Algorithmic Embeddings for Comparing Large Text Streams

Graham Cormode
Shanmugavelayutham Muthukrishnan

Strings

Strings of text are ubiquitous:

- Short: SMS (wireless text messages), email
- Huge: autogenerated reports, biological sequences

Modeled as a one dimensional sequence, S

Each S[i] $\in \Sigma$ for some (finite) alphabet Σ

We may be dealing with many, huge strings

```
To be or not to be, that is the question.  Whether tis nobler in the mind to suffer...
AGCTAGACAATGACACATAACAAGATACACGTTATATGACACAGTACGAGTCACGTAGCAATGGACATGAGAGAT…
WE ARE TOP OFFICIALS OF THE FEDERAL GOVERNMENT OF NIGERIA CONTRACT REVIEW PANEL WHO ARE…
The truth is, I don’t think we should see each other any more.  It’s not you, it’s me…
```

Problems on Strings

- Finding similar strings
- Clustering into groups of similar strings
- Measuring how similar two strings are
- Finding most similar string in the database to a new string (Approximate Nearest Neighbors)
- Other similarity problems

What is similar?

- Identical or different: too simplistic
- Hamming distance: number of different characters
- L_1 distance: treat strings like vectors, measure difference between corresponding entries
- **Edit distances**: insert/delete/change/move

```
Call me Ishmael. Some years ago - never mind how long precisely -
Call me Dwight. Some years ago - never mind how long precisely -
```

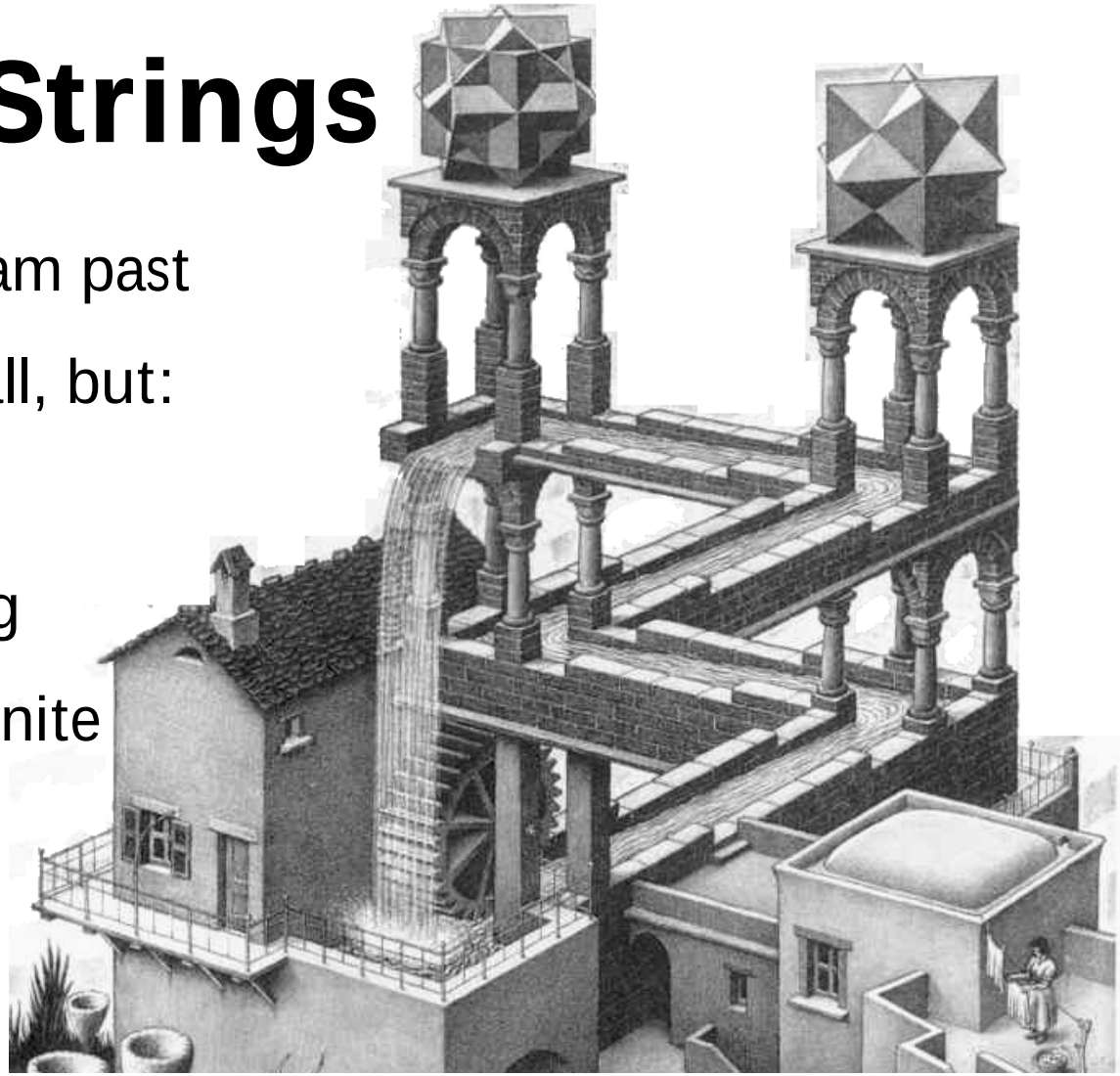
Streaming Strings

Many large strings stream past

We can observe them all, but:

- Our memory is finite
- Can’t store everything
- Processing power is finite

Strings go past character by character: always in order, but arbitrarily interleaved



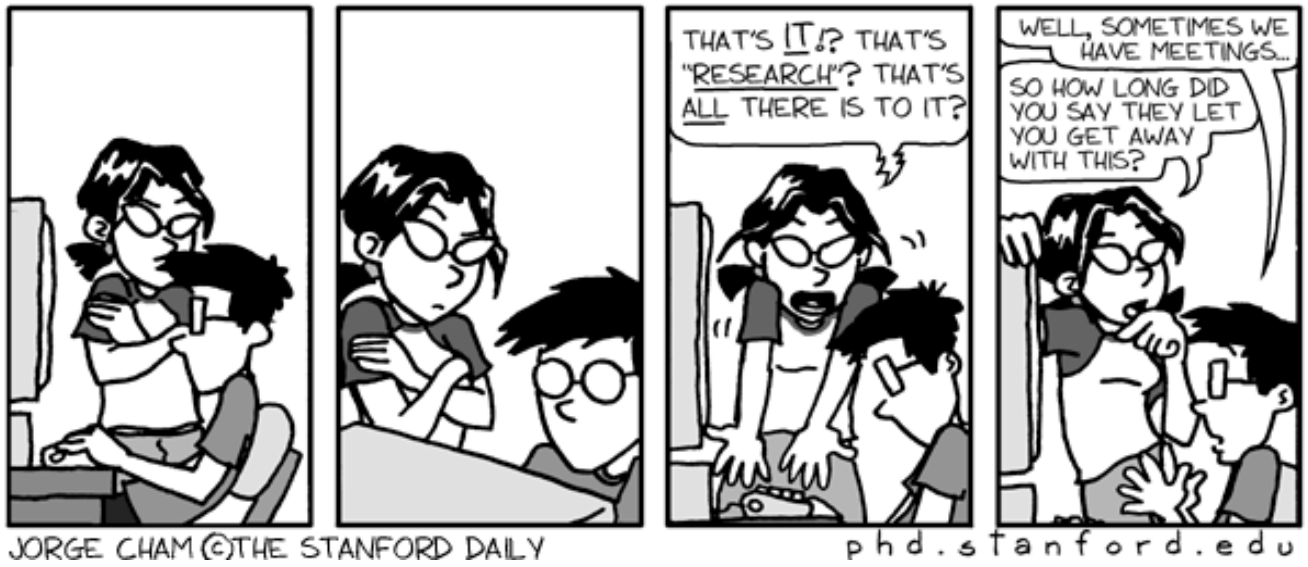
Escher Listening and Observing Network (EscherLON)

Edit Distance with Moves

d(A,B) = smallest no. of editing operations to turn A into B

- insert or delete a character
- move a substring

Approach: transform the edit distance into L_1 distance, then use L_1 distance algorithms to solve the problems



Special Case of Edit Distance

Return to main problem of Edit Distance with moves

We first consider a special case, where each character is unique (only appears once in any string)

Approximation Scheme

Each string is some ordering of a subset of the alphabet

Assume the alphabet is 1… n

We give a simple constant factor approximation:

- Define a breakpoint as a pair (i,j) adjacent in one string but not in the other
- The number of breakpoints is a 3-approximation of the distance

Approximation Scheme

We start with some number of breakpoints, we end with none, so we aim to remove breakpoints

- An insert or delete removes one or two breakpoints
- A move can remove at most three breakpoints
- Can remove at least one breakpoint per move

B: $0\ B_1 \ldots B_i\ B_{i+1} \ldots \ldots \ldots B_n\ n+1$

A: $0\ B_1 \ldots B_i\ A_j \ldots B_{i+1} \ldots A_n\ n+1$

Streaming approximation

Suppose the sequence is streaming past:

$$\ldots A_i, A_{i+1}, A_{i+2}, \ldots$$

Represent this as (**a**, $[A_i, A_{i+1}]$, 1), (**a**, $[A_{i+1}, A_{i+2}]$, 1)…

And feed this into the algorithm for L_1 distance

Then, L_1 distance between these (binary) vectors is exactly twice the number of breakpoints

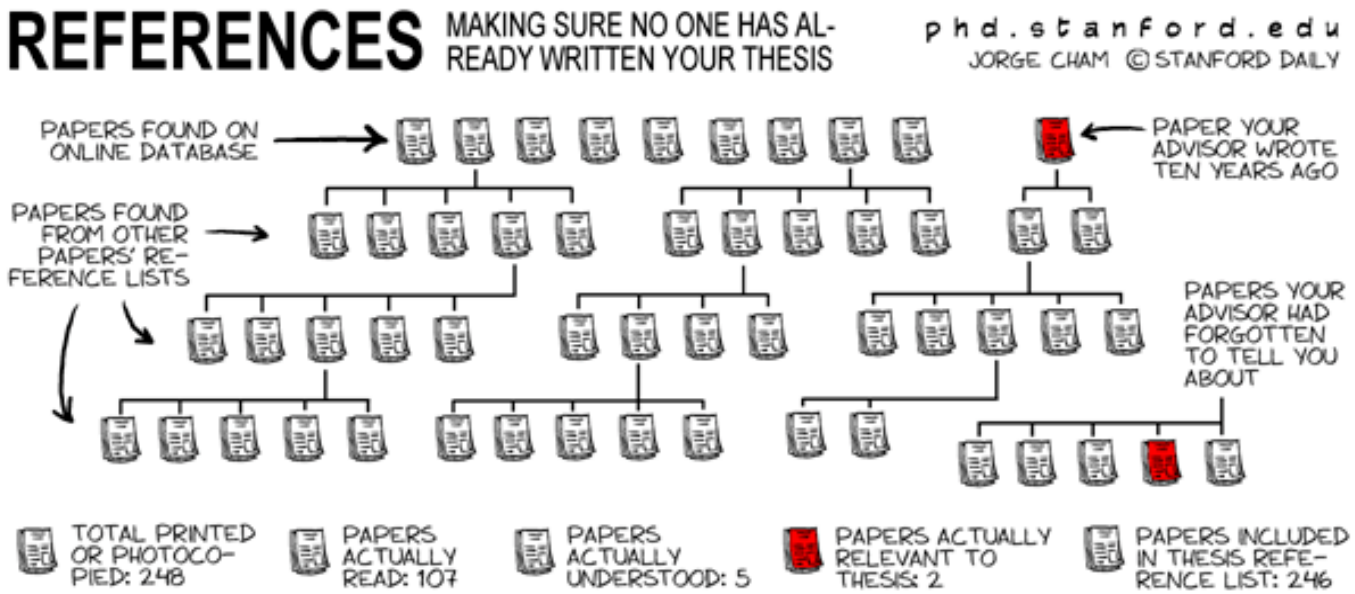
This embedding of an edit distance into L_1 space gives a 3-approximation of the edit distance with moves

General Texts

For more general strings, things get more difficult

Use the same idea: use local features to build a vector repn.

History: ideas in C-Paterson-Sahinalp-Vishkin'00, developed in Muthukrishnan-Sahinalp'00 and C-Muthukrishnan'02



q-grams

Embedding ideas have been used in strings for a while… not always deliberately!

q-grams: A *q*-gram is just a substring of length *q*

The *q*-gram representation of a string A is the histogram of *q*-grams of that string. Call this $F_q(A)$.

We can then look at $||F_q(A) - F_q(B)||_1$ as a measure of string distance (Ukkonen 92).

But : $F_q(A) = F_q(B)$ does not mean $A = B$, so not a metric

Still a good heuristic, often used in database applications.

Other Failed Ideas

We want to take same approach to strings as permutations

Look for Combinatorial features that capture edit distances

Frequency of fixed length substrings: *q*-grams don't work.

Try a binary tree structure: but a single character insert changes the substring set completely

Try **all** substrings of length 2^i: edits still have too much effect on the set

So we will need something more sophisticated

Same idea, different substrings...

Now describe a method that uses the same underlying idea: represent a string by a histogram of substrings so that L_1 difference of histograms approximates an editing distance.

Difference is that we obtain a guaranteed distortion embedding, poly-log in max length of string.

The embedding is fairly efficient to compute, based on parsing derived from deterministic coin tossing

Same ideas used in string matching by Sahinalp Vishkin 96, Mehlhorn Sundar Uhrig 97, Alstrup Brodal Rauhe 00.

Overview of Structure

We will build a 2-3 tree on the string

Each node corresponds to a substring that we will store in a histogram

Iterative procedure: parse the string into pairs and triples to make the nodes at the next level, then repeat on shorter string

Parsing has several parts:

- isolate simple patterns
- alphabet reduction on remainder
- mark certain features
- use these to divide into pairs and triples

Parsing for the Embedding

Embedding is based on parsing strings in a deterministic way

We parse the strings in a way so that edit operations have only a limited effect on the parsing — this will allow us to make the approximation.

Find ‘landmarks’ in the string based only on their locality.

- Repetitions (aaa) are easily identifiable landmarks
- Local maxima are good landmarks in varying sequences, but may be far apart — so reduce the alphabet to ensure landmarks occur often enough.

So: Isolate repetitions, leave substrings with no repeats.

Alphabet Reduction

Write each character as a bitstring ie a = 00000, b = 00001

Reduce the alphabet. For each character, find a new label as:

Smallest bit location where it differs from its left neighbor + Bit value there

e.g.

Char	b	d	a
Binary	00001	00011	00000
Location	-	001	000
Label	-	001**1**	000**0**

Alphabet Reduction

If starting alphabet is Σ, new alphabet has $2 \log |\Sigma|$ values

Repeat the procedure on the string iteratively until the alphabet is size 6, $\Sigma' = \{0,1,2,3,4,5\}$

Then reduce from 6 to 3, ensuring no adjacent pair are identical (first remove all 5s, then all 4s, then all 3s)

Properties of the final labels:

- Final alphabet is {0,1,2}
- No adjacent pair is identical
- Takes $\log^* |\Sigma|$ iterations
- Each label depends on $O(\log^* |\Sigma|)$ characters to left

Marking characters

Consider the final labels, and mark certain characters:

- Mark labels that are local maxima (greater than left & right)
- Also mark any local minima not adjacent to a marked char

Clearly, no two adjacent characters are marked.
Also, marked labels are separated by at most two labels

Text	c	a	b	a	g	e	f	a	c	e	d
Labels	-	010	001	000	011	010	001	000	011	010	011
Final	-	2	1	0	~~3~~	1	2	1	0	~~3~~	1	2	~~3~~	0

Group into pairs and triples

Now, whole string can be arranged into pairs and triples:

- For repeats, parse in a regular way

$$aaaaaaa \rightarrow (aaa)(aa)(aa)$$

- For varying substrings, use alphabet reduction, define pairs and triples based on the marked characters.

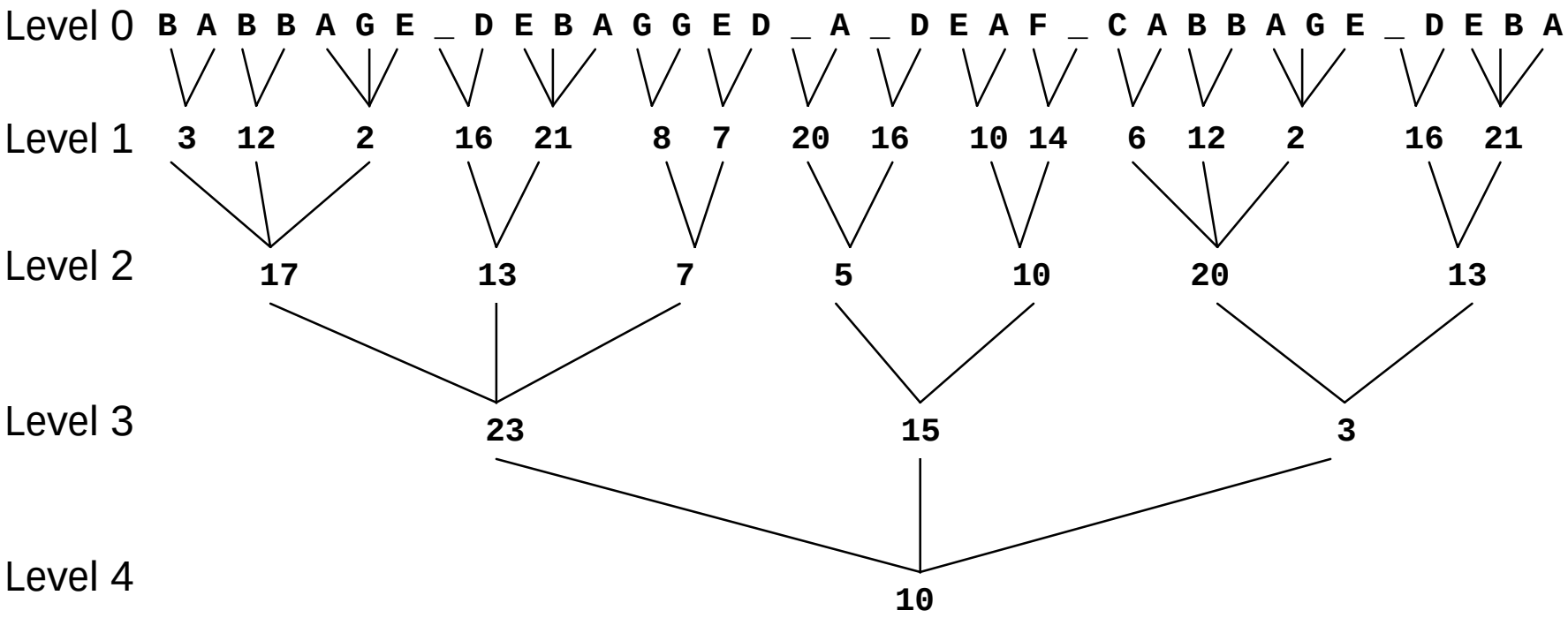
Text	c	a	b	a	g	e	f	a	c	e	d
Final	-	2	1	0	1	2	1	0	1	2	0

Parsing of each character depends on log*n + c neighborhood

Relabel each pair or triple in a consistent way, so same characters get the same new name

Build Hierarchical Structure

Given new labels, repeat the process… this builds a 2-3 tree



Can be constructed in time $O(n \log^* n)$

Vector Representation

From the structure, derive vector representation V recording occurrence frequency of each (level, label) pair:

(0,a)	(0,b)	(0,c)	(0,d)	(0,e)	(0,f)	(0,g)	(0,_)
8	7	1	4	6	1	4	5

(1,2)	(1,3)	(1,6)	(1,7)	(1,8)	(1,10)	(1,12)	(1,14)	(1,16)	(1,20)	(1,21)
2	1	1	1	1	1	2	1	3	1	2

(2,5)	(2,7)	(2,10)	(2,13)	(2,17)	(2,20)	(3,3)	(3,15)	(3,23)	(4,10)
1	1	1	2	1	1	1	1	1	1

Theorem: $\frac{1}{2}d(A,B) \leq \| V(A) - V(B) \|_1 \leq O(\log n \log^* n)\, d(A,B)$

Upper bound

$$\|V(A) - V(B)\|_1 \leq O(\log n \log^* n)\, d(A,B)$$

Consider the effect of each permitted edit operation:

- Insert or delete a character:
  Fairly straightforward, at most $\log^* n$ nodes can change per level

- Move a substring:
  Within the substring, there are no changes.
  At fringes, only $O(\log^* n)$ nodes change per level

As each operation changes V by $O(\log n \log^* n)$, so $\|V(A) - V(B)\|_1 / O(\log n \log^* n) \leq d(A,B)$

Hence the bound holds.

Lower bound

A constructive proof: we give an algorithm to transform A into B using at most $2\|V(A) - V(B)\|_1$ operations.

Be sure to keep hold of large pieces of the string that are common to both, so 'protect' enough pieces of A that are needed in B, and avoid changing these.

Then we will go through level by level to turn A into B:

- At the bottom, add or remove characters as needed.
- For each subsequent level, proceed inductively:

  Assume we have enough nodes of the level below.
  Then to make any node only need to move at most 2 nodes from the level below. ❑

Computation on the Stream

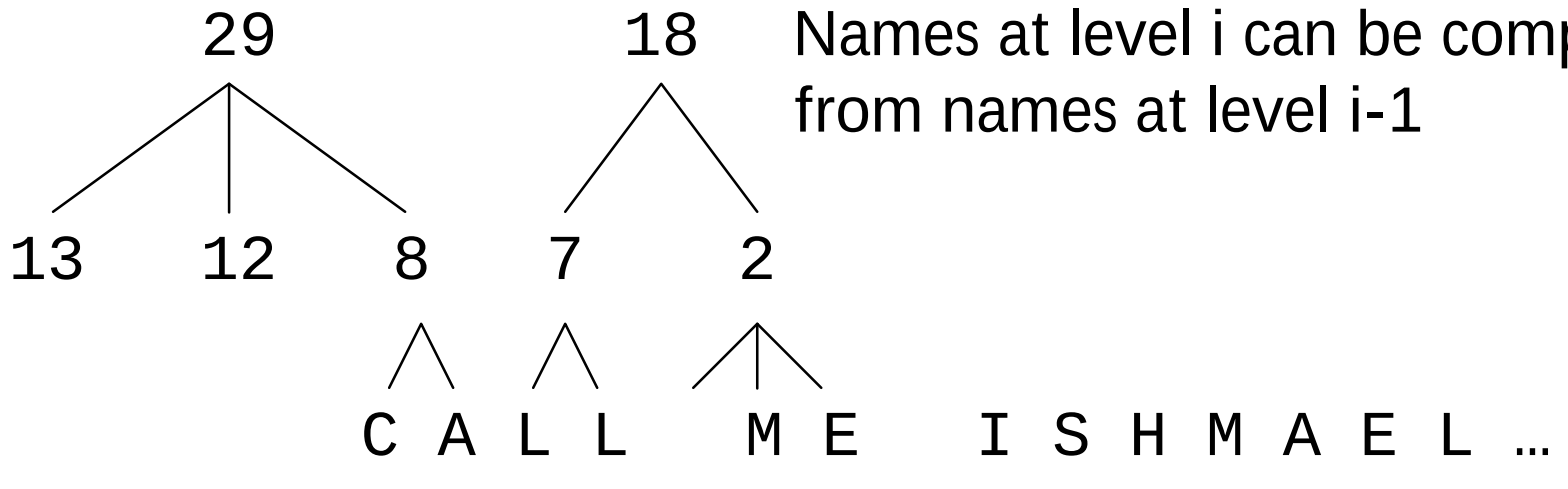
We can’t keep whole string in memory to make the parsing

But, the parsing of any section relies only on a local neighborhood, this is true at every level

So only keep in memory the O(log* n) nodes at each level needed to make the parsing

Use Karp-Rabin hash functions to name substrings. These can be combined to form the names of nodes.

Example of Stream Computation



Names at level i can be computed from names at level i-1

Parsed characters do not need to be stored

Feed into L_1 algorithm

Need log*n characters in memory to parse new characters

L_1 Distance Problem

A hard problem itself. Want to estimate the L_1 distance between two vectors presented as a stream.

Solution from Indyk'00, implemented CIKM'00

- Massive vectors **a**, **b** stream past us, not in order
- Initially, set **a**, **b** = **0**
- Receive updates (a,i,c) so **a**[i] $\leftarrow$ **a**[i] + c
- Approximate $\|\mathbf{a} - \mathbf{b}\|_1$ in small memory use

L_1 Distance Solution

Relies on using random distributions with certain properties:

$X, X_1, X_2, \ldots X_n$ are random variables with Cauchy dbn

$a_1X_1 + a_2X_2 + \ldots + a_nX_n \sim (|a_1| + |a_2| + \ldots |a_n|)\, X$

This is $\|\mathbf{a}\|_1\, X$, the L_1 norm of $\mathbf{a}$

So if we maintain $z(\mathbf{a}) = \mathbf{a} \cdot \mathbf{x}$ where each entry of $\mathbf{x}$ is drawn from Cauchy distribution, then

$z(\mathbf{a}) - z(\mathbf{b}) = z(\mathbf{a} - \mathbf{b}) \sim \|\mathbf{a} - \mathbf{b}\|_1\, X$

Streaming L_1 distance

For accuracy, we maintain a **vector z** of dot products

$$\mathbf{z}(\mathbf{a})[i] = \mathbf{a} \cdot \mathbf{x_i}$$

for different vectors $\mathbf{x_i}$, drawn from Cauchy distribution

Then approximate $\|\mathbf{a} - \mathbf{b}\|_1$ as median$|\mathbf{z}(\mathbf{a}) - \mathbf{z}(\mathbf{b})|$.

On each new update (a,i,c) we just update **z**(**a**) —

For each j: $\mathbf{z}(\mathbf{a})[j] \leftarrow \mathbf{z}(\mathbf{a})[j] + c\ \mathbf{x_j}[i]$

Fine Details

How big does **z** need to be?

If we pick **z** with $O(1/\varepsilon^2 \log 1/\delta)$ entries, this is a $1 \pm \varepsilon$ approximation with probability $1 - \delta$

Space requirements:

Storing **x** takes lots of space, more than keeping **a**

So, don't store **x**, but generate $\mathbf{x}_i[j]$ when needed, using pseudo random generators.

This is "good enough" to get good results.

Application to other problems

At the start, promised “clustering, nearest neighbors” etc.
In one slide:

- We can use this embedding into L_1 distance, then apply algorithms for clustering, nearest neighbors etc. in L_1 to the transformed problem, getting an $O(\log n \log^* n)$ approx
- Some care needed! The vectors have $O(|\Sigma|^n)$ entries, but only $O(n)$ entries are non-zero
- Slightly improved results possible when distances are large: $O(\log n/d \log^* n)$ approximation.

Extensions

O(log n log* n) is large, can it be improved?

Other edit distances: copies can be accommodated

Other applications of these parsing ideas?