values of the individual patterns P_i
- To compute each d_i we must truncate P_i to the length of the shortest pattern in $\mathcal{P}$, and that length will be m
- Other variants that perform well are extensions of BNDM
- Yet, bit-parallel algorithms are not useful for this case

Approximate Searching

- A simple string matching problem where not only a string P must be reported, but also text positions where P occurs with at most k ‘errors’
- Different definitions of what is an error can be adopted
  - The simplest definition is the **Hamming distance** that allows just substitutions of characters
- A very popular one corresponds to the so-called **Levenshtein or edit distance**:
  - A error is the deletion, insertion, or substitution of a single character
- This model is simple enough to permit fast searching, being useful for most IR scenarios
- This can be extended to approximate pattern matching

Dynamic Programming

- The classical solution to approximate string matching is based on dynamic programming
- A matrix $C[0..m, 0..n]$ is filled column by column, where $C[i, j]$ represents the minimum number of errors needed to match $p_1 p_2 \ldots p_i$ to some suffix of $t_1 t_2 \ldots t_j$
- This is computed as follows:

$$\begin{aligned}
C[0, j] &= 0, \\
C[i, 0] &= i, \\
C[i, j] &= \text{if } (p_i = t_j) \text{ then } C[i-1, j-1] \\
&\quad\ \text{else } 1 + \min(C[i-1, j], C[i, j-1], C[i-1, j-1]),
\end{aligned}$$

where a match is reported at text positions j such that $C[m, j] \leq k$

Dynamic Programming

- The dynamic programming algorithm to search for `'colour'` in the text `kolorama` with $k = 2$ errors

		k	o	l	o	r	a	m	a
	0	0	0	0	0	0	0	0	0
c	1	1	1	1	1	1	1	1	1
o	2	2	1	2	1	2	2	2	2
l	3	3	2	1	2	2	3	3	3
o	4	4	3	2	1	2	3	4	4
u	5	5	4	3	2	2	3	4	5
r	6	6	5	4	3	2*	3	4	5

- The starred entry indicates a position finishing an approximate occurrence

Dynamic Programming

- The previous algorithm requires $O(mn)$ time
- Several extensions of it have been presented that achieve $O(kn)$ time
  - A simple $O(kn)$ algorithm is obtained by computing each column only up to the point where one knows that all the subsequent cell values will exceed k
- The memory needed can also be reduced to $O(kn)$

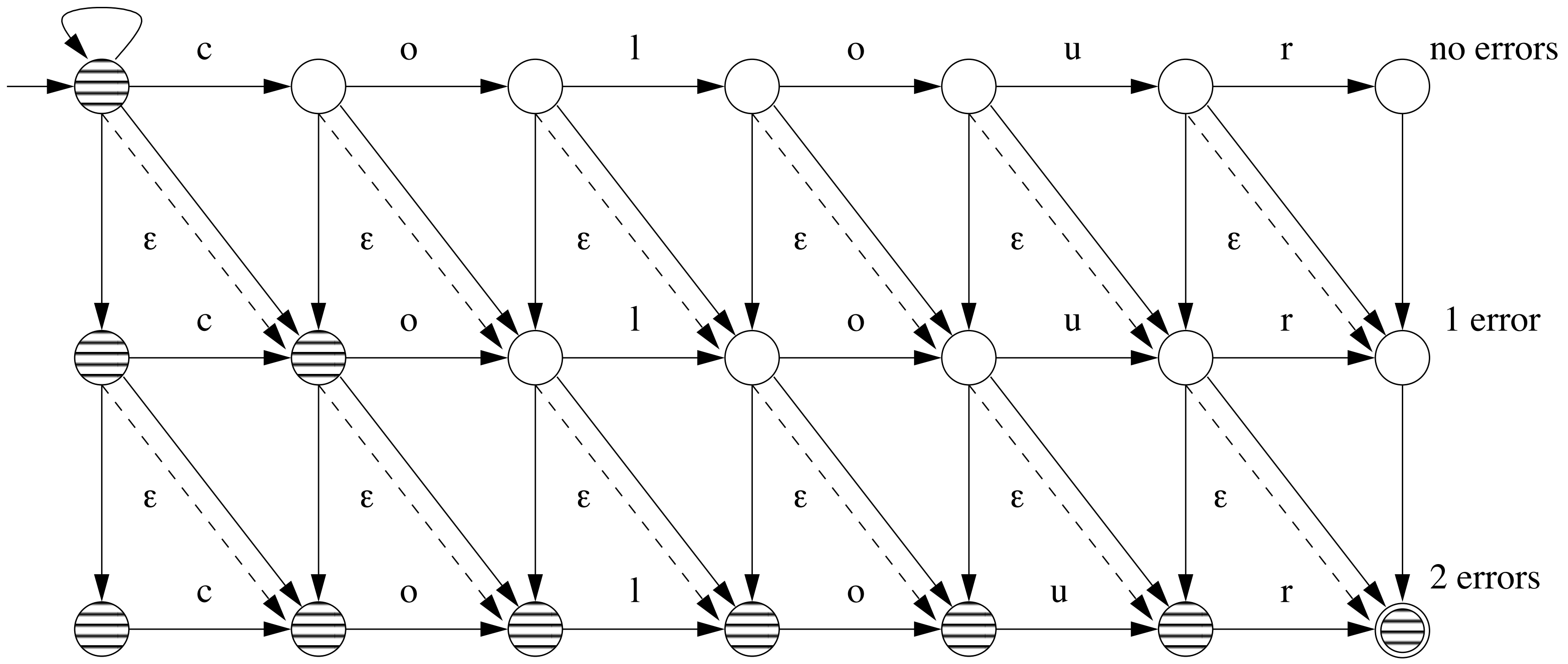
Dynamic Programming

- Figure below gives the pseudocode for this variant

```
**Approximate-DP** $(T = t_1t_2\dots t_n,\ P = p_1p_2\dots p_m,\ k)$
(1)  for $i \leftarrow 0\dots m$ do $C[i] \leftarrow i$
(2)  $last \leftarrow k + 1$
(3)  for $j \leftarrow 1\dots n$ do
(4)    $pC, nC \leftarrow 0$
(5)    for $i \leftarrow 1\dots last$ do
(6)      if $p_i = t_j$ then $nC \leftarrow pC$
(7)      else
(8)        if $pC < nC$ then $nC \leftarrow pC$
(9)        if $C[i] < nC$ then $nC \leftarrow C[i]$
(10)       $nC \leftarrow nC + 1$
(11)     $pC \leftarrow C[i]$
(12)     $C[i] \leftarrow nC$
(13)   if $nC \leq k$
(14)   then if $last = m$ then report an occurrence ending at position $i$
(15)          else $last \leftarrow last + 1$
(16)   else while $C[last - 1] > k$ do $last \leftarrow last - 1$
```

Automata and Bit-parallelism

- Approximate string matching can also be expressed as an NFA search
- Figure below depicts an NFA for approximate string matching for the pattern `‘colour’` with two errors



Automata and Bit-parallelism

- Although the search phase is $O(n)$, the NFA tends to be large ($O(kn)$)
- A better solution, based on bit-parallelism, is an extension of Shift-And
- We can simulate $k + 1$ Shift-And processes while taking care of vertical and diagonal arrows as well

Automata and Bit-parallelism

- Pseudocode for approximate string matching using the Shift-And algorithm

```
Approximate-Shift-And $(T = t_1 t_2 \ldots t_n,\ P = p_1 p_2 \ldots p_m, k)$
 (1)  for $c \in \Sigma$ do $B[c] \leftarrow 0$
 (2)  for $j \leftarrow 1 \ldots m$ do $B[p_j] \leftarrow B[p_j] \mid (1 << (j-1))$
 (3)  for $i \leftarrow 0 \ldots k$ do $D_i \leftarrow (1 << i) - 1$
 (4)  for $j \leftarrow 1 \ldots n$ do
 (5)     $pD \leftarrow D_0$
 (6)     $nD, D_0 \leftarrow ((D_0 << 1) \mid 1)\ \&\ B[t_i]$
 (7)     for $i \leftarrow 1 \ldots k$ do
 (8)        $nD \leftarrow ((D_i << 1)\ \&\ B[t_i]) \mid pD \mid ((pD \mid nD) << 1) \mid 1$
 (9)        $pD \leftarrow D_i$, $D_i \leftarrow nD$
(10)     if $nD\ \&\ (1 << (m-1)) \neq 0$
(11)     then report an occurrence ending at position $i$
```

Filtration

- Frequently it is easier to tell that a text position cannot match than to ensure that it matches with k errors
- Filtration is based on applying a fast filter over the text, which hopefully discards most of the text positions
- Then we can apply an approximate search algorithm over the areas that could not be discarded
- A simple and fast filter:
  - Split the pattern into $k+1$ pieces of about the same length
  - Then we can run a multi-pattern search algorithm for the pieces
  - If piece $p_j \ldots p_{j'}$ appears in $t_i \ldots t_{i'}$, then we run an approximate string matching algorithm over $t_{i-j+1-k} \ldots t_{i-j+m+k}$

Searching Compressed Text

- An extension of traditional compression mechanisms gives a very powerful way of matching much more complex patterns
- Let us start with phrase queries that can be searched for by
  - compressing each of its words and
  - searching the compressed text for the concatenated string of target symbols
- This is true as long as
  - the phrase is made up of simple words, each of which can be translated into one codeword, and
  - we want the separators to appear exactly as in the query

Searching Compressed Text

- A more robust search mechanism is based in **word patterns**
- For example, we may wish to search for:
  - Any word matching `‘United’` in case-insensitive form and permitting two errors
  - Then a separator
  - And then any word matching `‘States’` in case-insensitive form and permitting two errors
- This search problem can be modeled by means of an **automaton over codewords**

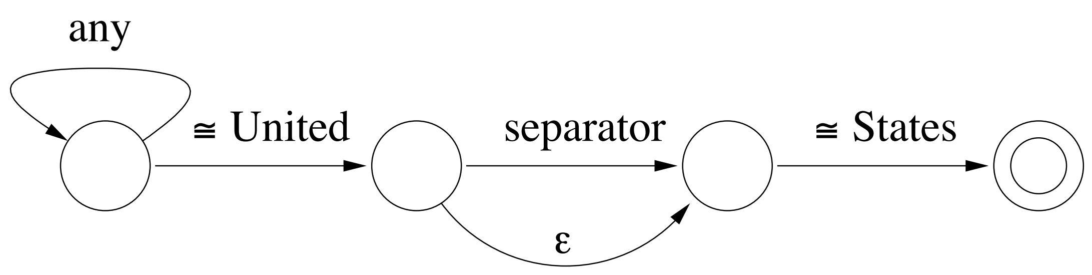
Searching Compressed Text

- Let $\mathcal{C}$ be the set of different codewords created by the compressor
- We can take $\mathcal{C}$ as an alphabet and see the compressed text as a sequence of atomic symbols over $\mathcal{C}$
- Our pattern has three positions, each denoting a class of characters:
  - The first is the set of codewords corresponding to words that match `‘United’` in case-insensitive form and allowing two errors
  - The second is the set of codewords for separators and is an optional class
  - The third is like the first but for the word `‘States’`

Searching Compressed Text

- The Figure below illustrates the previous example

Vocabulary (alphabet)	B[ ] table
, \n	010
\n	010
States	001
UNITED	100
United	100
Unnited	100
state	001
unates	101
unite	100



Searching Compressed Text

- This process can be used to search for much more complex patterns
- Assume that we wish to search for `‘the number of elements successfully classified’`, or something alike
- Many other phrases can actually mean more or less the same, for example:

  the number of elements classified with success

  the elements successfully classified

  the number of elements we successfully classified

  the number of elements that were successfully classified

  the number of elements correctly classified

  the number of elements we could correctly classify

  ...

Searching Compressed Text

- To recover from linguistic variants as shown above we must resort to **word-level** approximate string matching
- In this model, we permit a limited number of missing, extra, or substituted words
  - For example, with 3 word-level errors we can recover from all the variants in the example above

Multi-dimensional Indexing

Multi-dimensional Indexing

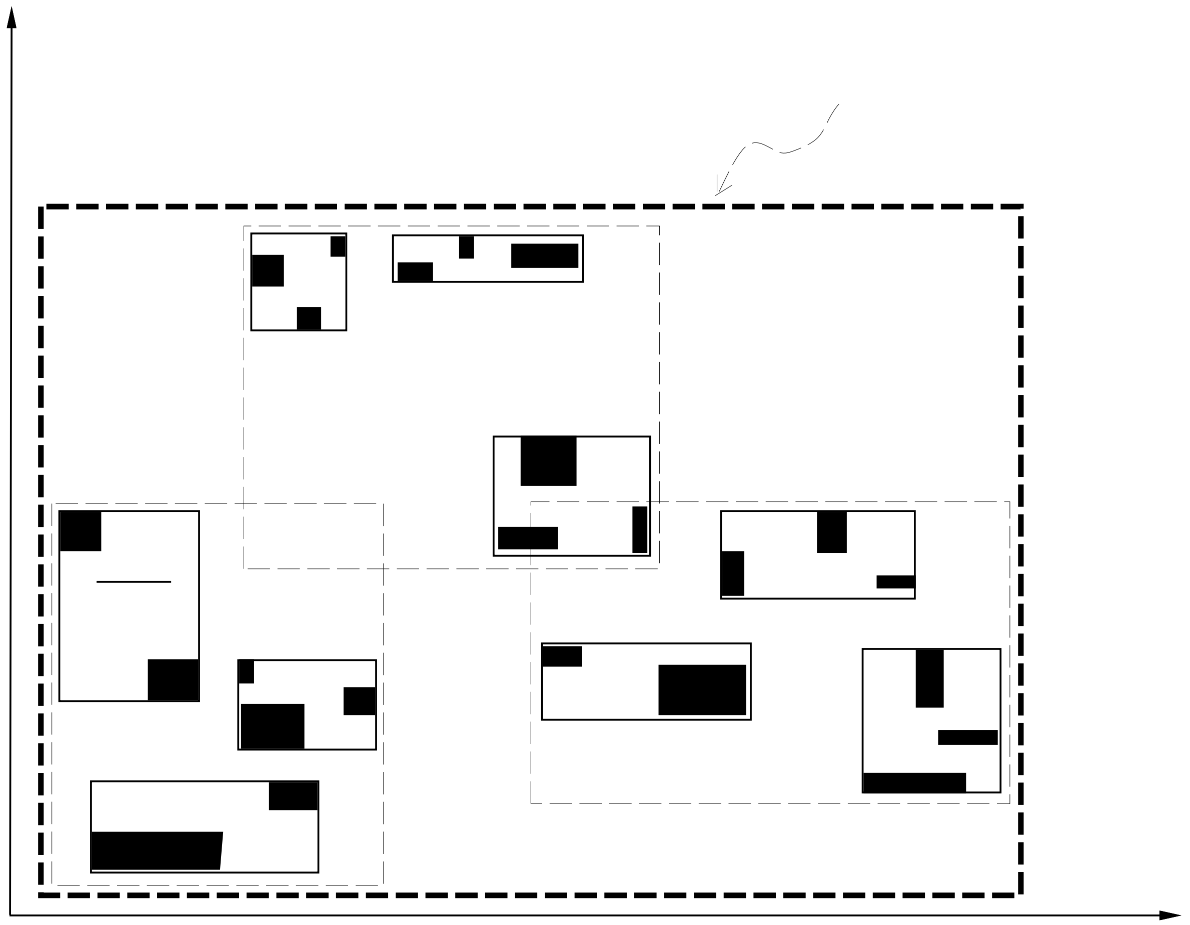
- In multimedia data, we can represent every object by several numerical features
- For example, imagine an image from where we can extract a color histogram, edge positions, etc
- One way to search in this case is to map these object features into points in a multi-dimensional space
- Another approach is to have a distance function for objects and then use a distance based index
- The main mapping methods form three main classes:
    - R^*-trees and the rest of the R-tree family,
    - linear quadtrees,
    - grid-files

Multi-dimensional Indexing

- The R-tree-based methods seem to be most robust for higher dimensions
- The R-tree represents a spatial object by its minimum bounding rectangle (MBR)
- Data rectangles are grouped to form parent nodes, which are recursively grouped, to form grandparent nodes and, eventually, a tree hierarchy
- **Disk pages** are consecutive byte positions on the surface of the disk that are fetched with one disk access
- The goal of the insertion, split, and deletion routines is to give trees that will have good clustering

Multi-dimensional Indexing

- Figure below illustrates data rectangles (in black), organized in an R-tree with fanout 3



Multi-dimensional Search

- A range query specifies a region of interest, requiring all the data regions that intersect it
- To answer this query, we first retrieve a superset of the qualifying data regions:
  - We compute the MBR of the query region, and then we recursively descend the R-tree, excluding the branches whose MBRs do not intersect the query MBR
  - Thus, the R-tree will give us quickly the data regions whose MBR intersects the MBR of the query region
- The retrieved data regions will be further examined for intersection with the query region

Multi-dimensional Search

- The data structure of the R-tree for the previous figure is (fanout = 3)