Approximate String Matching using Bidirectional Index

Gregory Kucherov
CNRS/Université Paris-Est, Marne-la-Vallée, France

Kamil Salikhov
Université Paris-Est, Marne-la-Vallée, France
Lomonosov Moscow State University, Russia

Dekel Tsur
Department of Computer Science, Ben-Gurion University of the Negev, Israel

CPM, June 17th 2014

Problem

Approximate matching of pattern P

- *Hamming* distance
- maximum k mismatches

Assumptions: text $\mathbf{T}$ is

- static
- given before queries are made
- available for preprocessing and storing in a data structure

Approximate matching in bioinformatics

Problem: mapping of Next Generation Sequencing reads

- reference genome sequence: long sequence on alphabet {A,C,G,T}
- large collections of *reads*: short strings

Goal: fast and accurate approximate matching of reads to the reference sequence

Indexing for approximate string matching: previous work

1 **mismatch**: [Myers 94], [Cobbs 95], [Amir et al. 99], [Buchsbaum et al. 00], [Navarro et al. 01], [Cole et al. 04], [Huynh et al. 04], [Lam et al. 05], [Maaß et al. 05], [Chan et al. 06]

k **mismatches**: [Cobbs 95], [Cole et al. 04], [Huynh et al. 04], [Lam et al. 05], [Maaß et al. 05], [Chan et al. 06], [Coelho et al. 06]

existing algorithms require exponential on k space or search time

FM-index [Ferragina & Manzini, 2000]

- based on Burrows-Wheeler transform & Compressed Suffix array
- supports *Count* (return number of pattern occurrences) and *Locate* (find all positions) operations
- performs *backward search*: given occurrences of string **S**, return occurrences of **c**S
- memory usage $O(n)$ bits ($2-4$ bits per character for DNA sequences)

Bidirectional search

backward search ($P \rightarrow cP$)

Bidirectional search

backward search ($P \to cP$)

forward search ($P \to Pc$)

Bidirectional search

backward search ($P \to cP$)

forward search ($P \to Pc$)

bidirectional search ($P \to cP$ or $P \to Pc$)

Bidirectional search

backward search ($P \rightarrow cP$)

forward search ($P \rightarrow Pc$)

bidirectional search ($P \rightarrow cP$ or $P \rightarrow Pc$)

[Lam et al. 09] showed how FM-index can be made bidirectional

Search with k mismatches using backtracking

(i) start with empty string

(ii) extend the current string with the corresponding letter of P, and with all other letters increasing the number of mismatches by 1

(iii) proceed until

- number of mismatches $> k$, or
- no occurrences of the current string are found

then backtrack

(iiii) complexity can be measured in terms of **number of enumerated strings** during the search

Bidirectional search with 1 mismatch

Key observation: if P is partitioned into two parts $P = P_1 P_2$, then one of them has no mismatches

Two *independent* searches instead of one:

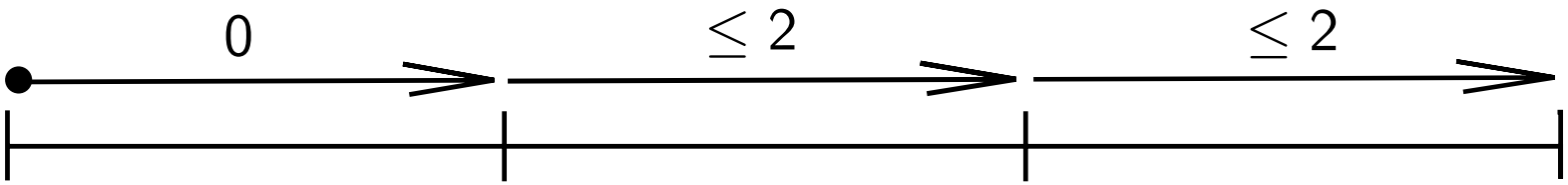
(i) Forward search with no mismatches in P_1, then up to 1 mismatch in P_2

(ii) Backward search with no mismatches in P_2, then up to 1 mismatch in P_1

[Lam et al. 09] scheme for 2 mismatches

Three searches: Pattern $P = P_1 P_2 P_3$

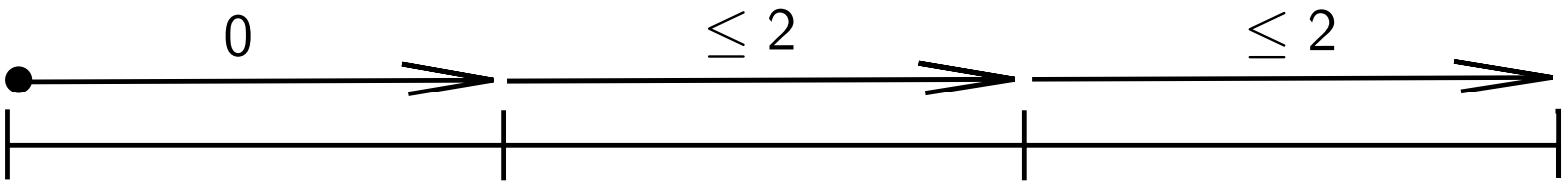
Forward S_f



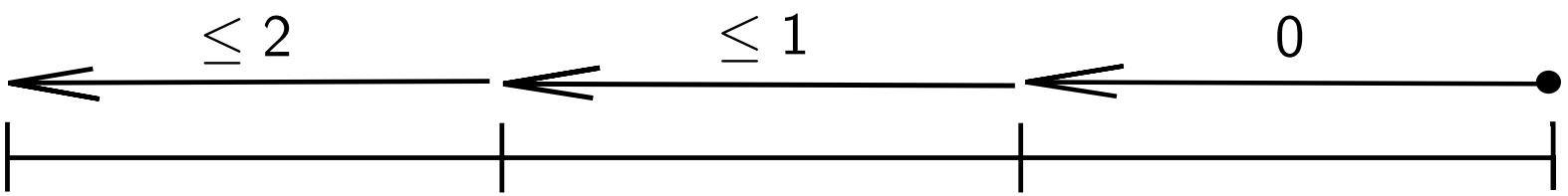
[Lam et al. 09] scheme for 2 mismatches

Three searches: Pattern $P = P_1 P_2 P_3$

Forward S_f



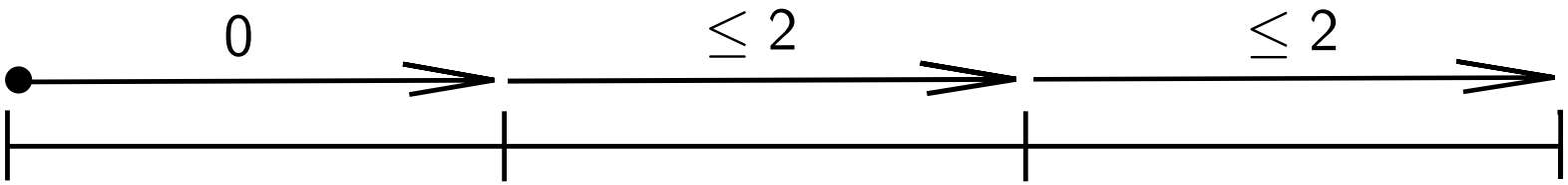
Backward S_b



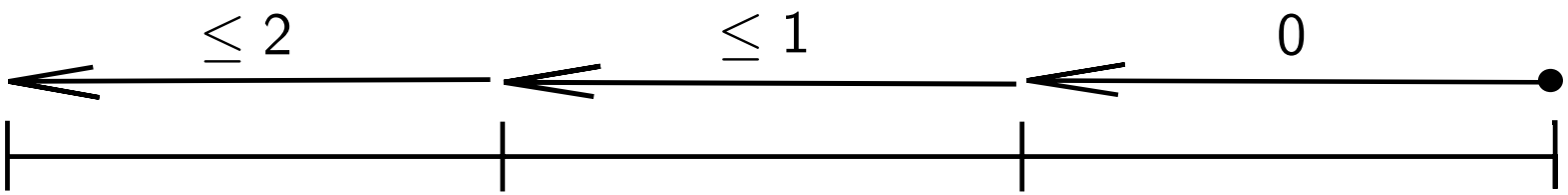
[Lam et al. 09] scheme for 2 mismatches

Three searches: Pattern $P = P_1 P_2 P_3$

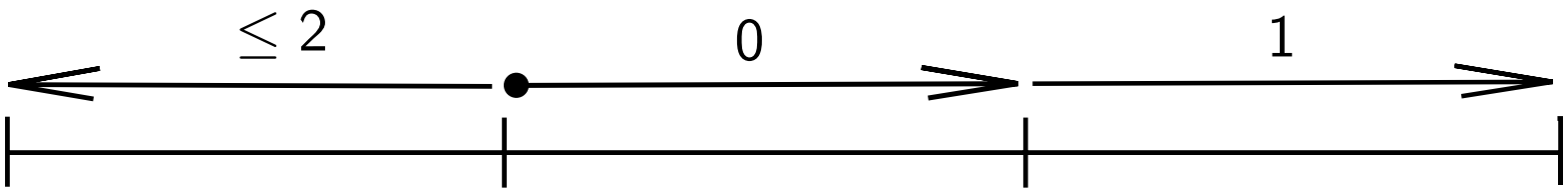
Forward S_f



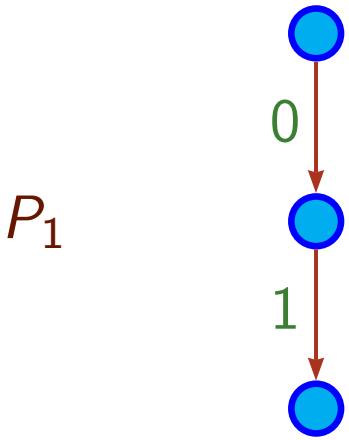
Backward S_b



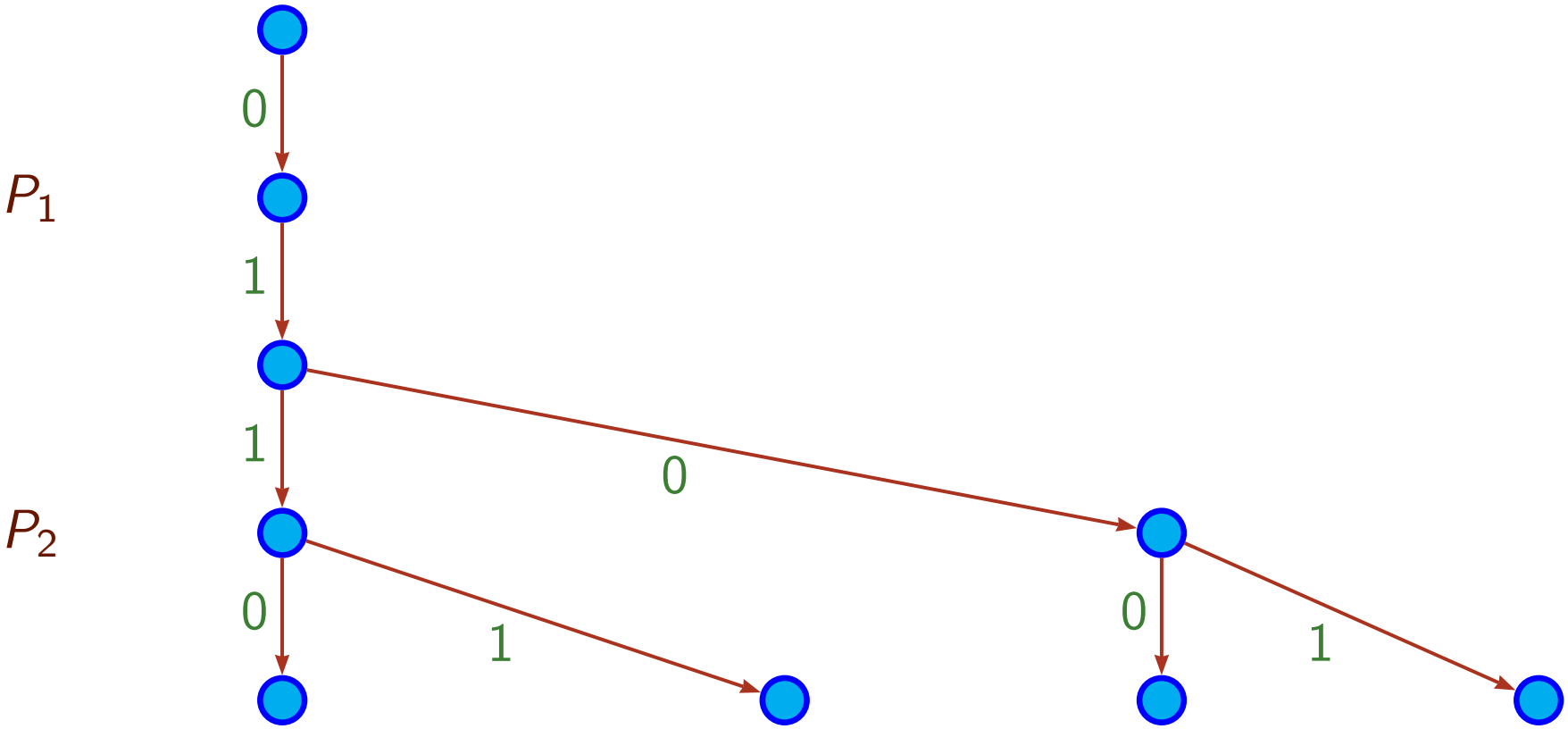
Bidirectional S_{bid}



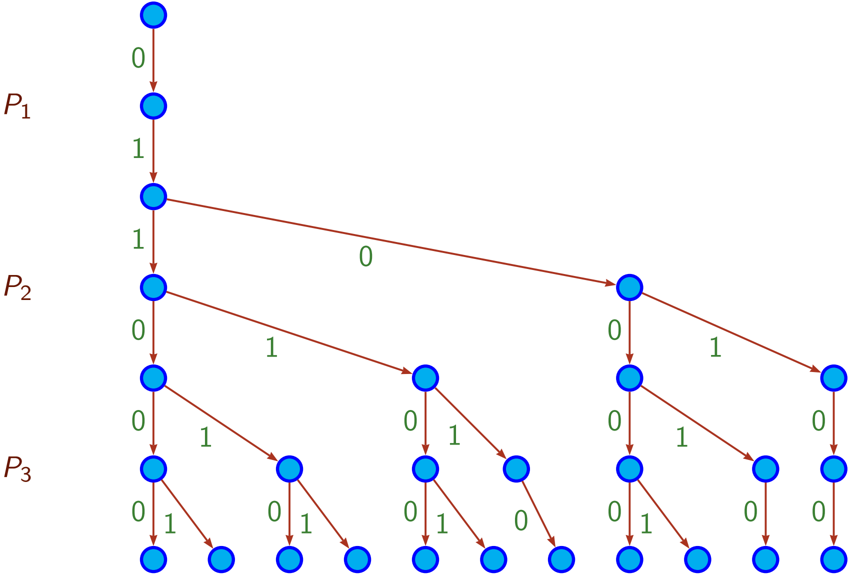
Enumeration tree for S_f for $P = 011000$ and $k = 2$



Enumeration tree for S_f for $P = 011000$ and $k = 2$



Enumeration tree for S_f for $P = 011000$ and $k = 2$



Our contribution

- General framework for approximate search on a bidirectional index
- Improved search schemes (generating smaller enumeration trees)

Search scheme: formal definition

A *search* is a triplet (π, L, U):

- π : order in which parts are processed
- $L[j]$ and $U[j]$: lower and upper bounds for number of accumulated mismatches when processing j-th part

Search scheme: formal definition

A *search* is a triplet (π, L, U):

- π : order in which parts are processed
- $L[j]$ and $U[j]$: lower and upper bounds for number of accumulated mismatches when processing j-th part

Search scheme $\mathcal{S}$: collection of searches that covers every possible distribution of errors among parts

Search scheme: formal definition

A *search* is a triplet (π, L, U):

- π : order in which parts are processed
- $L[j]$ and $U[j]$: lower and upper bounds for number of accumulated mismatches when processing j-th part

Search scheme $\mathcal{S}$: collection of searches that covers every possible distribution of errors among parts

Example for [Lam et al. 09] search scheme for 2 mismatches:

- S_f: $\pi = (1, 2, 3), U = (0, 2, 2), L = (0, 0, 0)$
- S_b: $\pi = (3, 2, 1), U = (0, 1, 2), L = (0, 0, 0)$
- S_{bid}: $\pi = (2, 3, 1), U = (0, 1, 2), L = (0, 1, 1)$

Efficiency of search scheme: random text and pattern

$\#str(S, X, \sigma, n)$ – number of strings enumerated by search S for partition X

Efficiency of search scheme: random text and pattern

$\#str(S, X, \sigma, n)$ – number of strings enumerated by search S for partition X

$\#str(S, X, \sigma, n) = \sum_{l \geq 1} \sum_{A \in A_l} Pr[A \text{ is a substring of } T]$, where A_l contains all possible enumerated strings of length l

Efficiency of search scheme: random text and pattern

$\#str(S, X, \sigma, n)$ – number of strings enumerated by search S for partition X

$\#str(S, X, \sigma, n) = \sum_{l \geq 1} \sum_{A \in A_l} Pr[A \text{ is a substring of } T]$, where A_l contains all possible enumerated strings of length l

Estimate with $\#str' = \sum_{l=1}^{\lceil \log_\sigma n \rceil + c} |A_l|(1 - e^{-n/\sigma^l})$

Efficiency of search scheme: random text and pattern

$\#str(S, X, \sigma, n)$ – number of strings enumerated by search S for partition X

$\#str(S, X, \sigma, n) = \sum_{l \geq 1} \sum_{A \in A_l} Pr[A \text{ is a substring of } T]$, where A_l contains all possible enumerated strings of length l

Estimate with $\#str' = \sum_{l=1}^{\lceil \log_\sigma n \rceil + c} |A_l|(1 - e^{-n/\sigma^l})$

$|A_l|$ can be computed using a recurrence relation (depending on the partition)

Two improvements

- **Uneven partition**:
  Partition the pattern into parts of *unequal* size

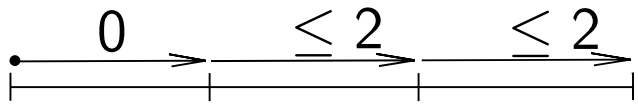
Two improvements

- **Uneven partition**:
  Partition the pattern into parts of *unequal* size

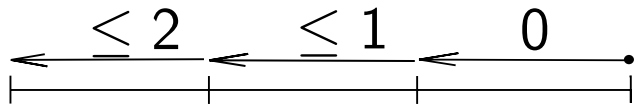
- **More parts**:
  Partition the pattern into $k + 2$ (or more) parts instead of $k + 1$

Uneven partition: intuition

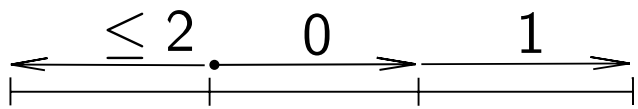
Forward S_f:



Backward S_b:



Bidirectional S_{bid}:



- S_f allows 2 mismatches in the second part
- S_b and S_{bid} allow only 1 mismatch in the second part

$\Rightarrow$ increasing $|P_1|$ may lead to a better performance

Properties of optimal schemes

Critical string of a search scheme : lexicographically maximal U-string among all searches

$\alpha(k, p)$: lexicographically minimal critical string among all search schemes for k mismatches and p parts

Lemma:

- $\alpha(k, k+1) = 013355...kk$ for odd k
- $\alpha(k, k+1) = 02244...kk$ for even k
- $\alpha(k, k+2) = 0123...(k-1)kk$

Properties of optimal schemes

Critical string of a search scheme : lexicographically maximal U-string among all searches

$\alpha(k, p)$: lexicographically minimal critical string among all search schemes for k mismatches and p parts

Lemma:

- $\alpha(k, k+1) = 013355...kk$ for odd k
- $\alpha(k, k+1) = 02244...kk$ for even k
- $\alpha(k, k+2) = 0123...(k-1)kk$

Critical string for $k+2$ parts is lexicographically smaller than for $k+1 \Rightarrow$ smaller enumeration trees

Computing a search scheme and an optimal partition

Optimal Search scheme is computed using a greedy algorithm

Computing a search scheme and an optimal partition

Optimal Search scheme is computed using a greedy algorithm

Optimal partition may be computed using

- *Naive way*:
  - count $\#str$ for all possible partitions (works well for small values of m and k)
- *Improved algorithm based on dynamic programming*:
  - time complexity $O(m^2 + (|S|Nk + mp)\sum_{i=1}^{p} C_{N-1}^{i-1})$ for a partition with p parts, $N = \lceil \log_\sigma n \rceil$

Experiments on genomic data

Text: human chromosome 14

Patterns for search: 1000000 substrings of the text with random mismatches

Experiments on genomic data: 2 mismatches

Table : Total time (in seconds) of searching for one million patterns in human chromosome 14

m	3 equal	3 unequal	4 equal	4 unequal
24	142	120 (85%)	117 (82%)	107 (75%)
30	101	84 (83%)	66 (65%)	68 (67%)
36	68	66 (97%)	49 (72%)	50 (74%)
42	45	45 (100%)	44 (98%)	38 (84%)

Experiments on genomic data: 2 mismatches

Table : Average number of enumerated strings

m	3 equal	3 unequal	4 equal	4 unequal
24	1049	882 (84%)	927 (88%)	816 (78%)
30	767	642 (84%)	523 (68%)	550 (71%)
36	538	529 (98%)	415 (77%)	432 (80%)
42	349	349 (100%)	359 (102%)	319 (91%)

Experiments on genomic data: 3 mismatches

Table : Total time (in seconds) of searching

m	4 equal	5 equal	5 unequal	
12	1442	1328 (92%)	1388 (96%)	1,2,6,1,2
15	2208	2061 (93%)	2095 (95%)	2,2,6,1,4
18	1698	1587 (93%)	1535 (90%)	4,2,6,1,5
21	1131	1006 (89%)	1033 (91%)	3,6,4,1,7

Experiments on genomic data: 3 mismatches

Table : Average number of enumerated strings

m	4 equal	5 equal	5 unequal
12	16463	15616 (95%)	15927 (97%)
15	31918	29288 (92%)	28658 (90%)
18	33141	27513 (83%)	26799 (81%)
21	27610	20488 (74%)	21442 (78%)

Summary of the results

- formalization of bidirectional *search schemes*
- two improvements confirmed by both analytical estimations and computational experiments:
  - partitioning the pattern into *unequal-size* parts; dynamic programming algorithm for designing partitions
  - using more than $k + 1$ parts

Future directions

- extend to the search to the *edit* distance
- simultaneous design of a search scheme and a partition

The end

Thank you!

Questions?