Searching for Jumbled Patterns in Strings

Ferdinando Cicalese Gabriele Fici Zsuzsanna Lipták

University of Salerno (Italy)

Bielefeld University (Germany)

PSC'09, September 1, 2009

The problem: Jumbled pattern matching

the text: string s over alphabet $\{a, b, c\}$ (length n)
the pattern: $(3, 1, 2)$ - any permutation of string *aaabcc* (length m)

b b a c a c c a b a b b a b c c a a a c

The problem: Jumbled pattern matching

the text: string s over alphabet $\{a, b, c\}$ (length n)
the pattern: $(3, 1, 2)$ - any permutation of string $aaabcc$ (length m)

b b a c [a c c a b a] b b a b c c a a a c

The problem: Jumbled pattern matching

the text: string s over alphabet $\{a, b, c\}$ (length n)
the pattern: $(3, 1, 2)$ - any permutation of string $aaabcc$ (length m)

b b a c [a c c a b a] b b [a b c c a a] a c

The problem: Jumbled pattern matching

the text: string s over alphabet $\{a, b, c\}$ (length n)
the pattern: $(3, 1, 2)$ - any permutation of string $aaabcc$ (length m)

b b a c a c c a b a b b a b c c a a a c

Definitions

- $\Sigma = \{a_1, \ldots, a_\sigma\}$ finite alphabet
- for $s \in \Sigma^*$, set $s \mapsto p(s)$ where
  $p(s)_k = |\{j \,:\, s_j = a_k\}|$,
  the multiplicity of a_k in s
- a Parikh vector $p \in \mathbb{N}^\sigma$ represents all strings s with $p(s) = p$
  $\rightsquigarrow$ "jumbled string"

Definitions

- $\Sigma = \{a_1, \ldots, a_\sigma\}$ finite alphabet
- for $s \in \Sigma^*$, set $s \mapsto p(s)$ where
  $p(s)_k = |\{j \, : \, s_j = a_k\}|$,
  the multiplicity of a_k in s
- a Parikh vector $p \in \mathbb{N}^\sigma$ represents all strings s with $p(s) = p$
  $\rightsquigarrow$ "jumbled string" (*aka* compomer, composition, composion, permuted pattern, ...)

Definitions

- $\Sigma = \{a_1, \ldots, a_\sigma\}$ finite alphabet
- for $s \in \Sigma^*$, set $s \mapsto p(s)$ where
  $p(s)_k = |\{j : s_j = a_k\}|$,
  the multiplicity of a_k in s
- a Parikh vector $p \in \mathbb{N}^\sigma$ represents all strings s with $p(s) = p$
  $\rightsquigarrow$ "jumbled string"

- p occurs in s if $\exists i, j : p(s_i \ldots s_j) = p$.
- define $p + q, p - q, p \leq q$ component-wise

Formal problem statement

Jumbled Pattern Matching (JPM)

Given string s, $|s| = n$, and query Parikh vector q, $|q| = m$.

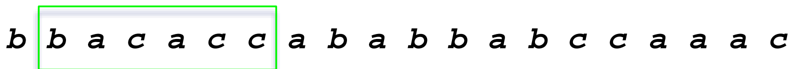
1. Does q occur in s? (decision)
2. Find all occurrences of q in s. (occurrence)

Naive solution

b b a c a c ***c a b a b b a b c c a a a c***

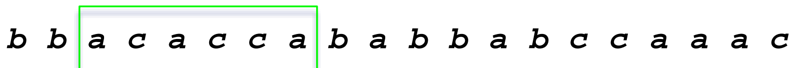
- sliding window of size $m = |q|$
- optimal for one query: $O(n)$ time, $O(\sigma)$ additional space.

Naive solution



- sliding window of size $m = |q|$
- optimal for one query: $O(n)$ time, $O(\sigma)$ additional space.

Naive solution



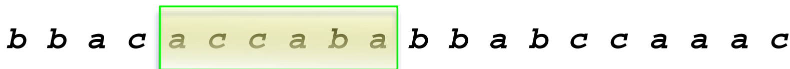
- sliding window of size $m = |q|$
- optimal for one query: $O(n)$ time, $O(\sigma)$ additional space.

Naive solution

$b\ b\ a\ \boxed{c\ a\ c\ c\ a\ b}\ a\ b\ b\ a\ b\ c\ c\ a\ a\ a\ c$

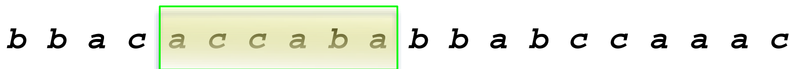
- sliding window of size $m = |q|$
- optimal for one query: $O(n)$ time, $O(\sigma)$ additional space.

Naive solution



- sliding window of size $m = |q|$
- optimal for one query: $O(n)$ time, $O(\sigma)$ additional space.

Naive solution



- sliding window of size $m = |q|$
- optimal for one query: $O(n)$ time, $O(\sigma)$ additional space.

What about many queries?

Comparison with exact pattern matching

Exact pattern matching

- one query: naive algorithm $O(nm)$ time
- one query: optimal $O(n+m)$ time (KMP, Boyer-Moore), sophisticated ideas
- many queries: $O(n)$ time preprocessing $\rightarrow$ $O(n)$ space data structure (suffix tree, ...), $O(Km)$ query time

Comparison with exact pattern matching

Exact pattern matching

- one query: naive algorithm $O(nm)$ time
- one query: optimal $O(n+m)$ time (KMP, Boyer-Moore), sophisticated ideas
- many queries: $O(n)$ time preprocessing $\rightarrow$ $O(n)$ space data structure (suffix tree, ...), $O(Km)$ query time

Jumbled pattern matching

- one query: naive algorithm $O(n)$ time - optimal!
- many queries: $O(Kn)$ time for K queries

Comparison with exact pattern matching

Exact pattern matching

- one query: naive algorithm $O(nm)$ time
- one query: optimal $O(n+m)$ time (KMP, Boyer-Moore), sophisticated ideas
- many queries: $O(n)$ time preprocessing $\rightarrow$ $O(n)$ space data structure (suffix tree, . . .), $O(Km)$ query time

Jumbled pattern matching

- one query: naive algorithm $O(n)$ time - optimal!
- many queries: $O(Kn)$ time for K queries

Question: Can we do better?

Our results

Two algorithms:

1. for binary alphabets: $O(n^2)$ preprocessing time, $O(n)$ space, $O(1)$ query time (only decision queries)
2. for general alphabets: $O(n)$ preprocessing time, no additional space, sublinear expected query time: $O(n\sqrt{\frac{\sigma}{\log \sigma}}\frac{\log m}{\sqrt{m}})$ query time

Our results

Two algorithms:

1. for binary alphabets: $O(n^2)$ preprocessing time, $O(n)$ space, $O(1)$ query time (only decision queries)
2. for general alphabets: $O(n)$ preprocessing time, no additional space, sublinear expected query time: $O(n\sqrt{\frac{\sigma}{\log \sigma}}\frac{\log m}{\sqrt{m}})$ query time

2a yesterday improved to: $O(n\frac{\sqrt{\sigma \log \sigma}}{\sqrt{m}})$, using wavelet trees

Binary alphabets: Interval algorithm

Let $|\Sigma| = 2$. Crucial property of binary strings:

Lemma

If $(x, m-x)$ and $(y, m-y)$ both occur in s, then so does $(z, m-z)$ for any $x \leq z \leq y$.

Corollary

All sub-Parikh vectors of s of length m build a set $\{(x, m-x) \; : \; \min(m) \leq x \leq \max(m)\}$.

Binary alphabets: Interval algorithm

With preprocessing:

- Preprocess: Compute all $\min(m)$ and $\max(m)$, for $1 \leq m \leq n$ in $O(n^2)$ time
- Query (q_1, q_2) with $m = q_1 + q_2$ occurs in s if and only if $\min(m) \leq q_1 \leq \max(m)$. Time $O(1)$.

Interval algorithm with lazy computation

With lazy computation:

- time $\begin{cases} O(n) & \text{for new length } m, \\ O(1) & \text{for queries with length } m \text{ already seen.} \end{cases}$
- Data structure finished in expected time $O(n^2)$ (after $O(n \ln n)$ many queries: coupon collector problem), assuming uniform distr. of query lengths.

In either case (preproc. or lazy):

- For K many queries, we need $O(n^2 + K)$ time.
- Better than naive algorithm, which has $O(KN)$ runtime, if $K = \omega(n)$.

A conjectured lower bound

Conjecture (Lower Bound)

There is no algorithm that can answer any $\Omega(n)$ queries in $o(n^2)$ time. Not even decision queries over binary alphabet.

General alphabets: Jumping Algorithm

Define $pr(i) = p(s_1 \ldots s_i)$: i'th prefix Parikh vector.

Basic idea:

$$q \text{ occurs in } s \text{ at position } (i+1, j) \iff pr(j) - pr(i) = q.$$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	b	*b*	*a*	*c*	*a*	*c*	*c*	*a*	*b*	*a*	*b*	*b*	*a*	*b*	*c*	*c*	*a*	*a*	*a*	*c*
a	0	0	1	**1**	2	2	2	3	3	**4**	4	4	5	5	5	5	6	7	8	8
b	1	2	2	**2**	2	2	2	2	3	**3**	4	5	5	6	6	6	6	6	6	6
c	0	0	0	**1**	1	2	3	3	3	**3**	3	3	3	3	4	5	5	5	5	6

e.g. $q = (312) = pr(10) - pr(4) = pr(18) - pr(12) = pr(19) - pr(13)$.

General alphabets: Jumping Algorithm

Define $pr(i) = p(s_1 \ldots s_i)$: i'th prefix Parikh vector.

Basic idea:

$$q \text{ occurs in } s \text{ at position } (i+1, j) \quad \Longleftrightarrow \quad pr(j) - pr(i) = q.$$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	b	*b*	*a*	*c*	*a*	*c*	*c*	*a*	*b*	*a*	*b*	*b*	*a*	*b*	*c*	*c*	*a*	*a*	*a*	*c*
a	0	0	1	**1**	2	2	2	3	3	**4**	4	**4**	5	5	5	5	6	**7**	8	8
b	1	2	2	**2**	2	2	2	2	3	**3**	4	**5**	5	6	6	6	6	**6**	6	6
c	0	0	0	**1**	1	2	3	3	3	**3**	3	**3**	3	3	4	5	5	**5**	5	6

e.g. $q = (312) = pr(10) - pr(4) = pr(18) - pr(12) = pr(19) - pr(13)$.

General alphabets: Jumping Algorithm

Define $pr(i) = p(s_1 \dots s_i)$: i'th prefix Parikh vector.

Basic idea:
q occurs in s at position $(i+1, j) \iff pr(j) - pr(i) = q$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	b	b	a	c	a	c	c	a	b	a	b	b	a	b	c	c	a	a	a	c
a	0	0	1	1	2	2	2	3	3	4	4	4	5	5	5	5	6	7	8	8
b	1	2	2	2	2	2	2	2	3	3	4	5	5	6	6	6	6	6	6	6
c	0	0	0	1	1	2	3	3	3	3	3	3	3	3	4	5	5	5	5	6

e.g. $q = (312) = pr(10) - pr(4) = pr(18) - pr(12) = pr(19) - pr(13)$.

General alphabets: Jumping Algorithm

Define $pr(i) = p(s_1 \dots s_i)$: i'th prefix Parikh vector.

Basic idea:

q occurs in s at position $(i+1, j) \iff pr(j) - pr(i) = q$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	b	b	a	c	a	c	c	a	b	a	b	b	a	b	c	c	a	a	a	c
a	0	0	1	**1**	2	2	2	3	3	**4**	4	**4**	**5**	5	5	5	6	**7**	**8**	8
b	1	2	2	**2**	2	2	2	2	3	**3**	4	**5**	**5**	6	6	6	6	**6**	**6**	6
c	0	0	0	**1**	1	2	3	3	3	**3**	3	**3**	**3**	3	4	5	5	**5**	**5**	6

e.g. $q = (312) = pr(10) - pr(4) = pr(18) - pr(12) = pr(19) - pr(13)$.

Of course we do not want to inspect each pair of $pr(i), pr(i+m)$.

Jumping algorithm: update rules

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	L ↓	***b***	***b***	***a***	***c***	***a***	***c***	***c***	***a***	***b***	***a***	***b***	***b***	***a***	***b***	***c***	***c***	***a***	***a***	***a***	***c***
a		0	0	1	1	2	2	2	3	3	4	4	4	5	5	5	5	6	7	8	8
b		1	2	2	2	2	2	2	2	3	3	4	5	5	6	6	6	6	6	6	6
c		0	0	0	1	1	2	3	3	3	3	3	3	3	3	4	5	5	5	5	6

Define $\texttt{firstfit}(p) = \min\{j \, : \, pr(j) \geq p\}$

- update R (first-fit-rule): $R \leftarrow \texttt{firstfit}\,\underbrace{(pr(L)+q)}_{\text{necessary char's}}$
- update L: if no match, then $L \leftarrow \texttt{firstfit}\,\underbrace{(pr(R)-q)}_{\text{unnecessary char's}}$,

  (good-suffix-rule/bad-character-rule)
  else (if match) $L \leftarrow L+1$.

Jumping algorithm: update rules

	0 (L)	1	2	3	4	5	6	7	8 (R)	9	10	11	12	13	14	15	16	17	18	19	20
		b	*b*	*a*	*c*	*a*	*c*	*c*	*a*	*b*	*a*	*b*	*b*	*a*	*b*	*c*	*c*	*a*	*a*	*a*	*c*
a		0	0	1	1	2	2	2	3	3	4	4	4	5	5	5	5	6	7	8	8
b		1	2	2	2	2	2	2	2	3	3	4	5	5	6	6	6	6	6	6	6
c		0	0	0	1	1	2	3	3	3	3	3	3	3	3	4	5	5	5	5	6

Define $\texttt{firstfit}(p) = \min\{j : pr(j) \geq p\}$

- update R (first-fit-rule): $R \leftarrow \texttt{firstfit}\ \underbrace{(pr(L) + q)}_{\text{necessary char's}}$
- update L: if no match, then $L \leftarrow \texttt{firstfit}\ \underbrace{(pr(R) - q)}_{\text{unnecessary char's}}$,

  (good-suffix-rule/bad-character-rule)
  else (if match) $L \leftarrow L + 1$.

Jumping algorithm: update rules

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
					L↓				R↓												
		b	***b***	***a***	***c***	***a***	***c***	***c***	***a***	***b***	***a***	***b***	***b***	***a***	***b***	***c***	***c***	***a***	***a***	***a***	***c***
a		0	0	1	1	2	2	2	3	3	4	4	4	5	5	5	5	6	7	8	8
b		1	2	2	2	2	2	2	2	3	3	4	5	5	6	6	6	6	6	6	6
c		0	0	0	1	1	2	3	3	3	3	3	3	3	3	4	5	5	5	5	6

Define $\texttt{firstfit}(p) = \min\{j : pr(j) \geq p\}$

- update R (first-fit-rule): $R \leftarrow \texttt{firstfit}\,\underbrace{(pr(L) + q)}_{\text{necessary char's}}$
- update L: if no match, then $L \leftarrow \texttt{firstfit}\,\underbrace{(pr(R) - q)}_{\text{unnecessary char's}}$,

  (good-suffix-rule/bad-character-rule)
  else (if match) $L \leftarrow L + 1$.

Jumping algorithm: update rules

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
					L						R										
		b	*b*	*a*	*c*	*a*	*c*	*c*	*a*	*b*	*a*	*b*	*b*	*a*	*b*	*c*	*c*	*a*	*a*	*a*	*c*
a		0	0	1	1	2	2	2	3	3	4	4	4	5	5	5	5	6	7	8	8
b		1	2	2	2	2	2	2	2	3	3	4	5	5	6	6	6	6	6	6	6
c		0	0	0	1	1	2	3	3	3	3	3	3	3	3	4	5	5	5	5	6

Define `firstfit`$(p) = \min\{j : pr(j) \geq p\}$

- update R (first-fit-rule): $R \leftarrow$ `firstfit` $\underbrace{(pr(L) + q)}_{\text{necessary char's}}$
- update L: if no match, then $L \leftarrow$ `firstfit` $\underbrace{(pr(R) - q)}_{\text{unnecessary char's}}$,

  (good-suffix-rule/bad-character-rule)
  else (if match) $L \leftarrow L + 1$.

Jumping algorithm: update rules

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
						L					R										
		b	b	a	c	a	c	c	a	b	a	b	b	a	b	c	c	a	a	a	c
a		0	0	1	1	2	2	2	3	3	4	4	4	5	5	5	5	6	7	8	8
b		1	2	2	2	2	2	2	2	3	3	4	5	5	6	6	6	6	6	6	6
c		0	0	0	1	1	2	3	3	3	3	3	3	3	3	4	5	5	5	5	6

Define `firstfit`$(p) = \min\{j : pr(j) \geq p\}$

- update R (first-fit-rule): $R \leftarrow$ `firstfit` $\underbrace{(pr(L) + q)}_{\text{necessary char's}}$
- update L: if no match, then $L \leftarrow$ `firstfit` $\underbrace{(pr(R) - q)}_{\text{unnecessary char's}}$,

  (good-suffix-rule/bad-character-rule)
  else (if match) $L \leftarrow L + 1$.

Jumping algorithm: update rules

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
						L								R							
		b	*b*	*a*	*c*	*a*	*c*	*c*	*a*	*b*	*a*	*b*	*b*	*a*	*b*	*c*	*c*	*a*	*a*	*a*	*c*
a		0	0	1	1	2	2	2	3	3	4	4	4	5	5	5	5	6	7	8	8
b		1	2	2	2	2	2	2	2	3	3	4	5	5	6	6	6	6	6	6	6
c		0	0	0	1	1	2	3	3	3	3	3	3	3	3	4	5	5	5	5	6

Define $\texttt{firstfit}(p) = \min\{j : pr(j) \geq p\}$

- update R (first-fit-rule): $R \leftarrow \texttt{firstfit}\,\underbrace{(pr(L)+q)}_{\text{necessary char's}}$
- update L: if no match, then $L \leftarrow \texttt{firstfit}\,\underbrace{(pr(R)-q)}_{\text{unnecessary char's}}$,

  (good-suffix-rule/bad-character-rule)
  else (if match) $L \leftarrow L+1$.

Jumping algorithm: update rules

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
												L ↓		*R* ↓							
		b	*b*	*a*	*c*	*a*	*c*	*c*	*a*	*b*	*a*	*b*	*b*	*a*	*b*	*c*	*c*	*a*	*a*	*a*	*c*
a		0	0	1	1	2	2	2	3	3	4	4	4	5	5	5	5	6	7	8	8
b		1	2	2	2	2	2	2	2	3	3	4	5	5	6	6	6	6	6	6	6
c		0	0	0	1	1	2	3	3	3	3	3	3	3	3	4	5	5	5	5	6

Define $\texttt{firstfit}(p) = \min\{j : pr(j) \geq p\}$

- update R (first-fit-rule): $R \leftarrow \texttt{firstfit}\,\underbrace{(pr(L)+q)}_{\text{necessary char's}}$
- update L: if no match, then $L \leftarrow \texttt{firstfit}\,\underbrace{(pr(R)-q)}_{\text{unnecessary char's}}$,

  (good-suffix-rule/bad-character-rule)
  else (if match) $L \leftarrow L+1$.

Jumping algorithm: update rules

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
		b	*b*	*a*	*c*	*a*	*c*	*c*	*a*	*b*	*a*	*b* (L)	*b*	*a*	*b*	*c*	*c*	*a*	*a* (R)	*a*	*c*
a		0	0	1	1	2	2	2	3	3	4	4	4	5	5	5	5	6	7	8	8
b		1	2	2	2	2	2	2	2	3	3	4	5	5	6	6	6	6	6	6	6
c		0	0	0	1	1	2	3	3	3	3	3	3	3	3	4	5	5	5	5	6

Define $\texttt{firstfit}(p) = \min\{j : pr(j) \geq p\}$

- update R (first-fit-rule): $R \leftarrow \texttt{firstfit}\ \underbrace{(pr(L)+q)}_{\text{necessary char's}}$
- update L: if no match, then $L \leftarrow \texttt{firstfit}\ \underbrace{(pr(R)-q)}_{\text{unnecessary char's}}$,

  (good-suffix-rule/bad-character-rule)
  else (if match) $L \leftarrow L+1$.

Jumping algorithm: update rules

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
													L						R		
		b	*b*	*a*	*c*	*a*	*c*	*c*	*a*	*b*	*a*	*b*	*b*	*a*	*b*	*c*	*c*	*a*	*a*	*a*	*c*
a		0	0	1	1	2	2	2	3	3	4	4	4	5	5	5	5	6	7	8	8
b		1	2	2	2	2	2	2	2	3	3	4	5	5	6	6	6	6	6	6	6
c		0	0	0	1	1	2	3	3	3	3	3	3	3	3	4	5	5	5	5	6

Define $\texttt{firstfit}(p) = \min\{j : pr(j) \geq p\}$

- update R (first-fit-rule): $R \leftarrow \texttt{firstfit}\ \underbrace{(pr(L) + q)}_{\text{necessary char's}}$
- update L: if no match, then $L \leftarrow \texttt{firstfit}\ \underbrace{(pr(R) - q)}_{\text{unnecessary char's}}$,

  (good-suffix-rule/bad-character-rule)
  else (if match) $L \leftarrow L + 1$.

Jumping algorithm: update rules

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
														L					R		
		b	*b*	*a*	*c*	*a*	*c*	*c*	*a*	*b*	*a*	*b*	*b*	*a*	*b*	*c*	*c*	*a*	*a*	*a*	*c*
a		0	0	1	1	2	2	2	3	3	4	4	4	5	5	5	5	6	7	8	8
b		1	2	2	2	2	2	2	2	3	3	4	5	5	6	6	6	6	6	6	6
c		0	0	0	1	1	2	3	3	3	3	3	3	3	3	4	5	5	5	5	6

Define $\texttt{firstfit}(p) = \min\{j : pr(j) \geq p\}$

- update R (first-fit-rule): $R \leftarrow \texttt{firstfit}\ \underbrace{(pr(L) + q)}_{\text{necessary char's}}$
- update L: if no match, then $L \leftarrow \texttt{firstfit}\ \underbrace{(pr(R) - q)}_{\text{unnecessary char's}}$,

  (good-suffix-rule/bad-character-rule)
  else (if match) $L \leftarrow L + 1$.

Jumping algorithm: update rules

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
		b	*b*	*a*	*c*	*a*	*c*	*c*	*a*	*b*	*a*	*b*	*b*	*a* (L)	*b*	*c*	*c*	*a*	*a*	*a* (R)	*c*
a		0	0	1	1	2	2	2	3	3	4	4	4	5	5	5	5	6	7	8	8
b		1	2	2	2	2	2	2	2	3	3	4	5	5	6	6	6	6	6	6	6
c		0	0	0	1	1	2	3	3	3	3	3	3	3	3	4	5	5	5	5	6

Define $\texttt{firstfit}(p) = \min\{j : pr(j) \geq p\}$

- update R (first-fit-rule): $R \leftarrow \texttt{firstfit}\ \underbrace{(pr(L) + q)}_{\text{necessary char's}}$
- update L: if no match, then $L \leftarrow \texttt{firstfit}\ \underbrace{(pr(R) - q)}_{\text{unnecessary char's}}$,

  (good-suffix-rule/bad-character-rule)
  else (if match) $L \leftarrow L + 1$.

Jumping algorithm: Putting it together

Algorithm *Jumping Algorithm*
Input: query Parikh vector q
Output: A set Occ of all occurrences of q in s

1.	set $m \leftarrow |q|; Occ \leftarrow \emptyset;\ L \leftarrow 0;$	
2.	**while** $L < n - m$	
3.	**do** $R \leftarrow \texttt{firstfit}(pr(L) + q)$;	update R
4.	**if** $R - L = m$	check match
5.	**then** add $L + 1$ to Occ;	
6.	$L \leftarrow L + 1$;	update L (match), or
7.	**else** $L \leftarrow \texttt{firstfit}(pr(R) - q)$;	update L (no match)
8.	**if** $R - L = m$	check match
9.	**then** add $L + 1$ to Occ;	
10.	$L \leftarrow L + 1$;	update L (match)
11.	**return** Occ;	

Jumping algorithm: Correctness

Lemma

1. *after each update of R:* $p(s_{L+1} \dots s_R) \geq q$*,*
2. *after each update of L:* $p(s_{L+1} \dots s_R) \leq q$*.*

Theorem (Correctness)

1. *Each reported occurrence is correct, and*
2. *each occurrence is found.*

Proof.

1. follows from the lemma, *2.* straightforward but a bit technical. □

Jumping algorithm: computing firstfit

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	b	*b*	*a*	*c*	*a*	*c*	*c*	*a*	*b*	*a*	*b*	*b*	*a*	*b*	*c*	*c*	*a*	*a*	*a*	*c*
a	0	0	1	1	2	2	2	**3**	3	4	4	4	5	5	5	5	6	7	8	8
b	**1**	2	2	2	2	2	2	2	3	3	4	5	5	6	6	6	6	6	6	6
c	0	0	0	1	1	**2**	3	3	3	3	3	3	3	3	4	5	5	5	5	6

I	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
a	0	3	5	**8**	10	13	17	18	19
b	0	**1**	2	9	11	12	14		
c	0	4	**6**	7	15	16	20		

$$\texttt{firstfit}(p) = \max\{\, I[k, p_k] \,:\, k = 1, \ldots, \sigma\}$$

Jumping algorithm: computing $pr(i)$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	b	*b*	*a*	*c*	*a*	*c*	*c*	*a*	*b*	*a*	*b*	*b*	*a*	*b*	*c*	*c*	*a*	*a*	*a*	*c*
a	0	0	1	1	2	2	2	3	3	4	4	4	5	5	5	5	6	7	8	8
b	1	2	2	2	2	2	2	2	3	3	4	5	5	6	6	6	6	6	6	6
c	0	0	0	1	1	2	3	3	3	3	3	3	3	3	4	5	5	5	5	6

I	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
a	0	3	5	8	10	13	17	18	19
b	0	1	2	9	11	12	14		
c	0	4	6	7	15	16	20		

$$pr(i)_k = \max\{\, j \;:\; I[k,j] \leq i \}$$

Jumping algorithm: Analysis

Algorithm *Jumping Algorithm*

```
1.   set m ← |q|; Occ ← ∅; L ← 0;
2.   while L < n − m
3.      do R ← firstfit(pr(L) + q);
4.         if R − L = m
5.            then add L + 1 to Occ;
6.                 L ← L + 1;
7.            else  L ← firstfit(pr(R) − q);
8.                 if R − L = m
9.                    then add L + 1 to Occ;
10.                        L ← L + 1;
11.  return Occ;
```

in each execution of while-loop:
update R
update L (match), or
update L (no match)

Jumping algorithm: Analysis cont.

- update R: $R \leftarrow \texttt{firstfit}(pr(L) + q)$
- update L: $L \leftarrow L + 1$,
  or $L \leftarrow \texttt{firstfit}(pr(R) - q)$

Jumping algorithm: Analysis cont.

- update R: $R \leftarrow \texttt{firstfit}(pr(L) + q)$
- update L: $L \leftarrow L + 1$,
  or $L \leftarrow \texttt{firstfit}(pr(R) - q)$

Time needed to

- compute `firstfit`:
- compute $pr(L)$:
- compute $pr(R)$:

Jumping algorithm: Analysis cont.

- update R: $R \leftarrow \texttt{firstfit}(pr(L) + q)$ $O(\sigma + \sigma \log m)$
- update L: $L \leftarrow L + 1$, $O(1)$
  or $L \leftarrow \texttt{firstfit}(pr(R) - q)$

Time needed to

- compute `firstfit`: $O(\sigma)$
- compute $pr(L)$: $O(\sigma \log(R - L)) = O(\sigma \log m)$
- compute $pr(R)$: $O(\sigma \log(R - L))$ not so easy to bound but ...

Intuition: The bigger $R - L$, the fewer jumps we are making. We pay $\log(R - L)$ but we jump $R - L$ far!

Jumping algorithm: Analysis new

- update R: $R \leftarrow \texttt{firstfit}(pr(L) + q)$
- update L: $L \leftarrow L + 1$,
  or $L \leftarrow \texttt{firstfit}(pr(R) - q)$

Notice that

- $\texttt{firstfit}(p) = \max_k\{select_k(p_k)\}$
- $pr(i)_k = rank_k(i), k = 1, \ldots, \sigma$

Jumping algorithm: Analysis new

- update R: $R \leftarrow \texttt{firstfit}(pr(L) + q)$
- update L: $L \leftarrow L + 1$,
  or $L \leftarrow \texttt{firstfit}(pr(R) - q)$

Notice that

- $\texttt{firstfit}(p) = \max_k \{select_k(p_k)\}$ $O(\sigma \log \sigma)$
- $pr(i)_k = rank_k(i), k = 1, \ldots, \sigma$ $O(\sigma \log \sigma)$

So using wavelet trees, we get

$O(\sigma \log \sigma)$ in each iteration.

Jumping algorithm: Analysis cont.

Let J = no. of iterations of while-loop (= no. of jumps). After a few simple steps we arrive at

$$\text{old runtime} = \sum_{\text{all jumps}} \text{update } R + \text{update } L = O(J\sigma \log(\frac{n}{J} + m))$$

and

$$\text{new runtime} = J \cdot (\text{update } R + \text{update } L) = O(J\sigma \log \sigma)$$

Jumping algo analysis: Estimating no. of jumps

We compute the expectation of J over a random string s (uniform i.i.d.) and balanced query $q = (\frac{m}{\sigma}, \ldots, \frac{m}{\sigma})$ as

$$\mathbb{E}(J) = \frac{n}{\mathbb{E}(\text{length of jump})}, \text{ and}$$

$$\mathbb{E}(\text{length of jump}) = Pr(\text{match}) \cdot 1 + Pr(\text{no match}) \cdot \mathbb{E}(R_{\text{new}} - L - m).$$

- $Pr(\text{match})$ = prob. that a random string of size m has Parikh vector $q = \frac{\binom{m}{\frac{m}{\sigma}, \ldots, \frac{m}{\sigma}}}{m^\sigma} \approx \sqrt{\frac{\sigma^\sigma}{m^{\sigma-1}}}$ - very small for $m >> \sigma$
- $\mathbb{E}(R_{\text{new}} - L - m)$ = length of wait before we have seen $\frac{m}{\sigma}$ many of all σ characters $= E_{m,\sigma} - m \approx \sqrt{2m\sigma \ln \frac{\sigma}{\sqrt{2\pi}}}$ (R. May, Coupon collecting with quotas, Electr. J Comb., 2008)

Jumping algo analysis: Average case

Altogether, we get (skipping many details):

$$\mathbb{E}(\textit{old}\,\textit{runtime}) = O(J\sigma\log(\frac{n}{J}+m)) = O(n\sqrt{\frac{\sigma}{\log\sigma}}\frac{\log m}{\sqrt{m}}).$$

$$\mathbb{E}(\textit{new}\,\textit{runtime}) = O(J\sigma\log\sigma) = O(n\frac{\sqrt{\sigma\log\sigma}}{\sqrt{m}}).$$

Recall: We assume a random string s (uniform i.i.d.) and balanced query $q = (\frac{m}{\sigma}, \ldots, \frac{m}{\sigma})$.

These are worst-case, both acc. to our model and our simulations:

- on non-random strings (DNA) J decreases
- on non-balanced Parikh vectors J decreases

Open problems and ongoing work.

- Prove or disprove the conjecture about lower bound for JPM.
- Tighter expected time analysis of Jumping algorithm.
- Efficient implementation of Jumping algo and experimentation data (master student).
- new Find best rank/select data structure for Jumping algo.

Diki.

Diki.

Děkuji.

Diki.

Děkuji.

Thank you.