Informationsintegration

Similarity Functions and Similarity Search

Ulf Leser

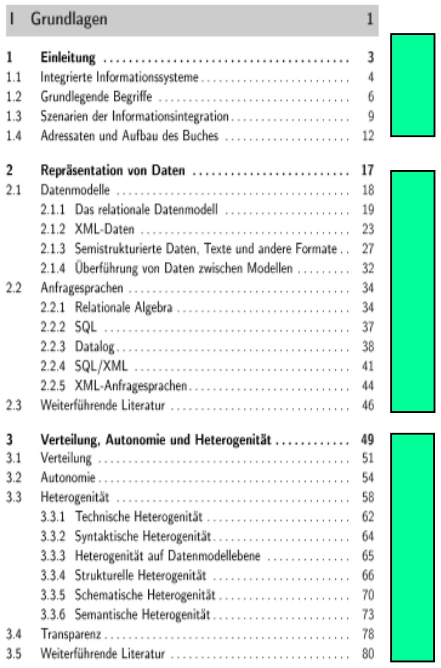
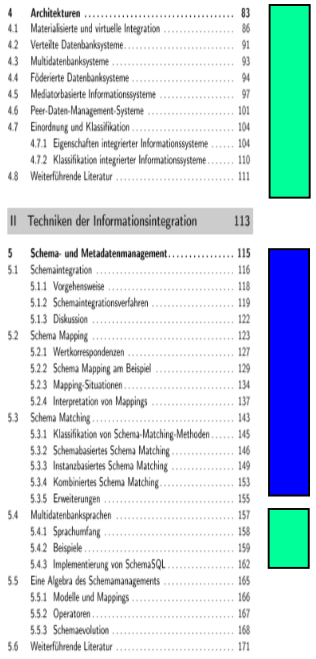
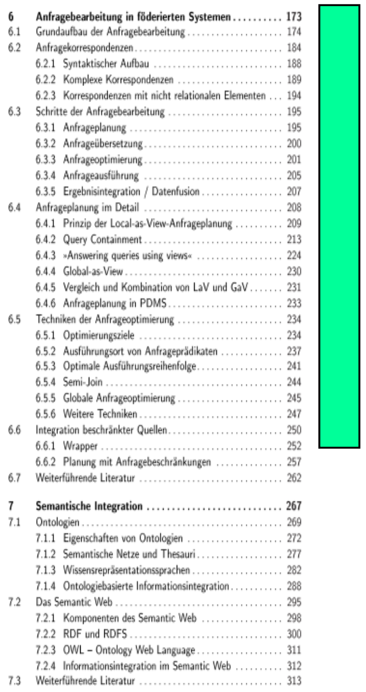
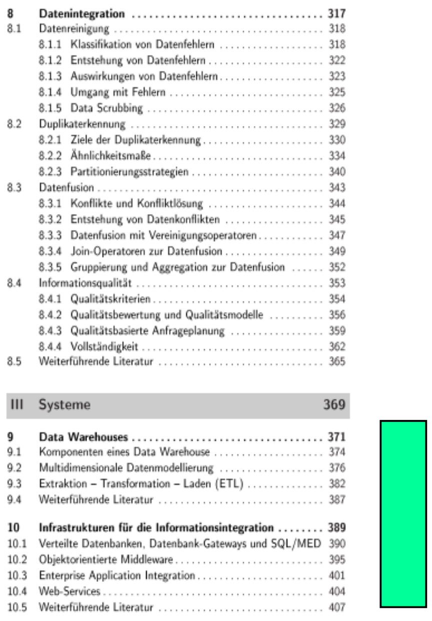
Wo sind wir – Big Picture

- Architekturen und Kriterien
  - Szenarien, Abgrenzung und Einsatzgebiete
  - Verteilung, Autonomie, Heterogenität
- Anfrageplanung
  - Multidatenbanksprachen
  - Anfrageplanung mit LaV und GaV
  - Logische Anfrageoptimierung
- Verteilte Anfrageoptimierung
  - Semi-Joins
  - Umgang mit beschränkten Quellen

Wo sind wir – Big Picture

- Similarity Functions
- Schemamanagement
  - Schema Matching
  - Schemaintegration
- Datenintegration
  - Duplikaterkennung
  - Datenfusion
- Semantische Integration
  - Ontologien und Beschreibungslogiken
  - Semantic Web

Buch

Topics Today

- Information Integration and Similarity
- Similarity Functions
- Similarity Search
- Appendix: Computing Edit Distance

Similarity versus Identity

- In several future topics, we will compare pairs of values to find "identical" objects
  - We compare their representation (strings, vectors, tuples, …)
  - We find to infer real-world identity
  - Examples: Duplicate detection, "same" schema elements
- In real-life, requiring identical representations is too strong
  - Because of errors in communication, differently curated data sources, different representations for the same object, …
  - Is "Peter Müller" = "Peter Mueller"?
  - Is "Stargarderstr. 67" = "Stagarderstrasse 67"
  - Is "Peter Müller, Badstr. 67" = "Peter Mueller, Badstr. 76"?
- Typical remedy: Similarity functions

Similarity for Information Integration

- Assume a set O of objects $O=\{o_1, \ldots, o_n\}$
- Similarity functions $sim: O \times O \rightarrow [0,1]$
  - Function that computes a similarity between pairs of objects
  - 1: Identical; high values: very similar; 0: maximally dissimilar
- Idea: High similarity –> high probability of being identical
  - That's something one has to show empirically
- Finding (presumably) identical pairs: Use a threshold t

$$id_{sim}(o,o')=\text{true iff } sim(o,o')>t$$

- But probability is not certainty – need to consider the quality of decisions given t

Quality of Decisions

- If high values of sim(o, o‵) imply high probability of o and o‵ being the same object
- … then the accuracy of id_{sim} depends on threshold t
  - Using a high value for t
    - All pairs tagged as identical will be identical
    - But identical pairs with similarity just below t will not be found
    - False negatives
  - Using a low value for t
    - Most identical pairs will be found, even if their similarity is not too high
    - But many pairs that actually are not identical will be tagged as such
    - False positives

Precision and Recall

Reality

id_{sim}

	Identical	Different
Identical	true-positive	false-positive
Different	false-negative	true-negative

- Precision = TP/(TP+FP)
  - What fraction of the set of tagged pairs are truly identical?
- Recall = TP/(TP+FN)
  - What fraction of the truly identical pairs have been identified as such?

Example

- Database with 10.000 customers
- A given combination of sim / t identifies 50 duplicates
- Truth
  - There are 55 real duplicates in the database
  - Of these, 42 were identified

	Identical	Different
Identical	TP = 42	FP = 8
Different	FN = 13	

- Precision = TP/(TP+FP) = 42/50 ~ 84%
- Recall = TP/(TP+FN) = 42/55 ~ 76%

Example – Extreme cases

- Let's set t=0

	Identical	Different
Identical	TP = 55	FP = 9945
Different	FN = 0	

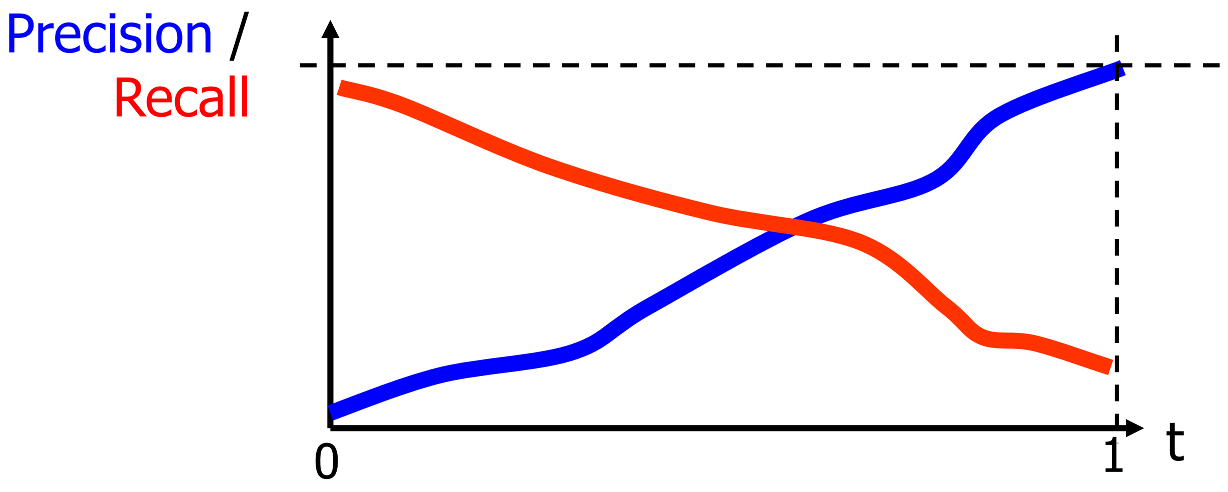
P~0, R=1

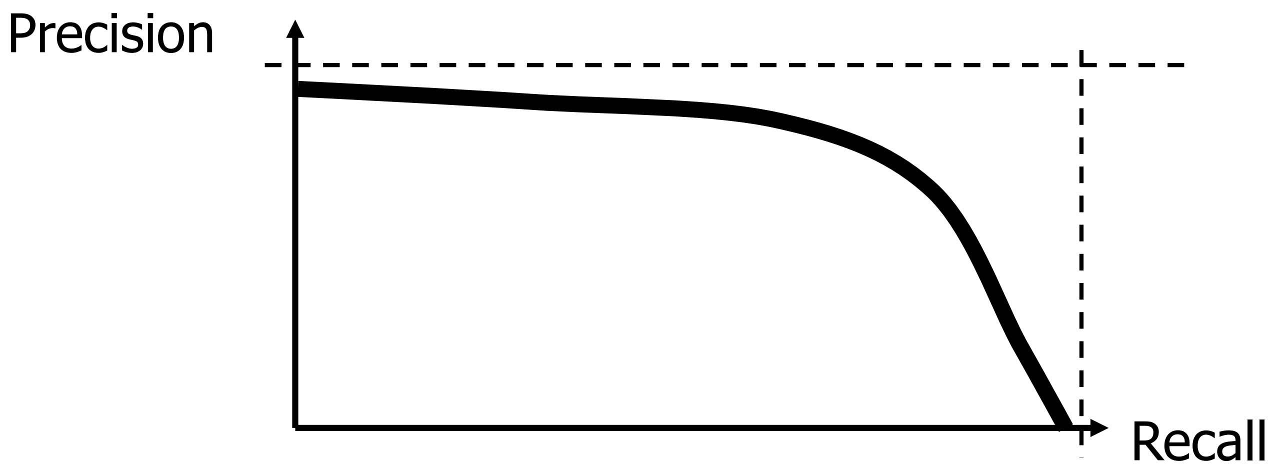
- Let's set t=1
  - Assume we find just 1 identical pair – a true one

	Identical	Different
Identical	TP = 1	FP = 0
Different	FN = 54	

P=1, R~0

Precision / Recall for different Thresholds





More Formally

- Definition

  A similarity function sim for a set O of object is a function

  $$sim: O \times O \rightarrow [0,1]$$

  with the following properties

  - *sim(o,o') = 1 if o=o'*
  - *sim(o,o') = sim(o',o)*

- Remarks
  - Sometimes, one also uses distance functions
  - Same purpose, invers semantics: low values = low distance = likely identical
  - We will later turn similarity in distance and vice versa

Remarks

- Similarity function: Decide upon identity of one pair
- Duplicate detection: Efficiently finding all identical pairs
- Be careful with transitivity of id_{sim}
  - One would naturally assume that
    $$id_{sim}(o_1,o_2)=true \wedge id_{sim}(o_2,o_3)=true \rightarrow id_{sim}(o_1,o_3)=true$$
  - This is not the case for a combination of sim / t!
    - Meier, Meyer, Mayer, Bayer, Bayes, Bades, …
    - Meier, Meir, Mer, Er, R, …
  - See lecture on duplicate detection

Topics Today

- Information Integration and Similarity
- Similarity Functions
- Similarity Search
- Appendix: Computing Edit Distance

Overview

- There exist ~1 zillion similarity functions
- We will discuss a few of them
  - Sets: Jaccard, tfidf
  - Strings: Hamming, edit, soundex, jaro, jaccard
  - Tuples: Weighted sums
  - Vectors, trees, graphs, images, songs, texts, ….: None (sadly ☹)

Similarity Functions for Sets

- Assume an object o to be a set of elements o={e_1, e_2,…e_n}
  - Sets: Order of elements is irrelevant
  - Different sets need not have the same number of elements
- Intuition: Sets are more similar, …
  - the more elements they share
  - the less elements they don't share
- This leads to Jaccard measure [Jaccard, 1902]

$$sim_{jaccard}(o,o') = \frac{|o \cap o'|}{|o \cup o'|}$$

- Example: o={1,2,3,4}, o'={2,4,5} -> sim(o,o')=2/5

Computing Jaccard

- Assume m to be the maximal size of an object, m=max(|o|)
- If we assume a fixed yet arbitrary order of elements
  - Keep elements in each set sorted
  - Then, Jaccard is computed over two ordered lists
  - Thus, $sim(o,o') \in O(|o|+|o'|) \in O(m)$
- If elements are not sorted
  - Union / intersection need to consider all pairs of elements
  - Thus, $sim(o,o') \in O(|o|*|o'|) \in O(m^2)$
  - Or sort first, or use hashing, or …

- Usually, Jaccard is linear in the sizes of the objects

Extension to Weighted Bags: tfidf

- Jaccard assumes all elements to be of equal importance
- Jaccard assumes sets, not bags (multi-sets)
- Sometimes, neither of this is the case
  - E.g. words in a document, products in a shopping basket
- Similarity of weighted bags: tfidf measure
  - Define tf(o,e) to be the relative frequency of e in o
  - Define idf(e) to be the inverse frequency of e in O
  - Define tfidf(o,e) = tf(o,e)*idf(e) =w(o,e)
  - Then

$$sim_{tfidf}(o,o') = \frac{\sum_{e \in o \cap o'} w(o,e) * w(o',e)}{\sqrt{\sum w(o,e)^2} * \sqrt{\sum w(o',e)^2}}$$

  - Same complexity as Jaccard for precomputed w-values

Example

- Assume o_1=ABC, o_2=AB, o_3=AC, o_4=B
- Idf / tf

A	B	C
1/3	1/3	½

	A	B	C
o_1	1/3	1/3	1/3
o_2	½	½	0
o_3	½	0	½

- Example: sim(o_1,o_2)=

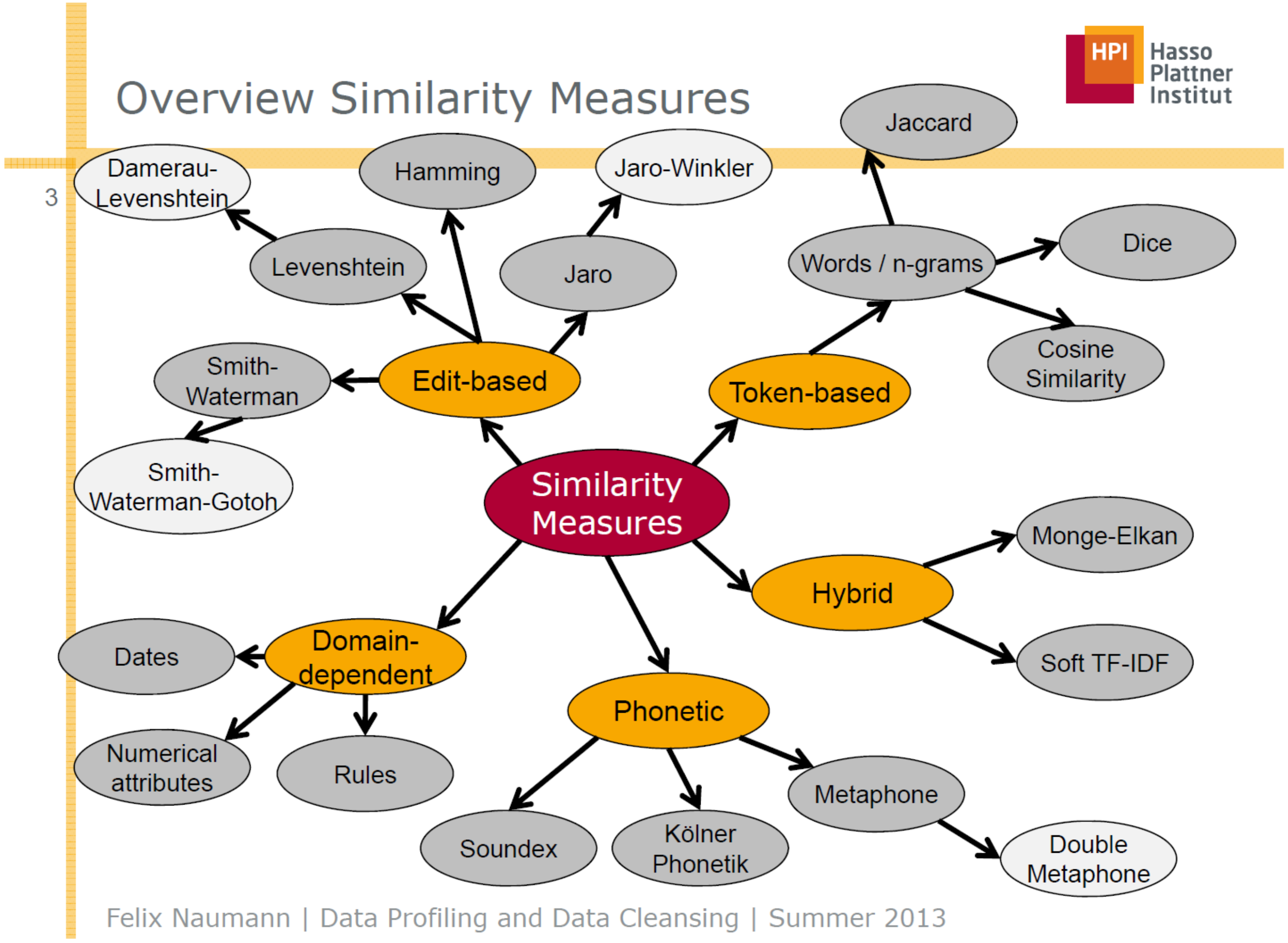
$$\frac{w(o_1,B)*w(o_2,B)}{\sqrt{w(o_1,A)^2+w(o_1,B)^2+w(o_1,C)^2}*\sqrt{w(o_2,A)^2+w(o_2,B)^2}}$$

- We will find $sim_{tfidf}(o_1,o_2)<sim_{tfidf}(o_1,o_3)$
  - Because C is less frequent in O than B, the weight of B drops
  - But: $sim_{jacc}(o_1,o_2)=sim_{jacc}(o_1,o_3)$

Similarity Functions for Strings

- An object o is a sequence of characters $o=<c_1, c_2,\ldots,c_n>$
  - Again, not all objects must have the same length
  - The order of characters is important
  - Examples: Strings, time series, log entries, …
- We discuss five classical string similarity functions
  - Hamming: Number of mismatching char for equal-length strings
  - Edit distance: Minimal number of edit operations
  - Soundex: Heuristic to capture acoustically similar words
  - Jaro: Heuristic especially for short words/names
  - Jaccard: Treating strings as bags of (positional) q-qrams
- There are many more …

Overview [Naumann, 2003]



Hamming distance

- Assume strings o=<c_1, c_2,…,c_n> and o'=<c'_1, c'_2,…,c'_n>
- Natural distance function: Hamming distance

$$dist_{ham}(o,o') = \sum_{i=1..n} 1(c_i \neq c_i')$$

- Remarks
  - Turn into similarity function with $sim_{ham}(o,o')=1-dist_{ham}(o,o')/n$
  - Strings must be of equal length
  - Intuition: Number of characters mismatching at equal positions
  - Roots in communication systems: Measure bit flips on the wire
    - But not loss/spurious insertions of bits or characters
  - Complexity of computation: O(|o|)

Edit Distance

- Intuition: The edit distance of two strings o, o’ is the minimal number of operations necessary to turn o in o’
- Operations: Insertion/deletion/replacement of a single char
  - Also called Levenshtein distance [Lev66]
  - Many variations: transpositions; affine gap costs; weighted character replacement costs; special treatment of pre/suffixes; …
  - See Bioinformatics lecture
- Examples
  -

```
 R I    R       R     D      R DR   R
MEYER  HASEN  LEVENST_EIN  RA_PUNZEL
MAY_R  RASEN  LIVENSTHEIN  GARFUNKEL
```

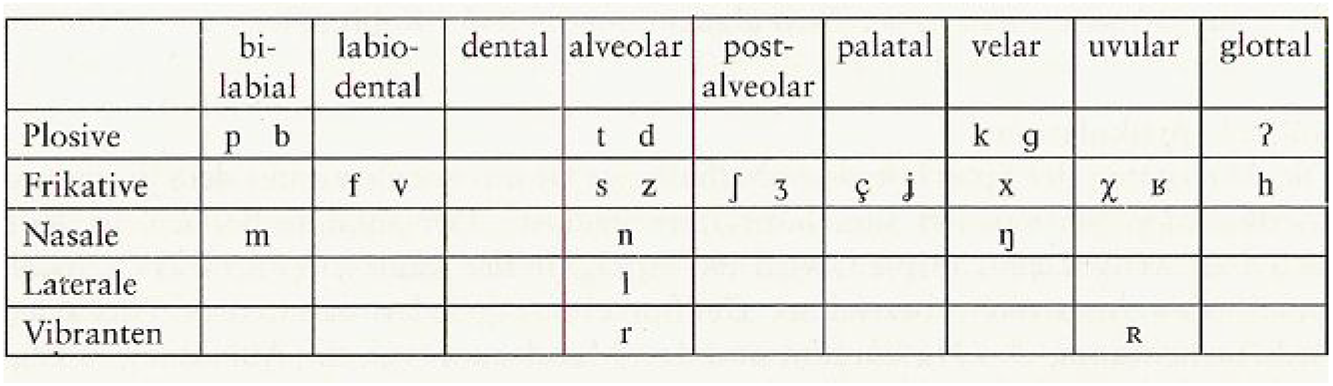
Normalized Edit Similarity

- Edit distance is bounded by |o| (not 1), and a distance, not a similarity
- Turn into similarity: Normalized edit similarity

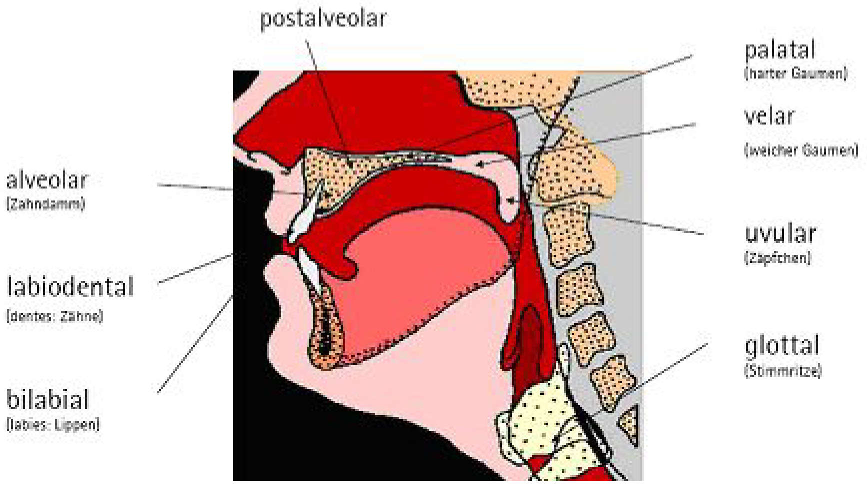
$$sim_{edit}(o,o') = 1 - \frac{dist_{edit}(o,o')}{\max(|o|,|o'|)}$$

- Remarks
  - Very popular measure
  - Often assumed as a kind-of gold standard: How well do other (faster) measures approximate edit distance?
  - In real applications, always use at least weighted replacements
    - Acoustic problems: r(y,i)~0, r(b,p)~0.2, r(i,t)~1, r(e,i)~0.3, …
    - Typos: Define r(X,Y) relative to distance of keys on keyboard

Similarity of Sounds



	bi-labial	labio-dental	dental	alveolar	post-alveolar	palatal	velar	uvular	glottal
Plosive	p b			t d			k g		ʔ
Frikative		f v		s z	ʃ ʒ	ç j	x	χ ʁ	h
Nasale	m			n			ŋ		
Laterale				l					
Vibranten				r				ʀ	



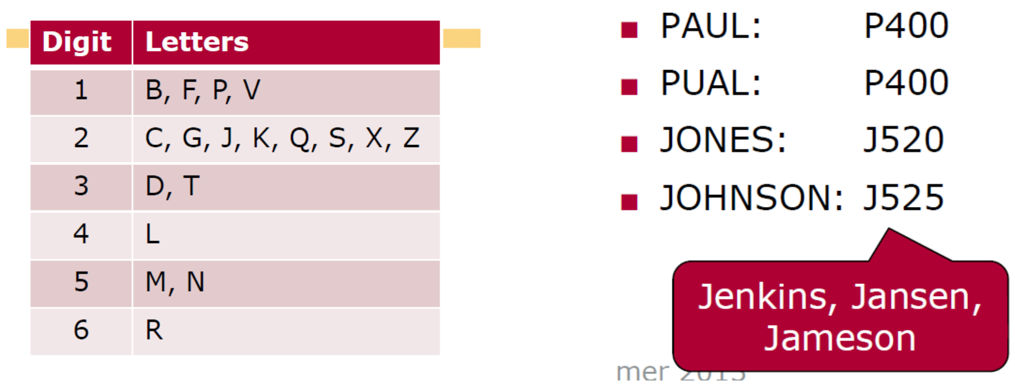
Source: https://www.mediensprache.net/

Algorithm

- Computing edit distance is a bit more involved
- In case you don’t know the algorithm – see Appendix
- Result: Complexity is $O(|o|*|o'|)$ using an edit matrix

Soundex [RO, 1918]

- Idea: Map strings into same codes that sound similar
  - Developed 1918 for census in US: Many acoustic transmissions
  - Soundex-Code: 1st char followed by char codes for next 3 consonants
    - Pad with 0 if less then three consonants exist
  - Similar consonants get same code (code(b)=code(p), d=t, …)
    - Vowels are ignored
  - Original algorithm creates only 1/0 similarities



Digit	Letters
1	B, F, P, V
2	C, G, J, K, Q, S, X, Z
3	D, T
4	L
5	M, N
6	R

PAUL:	P400
PUAL:	P400
JONES:	J520
JOHNSON:	J525

Jenkins, Jansen, Jameson

[Naumann, 2013]

Soundex - Remarks

- Needs language-specific codes (German, French, …)
- Highly application-dependent quality, many variations
- Complexity: O(|o|+|o'|)

Jaro Similarity [Jar89]

- Heuristic specifically for short words, especially names
  - Let h = ⌊min(|o|, |o'|)/2⌋ ; t=0;
  - Let c be the number of (overlapping) pairs of identical characters in both strings that are "close", i.e., less than h apart
  - Let i=1…c; if the i'th close char in o is different from the i'th close char in o': t=+1

$$sim_{jaro}(o,o') = \frac{\frac{c}{|o|} + \frac{c}{|o'|} + (c - \frac{t}{2})/c}{3}$$

- Examples
  - o="jon", o'="john" -> h=1, c=3, t=0, sim=0,97
  - o="melanie", o'="malenia" -> h=3, c=6, t=2, sim=0,85
- Complexity: O(|o|*|o'|)

Q-gram based String Measure: Jaccard for Strings

- Idea: Break strings into sets of q-grams and compute their Jaccard similarity
  - Could also be token of a document -> similarity of documents
  - q is an important parameter (usually q<<|o|)
- Complexity
  - Let m=max(|o|, |o'|)
  - Complexity: O(m*log(m))
    - String o has O(m-q+2)~O(m) q-grams
    - Sorting them requires O(m*log(m)) operations
    - Jaccard is linear in m
    - Together: O(2*m*log(m)+2*m)= O(m*log(m))
  - Note: Much lower complexity than edit distance

Examples

- o="Mueller", o'="Müller"
  - Assume q=3 and all lower cased
    - o ~ {mue, uel, ell, lle, ler}
    - o' ~ {mül, üll, lle, ler}
    - $sim_{jacc}(o,o') = 2/7$
  - Assume q=2 and all lower cased
    - o ~ {mu, ue, el, ll, le, er}
    - o' ~ {mü, ül, ll, le, er}
    - $sim_{jacc}(o,o') = 3/8$
- o="schlosstür", o'="türschloss", q=3
  - o ~ {sch, chl, hlo, los, oss, sst, stü, tür}
  - o' ~ {tür, ürs, rsc, sch, chl, hlo, los, oss}
  - $sim_{jacc}(o,o') = 6 / 10$ (!)

Properties

- Properties of Jaccard for strings
  - Deals with arbitrary-length strings (vs Hamming)
  - Is somewhat order-sensitive (within q-grams)
  - But sensitive to mismatches: A single mismatch destroys q q-grams
  - Can be combined with tfidf over q-grams
- Very popular
  - Use multiple q at once
  - Quite effective if identical strings are highly similar
    - But not very sensitive for lower similarities: Quickly decreasing scores
  - Faster than edit distance

Lower Bounding Edit Distance

- Since edit distance is a gold standard but costly to compute, two-phase algorithms are typical
  - Again, $m=\max(|o|,|o'|)$
  - Assume we are only interested in pairs with $sim_{edit}(o,o')>t$
  - Some math derives $dist_{edit}(o,o'):=d(o,o') < m * (1-t)$
  - Assume we have a (fast) function $f(o,o')$ with $f(o,o') \leq d(o,o')$
    - f is a lower bound for edit distance
  - Together: $f(o,o')\geq m*(1-t) \rightarrow d(o,o')\geq m*(1-t) \rightarrow sim_{edit}(o,o')\leq t$
- There are many lower bounds – with better complexity
  - Length: $||o|-|o'|| \leq dist_{edit}(o,o')$
  - Hamming: $dist_{ham}(o,o')+||o|-|o'|| \leq dist_{edit}(o,o')$
  - Jaccard: $(m+q-1-|q\text{-}set(o)\cap q\text{-}set(o')|) / q \leq dist_{edit}(o,o')$
  - Even better: Jaccard with positional q-grams (see literature)
- Generally: The higher the demands, the better the filtering

Similarity Functions for Tuples

- Tuples = Objects consisting of a flat set of attributes
  - Such as tuples in a RDBMS
  - Attributes have different types and different domains
- Weighted sum method
  - Assume a specific sim function $s_1, s_2, \dots s_n$ per attribute $A_1, A_2 \dots A_n$
  - Assume a weight vector $w=\{w_1, w_2, \dots w_n\}$ with $\Sigma w_i=1$
  - For objects $o=\{a_1, a_2, \dots a_n\}$ and $o'=\{a'_1, a'_2, \dots a'_n\}$

$$\begin{aligned}\mathrm{sim}(\mathrm{o},\mathrm{o}') &= w_1 * s_1(a_1,a_1') + w_2 * s_2(a_2,a_2') + \ \dots + w_n * s_n(a_n,a_n') \\ &= \sum w_i s_i(a_i,a_i')\end{aligned}$$

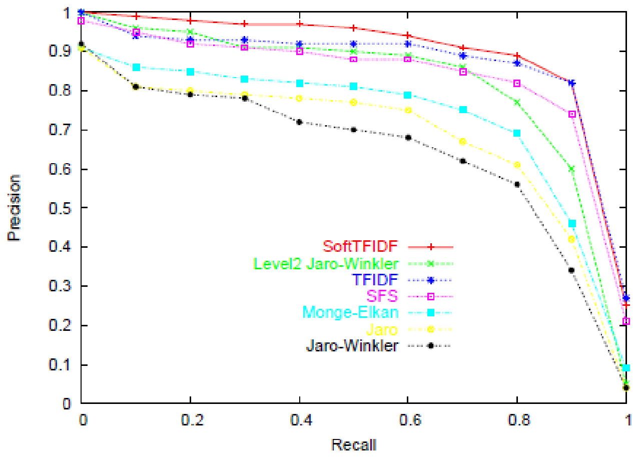
- How to obtain meaningful weights?
  - Educated guessing; learn from gold standard, e.g. linear regression

How to Choose the Right Similarity Function?

- Empirical: Have a gold standard, try many, select best
- Causal: Consider the source of deviations in duplicates
  - Example: Strings
  - Information transmitted acoustically: soundex
  - Information transmitted via networks: hamming, edit
  - Information typed in with keyboards: distance between keys, transpositions, strokes between two keys, …
- Knowledge-based: Have rules for common deviations
  - Especially abbreviations: Dr->Doctor, Str->Strasse, Str.->Str, …
  - In general: Domain-specific lists of synonyms
  - Usually necessary to achieve high accuracy
    - Though computer scientists hate it

Some Evaluation Results for Strings [CRF03]

Name	Src	#Strings	#Tokens
animal	1	5,709	30,006
bird1	1	377	1,977
bird2	1	982	4,905
bird3	1	38	188
bird4	1	719	4,618
business	1	2,139	10,526
game	1	911	5,060
park	1	654	3,425
fodorZagrat	2	863	10,846
ucdFolks	3	90	454
census	4	841	5,765



	MaxF1	AvgPrec
SFS	0.528	0.357
TFIDF	0.518	0.369
Jaccard	0.567	0.402
L2 Jaro-Winkler	0.746	0.770
SoftTFIDF	0.685	0.782
Jaro-Winkler	0.648	0.703
Jaro	0.687	0.731
NaiveAvgOverlap	0.697	0.731
AvgOverlap	0.701	0.736
Levenstein	0.832	0.901
Jaro	0.728	0.789
Scaled Levenstein	0.851	0.930
Levenstein	0.865	0.925

- Data sets with gold standard
- Some real, some artificial
- Census: Artificial; first name, name, street, number

- P/R for common methods across data sets
- SoftTFIDF: tfidf allowing mismatches in tokens
- Monge-Elkan: Edit dist with affine gap costs and weighted replacement costs

- Performance on census (person names)
- F1: Harmonic mean of precision and recall

Topics Today

- Information Integration and Similarity
- Similarity Functions
- Similarity Search
  - Inverted files
  - PETER
- Appendix: Computing Edit Distance

Similarity Search

- So far, we only looked at individual pairs
- More common tasks
  - Given a similarity function sim, an object o, a set O of objects
  - Similarity search: Find object o'∈O that is most similar to o
    - Actually: Find objects O' ⊆ O that are most similar to o
  - Top-k search: Fond the k objects from O most similar to o
  - Range search: Find subset O'⊆O with ∀o'∈O': sim(o,o')>t
- Naïve solution: Compare o to all o'∈O
  - Complexity: O(|O|)*O(sim) – slow
- Note: Using a B-tree is not simple: There is no "sim-sort"

Indexing

- Idea: Can we pre-process (index) O to improve speed?
  - There exists an astonishing wealth of published methods
  - Depending on object type, distance function, type-of-search, …
- We will look at two methods
  - Inverted files for speeding-up Jaccard (sets or strings)
  - PETER for speeding-up hamming and edit-distance (for strings)
- Many, many other
  - For arbitrary object types with metric distances: M-trees
  - For range search on multi-dimensional data: Grid-Files, kd-trees
  - For range search in 3D: Quad-trees
  - …

Inverted Files (or Inverted Index)

- Simple and effective index structure for sets (of tokens)
- Start from "objects containing tokens" and invert to "tokens appearing in objects"

```
d1:  t1,t3
d2:  t1
d3:  t2,t3
d4:  t1
d5:  t1,t2,t3
d6:  t1,t2
d7:  t2
d8:  t2
```

$\Rightarrow$

```
t1:  d1,d2,d4,d5,d6
t2:  d3,d5,d6,d7,d8
t3:  d1,d3,d5
```

Implementing Inverted Indexes (very basic)

- Index structure
  - Keep set of unique token (dictionary) in main memory
  - Use sorted list, hash table, or prefix tree
    - Searching a token o requires O(log(n)*|o|), ~O(|o|), O(|o|)
  - Keep list of objects containing token (posting) on disk
    - Searching posting for o requires one disk lookup + reading posting list
- See lecture / books on Information Retrieval
  - Efficient construction; with tfidf; compression; maintenance; …

Usage for Jaccard

- Given o, O
  - Build inverted index I over O
  - For each token $t_i \in o$: Find set of objects $O_i \subseteq O$ containing t_i using I
  - Build union $O' = \cup O_i$
    - These are all candidates: Objects o' having at least one token in common with o, i.e., having a sim(o,o')>0
    - Hope: $|O'| << |O|$
  - $\forall o \in O'$: Compute $sim_{jacc}(o,o')$
- Many tricks for further pruning with lower bounds
- We look at one simple trick: Size filtering
  - Other: Prefix filtering, frequency filtering, …
- There also exist specific set similarity search indexes
  - E.g. JOSIE, Vernica-Join, …

Size Filtering for Search Space Pruning

- Recall Jaccard: $\mathrm{sim}(o,o') = \frac{|o \cap o'|}{|o \cup o'|}$
- We show that: $\frac{1}{sim(o,o')} = \frac{|o \cup o'|}{|o \cap o'|} \geq \frac{|o'|}{|o|} \geq sim(o,o')$
  - Assume |o'|≥|o| (symmetric case similar and skipped)
    - Right inequation: Then |o'|/|o| ≥ 1 ≥sim(o,o')
    - Left inequation: Because |o∪o'|≥|o'| and |o∩o'|≤|o|
- Now assume we require sim(o,o')>t
  - Thus, we can require 1/t ≥ |o'|/|o| ≥ t, or |o|/t ≥ |o'| ≥ |o|*t
- Usage for pruning
  - Do not put o' into candidate set O' if this size constraints is hurt

Topics Today

- Information Integration and Similarity
- Similarity Functions
- Similarity Search
  - Inverted files
  - PETER
- Appendix: Computing Edit Distance

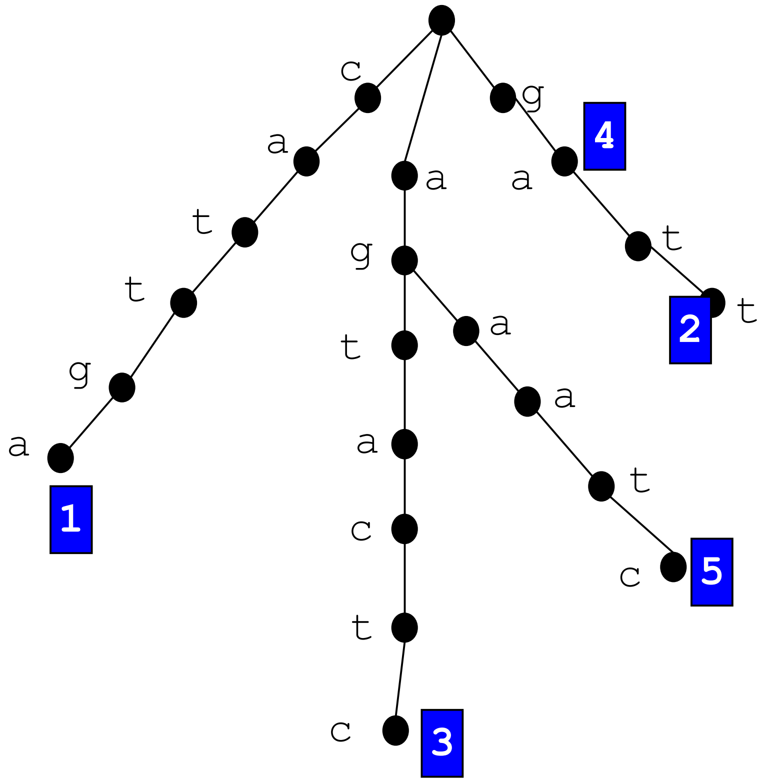
PETER [RKHL10]

- PETER: Prefix-tree based indexing algorithm for similarity search and similarity joins
  - Supports hamming distance and edit distance over strings
  - Especially suited for long strings
  - Also computes exact joins / search on large collections of long strings much faster than traditional DB technology
- There are many other (and more recent and better) string similarity search index structures
  - BED-Tree, HS-Tree, MASSJoin, …

Prefix-Trees

- Given a set O of strings
- Build a tree with
  - Labeled nodes
  - Outgoing edges have different label
  - Every string from O is spelled out on exactly one path from root
  - Mark all nodes where a string ends
- Common prefixes are represented only once

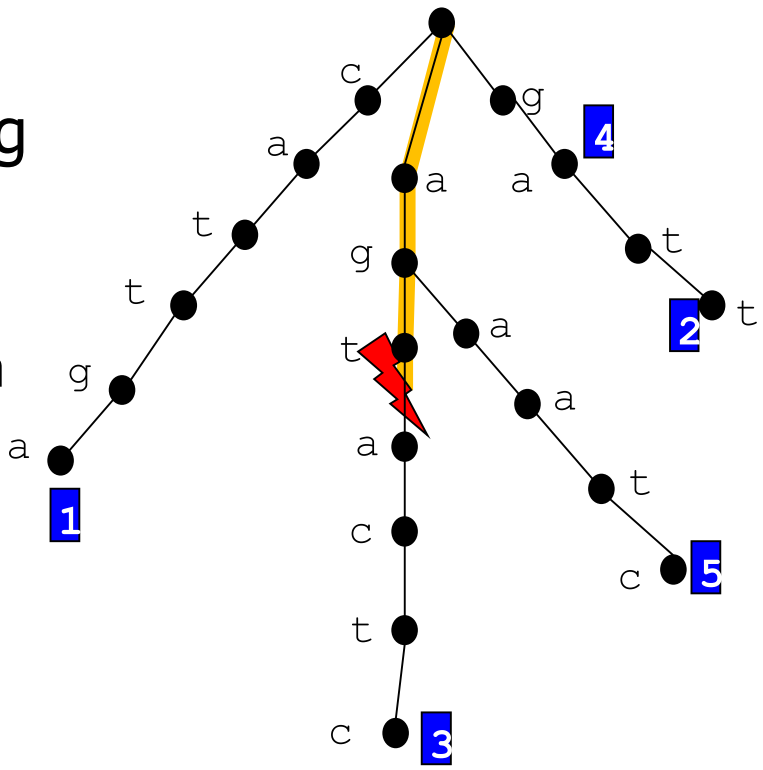
cattga, gatt, agtactc, ga, agaatc



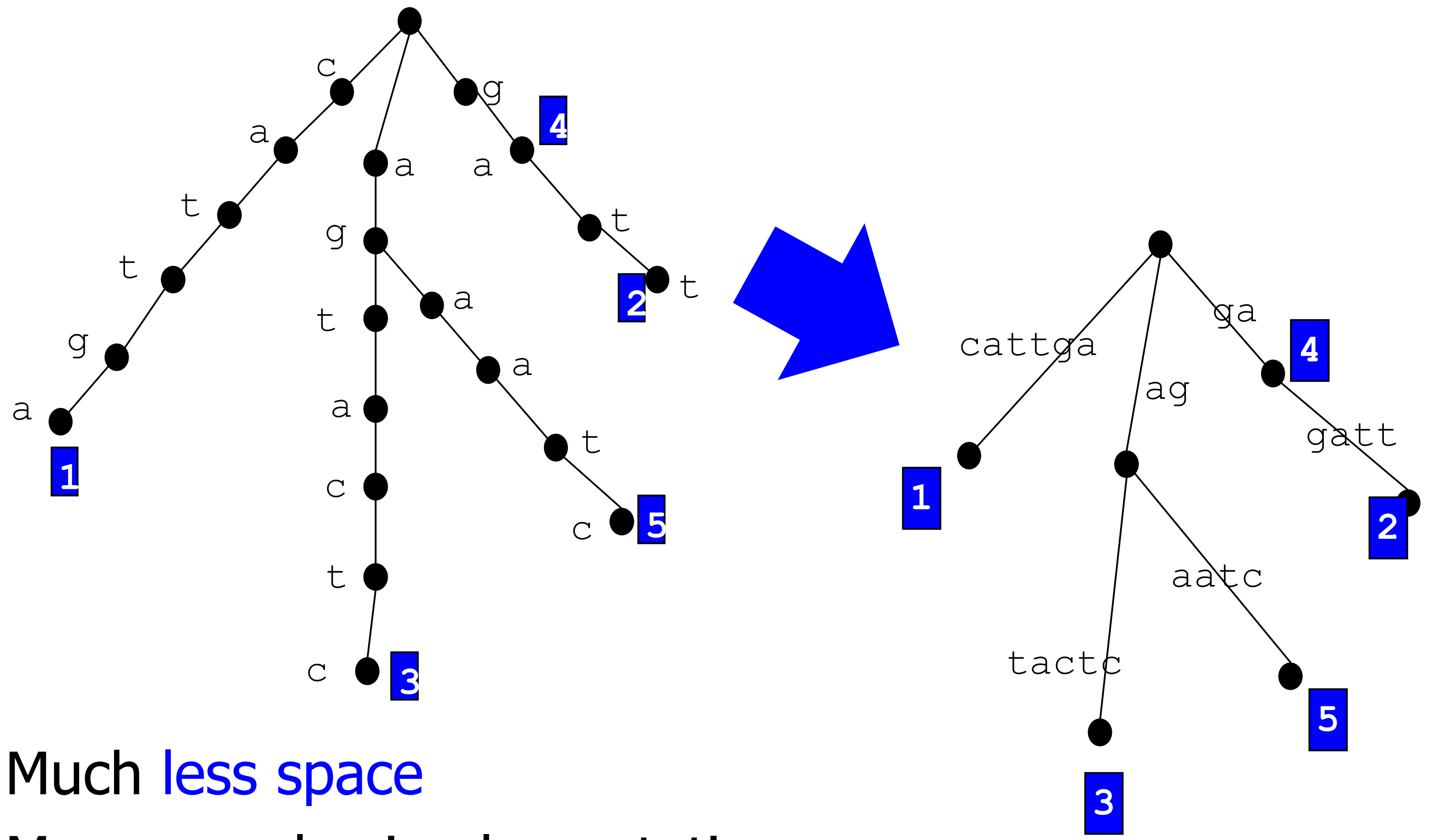
Searching Prefix-Trees

- Exact searching o in O
- Recursively match o char-by-char with a path starting from root of O
  - If no further match: $o \notin O$
  - If o matched completely at a marked node: $o \in O$
- Complexity
  - Only depends on |o|
  - Independent from |O|

Search t="agtcc"

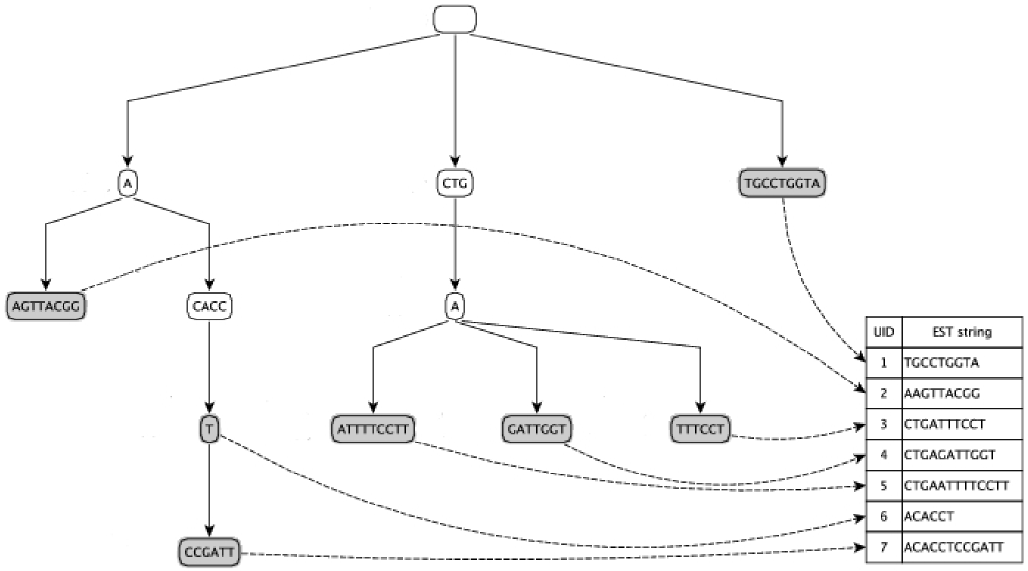


Compressed Prefix Trees (or Patricia Trees or Tries)



- Much less space
- More complex implementation
  - Different kinds of edges/nodes

Large Prefix Trees



UID	EST string
1	TGCCTGGTA
2	AAGTTACGG
3	CTGATTTCCT
4	CTGAGATTGGT
5	CTGAATTTTCCTT
6	ACACCT
7	ACACCTCCGATT

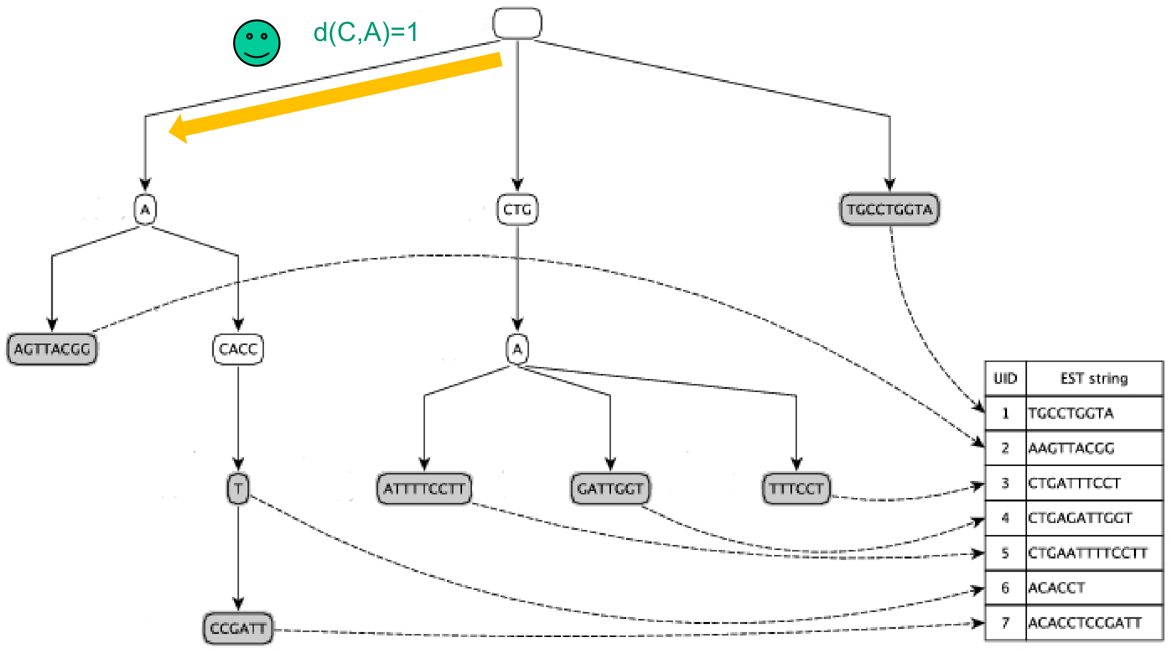
- Suffixes are stored on disk
- Tree of common prefixes is kept in main memory
  - Most failing searches never access disc
  - At most one disc IO per search
  - [If tree fits in main memory]

Similarity Search on Prefix-Trees

- In similarity search, a mismatch doesn't mean that the subtree contains no sufficiently similar o'
- Several mismatches might be allowed
  - Depending on error threshold
  - Depending on similarity function
- Idea
  - Depth-first search on the tree as usual
  - Keep a counter for the n# of errors occurring in the prefix so far
  - If counter exceeds threshold – stop searching in this branch
  - Pruning: Try to stop earlier by clever "guessing"

Example: Search

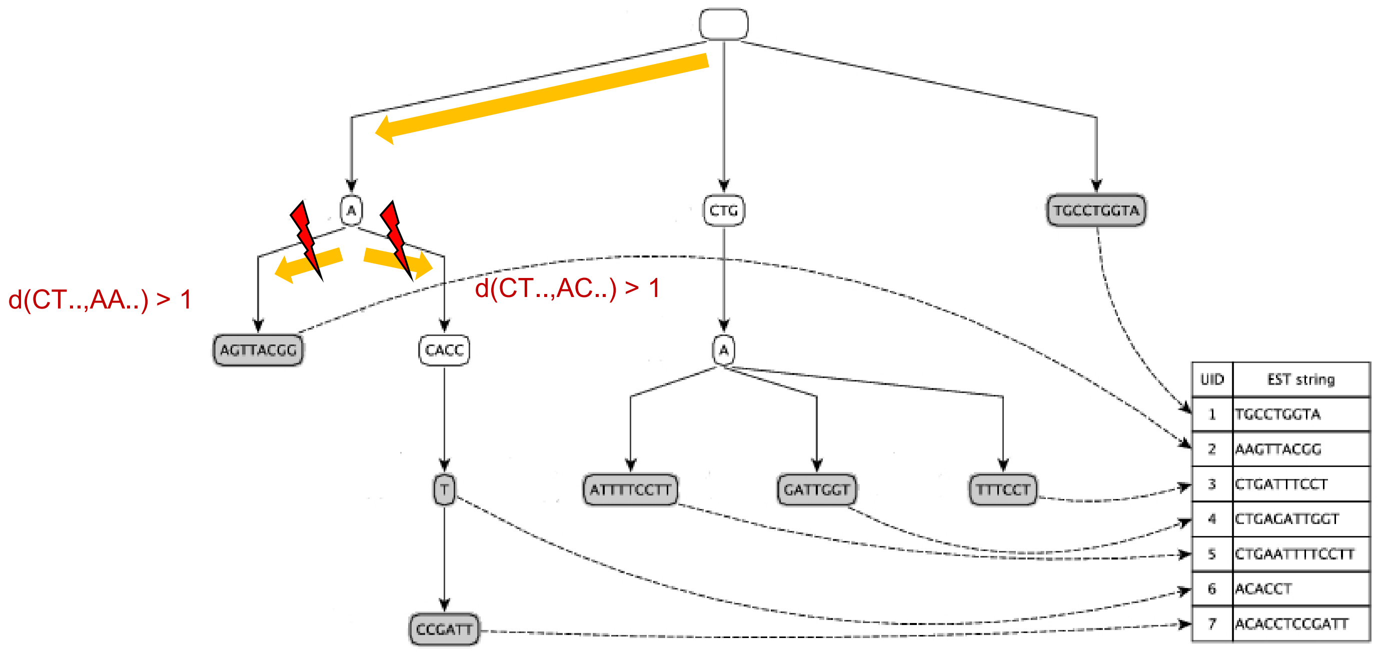
Hamming distance search for o = CTGAAATTGGT, t=1



UID	EST string
1	TGCCTGGTA
2	AAGTTACGG
3	CTGATTTCCT
4	CTGAGATTGGT
5	CTGAATTTTCCTT
6	ACACCT
7	ACACCTCCGATT

Example: Search

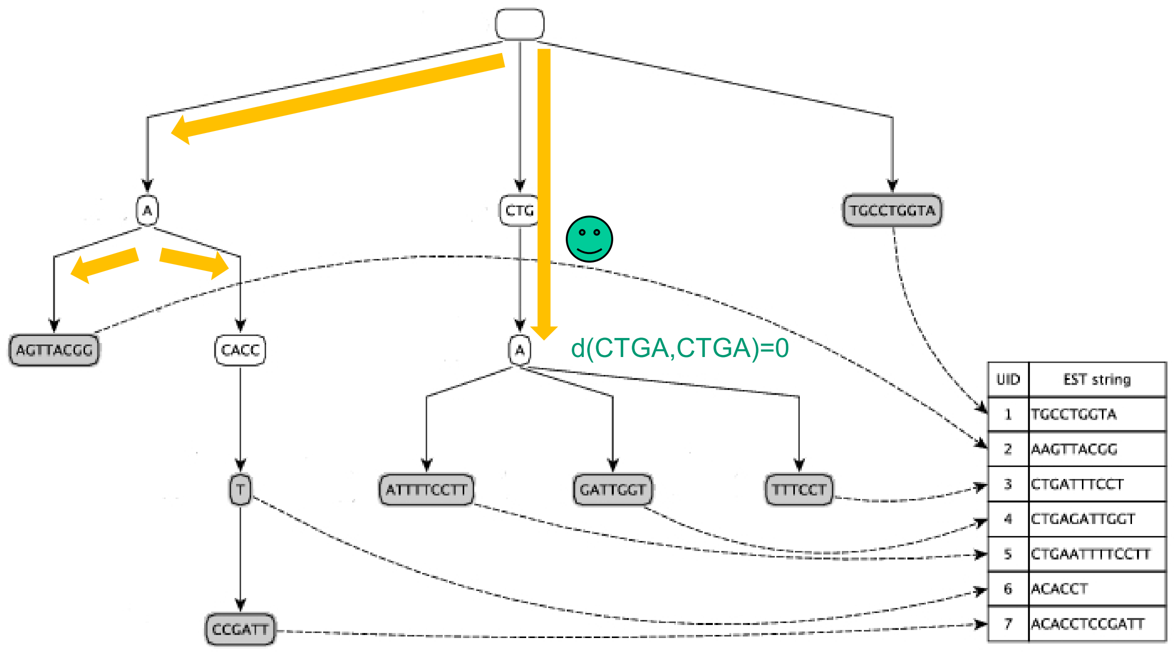
Hamming distance search for o = CTGAAATTGGT, t=1



UID	EST string
1	TGCCTGGTA
2	AAGTTACGG
3	CTGATTTCCT
4	CTGAGATTGGT
5	CTGAATTTTCCTT
6	ACACCT
7	ACACCTCCGATT

Example: Search

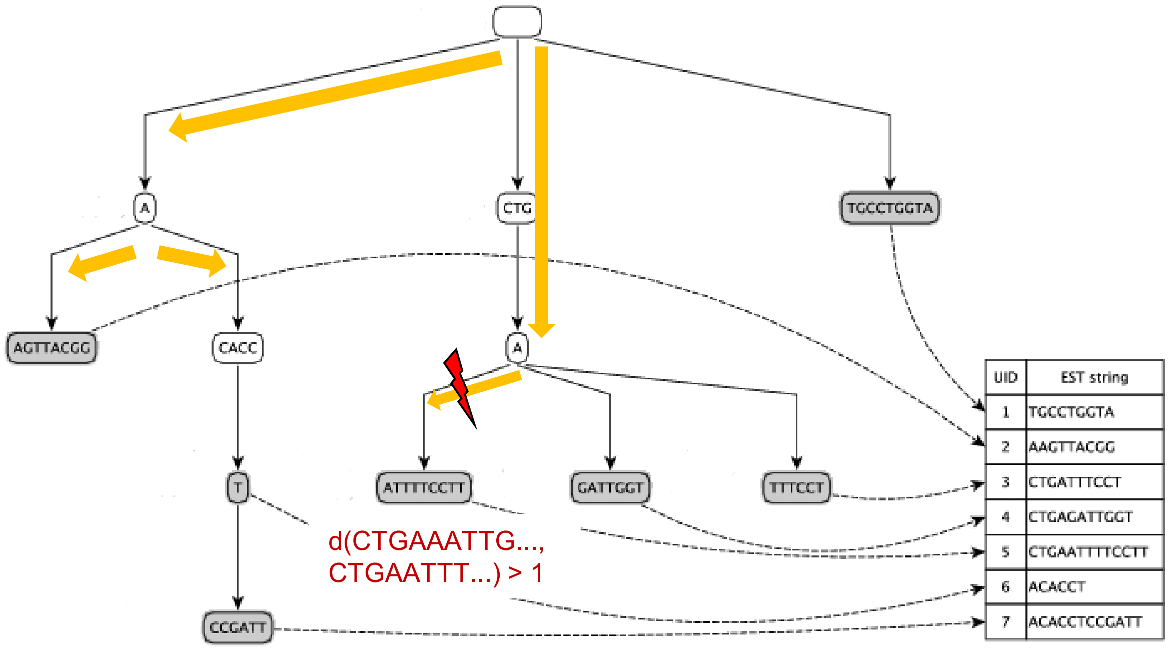
Hamming distance search for o = CTGAAATTGGT, t=1



UID	EST string
1	TGCCTGGTA
2	AAGTTACGG
3	CTGATTTCCT
4	CTGAGATTGGT
5	CTGAATTTTCCTT
6	ACACCT
7	ACACCTCCGATT

Example: Search

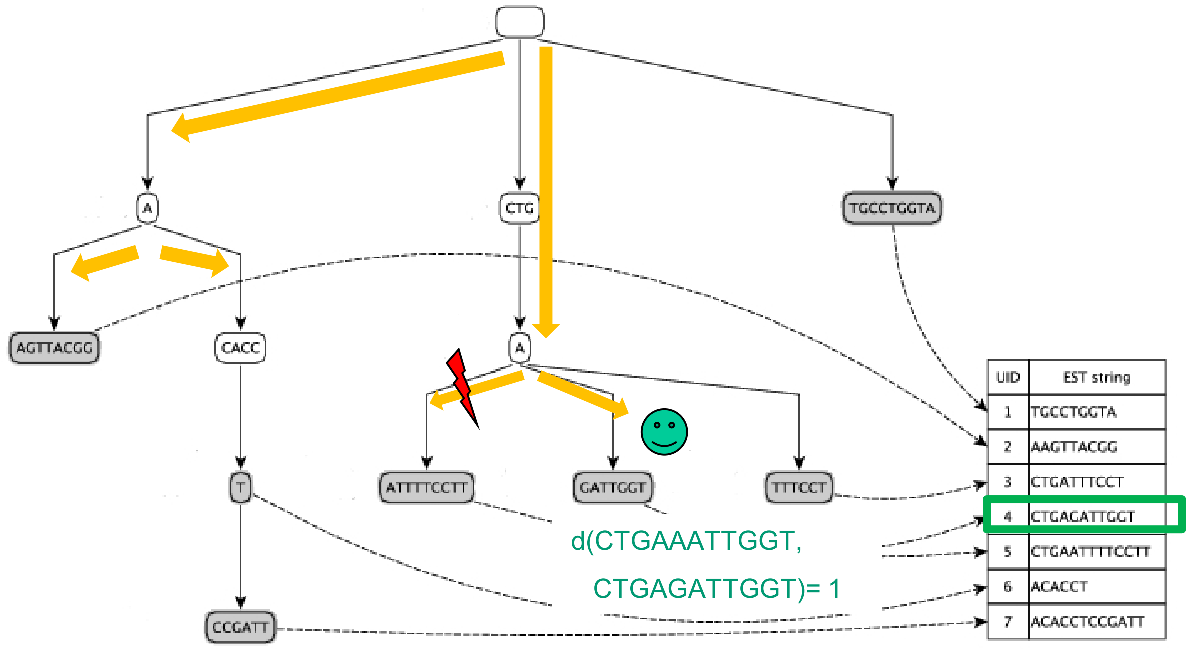
Hamming distance search for o = CTGAAATTGGT, t=1



UID	EST string
1	TGCCTGGTA
2	AAGTTACGG
3	CTGATTTCCT
4	CTGAGATTGGT
5	CTGAATTTTCCTT
6	ACACCT
7	ACACCTCCGATT

Example: Search

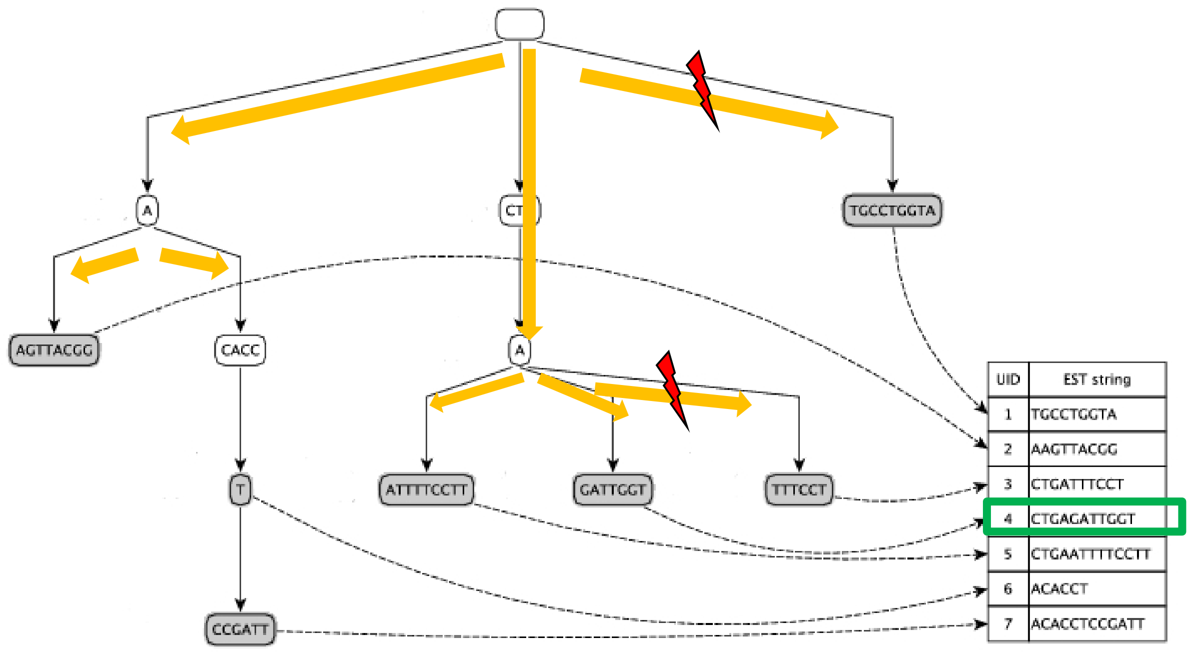
Hamming distance search for o = CTGAAATTGGT, t=1



UID	EST string
1	TGCCTGGTA
2	AAGTTACGG
3	CTGATTTCCT
4	CTGAGATTGGT
5	CTGAATTTTCCTT
6	ACACCT
7	ACACCTCCGATT

Example: Search

Hamming distance search for o = CTGAAATTGGT, t=1

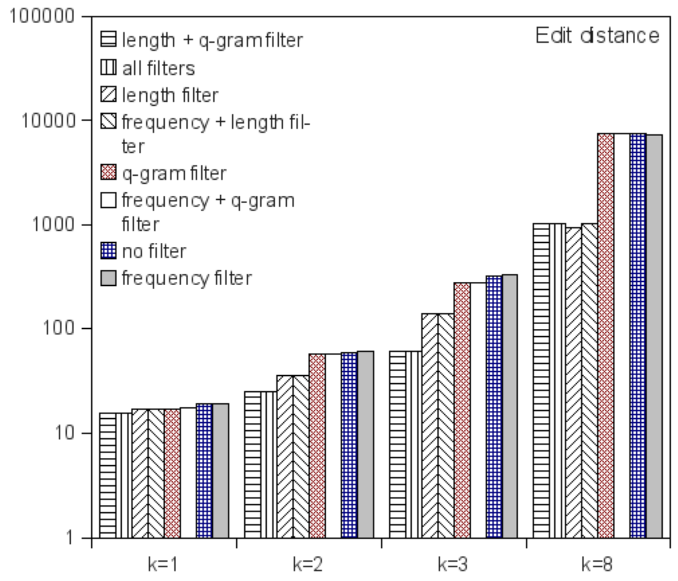


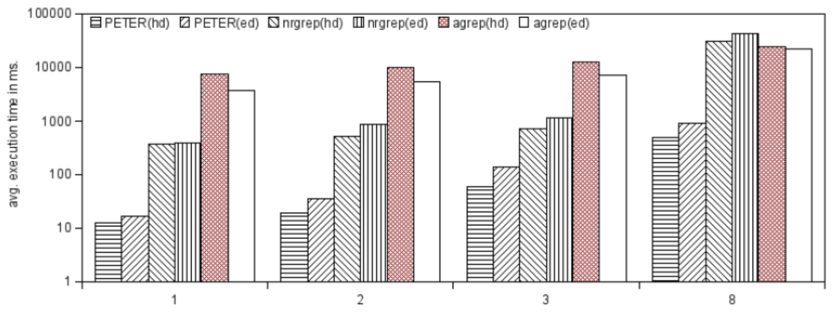
UID	EST string
1	TGCCTGGTA
2	AAGTTACGG
3	CTGATTTCCT
4	CTGAGATTGGT
5	CTGAATTTTCCTT
6	ACACCT
7	ACACCTCCGATT

Searching with Edit Distances (sketch)

- Iteratively build edit matrix when walking down the tree
  - A node represents a pair p=(prefix(o,l), prefix(o',l'))
    - Need not be of same length (i.e., l≠l'): Deletions, insertions
  - Always keep current edit matrix and minimal distance of p
  - When walking down to child: Extend matrix by one row / one col
- Further tricks
  - Use k-banded alignment (see lecture on Bioinformatics)
    - Much faster for small distances
  - Prune search space with several filters: Size filtering, frequency filtering, q-gram filtering (in leaves)
- PEARL: Parallel, main-memory based sim search and join
  - [RL11]

Some Results

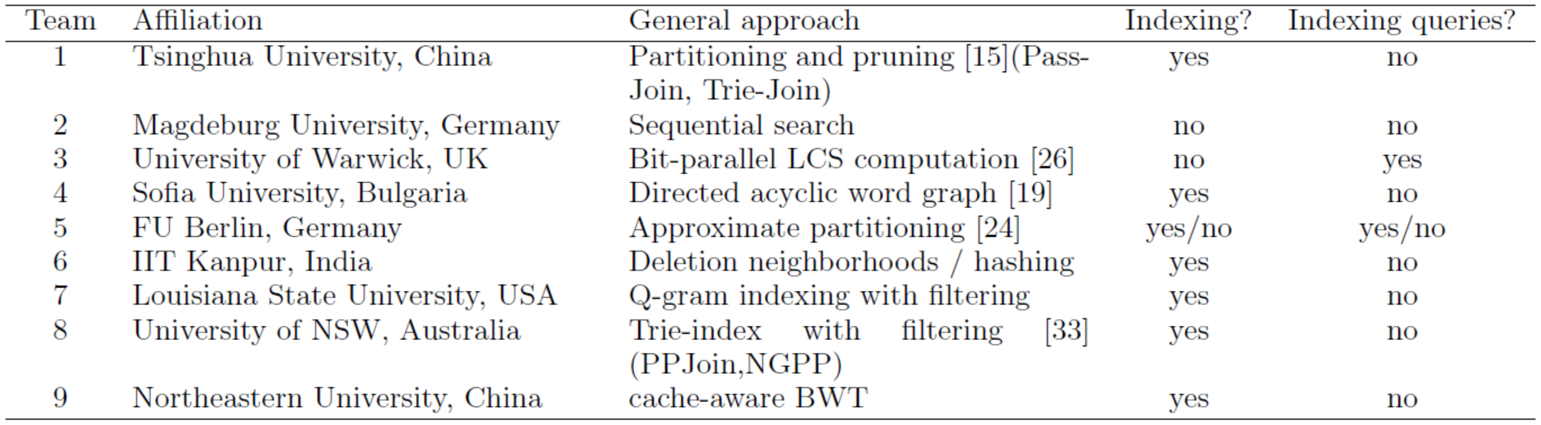




- Length filter very effective
- Sometimes, filter slow down search
- Impact dependent on threshold t and data set

- Indexing is orders-of-magnitude faster than online search (agrep, ngrep)
- For hamming and edit, for all thresholds

More Recent and Much Faster [WDG+14]



Team	Affiliation	General approach	Indexing?	Indexing queries?
1	Tsinghua University, China	Partitioning and pruning [15](Pass-Join, Trie-Join)	yes	no
2	Magdeburg University, Germany	Sequential search	no	no
3	University of Warwick, UK	Bit-parallel LCS computation [26]	no	yes
4	Sofia University, Bulgaria	Directed acyclic word graph [19]	yes	no
5	FU Berlin, Germany	Approximate partitioning [24]	yes/no	yes/no
6	IIT Kanpur, India	Deletion neighborhoods / hashing	yes	no
7	Louisiana State University, USA	Q-gram indexing with filtering	yes	no
8	University of NSW, Australia	Trie-index with filtering [33] (PPJoin,NGPP)	yes	no
9	Northeastern University, China	cache-aware BWT	yes	no

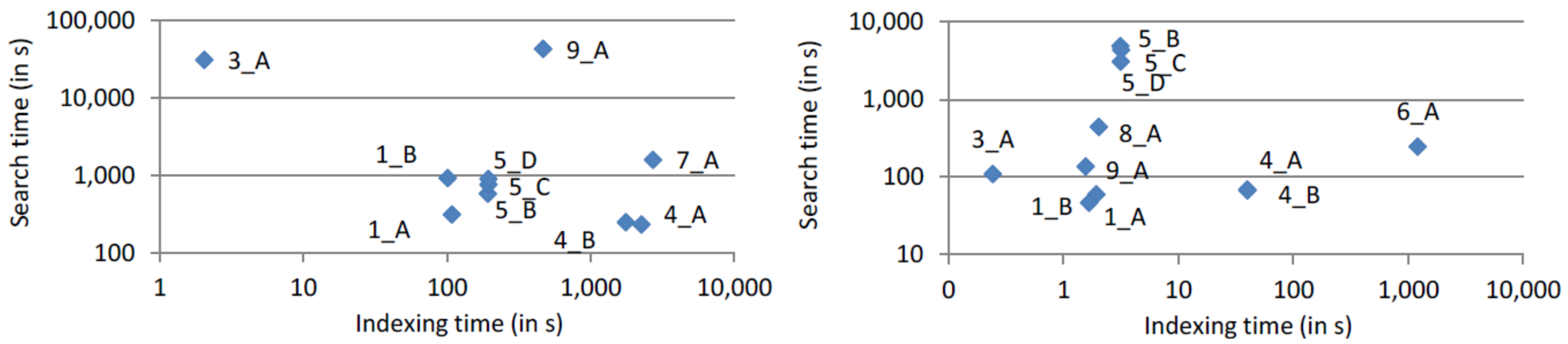


Figure 8: Search/Indexing times for READS-HUGE (left) and CITIES-HUGE (right) [time in seconds].

Literature

[CRF03]	Cohen, W., Ravikumar, P. and Fienberg, S. (2003). "A comparison of string metrics for matching names and records". Workshop on Data Cleaning and Object Consolidation
[DHI12]	Doan, A., Halevy, A. and Ives, Z. G. (2012). "Principles of Data Integration", Elsevier.
[Jac02]	Paul Jaccard: Lois de distribution florale dans la zone alpine, Bulletin de la Société Vaudoise des Sciences Naturelles, Band 38 (1902), S. 72,
[Jar89]	Jaro, M. A. (1989). "Advances in record linkage methodology as applied to the 1985 census of Tampa Florida". Journal of the American Statistical Association. **84** (406)
[Lev66]	Vladimir I. Levenshtein: Binary codes capable of correcting deletions, insertions, and reversals. In: Doklady Akademii Nauk SSSR. Band 163, Nr. 4, 1965, S. 845–848 (russisch, Englische Übersetzung in: Soviet Physics Doklady, 10(8) S. 707–710, 1966).
[RKHL10]	Rheinländer, A., Knobloch, M., Hochmuth, N. and Leser, U. (2010). "Prefix Tree Indexing for Similarity Search and Similarity Join on Genomic Data". SSDBM, Heideberg, Germany.
[RL11]	Rheinländer, A. and Leser, U. (2011). "Scalable Sequence Similarity Search and Join in Main Memory on Multi-Cores". 2nd Workshop on High Performance Bioinformatics and Biomedicine, France.
[RO18]	Robert C. Russell und Margaret King Odell , 1918, US-Patent 1,261,167
[WDG+14]	Wandelt, S., Deng, D., Gerdjikov, S., Mishra, S., Mitankin, P., Patil, M., Siragusa, E., Tiskin, A., Wang, W., Wang, J., et al. (2014). "State-of-the-art in String Similarity Search and Join." SIGMOD Record 43(1): 64-76.

Topics Today

- Information Integration and Similarity
- Similarity Functions
- Similarity Search
- Appendix: Computing Edit Distance

Editskripte

- Definition
  Ein Editskript e für zwei Strings A, B aus $\Sigma^{*} = \Sigma \cup$ "_" ist eine Sequenz von Editieroperationen
  - *I (Einfügen eines Zeichen $c \in \Sigma$ in A)*
    - *Dargestellt als Lücke in A; das neue Zeichen erscheint in B*
  - *D (Löschen eines Zeichen c in A)*
    - *Dargestellt als Lücke in B; das alte Zeichen erscheint in A*
  - *R (Ersetzen eines Zeichen in A mit einem anderen Zeichen in B)*
  - *M (Match, d.h., gleiche Zeichen in A und B an dieser Stelle)*

  so, dass e(A)=B
- Beispiel: A=„ATGTA", B=„AGTGTC"
  -
```
MIMMMR          IRMMMDI
A_TGTA          _ATGTA_
AGTGTC          AGTGT_C
```

Editabstand

- Offensichtlich gibt es immer unendlich viele Editskripte
- Definition
  - *Die Länge eines Editskript ist die Anzahl von Operationen o im Skript mit o∈{I, R, D}*
  - *Der Editabstand (oder Levenshtein-Abstand) zweier Strings A, B ist die Länge des kürzesten Editskript für A, B*
- Bemerkung
  - Matches zählen nicht – interessant sind nur die Änderungen
  - Es gibt oft verschiedene kürzeste Editskripte
  -
```
IMMMMMD     DMMMMMI
_AGAGAG     AGAGAG_
GAGAGA_     _GAGAGA
```

Alignment

- Definition
  - *Ein (globales) Alignment zweier Strings A,B ist eine Untereinanderanordnung von A und B mit beliebigen zusätzlichen Leerzeichen, ohne dass zwei Leerzeichen untereinander stehen*
    - Achtung: Untereinanderstehende Zeichen müssen nicht matchen
  - *Der Alignmentscore eines Alignment ist die Anzahl von Leerzeichen und Mismatches*
  - *Der Alignmentabstand zweier Strings A, B ist der minimale Alignmentscore aller Alignments der beiden Strings*
- Beispiele

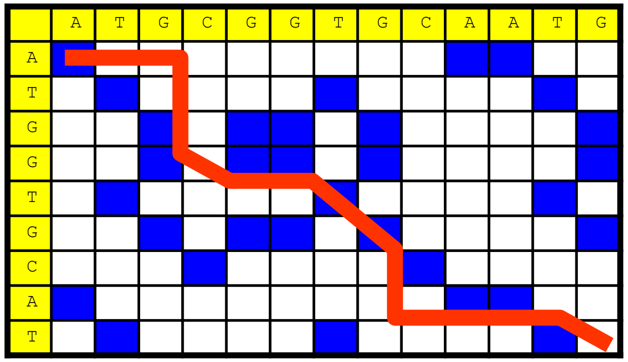
```
– A_TGT_A      A_T_GTA      _AGAGAG      AGAGAG_
  AGTGTC_      _AGTGTC      GAGAGA_      _GAGAGA
```

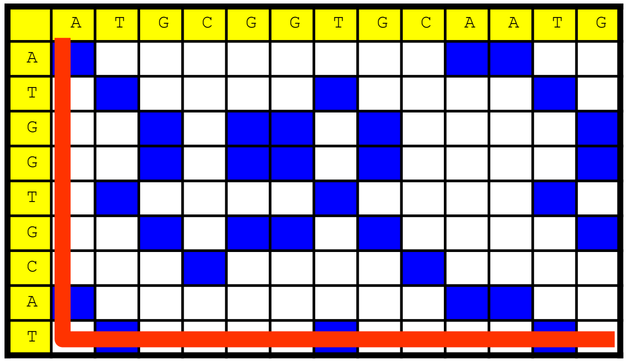
Score: **3** **5** **2** **2**

Alignments und Dotplots

- Übersetzung von Pfaden im Dotplot in Alignments
  - Dotplot: Sei A horizontal und B vertikal aufgetragen
  - Alignment: Sei A über B angeordnet
  - Schritt nach rechts: Nächstes Zeichen von A; „_“ in B
  - Schritt nach unten: Nächstes Zeichen von B; „_“ in A
  - Schritt nach rechts-unten: Nächstes Zeichen von A und B

```
ATG___CGGTG__CAATG          __________ATGCGGTGCAATG
___ATGG__TGCA____T          ATGGTGCCAT_____________
```

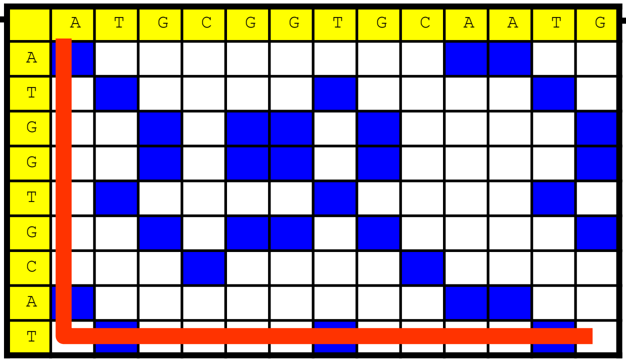




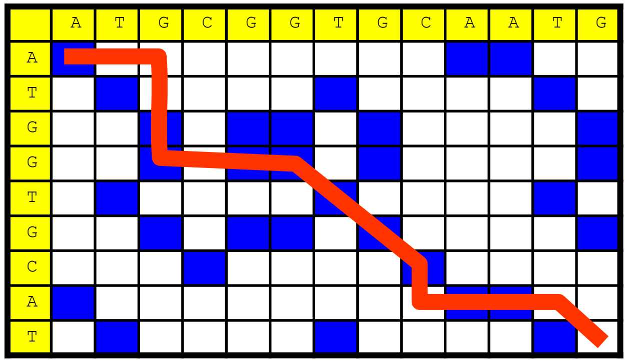
Pfadgüte

- „Gute Pfade“ haben viele Matches (1‘er Felder)
- Definition
  - *Die Güte eines Pfades P durch einen Dotplot M ist die Anzahl an diagonal durchquerten 1‘er Feldern*
  - *Die Länge eines Pfades P durch einen Dotplot M ist die Anzahl an Schritten, die nicht diagonal durch 1‘er Felder laufen*
- Bemerkung
  - Der beste Pfad kann also höchstens Güte min(m,n) haben

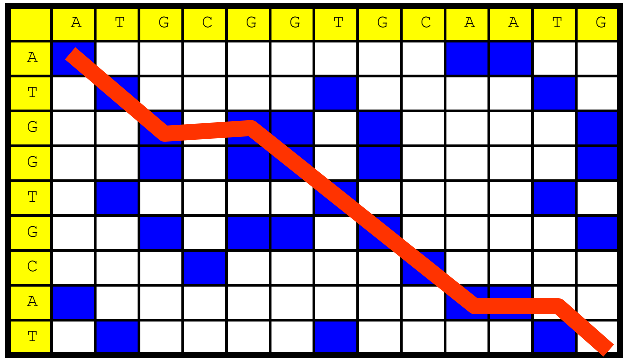
Beispiele



Pfadgüte: 0



Pfadgüte: 4



Pfadgüte: 9

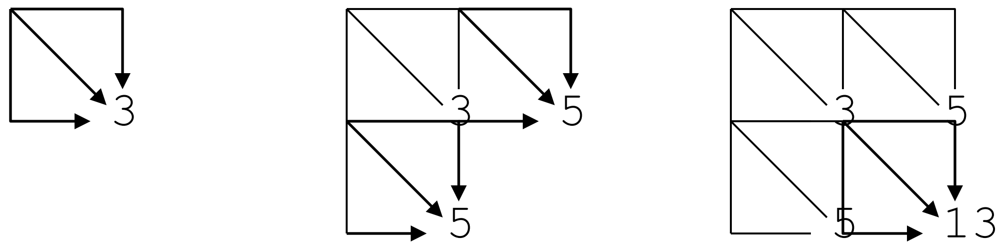
Maximale Güte?

Äquivalenzen

- Gegeben zwei Strings A,B und deren Dotplot M
- Die folgenden Probleme sind äquivalent
  - Finde das optimale Alignment von A und B (= Alignmentabstand)
  - Finde die minimale Menge an Editoroperationen von A nach B (= Editabstand)
  - Finde in M den Pfad mit minimaler Länge (= Pfadlänge)
- Beweis: Einfach
- Wir verwenden im Folgenden meistens Alignments
  - Einfacher zu lesen, weniger redundant, platzsparend

Algorithmus

- Naives Verfahren um den besten Pfad zu finden
  - Alle Pfade aufzählen
  - Das sind exponentiell viele

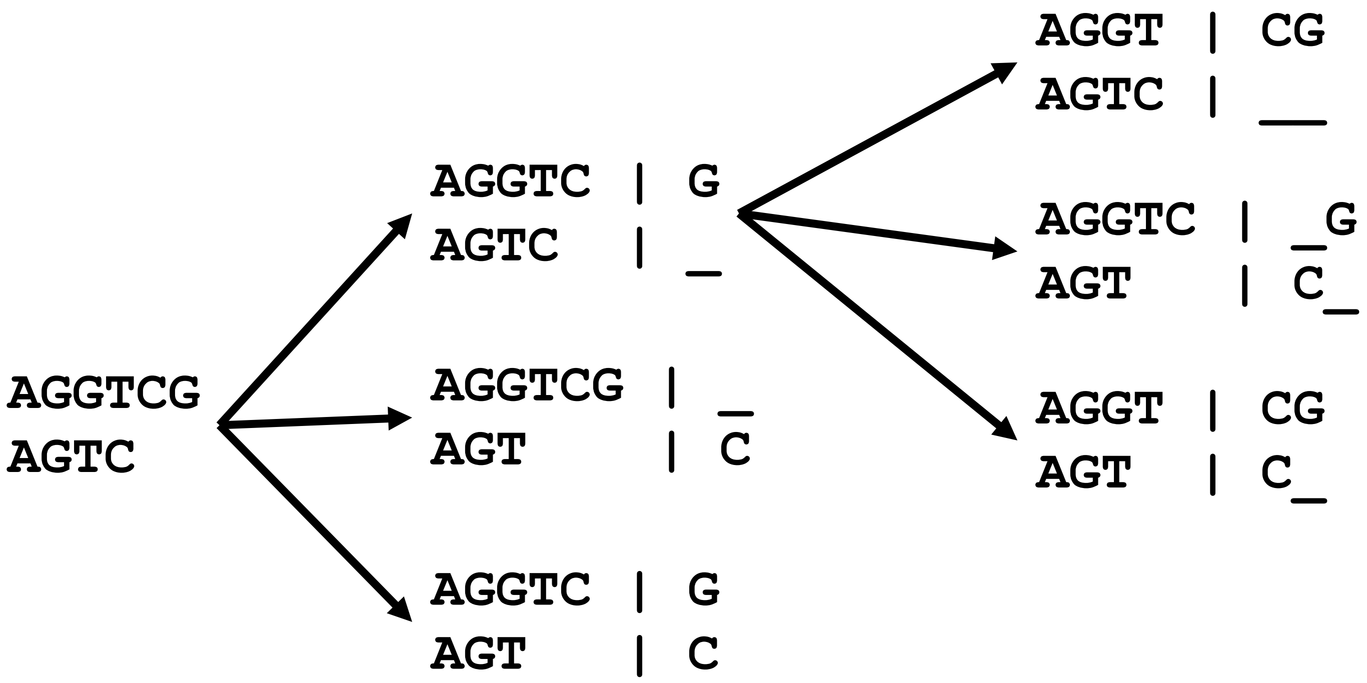


- Nur Pfade „um“ die Hauptdiagonale: $> 3^{\min(m,n)}$
  - Genaue Anzahl Pfade: Übungsaufgabe
  - Inakzeptable Laufzeit
- Tatsächliche Komplexität des Problems: O(m*n)

Editabstände

- Definition
  Gegeben zwei Strings A, B mit |A|=n, |B|=m
  – *Funktion dist(A,B) berechne den Editabstand von A, B*
  – *Funktion d(i,j), 0≤i≤n und 0≤j≤m, berechne den Editabstand zwischen A[1..i] und B[1..j]*
- Bemerkungen
  – Offensichtlich: d(n,m)=dist(A,B)
  – d(i,j) dient zur rekursiven Berechnung von dist(A,B)
  – Divide-and-Conquer: Wie kann man d(i,j) aus „kleineren“ d(x,y) Werten berechnen?

Rekursive Betrachtung



Zusammen

- Theorem
  - *Der Editabstand zweier Strings A,B mit |A|=n, |B|=m berechnet sich mit Startbedingung*

$$d(i,0) = i \qquad d(0,j) = j$$

als d(n,m) mit folgender Rekursionsgleichung

$$d(i,j) = \min\begin{Bmatrix} d(i,j-1)+1 \\ d(i-1,j)+1 \\ d(i-1,j-1)+t(i,j) \end{Bmatrix}$$

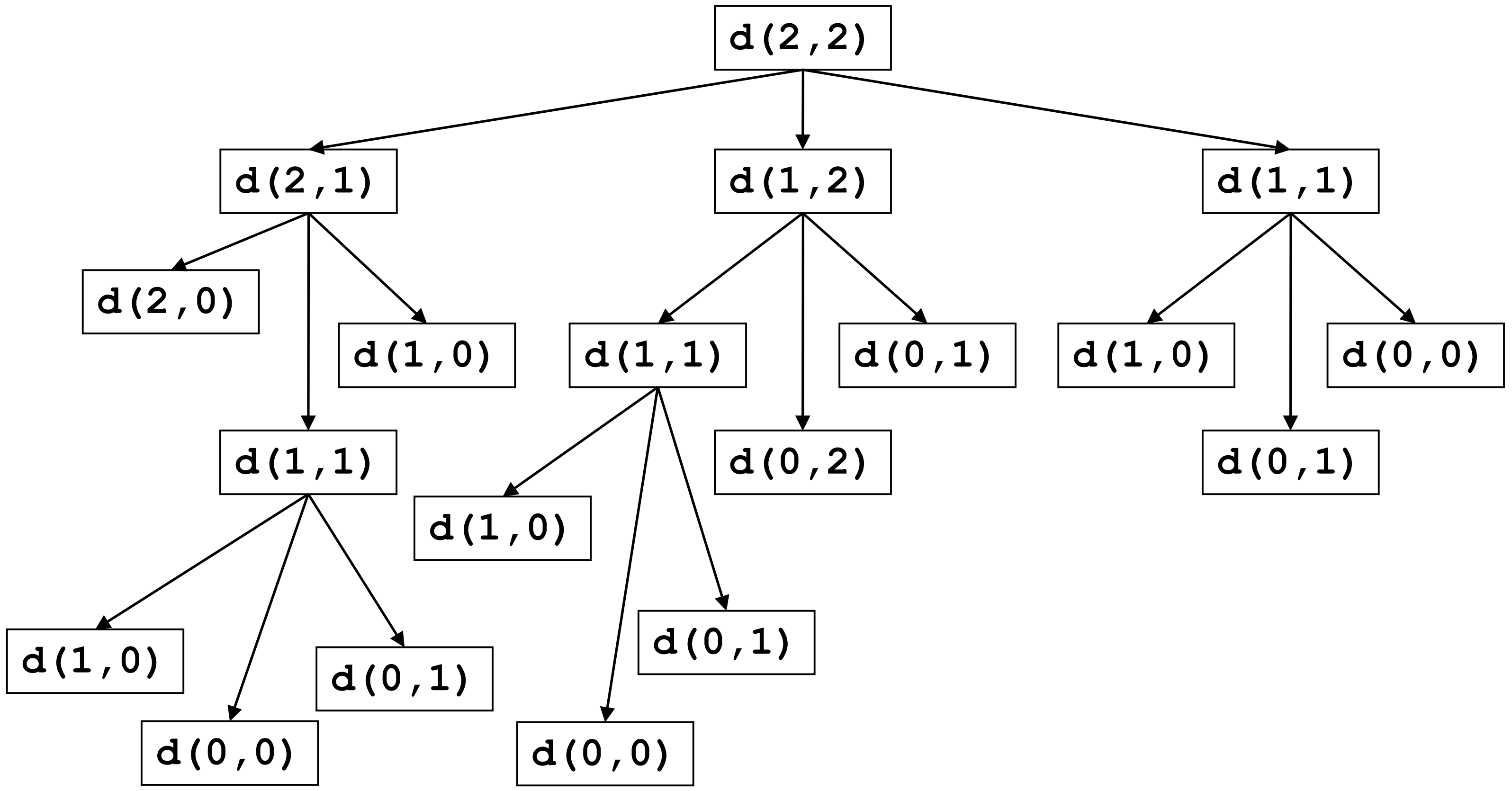
wobei t(i,j) = 0 wenn A[i]=B[j] sonst 1

Rekursiver Algorithmus

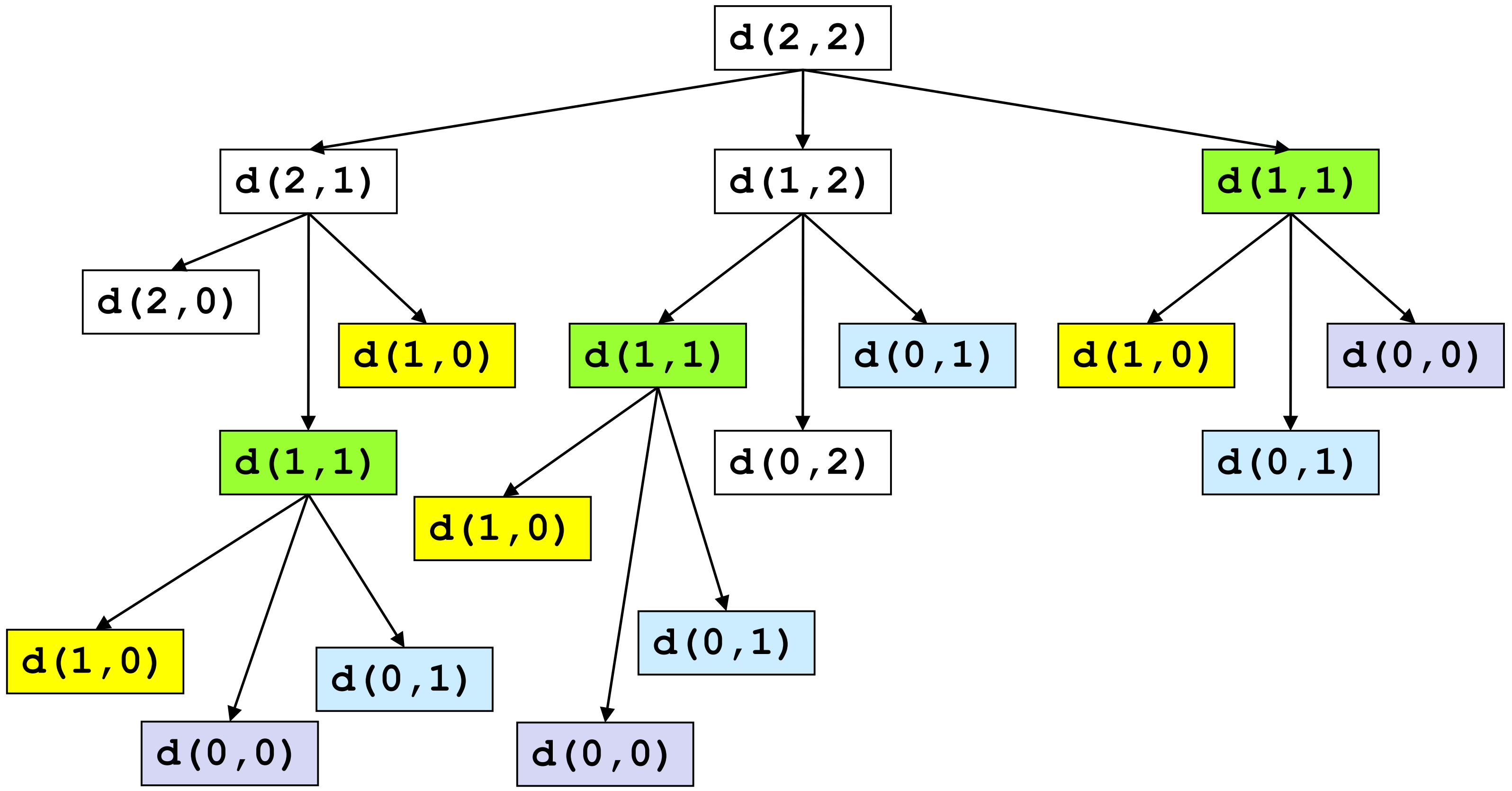
```
function d(i,j) {
        if (i = 0)                return j;
        else if (j = 0)           return i;
        else
                return min (   d(i-1,j) + 1,
                               d(i,j-1) + 1,
                               d(i-1,j-1) + t(A[i],B[j]));
}
function t(c₁, c₂) {
        if (c₁ = c₂)              return 0;
        else                      return 1;
}
```

- Komplexität?
  - Für (n,m) erfolgen 3 Aufrufe, die wiederum jeweils 3 Aufrufe auslösen, die ...
  - Komplexität damit mindestens $O(3^{\min(n,m)})$

Aufrufbaum


d(2,2)
d(2,1)
d(1,2)
d(1,1)
d(2,0)
d(1,0)
d(1,1)
d(0,1)
d(1,0)
d(0,0)
d(1,1)
d(0,2)
d(0,1)
d(1,0)
d(1,0)
d(0,1)
d(0,1)
d(0,0)
d(0,0)

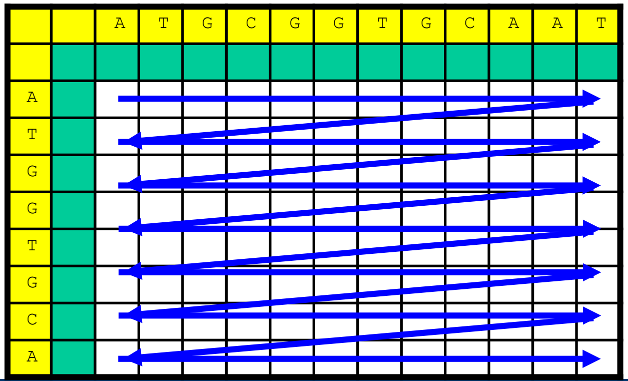
Redundanz

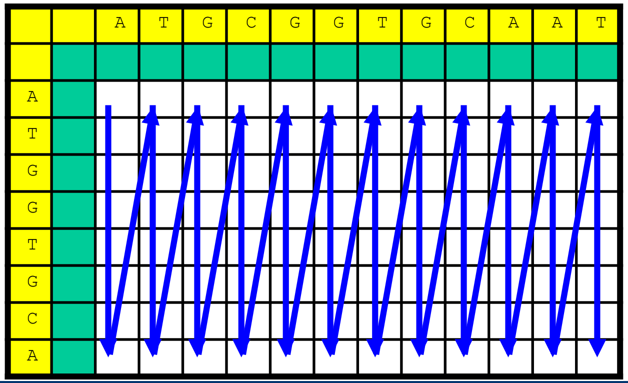


Es gibt nur (n+1)*(m+1) verschiedene Aufrufe

Tabellarische Berechnung

- Grundidee
  - Speichern der Teillösungen in Tabelle
  - Bei Berechnung: Wiederverwendung wo immer möglich
- Aufbau der Tabelle: Bottom-Up (statt rekursiv Top-Down)
  - Initialisierung mit festen Werten d(i,0) und d(0,j)
  - Sukzessive Berechnung von d(i,j) mit steigendem i,j
  - Für d(i,j) brauchen wir d(i,j-1), d(i-1,j) und d(i-1,j-1)
  - Verschiedene Reihenfolgen möglich





Beispiel

$$d(i,j) = \min\begin{Bmatrix} d(i,j-1)+1 \\ d(i-1,j)+1 \\ d(i-1,j-1)+t(i,j) \end{Bmatrix}$$

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1							
T	2							
G	3							
G	4							

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0						
T	2							
G	3							
G	4							

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2							
G	3							
G	4							

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3							
G	4							

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4							

		A	T	G	C	G	G	T
	0	1	2	3	4	5	6	7
A	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
G	3	2	1	0	1	2	3	4
G	4	3	2	1	1	1	2	3

Komplexität

- Berechnung einer Zelle betrachtet genau drei andere Zellen
- m*n Zellen
- Insgesamt: O(m*n)