Topic 7: Strings and Biological Data

- In some applications we store, search and analyze long sequences of discrete characters, which we call "strings"
- Typical Applications are Text Retrieval, Computational Biology, Signal Processing, etc.
- Queries on string sequences often allow errors in matches. Therefore an interesting and challenging subject is approximate string matching

Application 1: Information Retrieval

- A typical application of information retrieval is text searching; given a large collection of documents and some text keywords we want to find the documents which contain these keywords.
- In many cases search should allow errors. For example text collections digitized by OCR contain a large percentage of errors (7-16%). In addition, there could also be typing or spelling errors in documents.
- Therefore there are two research directions in text retrieval: exact search and approximate search. Naturally, approximate search is more difficult and expensive than exact search.

Application 2: Computational Biology

- The problem is similar in computational biology; here we have a long DNA sequence and we want to find subsequences in it that match approximately a query sequence.
- DNA and protein sequences can be seen as long texts over specific alphabets (e.g., {A,C,G,T}). Those sequences represent the genetic code of living beings.
- The similarity of two DNA substrings from different organisms may correspond to the same functional or physical relationship between these organisms.
- Exact searching here is of little use, since the query patterns rarely match the text exactly; the correct results may have small differences due to mutations and evolutionary alternations.

Application 3: Signal Processing

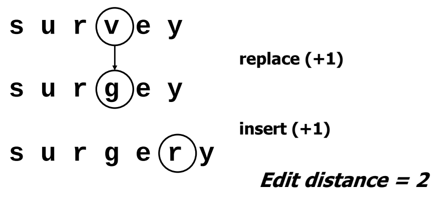
- In Speech Recognition the general problem is to determine a textual message from a transmitted audio signal. The signal could be compressed, or some words may not be pronounced well, so approximate matching is used.
- Another related problem is error correction. Compression introduces errors in the transmitted signals, so approximate search is needed for correcting these errors.

Similarity Metrics

- The similarity metric between two strings is typically dependent on the application and does not allow for general-purpose solutions.
- The most widely accepted similarity metric is the "edit distance". The edit distance between two strings is defined by the number of primitive operations (insert, delete, replace) necessary to transform one string to the other

Example of edit distance

- What is the edit distance between "survey" and "surgery"?



Edit Distance (cont'd)

- In the general version of edit distance, different operations may have different costs, or the costs depend on the characters involved.
- For example replacement could be more expensive than insertion, or replacing "a" with "o" could be less expensive than replacing "a" with "k".
- The general edit distance is powerful enough for a wide range of applications therefore most query processing algorithms consider it as a standard.
- The general edit distance does not satisfy the triangular inequality and thus it is *not* a metric.

Example of generic edit distance: String Alignment

- In biological string matching the similarity metric is often called "string alignment"
- Let *S* and *T* be strings. An alignment maps *S* and *T* into string *S'* and *T'* by inserting spaces into *S* and *T*, such that |*S'*|=|*T'*|.

- Example:

```
S=acgcaggtc
T=agcgtc

acgcaggtc
|||||||||
ag_cg__tc

acgcaggtc
|||||||||
a_gc__gtc
```

optimal alignment=distance

Other Similarity Metrics

- Edit distance with transpositions (e.g., ab→ba)
- LCS distance (allows only insertions/deletions)
- Hamming distance (allows only substitutions – only when |s1|=|s2|)
- Episode distance (allows only insertions – not symmetric)
- Reversals (allows reversing substribgs)
- Block distance (allows rearrangement and permutation of substrings)
- *q*-gram distance (based on finding common substrings of fixed length *q*).

Definition of the basic approximate search problem

- Given a query substring q and a long sequence t, find all substrings in t which are similar to q given a distance metric.
- Two versions of the problem:
  1. We are given a similarity threshold k and ask for all substrings within this distance from q
  2. We are given a number k and we ask for the k-Nearest Neighbors
- In a more general problem version, the long sequences t could be more than one.

A dynamic progamming algorithm for computing the edit distance

- Problem: find the edit distance between strings x and y.
- Create a (|x|+1)×(|y|+1) matrix C, where $C_{i,j}$ represents the minimum number of operations to match $x_{1..i}$ with $y_{1..j}$. The matrix is constructed as follows.
  - $C_{i,0} = i$
  - $C_{0,j} = j$
  - $C_{i,j} =$
    - C_{i-1},C_{j-1} if $x_i=y_i$,
    - $1+\min(C_{i-1},C_j, C_i,C_{j-1}, C_{i-1},C_{j-1})$, else.

Example:

Optimal alignment

		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1	0	1	2	3	4	5	6
u	2	1	0	1	2	3	4	5
r	3	2	1	0	1	2	3	4
v	4	3	2	1	1	2	3	4
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	2

How do we perform substring search?

- The same dynamic programming algorithm can be used to find the most similar substrings of a query sting q.
- The difference is that we set $C_{0,j}=0$ for all j, since any text position could be the potential start of a match.
- If the similarity distance bound is *k*, we report all positions, where $C_m \leq k$ (m is the last row – m = |q|).

Example: q=survey, t=surgery, k=2

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	2

Comments on the dynamic programming algorithm

- Observe that essentially it is the same algorithm used for Dynamic Time Warping.
- It is very flexible, since it can be used to compute most distance metrics under the "generic edit distance" definition. It also requires only O(|q|) space (only the previous column is necessary to compute the next one). Thus a single scan of t suffices.
- However, it is not efficient. The worst-case complexity is O(|q||t|).
- Several variations have been proposed to reduce its complexity.

Using diagonals of the matrix to compute the edit distance faster

- Improved versions of the basic dynamic programming algorithm are based on the observation that the diagonals of the matrix are monotonically increasing.
- These are called "diagonal transition" algorithms. The dynamic programming matrix is computed diagonal-wise instead of column-wise.
- The key to fast computation is to find in constant time where each *stroke* (i.e., diagonal sequence with the same error)

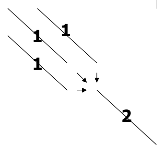
Example: q=survey, t=surgery, k=2

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r		2	1	0	1	2	2	
v			2	1	1	2		
e				2	2	1	2	
y						2	2	2

Strokes with error e≤k, ending at |q|th row indicate results

Comments on the diagonal approach

- The complexity is O(|t|k), since there are O(|t|) diagonals and at which we compute at most k strokes.
- We can use the ending of previous strokes to detect the beginning of new ones
- The hard part is to find the length of the next stroke in constant time.
- This is equivalent to finding the longest prefix of the remainder of q that matches t. It can be computed by building a suffix tree for tq.



Comments on the diagonal approach (cont'd)

- Several improvements on this method have been proposed.
- However still the algorithm has a relatively high complexity and needs to scan the whole string t.
- Thus the question is whether we can avoid scanning the whole database for each query.

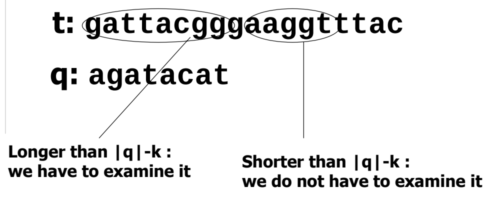
Filtering algorithms for approximate string matching

- They do not provide solutions, but aim to reduce the search effort.
- They are based on the idea that it is most probable for a text area not to match a query rather than to match it.
- They work in two steps.
- First a cheap heuristic is used to determine whether an area of the text t could match with the query q.
- If not search is abandoned for this area, otherwise an expensive search algorithm (i.e., dynamic programming) is applied

Dynamic Filtering

- Scans the text t linearly and at each position finds the longest substring that is included in q.
- Thus the text is transformed to a sequence of substrings in q, separated by non-matching characters. Each time an area of **k+1** substrings is examined.
- If this area is shorter than |q|-k, search is abandoned for it, otherwise dynamic programming is applied

Dynamic Filtering Example (k=2)

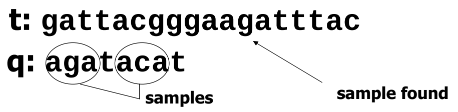

t: gattacgggaaggttac
q: agatacat
Longer than |q|-k : we have to examine it
Shorter than |q|-k : we do not have to examine it

A similar method is based on q-grams

- All substrings (q-grams) of a specific length *m* in q are computed.
- For each area of the text the number of q-grams that occur there are computed.
- If this number is smaller than (|q|-*m*+1-*km*) the text area is pruned, otherwise dynamic programming is applied.

A sampling method

- A sample of non-overlaping substrings of q is computed so that they have a fixed gap between them.
- If a text area does not contain any of the samples, it is pruned
- Otherwise the neighborhood around the found sample is verified.



Sub-linear algorithms

- In order to avoid applying the above methods at every position of the text, the text is split to blocks of (|q|-k)/2 size and substring checking is applied only at the beginning of each block.
- Blocks should be entirely contained in results, so if a block does not qualify, we immediately move to the next one.

t: gattacgggaaggtttac

q: agatacatacagatat

discarded block

move to next

Comments on the methods discussed so far

- They mainly focus on improving worst-case theoretical bounds
- In general, they require a linear scan of the database, so they are not attractive for very large problems
- Recently, there are efforts from DB researchers to make these methods more scalable

An adaptation of the suffix tree for secondary memory [Hunt et al., 2001]

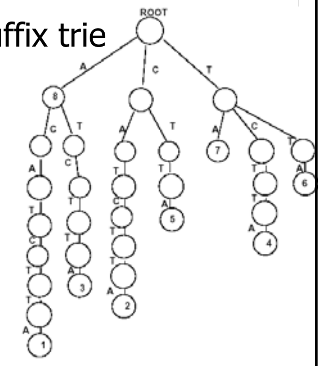
- Suffix trees are main memory structures used for exact substring search.
- Algorithms for performing approximate search on suffix trees are also available.
- However, there is no version of the suffix tree for secondary memory.
- A recent paper tries to solve this hard problem.

Example of a suffix trie

String

```
  1 2 3 4 5 6 7 8
S=A C A T C T T A
```

Suffix trie



Example of a suffix tree

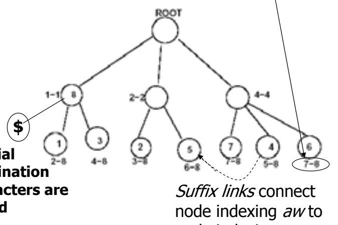
String

```
  1 2 3 4 5 6 7 8
S=A C A T C T T A
```

The *suffix tree* is a suffix trie, where unary paths are compressed

Suffix tree

Numbers showing the substring indexed by a node are added



Special termination characters are added

Suffix links connect node indexing *aw* to node indexing *w*

Suffix links

- Suffix links were introduced to improve the construction cost of the suffix tree to O(n).
- However they transform the tree to a graph, that cannot be managed in memory unless small. If we rely on the virtual memory of the machine a lot of swapping takes place and the construction algorithm is very slow.
- Biological sequences are very long (with millions or trillions of characters), thus we cannot use the suffix-link algorithm to construct them.

A new method for Suffix tree construction is proposed

- Suffix links are abandoned.
- The algorithm scans the string multiple times in order to construct the suffix tree for a subrange of suffixes in each pass.
- Abandoning suffix links means that the worst-case construction cost becomes O(n^2), but due to the pseudo-random nature of DNA, the average behavior is O(nlogn)

Description of the algorithm

- First the number of partitions is determined. If the expected size of the full suffix tree is S and the available memory is M, the number of partitions is given by $\lceil S/M \rceil$.
- Each partition corresponds to a set of suffixes. For each partition we build a separate suffix tree. These suffix trees are then connected via a common root.

Description of the algorithm (cont'd)

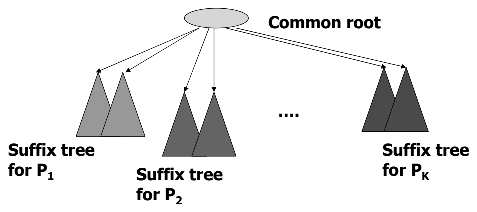
- The second step is to assign suffixes to partitions
- The partition for a suffix is determined upon its prefix. For example, suffix ATCTTA is indexed in the subtree following branches AT. Notice that these subtrees are independent.
- Thus the suffixes in the same group have the same prefix.

Description of the algorithm (cont'd)

- There are two ways to determine the partitions.
  - We scan the string once and count the occurencies of every 3-character substring. Then split them in groups with equal cardinality
  - We just partition lexicographically all 3-character strings, since we expect them to appear with equal probability in DNA

AAA	**32**	
AAC	**42**	**P_1**
AAG	**23**	
AAT	**76**	**P_2**
ACA	**43**	
ACC	**65**	
.		
.		
.		
TTT	**78**	
	3500	

Description of the algorithm (cont'd)

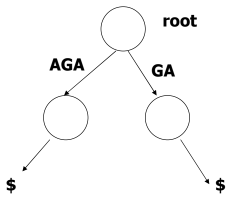


Description of the algorithm (cont'd)

- For each partition j
  - Initialize the suffix tree that corresponds to this partition. Then scan the string.
  - For each position i of the string, if the suffix s_i is in partition j, insert the suffix to the subtree which corresponds to this partition
- Thus one suffix-tree is created at each pass and these are stored independently on disk

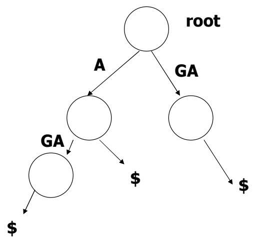
Example of Suffix tree creation

- Create a suffix tree for AGA
- The insertion order of suffixes is AGA\$, GA\$,A\$



Example of Suffix tree creation

- Create a suffix tree for AGA
- The insertion order of suffixes is AGA\$, GA\$,A\$



Overview of the partitioned suffix tree

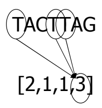
- The partitioned suffix tree is constructed for long sequences at no additional computational cost, despite the abandonment of suffix links.
- The experiments show that the construction time is linear to the sequence length.
- However the tree is tested only for exact matches, where only one path is traversed for each query.
- On the other hand, approximate search needs to combine information from multiple paths and it is questionable whether this method can perform well.

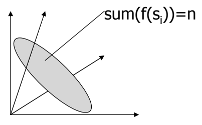
A Filtering Algorithm for a Database of Long Strings

- A new filtering algorithm for large datasets (that do not fit in memory) was recently proposed.
- The database S may contain many, potentially long strings $S = \{s_1, s_2, \ldots, s_d\}$
- The algorithm transforms the substrings in this database to high dimensional points and indexes them.
- A query is then applied on the indexes using a lower bound of the edit distance.

Idea: A Transformation and a Lower Bound for the Edit Distance

- Given an alphabet $\Sigma = \{\alpha_1, \alpha_2, \ldots, \alpha_d\}$, we can transform a string to a d-dimensional point, called *frequency vector*.
- Example m=4, Σ = {A,C,G,T}. String s=TACTTAG is transformed to f(s)=[2,1,1,3].
- The transformation maps strings of the same length n to points on the (n-1) dimensional plane





Relation between edit distance and frequency vector

- Edit operations (insert, delete, replace) has the following effect on the frequency vector:
  - A scalar increases (insert)
  - A scalar decreases (delete)
  - A scalar increases and another decreases (replace)
- Example:

$$\begin{tikzcd} \text{TACTTAG} \arrow[r] & \text{TCCTTAG} \\ {[2,1,1,3]} \arrow[r] & {[1,2,1,3]} \end{tikzcd}$$

Neighborhood and the frequency distance

- Two frequency vectors are called *neighbors*, if one can be transformed to the other by a single edit operation (e.g., [2,1,1,3] and [1,2,1,3] are neighbors).
- The *frequency* distance FD_1 between two vectors is defined by the minimum number of steps in order to go from one to the other by moving to a neighbor point at a time.
- The frequency distance is a *lower bound* of the edit distance between the corresponding strings.

Computing the frequency distance

- Let u, v be two d-dimensional vectors.
- Let posDist = 0, negDist = 0
- For each dimension i
  - If $u_i>v_i$ posDist += u_i-v_i;
  - Else negDist += v_i-u_i;
- Return max(posDist, negDist)
- Assume w.l.o.g. that posDist>negDist. The rationale is that we can combine each deletion in negDist with an insertion in posDist, and the map the remainder of posDist as an insertion.

Computing the frequency distance (example)

s_1 = TACTTAG u = [2,1,1,3]

s_2 = TTAGAG v = [2,0,2,2]

posDist = sum([0,0,1,0]) = 1

negDist = sum([0,1,0,1]) = 2

$FD_1(u,v) = 2$

$ED(s_1, s_2) = 4$

Use of the frequency distance

- Suppose we want to find all substrings s in t which match query q within edit distance k. If the frequency distance $FD_1(f(s),f(q))$ is larger than k we can prune s because the edit distance is also larger than k.

Using Wavelet Transforms to enrich the frequency vector

- For biological applications, where Σ = {A,C,G,T}, a 4-dimensional vector is too small to capture in detail the contents of the strings.
- Thus we need to capture also the *local frequencies* of the characters
- This can be done by applying a *wavelet* transformation to the string.

1st/2nd Wavelet Coefficients

- The idea is to break the string s in two parts s_1, s_2 of equal length and compute two vectors;
- The first vector is f(s).
- The second f'(s) is computed by array subtraction $f(s_1)-f(s_2)$. Thus f'(s) gives the difference between the frequencies in the left substring and the frequences in the right substring.
- Then we approximate the string by a new feature vector $\psi(s) = [f(s),f'(s)]$.

Example

s = TCACTTAG, $f(s) = [2,2,1,3]$

s_1 = TCAC, s_2= TTAG

$f(s_1) = [1,2,0,1]$, $f(s_2) = [1,0,1,2]$

$f(s') = f(s_1) - f(s_2) = [0,2,-1,-1]$

$\psi(s) = [f(s), f(s')] = [[2,2,1,3],[0,2,-1,-1]]$

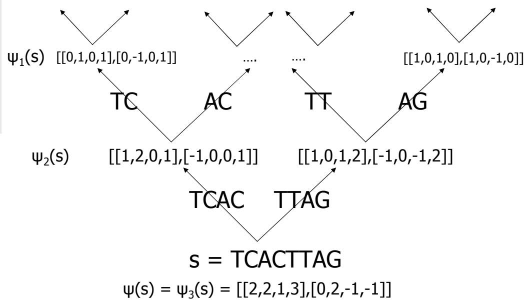
New Lower Bound

- The new edit distance lower bound between two strings x and y is then defined by
  - $FD(x,y) = \max\{FD_1(f(x),f(y)),FD_2(\psi(x),\psi(y))\}$
  - FD_1 is the previously defined frequency distance between frequency vectors
  - FD_2 is a more complex distance defined on the wavelet trasforms (details in the paper).

Generalizing the Idea

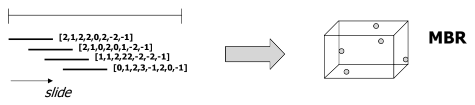
- We can generalize the idea to derive kth-level wavelet coefficients $\psi_k(s)$, longer vectors and higher accuracy.
- $\psi_k(s)$ divides the string into $|s|/2^k$ substrings and gives the Wavelet Coefficient for each substring, recursively.
- larger k provides more information, but is more expensive to evaluate.

Example for k=3


ψ1(s) [[0,1,0,1],[0,-1,0,1]] …. …. [[1,0,1,0],[1,0,-1,0]]
TC AC TT AG
ψ2(s) [[1,2,0,1],[-1,0,0,1]] [[1,0,1,2],[-1,0,-1,2]]
TCAC TTAG
s = TCACTTAG
ψ(s) = ψ3(s) = [[2,2,1,3],[0,2,-1,-1]]

The Indexing Method

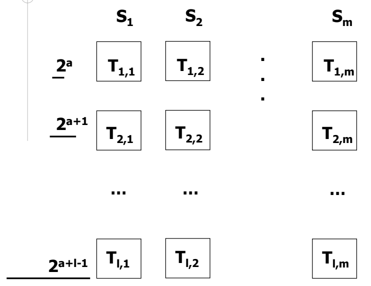
- Assume that we want to index a long sequence S.
- Let 2^a, be the minimum length of a query (a power of 2).
- We slide a window of length 2^a to each position of S and compute the first two wavelet coefficients to a vector v.
- Thus each position is indexed by a 2d-dimensional point.
- The points are then indexed in an R-tree-like index.



The Indexing Method (cont'd)

- We do the same for all window sizes up to a maximum length 2^b.
- Therefore for each long sequence S a number of MBR-lists for various lengths are created.
- If we have multiple long sequence, a matrix-like index structure is created, where each row corresponds to a resolution level and each column to an indexed sequence.

The Indexing Method (example)



Query Processing

- The query is chopped to as large pieces as possible in order to search the lowest-possible row of the index.
- Example: q=AGAGTATTTACCTG is chopped to AGAGTATT, TACC, and TG
- The pieces are used to search the corresponding index raw.
- The qualifying results are merged to get the candidate sequences and then evaluated using dynamic programming.

Comments

- This index can also be used to evaluate nearest-neighbor queries
- It performs well experimentally.
- However:
  - It relies on the quality of the MBRs in the index.
  - It relies on the effectiveness of the wavelet coefficients to approximate the structure
  - It has a high preprocessing cost (i.e., building the index is costly).

Summary

- There are many methods proposed for approximate string matching. Most are based on the generic edit distance function.
- Some improve the speed of the dynamic programming method.
- Some extend exact string search methods (like suffix trees) for approximate search
- Some apply filtering techniques that avoid expensive comparisons in large parts of the queried sequence.
- Some apply filtering after transforming the data to an approximate space appropriate for conventional search techniques.

Summary (cont'd)

- Although there have been many efforts for efficient approximate string matching, there is still room for improvement.
- This is because of the complex distance functions which make it hard to use traditional search techniques.
- Biologist are still not (and will never be!) satisfied by computational techniques, since they expect them to be general enough to solve every special search problem they encounter.

References

- G. Navarro, A Guided Tour to Approximate String Matching, ACM Computing Surveys, 33(1): 31-88, March 2001.
- Ela Hunt, Malcolm P. Atkinson, Robert W. Irving: A Database Index to Large Biological Sequences. VLDB 2001.
- Tamer Kahveci, Ambuj K. Singh: Efficient Index Structures for String Databases, VLDB 2001.
- HKU CSIS7402 (Computer Technology for Bioinformatics) Lecture Notes, http://i.csis.hku.hk/~c7402/information.php
- **Special thanks** to *Lok Lam Cheng*