More on Reconstructing from Random Traces: Insertions and Deletions

Sampath Kannan and Andrew McGregor, UPenn

Random Traces

- Transmit a length n binary string t
- Channel introduces errors:
  - Delete a bit with probability q_1
  - Insert a bit with probability q_2
  - Flip a bit with probability p
- Transmit m times to generate m independent received strings $r_1, r_2, ..., r_m$

Previous Work

- Levenshtein '01:

  Combinatorial Channels - eg. how many distinct subsequences are required to uniquely determine *t*?

  Probabilistic Channels - only treatment of memoryless channels

- Dudik & Shulman '03:

  Combinatorial Channels - how large must *k* be such that knowing all length *k* subsequences (and their multiplicities) is sufficient to deduce *k*?

- Batu, Kannan, Khanna & McGregor '04:

  Deletions only...

Our Results

	p	q_1	q_2	m	*Comments*
Previous Work	0	0	$O(\log^{-1} n)$	$O(\log n)$	Almost all strings
	0	0	$O(n^{-1/2-\epsilon})$	$O(1/\epsilon)$	Long runs approximated
This Work	$O(1)$	$O(\log^{-2} n)$	$O(\log^{-2} n)$	$O(\log n)$	Almost all strings
	0	$O(n^{-1/2-\epsilon})$	$O(n^{-1/2-\epsilon})$	$O(1/\epsilon)$	No long runs and long alternating sequences approximated

Defn:

A run: ...1111111... or ...00000000...

An alternating sequence: ...01010101010...

A substring is long if its length is greater than n^ϵ

The “Bit-Wise Majority” Algorithm

The "Bit-wise Alignment" Algorithm

- Frugally insert blanks to align the strings

r_1: `11101011101001011110...`

r_2: `11010010101101001 01...`

r_3: `11010000100101011110...`

r_4: `10100001011101011110...`

r_5: `11000000010110101110...`

r_m: `11000000101100101110...`

The "Bit-wise Alignment" Algorithm

- Frugally insert blanks to align the strings

r_1:	11101011101001011110...
r_2:	1101001010110100101...
r_3:	1101000010010101110...
r_4:	1010000101110101110...
r_5:	1100000001011010110...
r_m:	1100000010110010110...
t:	1

The "Bit-wise Alignment" Algorithm

- Frugally insert blanks to align the strings

r_1:	11101011101001011 10...
r_2:	11010010101101001 01...
r_3:	11010000100101011 10...
r_4:	1*0100001011101011 10...
r_5:	11000000010110101 10...
r_m:	11000000101100101 10...
t:	11

The “Bit-wise Alignment” Algorithm

- Frugally insert blanks to align the strings

r_1:	11*10101110100101110...
r_2:	1101001010110100101...
r_3:	1101000010010101110...
r_4:	1*010000101110101110...
r_5:	1100000001011010110...
r_m:	1100000010110010110...
t:	110

The "Bit-wise Alignment" Algorithm

- Frugally insert blanks to align the strings

r_1:	11*10101110100101110...
r_2:	110100101011010010 1...
r_3:	1101000010010101110...
r_4:	1*010000101110101110...
r_5:	110*0000001011010110...
r_m:	110*0000010110010110...
t:	1101

The "Bit-wise Alignment" Algorithm

- Frugally insert blanks to align the strings

r_1:	`11*101011101001011 10...`
r_2:	`11010010101101001 01...`
r_3:	`11010000100101011 10...`
r_4:	`1*010000101110101 110...`
r_5:	`110*0000001011010 110...`
r_m:	`110*0000010110010 110...`
t:	`11010`

The "Bit-wise Alignment" Algorithm

- Frugally insert blanks to align the strings

r_1:	`11*10*10111010010111 0...`
r_2:	`11010010101101001 01...`
r_3:	`1101000010010101110...`
r_4:	`1*0100001011101011 10...`
r_5:	`110*0000001011010110...`
r_m:	`110*0000010110010110...`
t:	`110100`

The “Bit-wise Alignment” Algorithm

- Frugally insert blanks to align the strings

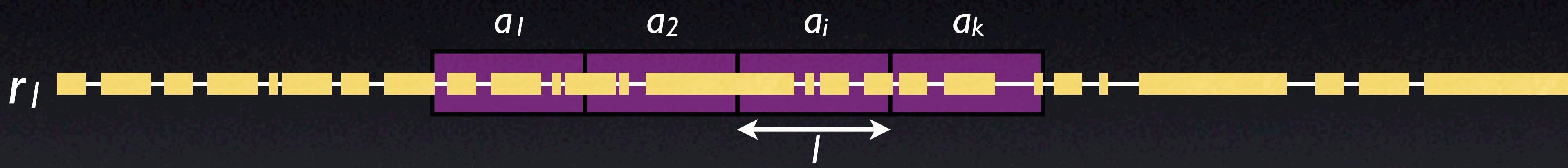
```
r1:    11*10*101110100101110...
r2:    1101001010110100101...
r3:    1101000010010101110...
r4:    1*0100001011101011 10...
r5:    110*00000010110100110...
rm:    110*00000101100 10110...
       ---------------------------
t:     110100...
```

- Analysis for a randomly chosen t: alignment of r_i with t can be modeled using random walk

The “Velcro” Algorithm

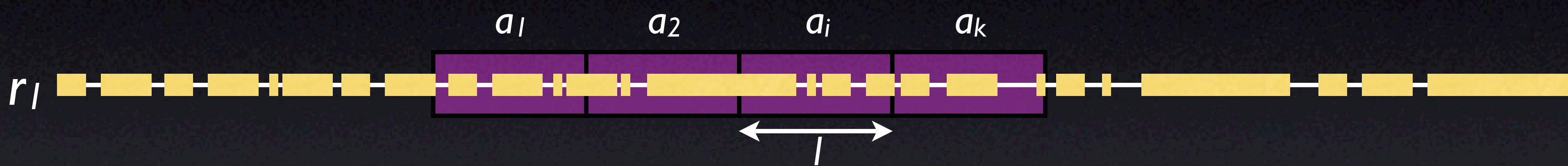
The "Velcro" Algorithm

- Consider the middle kl bits of r_l: k possible length l anchors



The “Velcro” Algorithm

- Consider the middle *kl* bits of r_1: *k* possible length *l* anchors



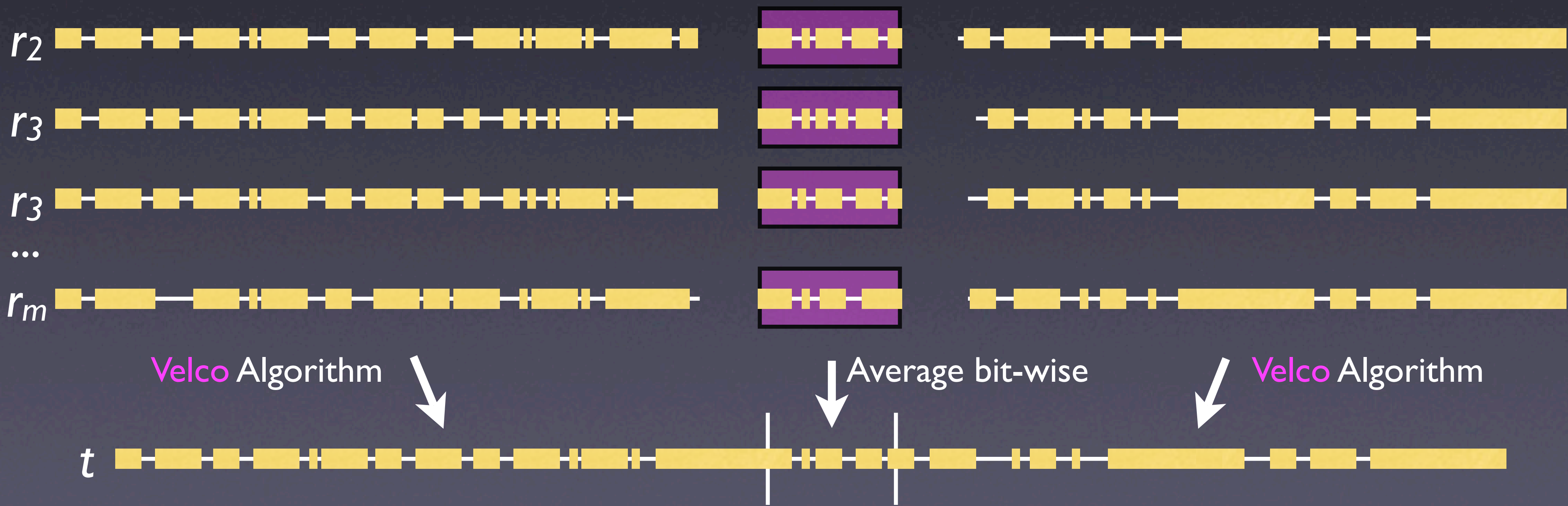
- For each a_i, find the “best” match in other received strings

The “Velcro” Algorithm

- Consider the middle kl bits of r_1: k possible length l anchors
- For each a_i, find the “best” match in other received strings
- If a_i has a “good” match in all received strings, recurse on the strings either side of each match

The "Velcro" Algorithm

- Consider the middle kl bits of r_1: k possible length l anchors
- For each a_i, find the "best" match in other received strings
- If a_i has a "good" match in all received strings, recurse on the strings either side of each match



Analysis

- Defn: Match is good if Hamming distance is less than $(p - p^2 + 1/4)l$
- Lemma:

  a) One of *k* anchors has a good match with all received strings with probability at least

$$1 - \left(mql + m \left[\frac{e^{\delta}}{(1+\delta)^{1+\delta}} \right]^{(2p-2p^2)l} \right)^k$$

  b) If a_i has a good match with all received strings then "splitting-off" at a_i is legitimate with probability as least

$$1 - kne^{-l(1/2-2p+2p^2)/4}$$

Analysis

- Defn: Match is good if Hamming distance is less than $(p - p^2 + 1/4)l$
- Lemma:

a) One of k anchors has a good match with all received strings with probability at least

$$1 - \left(mql + m \left[\frac{e^{\delta}}{(1+\delta)^{1+\delta}} \right]^{(2p-2p^2)l} \right)^k \longrightarrow \; > 1 - 1/n^2$$

b) If a_i has a good match with all received strings then "splitting-off" at a_i is legitimate with probability as least

$$1 - kne^{-l(1/2-2p+2p^2)/4} \longrightarrow \; > 1 - 1/n^2$$

Set $m = O(\log n)$, $l = O(\log n)$, $k = O(\log n)$ and $q = O(1/\log^2 n)$

The "Simple but Incredibly Tedious to Analyze" Algorithm

The “Simple but...” Algorithm

Promises, promises...

- Deletion and insertion probabilities are $q = O(n^{-1/2-\epsilon})$ and zero flip probability
- Lemma (Promises): With high probability, if $m = O(1)$

  (P1): In each transmission, the first bit of t was transmitted without error

  (P2): Among all transmissions, at most one error occurred in the transmission of any four consecutive runs

  (P3): For all alternating sequence of length $l > \sqrt{n}$, if an error occurs at the start of the alternating sequence (in any transmission) then, in all transmissions, there are no errors during the transmission of the final $\log n \sqrt{l}$ bits of the maximal alternating sequence and the next two bits of the delimiting run

  (P4): For all alternating sequence, if an error occurs at the start of the alternating sequence (in any of the m transmissions) then in all the m transmissions, there are no errors during the transmission of the final n^ϵ (or the rest of the alternating sequence if the length of the alternating sequence is less than n^ϵ) bits of the maximal alternating sequence and the next two bits of the delimiting run

  (P5): For each length $\sqrt{n}$ substring x of t, in the majority of transmissions, x is transmitted without errors

  (P6): For each substring x of t of length $> n^\epsilon$, in each transmission, there are fewer than $q\,|x| \log n$ errors in the transmission of x

The “Simple but...” Algorithm

Promises, promises...

- Given the promises we can usually locally correct the errors:

r_1: `11101100...`
r_2: `11101100...`
r_3: `11111000...`
r_4: `11101100...`
r_5: `11101100...`
r_m: `11101100...`

The "Simple but..." Algorithm

Promises, promises...

- Given the promises we can usually locally correct the errors:

r_1:	`11101100...`
r_2:	`11101100...`
r_3:	`111*11000...`
r_4:	`11101100...`
r_5:	`11101100...`
r_m:	`11101100...`

The "Simple but..." Algorithm

Promises, promises...

- Given the promises we can usually locally correct the errors:

r_1: `11101100...`
r_2: `11101100...`
r_3: `111*11000...`
r_4: `11101100...`
r_5: `11101100...`
r_m: `11101100...`

- But not always:

r_1: `10101010101...`
r_2: `10101010101...`
r_3: `11010101010...`
r_4: `10101010101...`
r_5: `10101010101...`
r_m: `10101010101...`

The "Simple but..." Algorithm

Promises, promises...

- Given the promises we can usually locally correct the errors:

r_1:	11101100...
r_2:	11101100...
r_3:	111*11000...
r_4:	11101100...
r_5:	11101100...
r_m:	11101100...

"Delimitating" Run

- But not always:

r_1:	1010101010 1...	...1010101011 01
r_2:	1010101010 1...	...1010101011 01
r_3:	1101010101 0...	...1101010101 10
r_4:	1010101010 1...	...1010101101 01
r_5:	1010101010 1...	...1010101011 01
r_m:	1010101010 1...	...1010101011 01

Conclusions & Further Work

	p	q_1	q_2	m	*Comments*
Previous Work	0	0	$O(\log^{-1} n)$	$O(\log n)$	Almost all strings
	0	0	$O(n^{-1/2-\epsilon})$	$O(1/\epsilon)$	Long runs approximated
This Work	$O(1)$	$O(\log^{-2} n)$	$O(\log^{-2} n)$	$O(\log n)$	Almost all strings
	0	$O(n^{-1/2-\epsilon})$	$O(n^{-1/2-\epsilon})$	$O(1/\epsilon)$	No long runs and long alternating sequences approximated

- What about constant insert/delete probabilities?

- Thanks.

The "Simple but..." Algorithm

Using the Promises

- Look at length of first run in each received string (wlog it's a run of 1's)
- Lemma (Tedious Case Analysis): Let y be the average length of this run and x^i be the length of the run in received string i
  - $x^i = y$: No errors have occurred in the i th transmission of this run
  - $x^i = y + 1$: Either one "1" was inserted in the ith transmission of this run or that, on the condition that the next two runs are of length one, one "0" was deleted from next .
  - $x^i > y + 1$: One "0" was deleted in the ith transmission of the next run.
  - $x^i = y - 1$: Either one "1" was deleted in the ith transmission of this run or that one "0" was inserted before the last bit of this run was transmitted.
  - $x^i < y - 1$: One "0" was inserted into this run.