String Edit Distance Matching Problem with Moves

Graham Cormode, S. Muthukrishnan
grahamc@dcs.warwick.ac.uk
muthu@research.att.com

Pattern Matching

We want to find good matches of P in T as measured by d(-,-) where d is some string edit distance.

General setting: for each i, find

$$D[i] = \min_j d(T[i:j],P)$$

Pattern Matching Problems

Hamming distance in time

$O(nm^{1/2})$	Abrahamson 87
$O(1/\varepsilon^2\, n \log^3 n)$	Karloff 93 ($1 + \varepsilon$ approx)
$O(1/\varepsilon^2\, n \log n)$	Indyk 98 ($1 + \varepsilon$ approx)

Edit distance in time

$O(nm)$	Dynamic Programing

Other solutions parametrized by k (largest distance) still have $O(nm)$ worst case perfomance in general

We want $o(nm)$ time solutions, ideally close to $O(n)$.

Our results

We make a simplification, and allow approximations of each D[i]

We will study the string edit distance *with moves*:

d(X,Y)= smallest number of following operations to turn X into Y

- insert a character
- delete a character
- replace a character
- move a substring

Substring moves are relevant to many situations, eg Computational Biology, Text Editing, Web Page updates etc.

We will find each D[i] up to a factor of O(log n log*n)

Main Features

- Embed the string distance into the L_1 vector distance, up to a $O(\log n \log^* n)$ factor
- Compute this vector embedding quickly with a single pass over the string
- Quickly find the representation for any substring of T
- Only need to consider $O(n)$ substrings
- Solve the whole problem approximately but deterministically in time $O(n \log n)$

Parsing for the Embedding

The 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.

Procedure: Isolate repetitions, leaving 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 the starting alphabet is Σ, the 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 the $O(\log^* |\Sigma|)$ characters to its left

Marking characters

Consider the final labels, and mark certain characters:
- Mark any labels that are local maxima (greater than left & right)
- Also mark any local minima if not adjacent to a marked char.

Clearly, no two adjacent characters are marked.
Also, successive 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	-	$\underline{2}$	1	$\underline{0}$	~~3~~ 1	$\underline{2}$	1	$\underline{0}$	~~3~~ 1	$\underline{2}$	~~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 => (aaa)(aa)(aa)

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

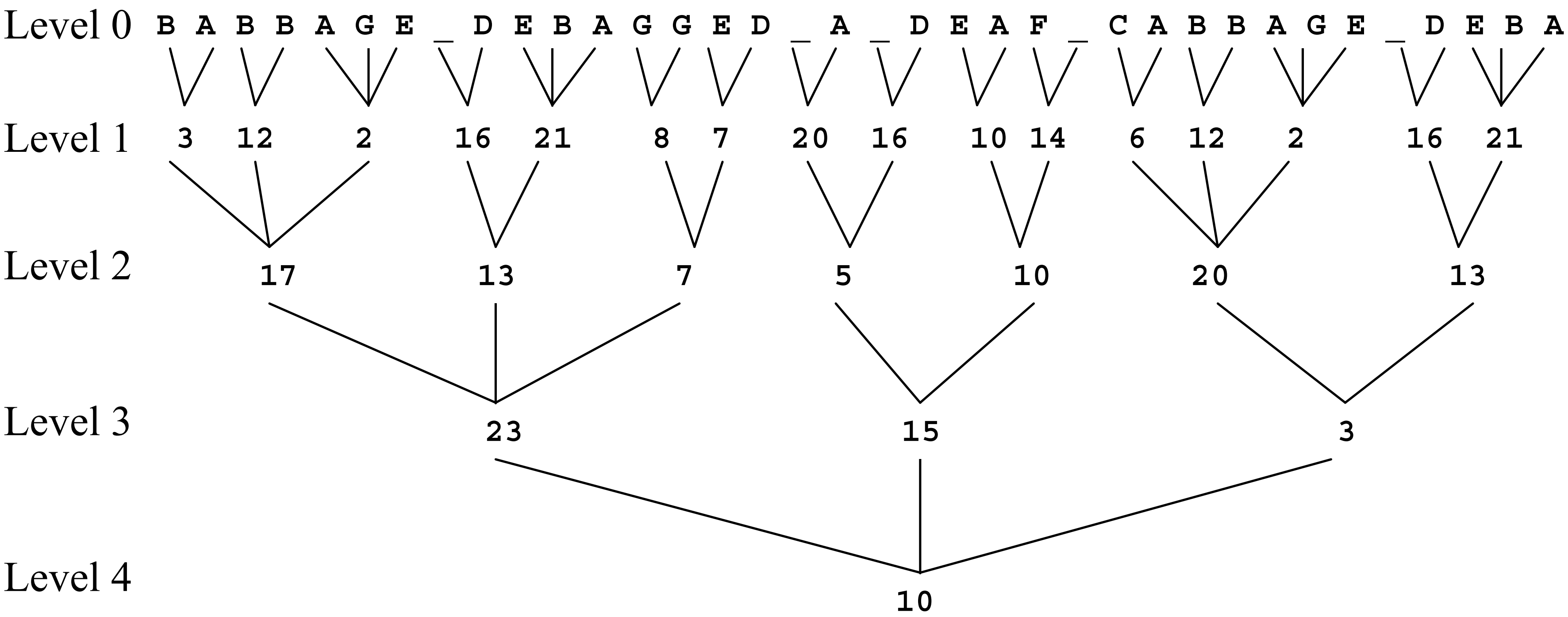
Text	c	a	b	a	g	e	f	a	c	e	d
Final	-	$\underline{2}$	1	$\underline{0}$	1	$\underline{2}$	1	$\underline{0}$	1	$\underline{2}$	0

Relabel each pair or triple — can do this deterministically, building a dictionary of labels using Karp-Miller-Rosenberg labelling.

The parsing of each character depends on a log*n + c neighborhood

Build Hierarchical Structure

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



Can be constructed in time O(*n* log**n*)

Vector Representation

From this structure, derive a vector representation V recording the frequency of occurrence 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: $\frac{1}{2}d(X,Y) \leq \| V(X) - V(Y) \|_1 \leq O(\log n \log^* n)\, d(X,Y)$

Upper bound

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

Consider the effect of each permitted edit operation:

- Insert / change / 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 the fringes, only $O(\log^* n)$ nodes change per level

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

Hence the bound holds.

Lower bound

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

We want to make sure we keep hold of large pieces of the string that are common to both X and Y, so we will go through and protect enough pieces of X that will be needed in Y, and we avoid changing these in the manipulation.

Then we will go through level by level to turn X into Y:

- At the bottom, we 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 we only need to move at most 2 nodes from the level below. ❑

Application to String Matching

To find $D[i]$, we need to compare every substring of T against P — this is $O(n^2)$. We reduce this to $O(n)$ substrings.

$$d(T[l:l+m-1],P) \le d(T[l:l+m-1],T[l:r]) + d(T[l:r],P)$$

by triangle inequality

$$= |(r - l + 1) - m| + d(T[l:r],P)$$

$|(r - l + 1) - m| \le d(T[l:r], P)$ since we need at least $|(r-l+1) - m|$ operations to make $T[l:r]$ the same length as P. So

$$d(T[l:l+m-1],P) \le 2d(T[l:r],P)$$

So we only need to consider the $O(n)$ substrings of length m and this will be a 2-approximation of the optimal matching.

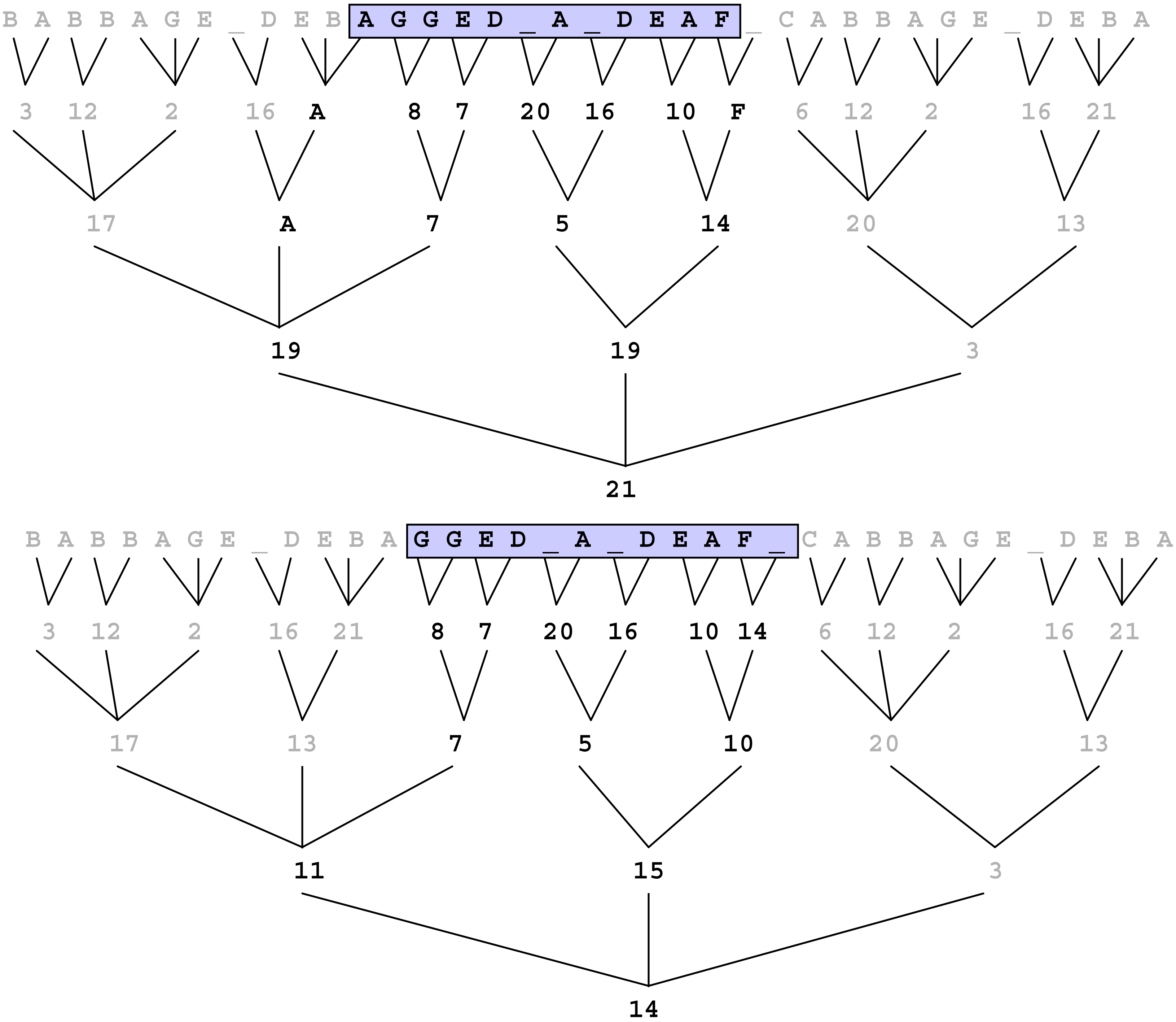
Final algorithm

By construction, a subtree of an ESP tree induced by any substring has the same properties: the L_1 distance of the vector embedding approximates the edit distance with moves.

String matching algorithm:

- Create a naming function for T and P using
  Karp-Miller-Rosenberg Labelling.
- Compute parse trees for T and P
- Find $\|V(T[1{:}m]) - V(P)\|_1$
- Iteratively compute $D[i] \approx \|V(T[i{:}i+m-1]) - V(P)\|_1$

Overall cost is $O(n \log n)$ for the whole algorithm.



B A B B A G E _ D E B A G G E D _ A _ D E A F _ C A B B A G E _ D E B A
3 12 2 16 A 8 7 20 16 10 F 6 12 2 16 21
17 A 7 5 14 20 13
19 19 3
21
B A B B A G E _ D E B A G G E D _ A _ D E A F _ C A B B A G E _ D E B A
3 12 2 16 21 8 7 20 16 10 14 6 12 2 16 21
17 13 7 5 10 20 13
11 15 3
14

Conclusion

Advantages of this embedding approach:

- General: applicable to many other problems
  eg Approximate Nearest Neighbor, Clustering
- Easy to compute, can be made probabilistically in
  the streaming model

Disadvantages of this solution:

- Large approximation factor
- Does not obviously extend to Levenshtein edit distance

Open problems: remedy these disadvantages!