Fast Approximate Matching

Yong Kong
Keck Bioinformatics Resource
Yale University

Outline

- Introduction
- Distance functions
- Probability of an approximate match: an open problem
- Bit-vector algorithm
- Filter algorithms

Taxonomy

- Online search
  - Based on DP matrix
    - Worst case
    - Average case
  - Automaton
  - Filters
    - For moderate pattern
    - For very long pattern
  - Bit-vector
    - Based on automata
    - Based on DP matrix
- Indexed search
- Heuristic

Applications

- Computational biology: many applications (you cannot find many "exact" things in biology, so "approximate" is the rule), such as
  - Adaptor/vector trimming
  - Primer designs
  - siRNA design/microRNA finding
  - Mapping
  - Basic operations for other applications, such as sequence assembly, motif finding, etc.
- Other fields (signal processing, text retrieval, etc.)

Distance functions

- Given two sequence, how similar (or dissimilar) they are? What is the distance between them?
- Hamming distance (only substitutions are allowed)
- Edit distance (substitutions, insertions, deletions)
- Substitution matrix

Hamming distance

- Hamming distance: number of different positions

  ATTGTC

  ACTCTC

  x x

- Applies to ungapped alignments of two sequences with equal length
- The smaller the distance, the closer the sequences: we want minimize the distance

Edit distance (Levenshtein distance)

- Minimal number of editing operations to transform one sequence to another
- Editing operations are insertion, deletion, and substitution
- If all the operations cost 1: simple edit distance
- If different operations have different costs or the costs depend on the characters involved: general edit distance

Edit distance

```
-TGC-ATAT
ATCCGAT--
```

Edit distance = 1 insertion (A)

+ 1 substitution (G->C)

+ 1 insertion (G)

+ 2 deletions (AT)

= 5

- We can use 5 operations to change "TGCATAT" into "ATCCGAT", so the edit distance is *at most* 5.

Edit distance

```
-TGCATAT
ATCC-GAT
```

Edit distance = 1 insertion (A)

+ 1 substitution (G->C)

+ 1 deletion (A)

+ 1 substitution (T->G)

= 4

Edit distance

- Actually, This is another “optimal” alignment with edit distance as 4:

```
TGCATAT
ATCCGAT
```

Edit distance = 4 substitutions

(T->A, G->T, A->C, T->G)

= 4

- Any other alignments with edit distance as 4?
- How can we find out?

How many alignment are there?

- For two sequences of lengths m and n, the number of all possible alignments is

$$B = \binom{m+n}{m}$$

- If m = n, then

$$B \sim 2^{2n}/(\pi n)^{1/2}$$

- B grows exponentially.

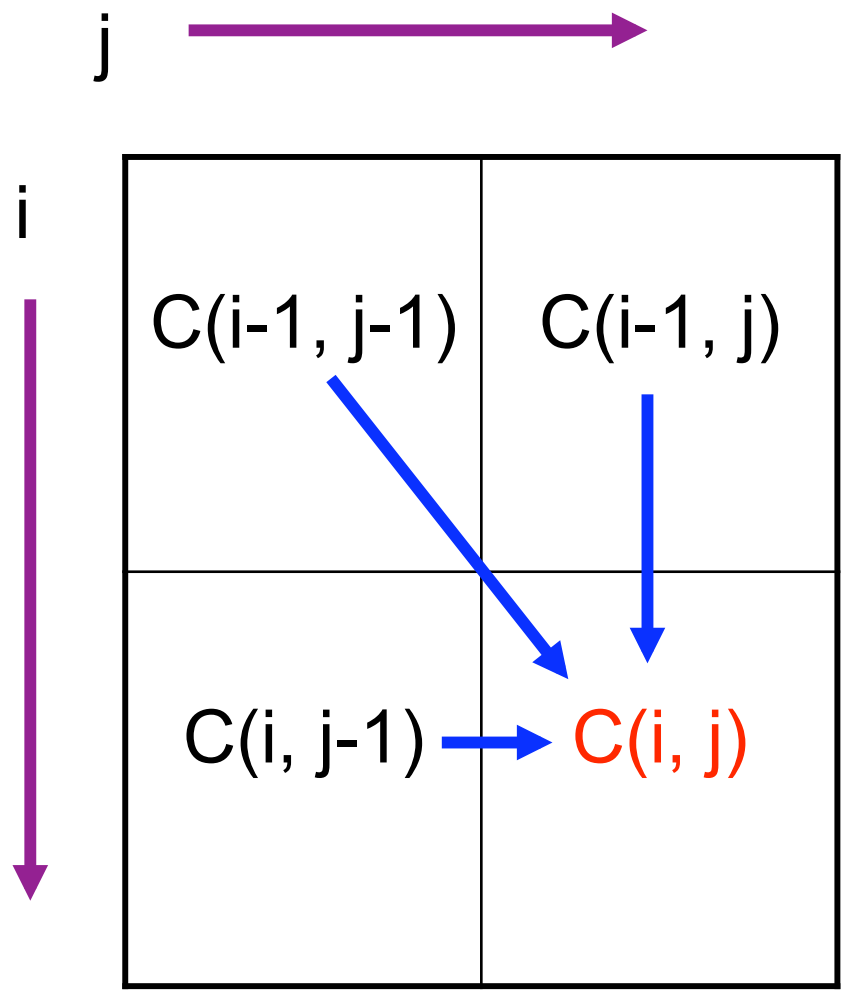
Many, many alignments

- Suppose m=1000, n=1000, we have
- $B = 2^{2000}/(1000\pi)^{1/2} \sim 10^{600}$

- Not feasible to exam all of them by using naïve enumeration
- The classical method: dynamic programming

Dynamic programming

- C(i, j) = min{
  C(i-1, j-1) +
  (pi == tj ? 0 : 1),
  C(i-1, j) + 1,
  C(i, j-1) + 1
  }
- The three neighboring cells in the up-left directions are used to update C(i,j)



Initial conditions

		T	G	C	A	T	A	T
	0	1	2	3	4	5	6	7
A	1							
T	2							
C	3							
C	4							
G	5							
A	6							
T	7							

First step…

		T	G	C	A	T	A	T
	0	1	2	3	4	5	6	7
A	1	1						
T	2							
C	3							
C	4							
G	5							
A	6							
T	7							

Next…

		T	G	C	A	T	A	T
	0	1	2	3	4	5	6	7
A	1	1	2					
T	2	1						
C	3							
C	4							
G	5							
A	6							
T	7							

The final matrix

		T	G	C	A	T	A	T
	0	1	2	3	4	5	6	7
A	1	1	2	3	3	4	5	6
T	2	1	2	3	4	3	4	5
C	3	2	2	2	3	4	4	5
C	4	3	3	2	3	4	5	5
G	5	4	3	3	3	4	5	6
A	6	5	4	4	3	4	4	5
T	7	6	5	5	4	3	4	4

Approximate occurrences

- This is another important variant of global alignment
- Suppose we have *a short pattern P*, and *a long sequence T*
- Given a parameter t, a substring T' of T is said to be an approximate occurrence of P if and only if the optimal alignment of P to T' has value at most t – (for edit distance, etc.)

Approximate occurrences

- Suppose x is the long target sequence, y is the short pattern. The only change for the algorithm is the initial condition:

$$C(0, j) = 0$$

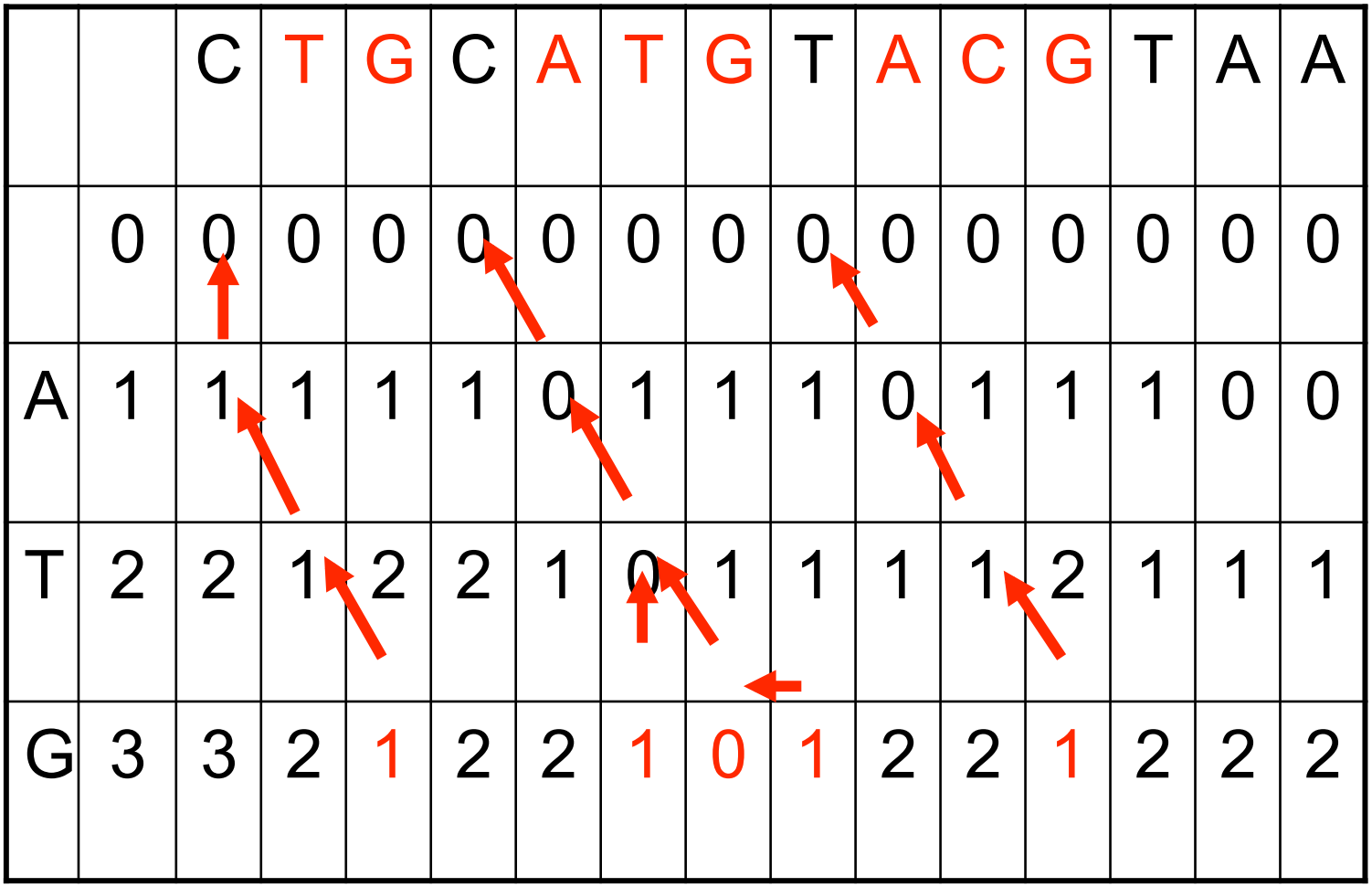
- There is an approximate occurrence ending at position j if and only if

$$C(n, j) <= t \text{ (for edit distance)}$$

- The substring is at [k..j], where k is determined by a path of traceback from [n,j] to [0,k].

Approximate occurrences

- A toy example: t = 1 (edit distance)



		C	T	G	C	A	T	G	T	A	C	G	T	A	A
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	1	1	1	1	1	0	1	1	1	0	1	1	1	0	0
T	2	2	1	2	2	1	0	1	1	1	1	2	1	1	1
G	3	3	2	1	2	2	1	0	1	2	2	1	2	2	2

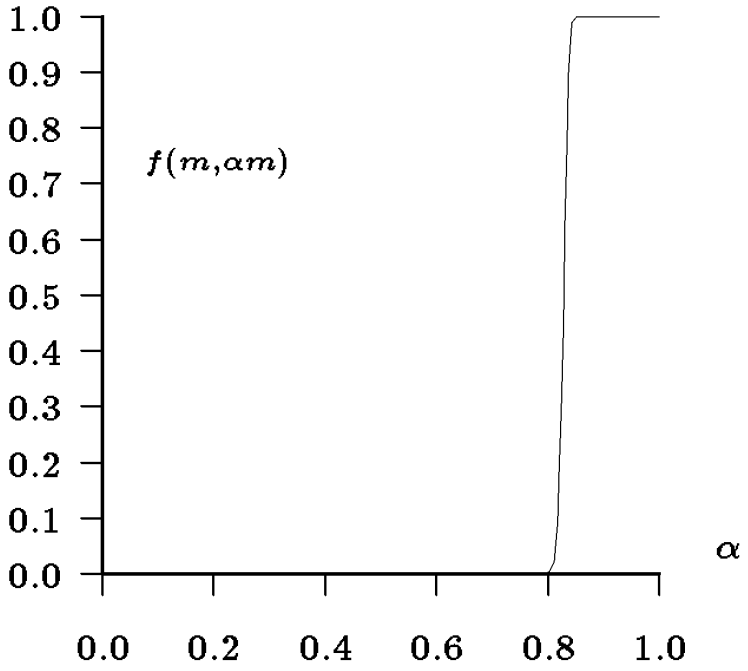
Time complexity

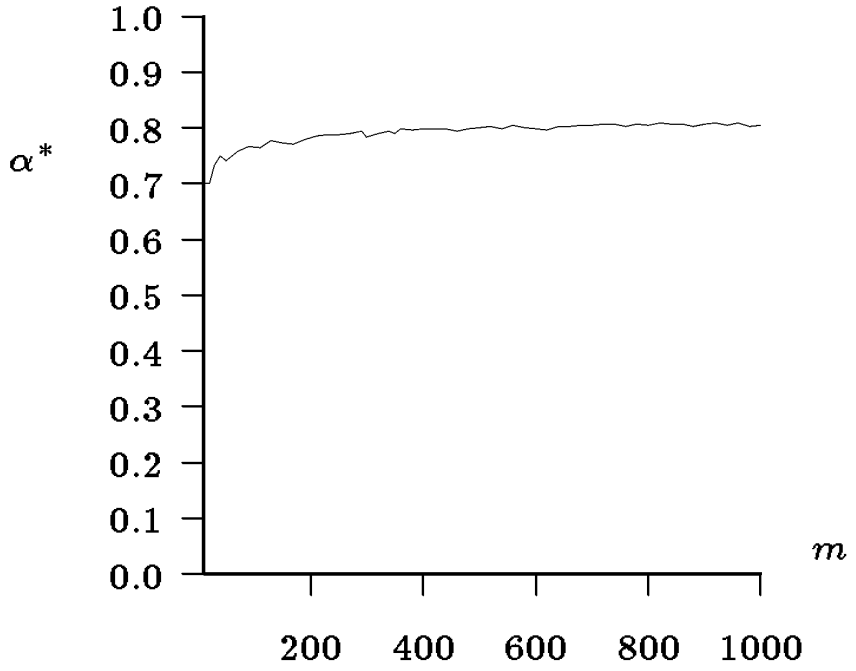
- From the algorithm, we see that we have (n +1) x (m+1) numbers in the matrix, m+n+1 of which are initial conditions, and the rest n x m can be calculated in constant time (three sums and a max/min), so the time complexity is O(mn).
- Compared with exponential number of alignments, this is a significant improvement!
- But this can be proved to O(n)!

Matching probability

- What is the probability of a match with <= k errors?
- It is still an open problem. Only rough simulation results for equal letter probability.
- The probability has a "phase-transition" like behavior: when error level α=k/m is below α^*, there are few matches; when $\alpha > \alpha^*$, there are a lot of matches
- Simulation results: $\alpha^* = 1 - 1.09/\sqrt{\sigma}$

Matching probability

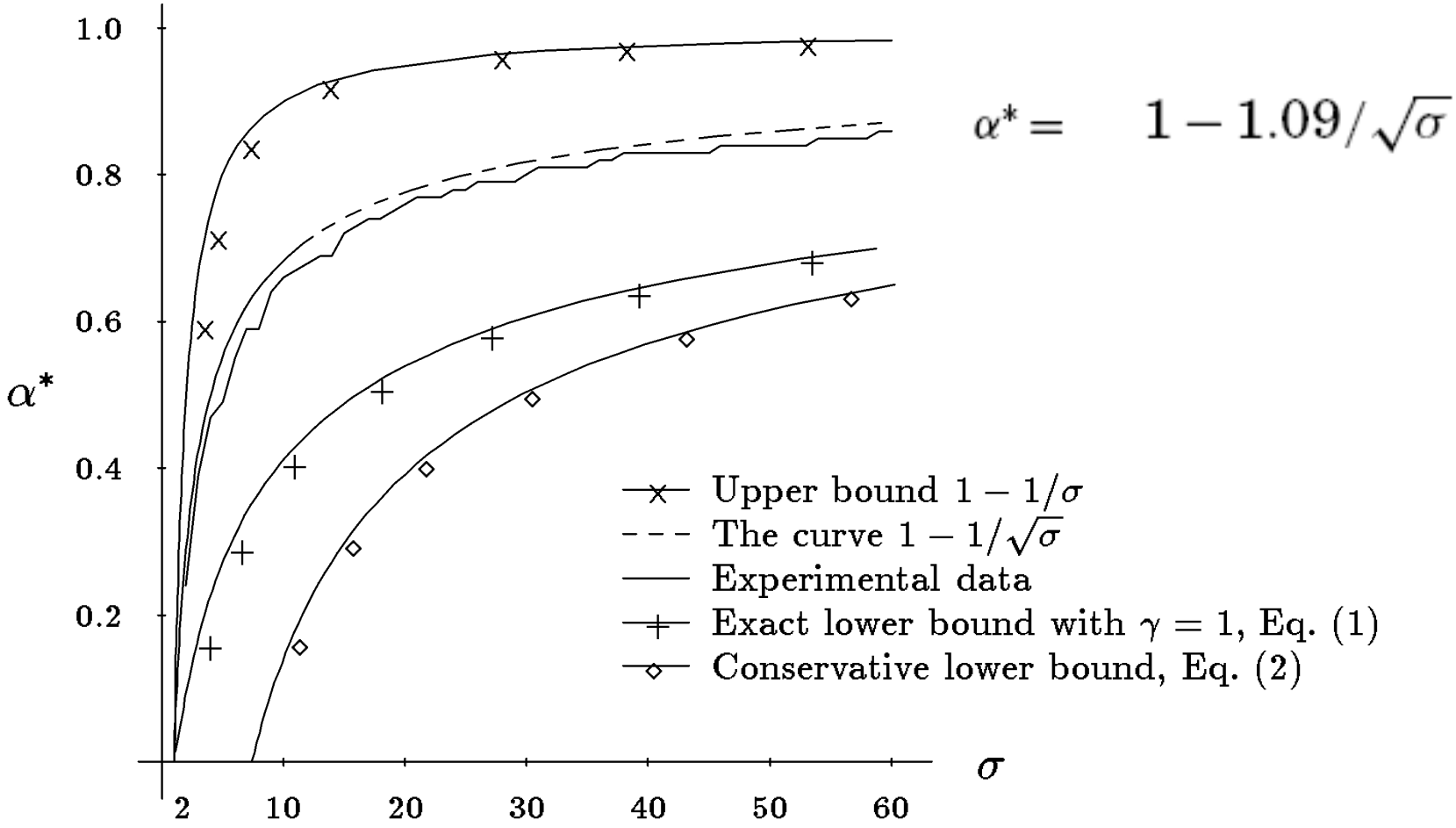




Simulation results using equal letter probability, n=10MB

with $\sigma = 32$.

Matching probability



Some (computer) terminology

- Bit, byte, word
  - A bit is a binary digit, taking a value of either 0 or 1.
  - A byte is usually 8 bits (a unit of memory addressing)
  - A machine word is usually 4 bytes (32-bit machine), or 8 bytes (64-bit machine)
  - A machine word is “the natural unit of data” for numerical calculations

Word sizes

- On a typical 32-bit machine:

short	int	long	long long
2	4	4	8

- On a typical 64-bit machine:

short	int	long	long long
2	4	8	8

Bitwise operators

- Bitwise NOT (~) `~(01001001) = 10110110`
- Bitwise AND (&)

```
01001001 &
10111010 =
--------
00001000
```

- Bitwise OR (|)

```
01001001 |
10111000 =
--------
11111001
```

Bitwise operators

- Bitwise Exclusive-Or (XOR) (^)

```
 01110011 ^
10101010
--------
11011001
```

- Bitwise shift (<< and >>)

```
11011001 << 2 = 01100100
11011001 >> 2 = 00110110
```

Bit-vector algorithms

- Bit-vector algorithms take advantage of the parallelism of bitwise operators
- For one operation (such as AND), 32 or 64 bit are changed simultaneously
- These algorithms are also called bit-parallelism algorithms
  - Wu and Manber (1992), Baeza-Yates and Gonnet (1992): automata
  - Myers (1999): dp matrix

Highlights of Myers algorithm

- Basic algorithm: O(n [m/w])
  - Independent of k, the error threshold
  - When w < m, O(n)
- Extended algorithm: O(kn/w)
  - Use basic algorithm as a subroutine to calculate w rows in dp matrix at a time
- Easy to extend to search set of letters, wild cards, etc., which are commonly encounter in Bioinformatics, at no additional cost
- Elegant and code is short

Bit-vector algorithms

- One of the key insights is that for edit distance, the dp matrix has the following property:

The difference between adjacent cells in any row or column is either 1, 0, or -1.

Relocatable dp matrix

- So what the algorithm calculates is the difference of the dp matrix (relocatable dp matrix):

$$\Delta h(i,j) = C(i, j) - C(i, j-1)$$

$$\Delta v(i,j) = C(i, j) - C(i-1, j)$$

- Since each $\Delta h(i,j)$ and $\Delta v(i,j)$ can take 3 values, 2 bit-vectors are needed:

$$Pv_j(i) \equiv (\Delta v[i, j] = +1)$$

$$Mv_j(i) \equiv (\Delta v[i, j] = -1)$$

Difference between adjacent cells

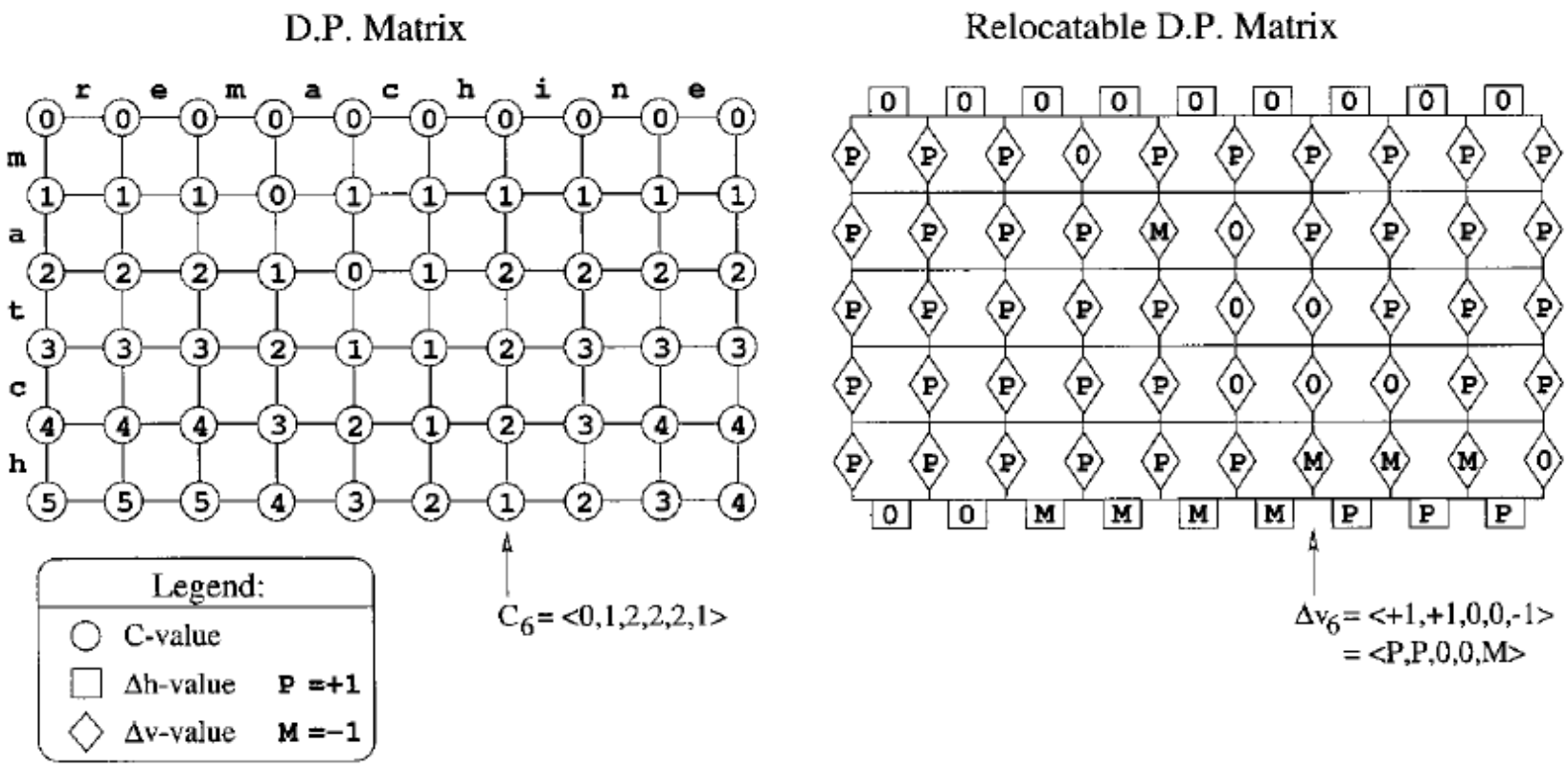


FIG. 1. Dynamic programming (d.p.) matrices for P = *match* and T = *remachine*.

DP expressed as deltas

$$\Delta v[i,j] = \min\{-Eq[i,j], \Delta v[i,j-1], \Delta h[i-1,j]\} + (1 - \Delta h[i-1,j])$$

$$\Delta h[i,j] = \min\{-Eq[i,j], \Delta v[i,j-1], \Delta h[i-1,j]\} + (1 - \Delta v[i,j-1])$$

$$Eq_j(i) \equiv (p_i = t_j)$$

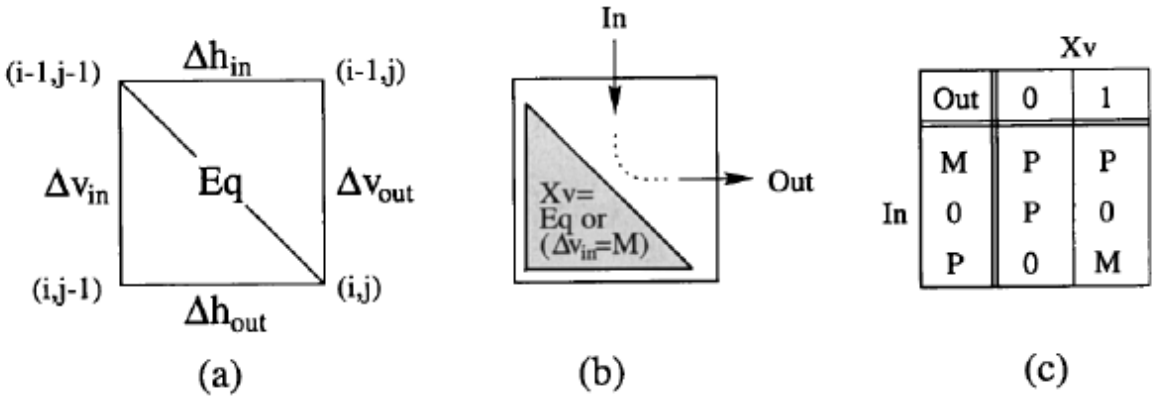


FIG. 2. D.P. cell structure and input/output function.

DP expressed as deltas

- There are 2x3x3=18 possible inputs for each equation; by a brute force enumeration, the following can be obtained:

$$
\begin{aligned}
Xv &= Eq \text{ or } Mv_{in} \\
Pv_{out} &= Mh_{in} \text{ or not } (Xv \text{ or } Ph_{in}) \\
Mv_{out} &= Ph_{in} \text{ and } Xv
\end{aligned}
\qquad (4a)
$$

In \ Xv	0	1
M	P	P
0	P	0
P	0	M

$$
\begin{aligned}
Xh &= Eq \text{ or } Mh_{in} \\
Ph_{out} &= Mv_{in} \text{ or not } (Xh \text{ or } Pv_{in}) \\
Mh_{out} &= Pv_{in} \text{ and } Xh
\end{aligned}
\qquad (4b)
$$

Text preprocessing

- We need pre-process the text to get an integer Eqj for each text position j in O(1) time. To do this, build a table of vectors for all possible text characters based on the pattern m.
- For example, for pattern `TGCATAT`, the table would be

	T	A	T	A	C	G	T
A		1		1			
C					1		
G						1	
T	1						1

σ

m

Scanning step

- Goal: given the vertical delta on the left, the scan text one letter a time, to obtain the vertical delta on the right
- Formulas we have now are mix of horizontal and vertical deltas.
- Two-stage scan has to be used:
  - Vertical left → horizontal bottom
  - Horizontal top → vertical right

Scanning step

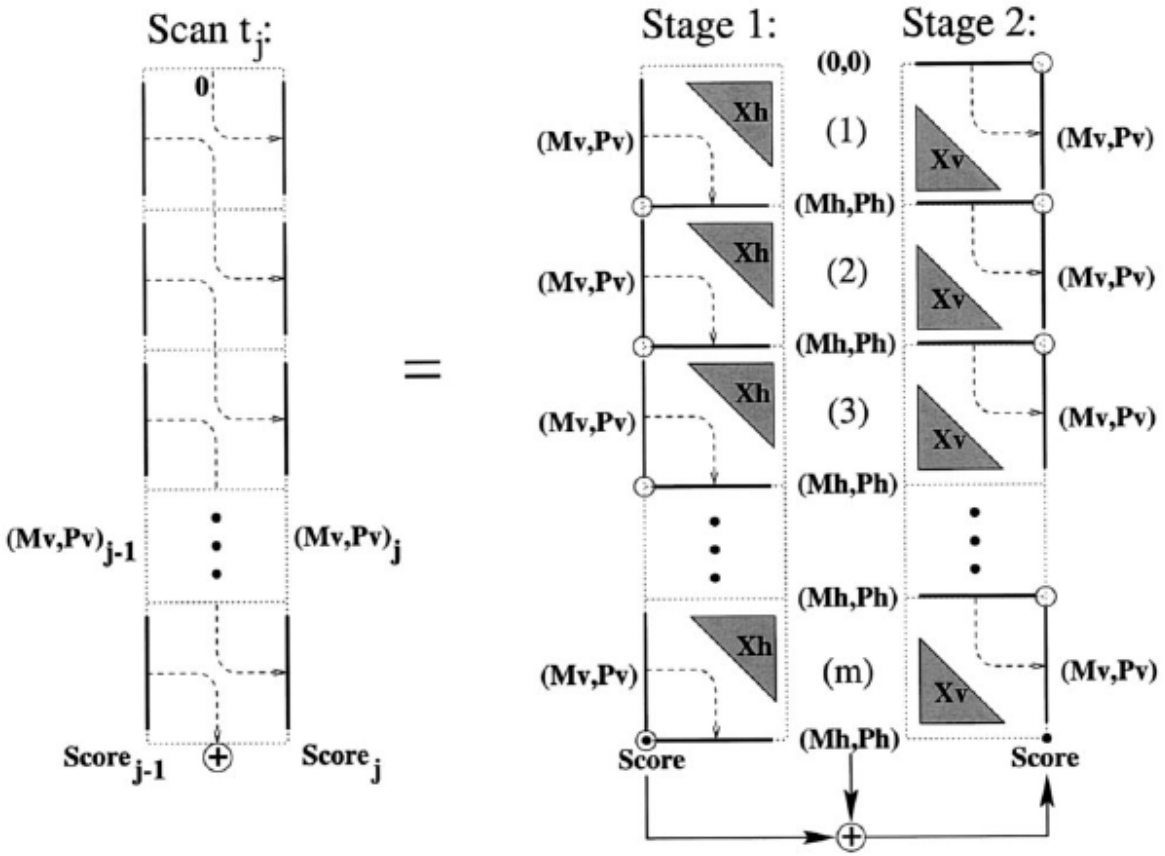


FIG. 3. The two stages of a scanning step.

The X-factors

- The two X-factors are:

$$Xv_j(i) = Peq[t_j](i) \text{ or } Mv_{j-1}(i)$$
$$Xh_j(i) = Peq[t_j](i) \text{ or } Mh_j(i-1)$$

- There is no problem for Xv_j: since Mv_{j-1} is the input.
- For Xh_j, however, is difficult: we need Mh_j to calculate Xh_j, but to calculate Mh_j we need Xh_j. To unwind the loop is the most difficulty part of the algorithm.

Put together

- The result is

$$Xh_j = (((Peq[t_j] \& Pv_{j-1}) + Pv_{j-1}) \hat{} Pv_{j-1}) | Peq[t_j]$$

- The score is calculated as

$$Score_j = Score_{j-1} + (1 \text{ if } Ph_j(m)) - (1 \text{ if } Mh_j(m))$$

- The initial conditions:

$$\begin{aligned} Pv_0(i) &= 1 \\ Mv_0(i) &= 0 \\ Score_0 &= m \end{aligned}$$

The code

```
1.   Precompute Peq[σ]                                  Pre-processing
2.   Pv = 1^m                                           Initial conditions
3.   Mv = 0
4.   Score = m
5.   for j = 1, 2, ...n do                             Main loop
6.    { Eq = Peq[t_j]
7.      Xv = Eq | M                                     X-factors
8.      Xh = (((Eq & Pv) + Pv) ^ Pv) | Eq
9.      Ph = Mv | ~ (Xh | Pv)                           Stage 1
10.     Mh = Pv & Xh
11.     if Ph & 10^(m-1) then                           Score
12.        Score += 1
13.     else if Mh & 10^(m-1) then
14.        Score -= 1
15.     Ph <<= 1                                        Stage 2
16.     Mh <<= 1
17.     Pv = Mh | ~ (Xv | Ph)
18.     Mv = Ph & Xv
19.     if Score ≤ k then
20.        print "Match at " · j
      }
```

Extension

- Due to the structure of the algorithms, some extensions are easy, some are not.
- It is easy to search a set of letters. The only change is in the pre-processing step.
- For example, if the pattern is `T[AG]CATAT` (either A or G in the second position):

	T	A	T	A	C	G	T
A		1		1		1	
C					1		
G						1	
T	1						1

Filtering algorithms

- Key idea: it is easier to tell that a text position *does not match* than to tell that it matches
- Filtering: to discard areas of the text that cannot contain a match
- The filtering algorithms themselves cannot discover the matches; a non-filtering method is needed to verify the potential positions
- Sensitive to error level: α=k/m, work better at lower α

Filtering algorithms

- Algorithm 1: if a pattern is cut in k + 1 pieces, then at least one of the pieces must appear unchanged in an approximate match with <= k errors: k errors cannot change k+1 pieces
- Example: if neither “fil” nor “ter” appear in the text, then “filter” cannot occur with <=1 error
- Cut the pattern into k+1 pieces, search the pieces exactly, then verify in the neighbors of the matches

Filtering algorithms

- Algorithms 2: in a sliding window along the text, count the number of letters that belong to the pattern
- For any matches with <= k errors, at least m-k letters belong to the pattern