Algoritmo Boyer Moore in versione approssimata per sequenze geniche

Seminario di Bioinformatica

AA 2007-2008

Broccolo Daniele – Marcon Lorenzo – Vandoni Riccardo

{broccolo, marcon, vandoni}@cli.di.unipi.it

Sommario /1

- Pattern matching: cos'è e perché
- Algoritmi per il pattern matching esatto
  - Boyer-Moore e sue varianti

Sommario /2

- Algoritmi per il pattern matching approssimato
  - Baeza-Yates-Gonnet (1992)
  - Tarhio-Ukkonen (1993)
  - FAAST (2005)
  - Boyer Moore ottimizzato per sequenze geniche (2007)
- Dimostrazione live

Pattern Matching: cos'è?

Il pattern matching

I. Esatto

II. Approssimato
   - k-mismatch
   - k-difference

Pattern Matching: perché? /1

- Ricostruire una sequenza di DNA dai singoli pezzi ottenuti da esperimenti

- Ricercare delle specifiche caratteristiche nelle sequenze di DNA

- Determinare quanto sono differenti due sequenze di codice genetico

Pattern Matching: perché? /2

OFRG: Oligonucleotide Fingerprinting of Ribosomal RNA Genes

- E’ un metodo che permette l’identificazione di geni ribosomali RNA allineati attraverso una serie di esperimenti di ibridizzazione usando piccole sonde di DNA.
- Questo approccio coinvolge un’innovativa strategia di analisi dei dati, fornisce un buon rapporto qualità/prezzo e potrebbe avere applicazioni in medicina, biotecnologie e studio dell’ecosistema.
- L’utilizzo di questo nuovo approccio sperimentale porta e porterà ad una maggiore comprensione degli organismi che popolano il nostro pianeta, la loro funzione e il loro potenziale per la biotecnologia.

Boyer-Moore /1

Algoritmo di string matching *esatto* (1977).

L'idea: **preprocessing sul pattern**

Vengono utilizzate due euristiche:

1. Bad character
2. Good suffix

Boyer-Moore /2

Bad character shift table

$$B[c] = \min\{\ k>=1 : p(k+1) = c\ \}$$

min{ } = m per convenzione.

Esempio:

Pattern: GCTTGAGGA

char	A	G	T	C
shift	3	1	5	7

Ogni altro carattere ha un valore di shift pari ad m.

Boyer-Moore /3

Good suffix shift table

G[i] = min { k >= 1 : [p(min{m,i+k-1})..p(k+1)] fa match con un suffisso di [p(i-1)..p(1)] quando i>=2 e p(i+k) != p(i) quando i+k <= m }

Pattern: **GCTTGAGGAA**

i	9	8	7	6	5	4	3	2	1
char	G	C	T	T	G	A	G	G	A
shift	9	9	9	9	9	9	3	9	1

Boyer-Moore /4

Esempio:

Text: **GCTGCTTGAGGAATGATGC**

Pattern: **GCTTGAGGA**

char	A	G	T	C
shift	3	1	5	7

i	9	8	7	6	5	4	3	2	1
char	G	C	T	T	G	A	G	G	A
shift	9	9	9	9	9	9	3	9	1

In caso di mismatch lo shift avrà un valore pari a max {G[i], B[c] - i + 1}

L'algoritmo restituisce l'indice dove ha inizio l'occorrenza trovata, cioè 3

Boyer-Moore /5

- La complessità in fase di preprocessing è **O(m + σ)**
- La complessità in fase di ricerca è **O(m * n)**

- Nel caso pessimo saranno effettuati ~ **3 * n** confronti

- Lo spazio occupato è pari a **O(*m* + σ)**

Varianti del Boyer-Moore

1. **Turbo-BM**
   - Più spazio occupato
   - Nel caso pessimo sono necessari **2 * n** confronti
2. **Boyer-Moore-Horspool (1980)**
   - Meno spazio necessario, non necessita della seconda tabella in quanto la bad character table è calcolata in maniera diversa (evita i valori negativi).
   - Nel caso pessimo sono necessari **m * n** confronti

Perché ricerca approssimata?

- Gli algoritmi finora presentati effettuano un matching esatto

- In biologia il pattern matching approssimato è più interessante rispetto a quello esatto per questioni legate alla struttura delle sequenze geniche

K-mismatch

Il problema è quello di cercare tutte le occorrenze del pattern con al massimo k differenze.

Baeza-Yates-Gonnet (1992)

- Risolve il problema del pattern matching attraverso lo sviluppo del match-count

- L'idea è quella di voler rappresentare ogni stato della ricerca come una sequenza di bit, in modo da poter computare simultaneamente più stati attraverso operazioni logiche (shift-add method)

Complessità: **O(mn * log m/w)**

Tarhio-Ukkonen /1

- Basato sul Boyer-Moore-Horspool
- Ne mantiene le caratteristiche fondamentali:
  - Preprocessing sul pattern
  - Scansione da destra a sinistra
- Risolve il problema del k-mismatch

Tarhio-Ukkonen /2

- Preprocessing:
  - calcolo la shift-table in cui memorizzo per ogni carattere i di P le distanze dei diversi simboli ∈ Σ

$$d_k[i,a] = \min\{\{m-k\} \cup \{s \mid p_{i-s} = a, s \in [1,i-1], a \in \Sigma\}\}$$

- Esecuzione
  - La distanza di shift è calcolata in modo che almeno uno degli ultimi k caratteri restituisca un match.

$$d_k[t_{j-k}, t_{j-k+1}, \ldots, t_j] = \min\{d_k[m-i, t_{j-1}], i \in [0,k]\}$$

Tarhio Ukkonen: Esempio

k=2, m=10, n=20

K+1 mismatch

T: AACTGTTAACTTGCGACTAG

P: AAGTCGTAAC

AAGTCGTAAC

Shift=d_k[AAC]=
min{d_k[8,A], d_k[9,A], d_k[10,C]}= 1

posizione	A	C	G	T
8	6	3	2	1
9	1	4	3	2
10	1	5	4	3

Complessità: $O(kn(1/m-k+k/c)$,dove c=# Σ

FAAST

- Algoritmo per il k-mismatch
- Rispetto al Tarhio-Ukkonen viene introdotto il parametro x che indica il numero di match che la sottostringa deve soddisfare.
- Non mi accontento più quindi di almeno 1 match su $k + 1$, ma ne devo ottenere almeno x su $k + x$.

FAAST
L'algoritmo /1

- Preprocessing:
  - Cerco le istanze dei simboli che precedono un carattere e le inserisco tutte in tabella.
    - $\mathcal{U}_{kx}[i,a]=\{s \mid p_{i-s}=a,\ s\in[1,i-1], a\in\Sigma\ \}$

Esempio con k=2 e x=3

1 2 3 4 5 6 7 8 9 10

P: A A G T C G T A A C

posizione	A	C	G	T
6	4,5	1	3	2
7	5,6	2	1,4	3
8	6,7	3	2,5	1,4
9	1,7,8	4	3,6	2,5
10	1,2,8,9	5	4,7	3,6

FAAST
L'algoritmo /2

– $\mathcal{V}_{kx}[t_{j-k-x+1} \ldots t_j, l] = \{ l \mid \mathcal{U}_{kx}[m-i, t_{j-1}], i \in [0, k+x-1] \}$

l	1	2	3	4	5	6	7	8
AAAAA	0,1	0		4	3,4	2,3	1,2	0,1
…								
GCGAC	1	2,3	4		0,2		1	1
…								
GTCGT			0,1,2,3,4			0,1		
…								
TTAAC	1	4	3		0	2	1,2	1
…								
TTTTT	2	1,4	0,3	2	1	0		

FAAST
L'algoritmo /3

- La Distanza di Shift è calcolata in modo da ottenere al massimo k mismatch tra i k-x caratteri:
  - $d_{kx}[t_{j-k-x+1} \dots t_j] =$

    $min\{ l \mid\mid \mathcal{V}_{kx}[t_{j-k-x+1} \dots t_j, l] \mid \geq min\{x, m-k-l, l \in [1, m-k]\}$

FAAST
L'algoritmo /4

- Calcolo di $\mathcal{V}_{kx}$ e di d_{kx} :

l	1	2	3	4	5	6	7	8	d_{kx}
AAAAA	0,1	0		4	3,4	2,3	1,2	0,1	6
…									
GCGAC	1	2,3	4		0,2		1	1	7
…									
GTCGT			0,1,2,3,4			0,1			3
…									
TTAAC	1	4	3		0	2	1,2	1	7
…									
TTTTT	2	1,4	0,3	2	1	0			8

FAAST: Esempio

mismatch

k=2, x=3, m=10, n=20

Shift 1 : d_{kx}[TTAAC]= 7

l	1	2	3	4	5	6	7	8	d_{kx}
…									
TTAAC	0	4	3		0	2	1,2	1	7
…									

FAAST: Correttezza /1

- *<u>Teorema:</u> Dato un qualsiasi allineamento tra Pattern P e* $t_{j-k-x+1} \ldots t_j$ in T, P può essere shiftato di $d_{kx}[t_{j-k-x+1} \ldots t_j]$ caratteri verso destra senza saltare nessuna occorrenza approssimata di P in T.

FAAST: Correttezza /2

- *Dimostrazione:*

Esmepio: k=2, x=1

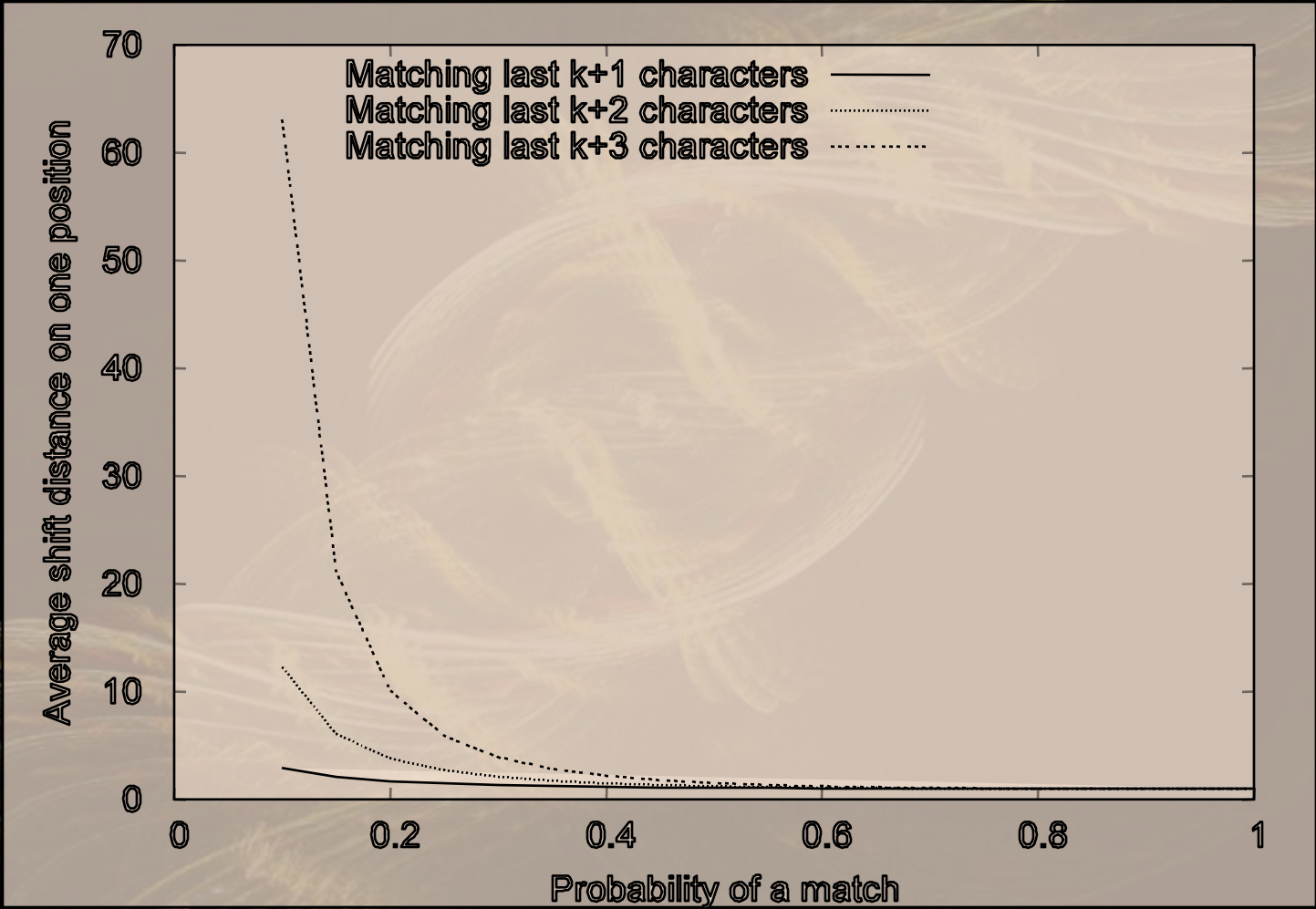
$d_{kx}[t_{j-k-x+1} \ldots t_j] = j-i = 3$

Assumiamo per assurdo che nello shift sia passata un' occorrenza, questo significa che esiste i' tale per cui $p_{i'-k-x+1} \ldots p_{i'}$ allineato con $t_{j-k-x+1} \ldots t_j$ produce al massimo k mismatch e i' > i . Ma questo è in contraddizione con la definizione di d_{kx}.

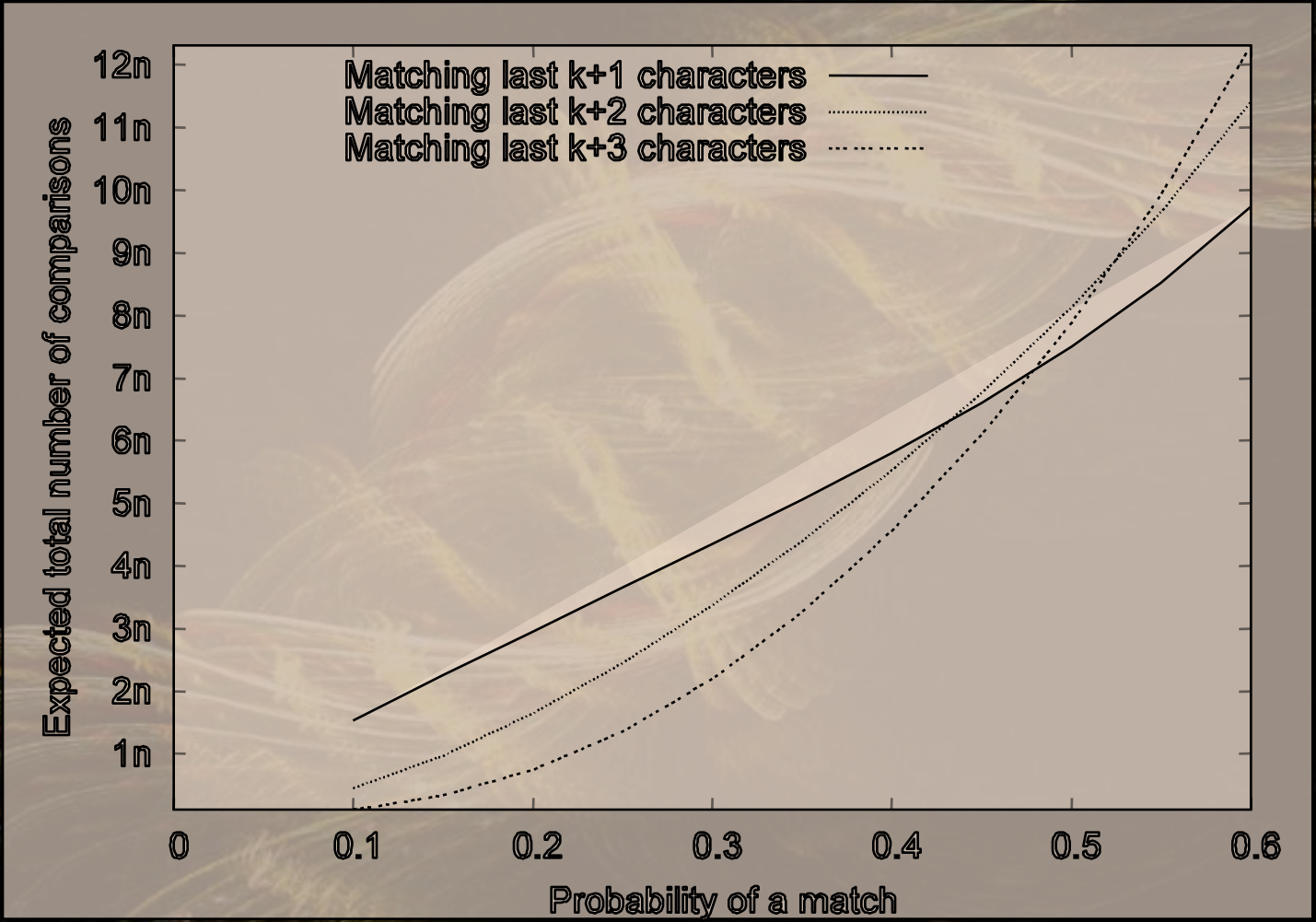
FAAST: Complessità

- Preprocessing:
  - Calcolo di $\mathcal{U}_{kx}$:
    - Spazio: $O(m(k+x))$
    - Tempo: $O((m-k)(k+x)c)$,dove c= #Σ = 4 nel DNA
  - Calcolo di d_{kx}:
    - Spazio: $O(c^{k+x})$
    - Tempo: $O(c^{k+x}(m-k)(k+x))$
  - TOTALE:
    - Spazio: $O(c^{k+x}+c(m-k)(k+x))$
    - Tempo: $O((k+x)((m-k)c^{k+x}+m))$
- *Il matching richiede $O(mn)$ nel caso peggiore.*

FAAST:
Distanze medie di shift al variare di x



FAAST:
Quantità di caratteri comparati



Risultati con sequenze simulate

- Text: 2M bases sequence, Pattern: 39 bases, k=3.

x	1	2	3	4	5	6	7
Ave. shift dist.	1.41	2.76	5.59	16.38	31.31	37.37	38.87
Total comp.	6.70	3.68	1.86	0.65	0.34	0.28	0.27
Running time(sec.)	210.2	114.4	58.1	20.6	11.2	10.8	16.7
Prepro. Time(sec.)	0.01	0.01	0.03	0.08	0.36	1.58	6.90

Risultati con sequenze reali

- Text: 150 bacteria DNA sequences, k=3

x	1	2	3	4	5	6	7
Running time (sec.)	18.87	13.05	7.74	3.84	2.63	3.21	8.55
Prepro. Time(sec.)	0.01	0.01	0.02	0.09	0.35	1.57	6.96
matching Time(sec.)	18.77	13.04	7.72	3.75	2.28	1.64	1.59

- Text: 150 fungi DNA sequences, k=3

x	1	2	3	4	5	6	7
Running time (sec.)	16.45	11.43	9.24	6.78	5.62	8.24	26.48
Prepro. Time(sec.)	0.02	0.03	0.08	0.32	1.34	5.77	23.86
matching Time(sec.)	16.43	11.40	9.16	6.46	4.28	2.47	2.62

Degenerazione /1

- Molti amminoacidi sono specificati da più di un codone
- Un problema comune è quello di cercare dei pattern che tengano conto di tale caratteristica

Ala	A	GCU, GCC, GCA, GCG	Leu	L	UUA, UUG, CUU, CUC, CUA, CUG
Arg	R	CGU, CGC, CGA, CGG, AGA, AGG	Lys	K	AAA, AAG
Asn	N	AAU, AAC	Met	M	AUG
Asp	D	GAU, GAC	Phe	F	UUU, UUC
Cys	C	UGU, UGC	Pro	P	CCU, CCC, CCA, CCG
Gln	Q	CAA, CAG	Ser	S	UCU, UCC, UCA, UCG, AGU, AGC
Glu	E	GAA, GAG	Thr	T	ACU, ACC, ACA, ACG
Gly	G	GGU, GGC, GGA, GGG	Trp	W	UGG
His	H	CAU, CAC	Tyr	Y	UAU, UAC
Ile	I	AUU, AUC, AUA	Val	V	GUU, GUC, GUA, GUG
start		AUG, GUG	stop		UAG, UGA, UAA

Degenerazione /2

- Per risolvere questo problema vengono utilizzate delle lettere speciali per rappresentare piu basi
  - secondo la IUPAC-IUB :
    - R=G/A
    - Y=T/A
    - H=A/C/T

FAAST per i codici degenerati /1

- Un metodo “Naive” potrebbe essere quello di usare più pattern, espandendo i caratteri degenerati.
- Un metodo più veloce invece prevede due modifiche :
  - Nuova definizione di Match
  - Ridefinire lo Shift

FAAST per i codici degenerati /2

- Due caratteri degenerati fano match se hanno in comune aratteri non degenrati
- Esempio:
  - R = A/G
  - H = A/C/T
  - R e H fanno match.

FAAST per i codici degenerati /3

- Nel calcolo delo shift il FAAST tratta i caratteri degenerati come uno qualsiasi dei suoi caratteri non-degenerati corrispondenti e prende la distanza minima nel pattern.
- Per i resto la procedura rimane inalterata

K-problems/1

Trovare tutte le sottostringhe del testo T tali che:

la **distanza** tra tale sottostringa e il pattern P sia al più k.

distanza:

K-mismatch → sostituzioni (Hamming distance)

K-difference → sostituzioni,inserimenti,cancellazioni (Edit distance)

K-problems/2

Generici algoritmi di APPROXIMATE STRING MATCHING sono progettati per lavorare su testi

alfabeti grandi, basso tasso di errori

Algoritmi/1

- Algoritmo 1: k-mismatch problem per sequenze geniche
- Algoritmo 2: modifica del precedente per k-difference problem
- Algoritmo 3: raffinamento dell'algoritmo 2

Algoritmi/2

- Due fasi: PREPROCESSING e SEARCHING

- Basati su programmazione dinamica

Algoritmo 1 (K-mismatch)/1

Fase 1: preprocessing

E’ sufficiente operare sul pattern

k-gramma: stringa lunga k di caratteri contigui

Esempio:

t= GATTACA, k = 3
k-grammi di T:
GAT, ATT, TTA, TAC, ACA

Algoritmo 1 (K-mismatch)/2

Fase 1: preprocessing

Calcola la distanza di Hamming fra ogni (k+x)-gramma dell'alfabeto e ogni (k+x)-gramma del pattern.

Esempio:

t = GCATA, p = GGCAA, k = x = 2

(k+x)-gramma finale di t: CATA

(FAAST effettuava questo calcolo nella fase di searching)

Algoritmo 1 (K-mismatch)/3

Fase 1: preprocessing

$D[i,j] = D[i-1,j-1] + \alpha$ dove $\alpha = \begin{cases} 0 & \text{se } t_{i-1} = p_{j-1} \\ 1 & \text{altrimenti} \end{cases}$

			G	G	C	A	A
	i\j	*0*	*1*	*2*	*3*	*4*	*5*
	0	0	0	0	0	0	0
C	*1*	0	1	1	0	1	1
A	*2*	0	1	2	2	0	1
T	*3*	0	1	2	3	3	1
A	*4*	0	1	**2**	3	3	**3**

shift corretto:

m-j

In questo caso:

m-j = 5-2 = 3

Algoritmo 1 (K-mismatch)/4

Fase 1: preprocessing

non è necessario quindi tenere in memoria tutta la matrice D.

Strutture dati associative, in cui per ogni (*k+x*)-gramma :

M = contiene le distanze di Hamming

D_{kx} = contiene gli shift precalcolati come mostrato (*m-j*)

Algoritmo 1 (K-mismatch)/5

Fase 2: searching

Presi gli ultimi *k+x* caratteri del testo t, questi corrispondono ad un (*k+x*)-gramma calcolato in precedenza, mappato con y nei vettori *M* e D_{kx}

Cerchiamo un'occorrenza del pattern nel caso in cui $M[y] \leq k$.

Esempio per *k=2*:

M["CATA"] = 2 OK! 

M["ATCG"] = 3 KO! 

Infine, shiftiamo di $D_{xk}[y]$ caratteri.

Algoritmo 2 (K-difference)/1

Modifica dell'algoritmo mostrato per il k-mismatch problem

L'equazione che riempie la matrice D è ora la seguente:

$$D[i,j] = \min \begin{cases} D[i-1,j-1] + \alpha \\ D[i-1,j] + 1 \\ D[i,j-1] + 1 \end{cases} \quad \text{dove} \quad \alpha = \begin{cases} 0 & \text{se } t_{i-1} = p_{j-1} \\ 1 & \text{altrimenti} \end{cases}$$

A partire da questa otteniamo i vettori M e D_{kx}

Algoritmo 2 (K-difference)/2

La fase di search inizia allineando il pattern al prefisso di testo che termina alla posizione *m-k-1*

detta ***s*** la posizione in cui termina l'allineamento:

$y = t_{s-(k+x)+1} \dots t_s$ (sottostringa del testo)

Se $M[y] \leq k$, costruiamo per ogni allineamento una matrice di dimensione $(m+x+1)$ x $(m+1)$ contenente le distanze (di edit)

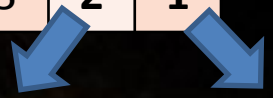
Si riporta un match se $D[m+k,m] \leq k$, dopodiché si shifta di $D_{kx}[y]$

Algoritmo 2 (K-difference)/3

Esempio:

p = GGCAA t = AGGCATA k=2 x=2

			G	G	C	A	A
	i\j	*0*	*1*	*2*	*3*	*4*	*5*
	0	0	0	0	0	0	0
C	*1*	0	1	1	0	1	1
A	*2*	0	1	2	1	0	1
T	*3*	0	1	2	2	1	1
A	*4*	0	1	2	3	**2**	**1**



D_{xk}[“CATA”] = 5 - 4 = 1

M[“CATA”] = 1

Algoritmo 2 (K-difference)/4

p = GGCAA t = AGGCATA k = 2 x = 2

Costruiamo la matrice *D* di dimensione *(m+k+1)* x *(m+1)*

			G	G	C	A	A
	i\j	*0*	*1*	*2*	*3*	*4*	*5*
	0	0	0	0	0	0	0
A	*1*	0	1	1	1	0	0
G	*2*	0	0	1	2	1	1
G	*3*	0	0	0	1	2	2
C	*4*	0	1	1	0	1	2
A	*5*	0	1	2	1	0	1
T	*6*	0	1	2	2	1	1
A	*7*	0	1	2	3	2	**1**

$D[m+k,m] = 1 \leq k$

Match alla posizione s!

Algoritmo 2 (K-difference)/5

Nell'algoritmo 1 non c'era bisogno di rileggere gli ultimi *k+x* caratteri quando controllavamo un'occorrenza, perché conoscevamo già quanti mismatch c'erano grazie a *M*.

Nell'algoritmo 2 il testo allineato è letto in avanti durante la fase di search, mentre nel preprocessing la tabella D è stata generata per gli ultimi caratteri del pattern.

Calcolare una tabella invertita velocizza l'algoritmo nella fase di search

Algoritmo 3 (K-difference)/1

L'introduzione di D_{inv} modifica l'algoritmo 2

D_{inv} è così inizializzata:

$D[0,j] = j \qquad j = 0..m$

$D[i,0] = i \qquad i = 0..k+x$

Cambia la condizione di α

$$\alpha = \begin{cases} 0 & \text{se } t_{k+x-i} = p_{m-j} \\ 1 & \text{altrimenti} \end{cases}$$

Algoritmo 3 (K-difference)/2

p = GGCAA t = AGGCATA k = 2 x = 2

La distanza di edit fra stringa e pattern è la stessa, ma abbiamo ancora bisogno di D_{kx} e M dalla tabella non invertita!

D_{inv}

			A	A	C	G	G
	i\j	*0*	*1*	*2*	*3*	*4*	*5*
	0	0	1	2	3	4	5
A	*1*	1	0	1	2	3	4
T	*2*	2	1	1	2	3	4
A	*3*	3	2	1	2	3	4
C	*4*	4	3	2	1	2	3
G	*5*	5	4	3	2	1	**2**
G	*6*	6	5	4	3	2	**1**
A	*7*	7	6	5	4	3	**2**

Il match viene cercato nell'ultima colonna

2k+1 celle da controllare (insertion e deletion)

Algoritmo 3 (K-difference)/3

Se $D_{inv}[m+i,m] \leq k$, $i \in -k \ldots k$:

allora $t_{s-(m+i)+1} \ldots t_s$ match $p_0 \ldots p_{m-1}$
con al più k differenze

Risultati ottenuti/1

Tempi di search in secondi per il problema k-mismatch

	k=1			k=2		
m	ABM	FAAST	Alg.1	ABM	FAAST	Alg.1
15	7.28 (0.04)	1.17 (0.48)	0.64 (0.03)	15.65 (0.04)	2.17 (1.76)	1.21 (0.16)
20	7.28 (0.07)	0.92 (0.65)	0.54 (0.03)	15.65 (0.04)	1.68 (2.58)	0.98 (0.14)
25	7.24 (0.09)	0.78 (0.87)	0.44 (0.04)	15.63 (0.09)	1.47 (3.13)	0.81 (0.22)
30	7.22 (0.15)	0.68 (0.98)	0.40 (0.06)	15.71 (0.10)	1.30 (3.70)	0.69 (0.20)
35	7.34 (0.18)	0.60 (1.22)	0.36 (0.05)	15.65 (0.16)	1.22 (4.16)	0.53 (0.24)
40	7.31 (0.24)	0.53 (1.42)	0.33 (0.05)	15.69 (0.19)	1.11 (4.73)	0.54 (0.27)

Tempi di preprocess fra parentesi

Risultati ottenuti/2

Tempi di search in secondi per il problema k-difference

	k=1					k=2				
m	ABM	Myers	BYP	Alg.2	Alg.3	ABM	Myers	BYP	Alg.2	Alg.3
15	8.82	7.35	2.85	1.98	1.65	38.58	7.33	6.90	6.70	5.04
20	8.27	7.41	2.74	1.63	1.44	27.24	7.36	4.50	5.75	4.53
25	7.99	7.34	2.69	1.41	1.34	19.49	7.37	3.79	5.58	4.09
30	8.07	7.37	2.67	1.32	1.15	14.80	7.37	3.89	5.61	4.03
35	8.07	-	2.62	1.29	1.13	12.48	-	3.73	5.77	4.00
40	7.99	-	2.63	1.23	1.05	11.08	-	3.94	5.95	4.04

Risultati ottenuti/3

I tempi di preprocessing sono maggiori per l'algoritmo 3

	Preprocessing		Search	
x	Alg.2	Alg.3	Alg.2	Alg.3
1	<0.01	<0.01	977.30	724.61
2	0.01	0.01	213.43	144.53
3	0.02	0.05	45.57	28.92
4	0.10	0.18	11.64	7.08
5	0.37	0.71	3.94	2.44
6	1.59	2.76	1.84	1.44
7	6.38	11.35	1.63	1.51
8	25.27	46.50	3.06	2.94
9	101.09	188.38	4.03	4.06

Dimostrazione live

Eseguiamo ora dei test

Codice gentilmente fornito da Leena Salmela