Universität Potsdam
Institut für Informatik
Lehrstuhl Maschinelles Lernen

Indexing and Search

Uwe Dick

Overview

- Index data structures
  - Inverted index
  - Suffix tries and suffix arrays

- Search algorithms
  - Boolean search
  - Sequential Search without index
  - String matching
  - String matching with index

Inverted Index

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.

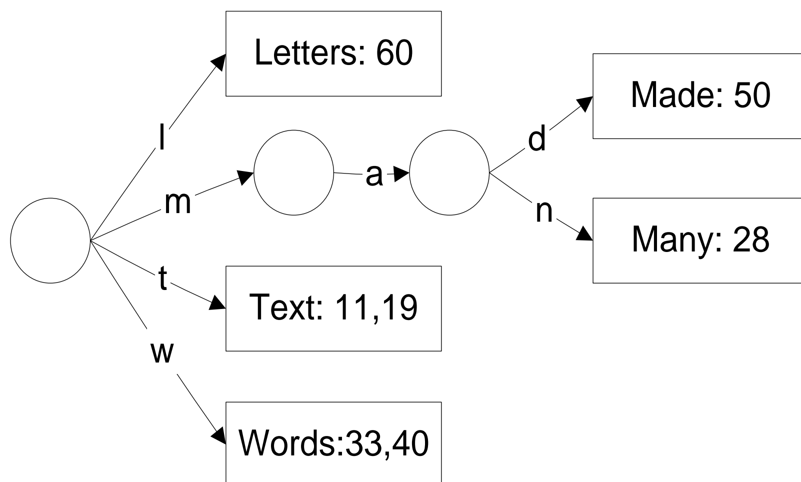
Terme	Vorkommen
Letters	60
Made	50
Many	28
Text	11, 19
words	33, 40

- Mapping from terms to documents / text positions.
- Additional information can also be stored
  - Typeset, HTML tags

Inverted Index
Tries

- Elementary data structure for re*trie*val tasks
- Edges are labeled with letters.
- Nodes are either empty or labeled with words and position

```
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.
```



Inverted Index
Construction of Tries

- Start with empty trie and iteratively insert words into the trie:
- Start at root, iterate over letters
  - If a terminal node is reached that is labeled with the same word, add position; done.
  - If a terminal node is reached that is labeled with a different word, replace node with internal nodes until there is a difference between old and new word. Add two new terminal node. Done.
  - If an edge accepts current letter, follow edge.
  - Otherwise add new edge, label with current letter, and add new terminal node that is labeled with current word and position.

Inverted Index
Construction of an Inverted Index

- Iterate over all texts, iterate over all positions in text where a new word start.
  - Insert (word, position) into a trie (s.t. trie is ordered).
- Traverse trie and write words into inverted index.

Inverted Index

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.

Terme	Vorkommen
Letters	60
Made	50
Many	28
Text	11, 19
words	33, 40

- Mapping from terms to documents / text positions.
- Unfortunately, memory usage is usually a problem for real world index construction.

Inverted Index
Construction of an Inverted Index

- Iterate over all texts, iterate over all positions in text where a new word start.
  - Insert (word, position) into a trie (s.t. trie is ordered).
  - If memory full, store trie, load a subtree in memory, consider only words contained in this subtree. Store and take next subtree, reiterate .
- Traverse trie and write words into inverted index.

Inverted Index

- Memory usage for vocabulary: between $n^{0.4}$ and $n^{0.6}$.
- Memory usage for position lists: between 0.3 x length of text and 0.4 x length of text.
- If only documents are indexed (no word positions), 0.2 to 0.3 x length of text.

- **Problem**: High memory usage.
- **Solution**: Memory reduction by block addressing: e.g. division of texts into 256 blocks, uses only 0.05 x length of text.

Inverted Index
Block Addressing

1	2	3	4
This is a text.	A text has many	words. Words are	made from letters.

Terme	Vorkommen
Letters	4
Made	4
Many	2
Text	1, 2
words	3

- Exact text position can be determined with sequential search.

Inverted Index
Large Data Collections

- Iterate over documents, parse tokens (words).
- Insert tokens and text positions into unordered list.
- Concatenate all unsorted lists.
- Eliminate duplicate terms, merge all positions into a list.
- Sort list.
- **Requirement**: List has to fit into memory.

Inverted Index
Large Data Collections

- Read n documents in k blocks.
- Create term list for each block.
- Sort local index for block, store block.
- Merge local indices in $\log_2$ k layers.
  - c.f. merge sort
  - For layer index $l = 0 \ldots \log_2 k - 1$:
    - For block index $b = 0 \ldots \frac{k}{2(l+1)} - 1$:
      - Merge blocks block_{2b} and block_{2b+1} by sequentially reading both blocks and writing new block with size $2(l+1)\frac{n}{k}$

Scheffer/Dick, Language Technology

Inverted Index
Search

- Let q be a search query.
- Search for individual terms from q in index. Use binary search in term list.
- Access retrieved documents.
- Solve phrase-, neighbor-/context-, and boolean queries by comparing documents.
- (In case of block addressing, search in blocks.)

Inverted Index
Application: Spell Checking

- Occasionally, search queries do not return answers:
  - E.g. if query contains misspellings.
- Alter query by adding terms that have edit distance of at most n.
- May slow down search speed significantly.

Inverted Index
Context Queries

- Find list of occurrences of each query term.
  - Lists are ordered according to position.
- Set pointer on first element.
- Iterate until one of the lists is empty:
  - If positions of all list elements are consecutive (or at least close to each other), add range to return list.
  - Increment list pointer that points to lowest text position. Continue.
- Complexity: $O(n^{0.4 \ldots 0.8})$

Inverted Index
Problems

- Not suited for neighbor queries.
- Needs tokenization (Doesn‘t work for DNA-Sequences).

- Better: Do not consider tokens. Consider all suffixes of a text.

- Idea of Suffix trees: Complete text is a string. Each position is a suffix (From position to end of text).
- Trie over all suffixes.

Inverted Index
Suffix Trie

- Index pointers can be beginning of words or all string positions.
- Text starting at index pointer: Suffix.

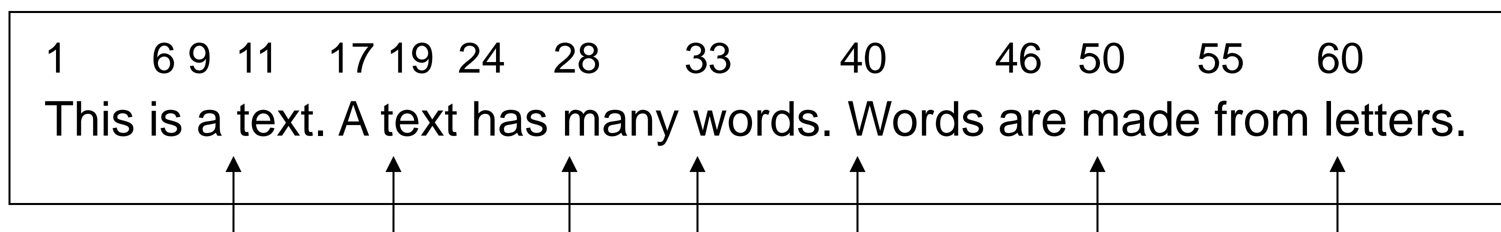
This is a text. A text has many words. Words are made from letters.

Suffixes:

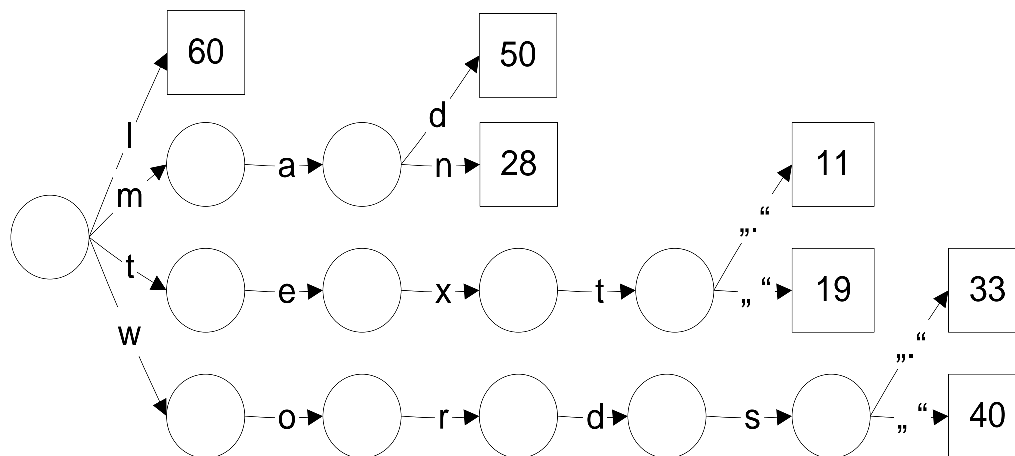
text. A text has many words. Words are made from letters.
text has many words. Words are made from letters.
many words. Words are made from letters.
words. Words are made from letters.
Words are made from letters.
made from letters.
letters.

Inverted Index
Suffix Trie

- Construction of a suffix trie:
- For all index points:
  - Insert suffix starting at index pointer into trie.



Suffix-Trie:

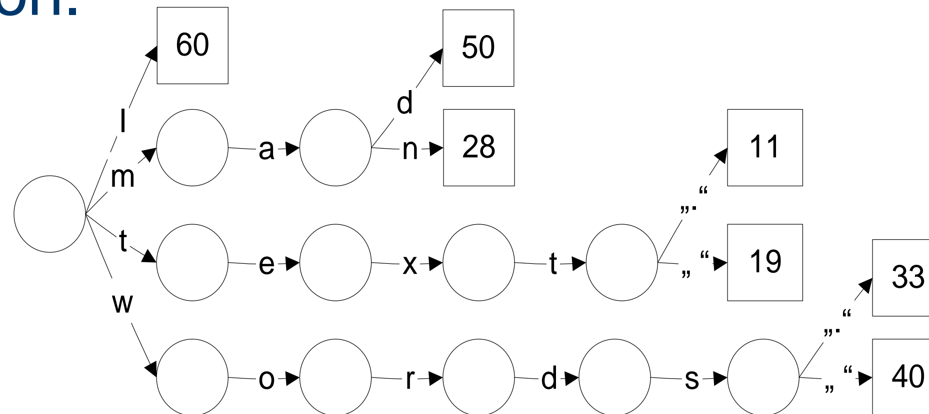


Scheffer/Dick, Language Technology

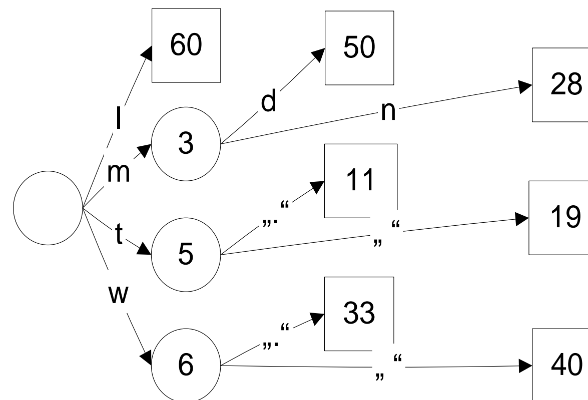
Inverted Index
Patricia Trie / Radix Tree

- Replace sequences of internal nodes which only have a single child with a single node and label it with number of letters to skip until next letter comparison.

Suffix trie:



Patricia trie:



Inverted Index
Search in Suffix Tries

- **Input**: search string s, root node.

1. $i = 0, n = root$
2. Iterate
   1. If n leaf node, check whether s can be found at given text positions and return positions. Stop.
   2. Set $i = i + 1$.
      1. If $i > len(s)$, find all leaf nodes in subtree starting at n and return list of all occurrences. Stop.
      2. Follow edge which accepts i-th letter of s. If there is none, return empty list and stop. Otherwise set n to current node.

Inverted Index
Search in Patricia Tries

- **Input**: search string s, root node.

1. $i = 0, n = root$
2. Iterate
    1. If n leaf node, check whether s can be found at given text position and return position.
    2. n is skip node with label k, set $i = i + k$.
        1. If $i > len(s)$, find all leaf nodes in subtree starting at n and check each text position for s. Return list of all occurrences. Stop.
        2. Follow edge which accepts i-th letter of search string. If there is none, return empty list. Otherwise set n to current node.

Inverted Index
Suffix Trie

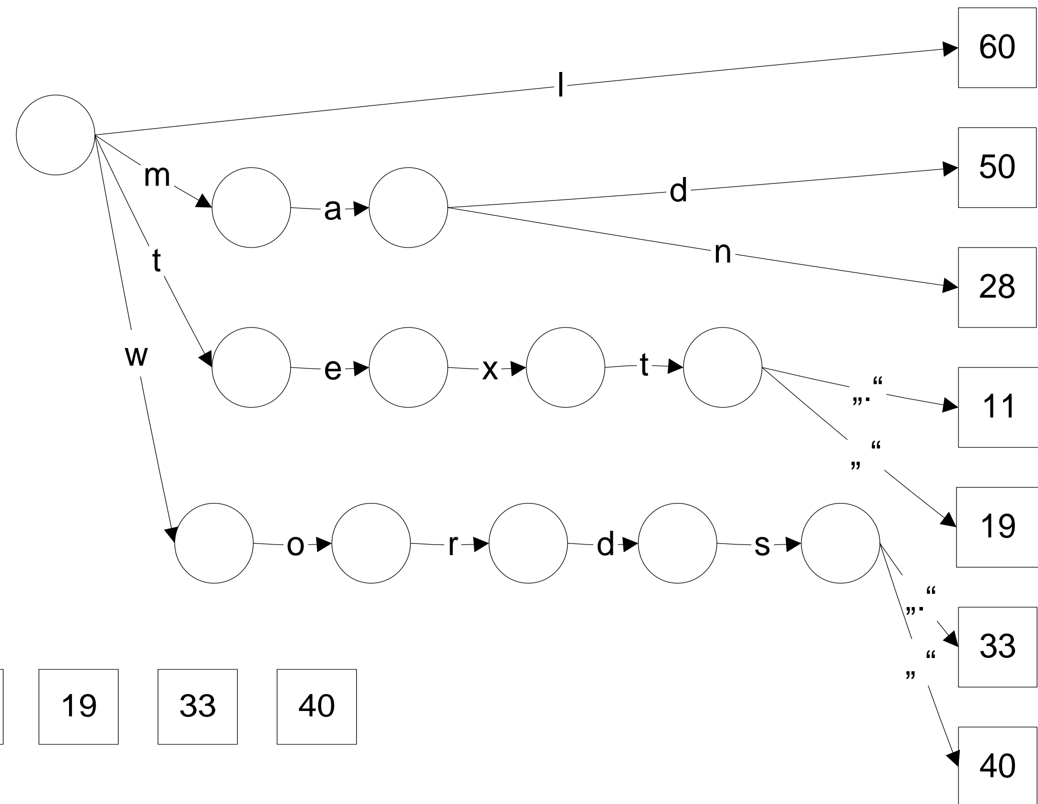
- Construction: O(Length of text).
- Algorithm not optimal if trie doesn‘t fit into memory.
- **Problem**: Memory consumption rather high
  - approx. 120-240% of texts, even if only beginnings of words are indexed.

- **Solution**: Suffix arrays
  - More compact representation.

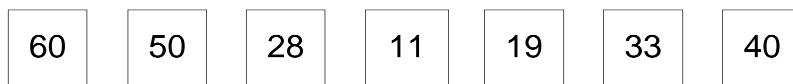
Inverted Index
Suffix Array

- Sort suffix trie lexicographically.
- Suffix array = Sequence of index positions.

Lexicographically sorted suffix trie.



Suffix array:



Inverted Index
Search in Suffix Arrays: Binary Search

- Input: search string s, suffix array S

1. $A = 1$
2. $E = |S|$
3. Repeat until $A = E$
   1. $M = (A + E)/2$.
   2. If $s = Text(S[M])$, return text position. Stop
   3. If $s < Text(S[M])$, $E = M$.
   4. Otherwise $A = M$.

- Suffix tries: $O(|s|)$
- Suffix arrays: $O(\log(len(text)))$

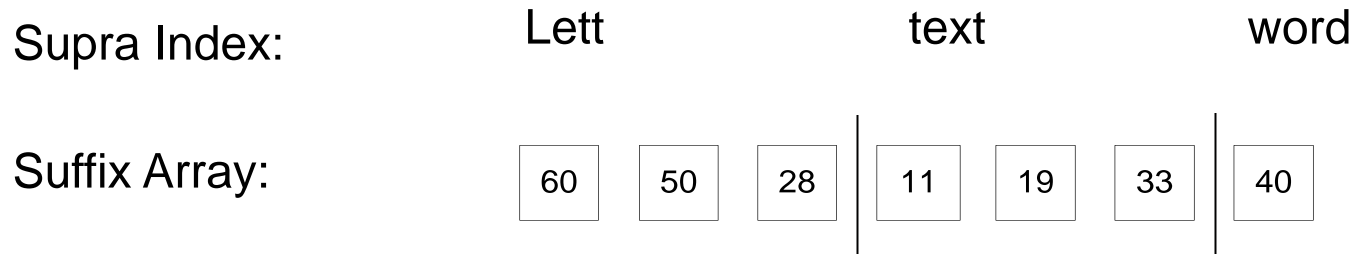
Runtime is worse!!

Inverted Index
Search in Suffix Arrays

- Binary search: $\log(len(text))$
- **Problem**: Many data base accesses
- **Solution**: Accelerate with supra index.

- Store first letters of every k-th entry in array.
- Perform binary search in this supra index, without data base access.
- Then binary search in corresponding part of suffix array with data base access.

Inverted Index
Search in Suffix Arrays with Supra Index



- Binary search in supra index (Without data base access)
- Then binary search in corresponding part of suffix array (With data base access)

- Storage size comparable to inverted index.

Inverted Index
Suffix Arrays for Large Data Collections

- Read text blockwise, create local suffix arrays, store.

- For all local suffix arrays:
  - Read full text incrementally.
  - Search all suffixes in local array and count how many suffixes are ‚missing' between entries. This yields positions in global suffix array for each entry in local array. Store position in local array.

- Merge arrays incrementally without access to text. Use stored positions only.

Sequential Search

- Search string of length m in text of length n without index.
- Brute Force:
  - For all positions of text, iterate over search string and compare.
    - Full Match: return position.
    - Otherwise: next text position.

- $O(nm)$.
- $O(n)$ possible?

Knuth-Morris-Pratt

- Algorithm for finding string patterns
  - Uses structure of search pattern by skipping characters / symbols in case of a mismatch.
- Consists of two parts:
  - Preprocessing / table building algorithm:
    - Generates prefix table, which defines by how many positions the pattern can be shifted in case of a mismatch.
  - Search algorithm:
    - Searches text for pattern using the prefix table.

Knuth-Morris-Pratt
Table Building Algorithm

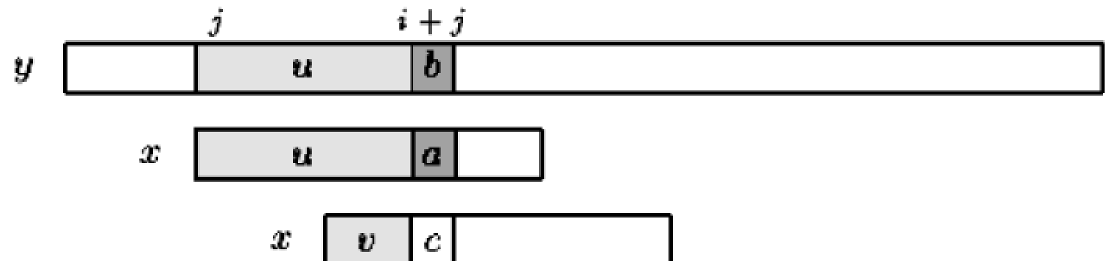
```
kmpPreprocess(p)
i = 1; j = 0;
b[0] = 0;
while (i < m) do
  if (p[j] == p[i]) // Match
    b[i] = j + 1;
    i = i + 1; j = j + 1;
  else        // Missmatch
    if (j > 0)
      j = b[j-1];
    else
      b[i] = 0;
      i = i + 1;
```

Example:

j	0	1	2	3	4
p	d	a	d	a	b
b[j]	0	0	1	2	0

Pattern

Prefix array



- Runtime O(m)
- b[j] is longest suffix of p[0…j] which is prefix of p[0... m].

Knuth-Morris-Pratt
Search Algorithm

```
kmpSearch(t,p)
b = kmpPreprocess(p);
i = 0; j = 0;
while (i < n) do
  if (p[j] == t[i]) // Match
    if (j == m-1)
      return i-m+1;
    i = i+1; j = j+1;
  else              // Mismatch
    if (j > 0)
      j = b[j-1];
    else
      i = i + 1;
```

precomputed

j	0	1	2	3	4
p	d	a	d	a	b
b[j]	0	0	1	2	0

Example:

i	0	1	2	3	4	5	6	7	8	9
t	d	a	d	a	c	d	a	d	a	b
p	d	a	d	a	b					

- Runtime O(n)

Knuth-Morris-Pratt
Search Algorithm

```
kmpSearch(t,p)
b = kmpPreprocess(p);
i = 0; j = 0;
while (i < n) do
  if (p[j] == t[i]) // Match
    if (j == m-1)
      return i-m+1;
    i = i+1; j = j+1;
  else              // Mismatch
    if (j > 0)
      j = b[j-1];
    else
      i = i + 1;
```

j	0	1	2	3	4
p	d	a	d	a	b
b[j]	0	0	1	2	0

Example:

i	0	1	2	3	4					
t	d	a	d	a	c	d	a		a	b
p	d	a	d	a	b					

Mismatch at i = 4 and j = 4:
j = b[4-1] = 2

- Runtime O(n)

Knuth-Morris-Pratt
Search Algorithm

```
kmpSearch(t,p)
b = kmpPreprocess(p);
i = 0; j = 0;
while (i < n) do
  if (p[j] == t[i]) // Match
    if (j == m-1)
      return i-m+1;
    i = i+1; j = j+1;
  else             // Mismatch
    if (j > 0)
      j = b[j-1];
    else
      i = i + 1;
```

j	0	1	2	3	4
p	d	a	d	a	b
b[j]	0	0	1	2	0

Example:

i	0	1	2	3	4	5	6	7	8	9
t	d	a	d	a	c	d	a	d	a	b
p	d	a	d	a	b					
			d	a	d	a	b			

- Runtime O(n)

Scheffer/Dick, Language Technology

Knuth-Morris-Pratt
Search Algorithm

```
kmpSearch(t,p)
b = kmpPreprocess(p);
i = 0; j = 0;
while (i < n) do
  if (p[j] == t[i]) // Match
    if (j == m-1)
      return i-m+1;
    i = i+1; j = j+1;
  else              // Mismatch
    if (j > 0)
      j = b[j-1];
    else
      i = i + 1;
```

j	0	1	2	3	4
p	d	a	d	a	b
b[j]	0	0	1	2	0

Example:

i	0	1	2	3	4		
t	d	a	d	a	c		
p	d	a	d	a	b		
			d	a	d	a	b

Mismatch at i = 4 and j = 2:
j = b[2-1] = 0

- Runtime O(n)

Knuth-Morris-Pratt
Search Algorithm

```
kmpSearch(t,p)
b = kmpPreprocess(p);
i = 0; j = 0;
while (i < n) do
   if (p[j] == t[i]) // Match
      if (j == m-1)
         return i-m+1;
      i = i+1; j = j+1;
   else              // Mismatch
      if (j > 0)
         j = b[j-1];
      else
         i = i + 1;
```

j	0	1	2	3	4
p	d	a	d	a	b
b[j]	0	0	1	2	0

Example:

i	0	1	2	3	4	5	6	7	8	9
t	d	a	d	a	c	d	a	d	a	b
p	d	a	d	a	b					
			d	a	d	a	b			
					d	a	d	a	b	

- Runtime O(n)

Knuth-Morris-Pratt
Search Algorithm

```
kmpSearch(t,p)
b = kmpPreprocess(p);
i = 0; j = 0;
while (i < n) do
  if (p[j] == t[i]) // Match
    if (j == m-1)
      return i-m+1;
    i = i+1; j = j+1;
  else              // Mismatch
    if (j > 0)
      j = b[j-1];
    else
      i = i + 1;
```

j	0	1	2	3	4
p	d	a	d	a	b
b[j]	0	0	1	2	0

Example:

```
i    0 1 2 3 4
t    d a d a d
p    d a d a b
         d a d a b
             d a d a b
```

Mismatch at i = 4 and j = 0: i = 5

- Runtime O(n)

Knuth-Morris-Pratt
Search Algorithm

```
kmpSearch(t,p)
b = kmpPreprocess(p);
i = 0; j = 0;
while (i < n) do
  if (p[j] == t[i]) // Match
    if (j == m-1)
      return i-m+1;
    i = i+1; j = j+1;
  else              // Mismatch
    if (j > 0)
      j = b[j-1];
    else
      i = i + 1;
```

- Runtime O(n)

j	0	1	2	3	4
p	d	a	d	a	b
b[j]	0	0	1	2	0

Example:

i	0	1	2	3	4	5	6	7	8	9
t	d	a	d	a	c	d	a	d	a	b
p	d	a	d	a	b					
			d	a	d	a	b			
					d	a	d	a	b	
						d	a	d	a	b

Knuth-Morris-Pratt
Runtime Analysis

- Preprocessing algorithm has runtime of $O(m)$.
  - At most $2m - 1$ comparisons.
- Search algorithm has runtime of $O(n)$.
  - At most $2n - m + 1$ comparisons.

- In general, m is much smaller than n. This yields an overall runtime of $O(n)$.

- Overall, there will be at most $2n + m$ comparisons.

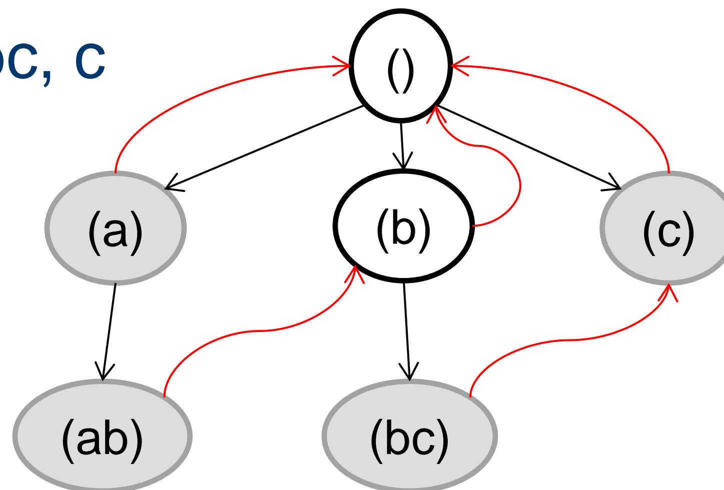
Aho-Corasick-Trie

- Aho-Corasick-Trie
- **Idea**: We want to search for several strings simultaneously.
  - Search strings are stored in automaton.
  - Edges accept characters.
  - If there is no edge that accepts the next term, jump to state that has largest agreement between text and a prefix of the search string.

Aho-Corasick-Trie
Construction

- 1. Construction of a prefix tree which contains all search terms.
  - Mark states that contain search terms.
- 2. Generate failure transitions:
  - For each node (except root), add a failure transition to the node that is longest suffix of the string of that node.

Example: a, ab, bc, c



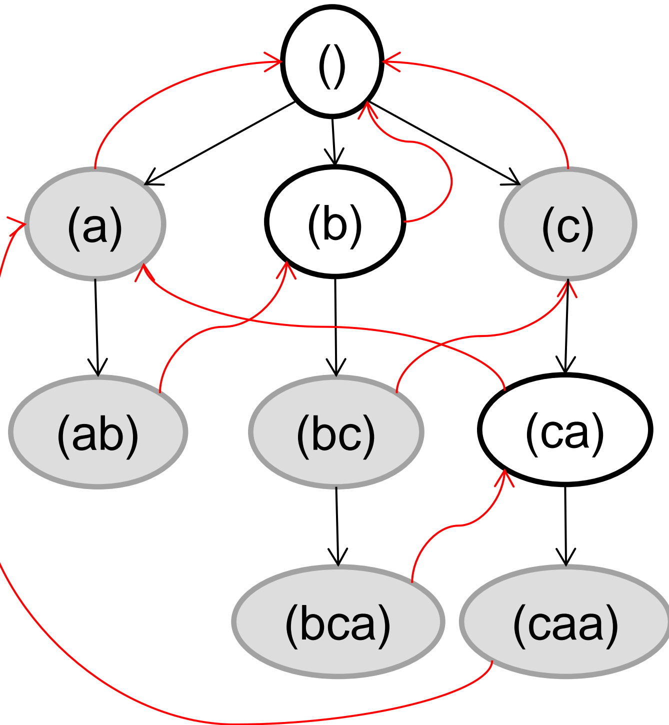
Aho-Corasick-Trie
Search

- State $S = root$
- Iterate over characters c of text.
  - If there exists edge from S labeled with c, follow edge to new state S.
  - Otherwise, follow failure transitions until a state is reached that has an outgoing edge labeled with c. Follow that edge to new state S.
  - Otherwise, Set $S = root$.

  - Output: Follow failure transitions from S to $root$. Return all strings of accepted states on path.

Aho-Corasick-Trie
Search Example

Automaton accepts strings a,ab,bc,bca,c,caa



String	State	Next State	Output
abccab	()	() -> (a)	a:1
bccab	(a)	(a) -> (ab)	ab:2
ccab	(ab)	(ab) -> (b) -> (bc)	c:3, bc:3
cab	(bc)	(bc) -> (c) -> () -> (c)	c:4
ab	(c)	(c) -> (ca)	a:5
b	(ca)	(ca) -> (a) -> (ab)	ab:6

Other String Matching Algorithms

- Wide Topic. $\rightarrow$ Bio informatics.
- Boyer-Moore. Idea: Compare strings from back to front. On average, more characters can be skipped.
- Suffix automata.

Approximate String Matching

- Search substrings in y which are *similar* to x. Their *distance* to x should be at most k.

- What is a good distance measure?
  - Edit distance.
  - Model for typing errors.
  - Language Model.
  - Semantic Model.

Approximate String Matching

- Search substrings in y which are *similar* to x. Their *distance* to x should be at most k.
- Dynamic programming (similar to Forward-Backward or Viterbi).
- Computation of C[0…m, 0…n]; C[i,j] = minimal # errors for matching x[1…i] with y[1…j].
- C[0, j] = j
- C[i, 0] = i
- C[i, j] = { C[i-1, j-1] , if x[i] = y[j]; 1+min {C[i-1, j], C[i, j-1], C[i-1, j-1]}, otherwise

- Complexity: $O(nm)$

Scheffer/Dick, Language Technology

Approximate String Matching
Example

- $C[0, j] = j$
- $C[i, 0] = i$
- $C[i, j] = \begin{cases} C[i-1, j-1], & \text{if } x[i] = y[j] \\ 1+\min\{C[i-1, j], C[i, j-1], C[i-1, j-1]\}, & \text{otherwise} \end{cases}$

		a	b	c	a	b	b	b	a	a
	0	1	2	3	4	5	6	7	8	9
c	1	1								
b	2									
a	3									
b	4									
a	5									
c	6									

Approximate String Matching
Example

- $C[0, j] = j$
- $C[i, 0] = i$
- $C[i, j] = \begin{cases} C[i-1, j-1], & \text{if } x[i] = y[j] \\ 1+\min\{C[i-1, j], C[i, j-1], C[i-1, j-1]\}, & \text{otherwise} \end{cases}$

		a	**b**	**c**	**a**	**b**	**b**	**b**	**a**	**a**
	0	1	2	3	4	5	6	7	8	9
c	1	1	2							
b	2									
a	3									
b	4									
a	5									
c	6									

Approximate String Matching
Example

- $C[0, j] = j$
- $C[i, 0] = i$
- $C[i, j] = \begin{cases} C[i-1, j-1], & \text{if } x[i] = y[j] \\ 1+\min\{C[i-1, j], C[i, j-1], C[i-1, j-1]\}, & \text{otherwise} \end{cases}$

		a	**b**	**c**	**a**	**b**	**b**	**b**	**a**	**a**
	0	1	2	3	4	5	6	7	8	9
c	1	1	2	2						
b	2									
a	3									
b	4									
a	5									
c	6									

Approximate String Matching
Example

- C[0, j] = j
- C[i, 0] = i
- C[i, j] = { C[i-1, j-1] , if x[i] = y[j]; 1+min {C[i-1, j], C[i, j-1], C[i-1, j-1]}, otherwise

		a	b	c	a	b	b	b	a	a
	0	1	2	3	4	5	6	7	8	9
c	1	1	2	2	3	4	5	6	7	8
b	2	2	1	2	3	3	4	5	6	7
a	3	2	2	2	2	3	4	5	5	6
b	4	3	2	3	3	2	3	4	5	6
a	5	4	3	3	3	3	3	4	4	5
c	6	5	4	3	4	4	4	4	5	5

Approximate String Matching
Example

		a	b	c	a	b	b	b	a	a
	0	1	2	3	4	5	6	7	8	9
c	1	1	2	2	3	4	5	6	7	8
b	2	2	1 =	2	3	3	4	5	6	7
a	3	2	2	2	2 =	3	4	5	5	6
b	4	3	2	3	3	2	3 =	4	5	6
a	5	4	3	3	3	3	3	4	4 =	5
c	6	5	4	3	4	4	4	4	5	5

Deletion

Insertion

Substitution

= No Change

cbabac -> ababac -> abcabac -> abcabbac -> abcabbbac -> abcabbbaa

Questions?