Bioinformatics Algorithms

(Fundamental Algorithms, module 2)

Zsuzsanna Lipták

Masters in Medical Bioinformatics
academic year 2017/18, spring term

Pairwise Alignment

Alignments

Alignment

- a way of visualizing similarities and differences between two strings
- we want to find a good way of doing this

Alignments

Alignment

- a way of visualizing similarities and differences between two strings
- we want to find a good way of doing this

Ex: five different alignments of $s = \texttt{ACCT}$ and $t = \texttt{CAT}$

-ACCT	ACCT	ACCT	ACC-T	---ACCT
CA--T	-CAT	CAT-	--CAT	CAT----

Alignments

Alignment

- a way of visualizing similarities and differences between two strings
- we want to find a good way of doing this

Ex: five different alignments of $s = \texttt{ACCT}$ and $t = \texttt{CAT}$

-ACCT	ACCT	ACCT	ACC-T	---ACCT
CA--T	-CAT	CAT-	--CAT	CAT----

Formal definition

An alignment $\mathcal{A}$ of $s, t \in \Sigma^*$ is a matrix with two rows, entries from $\Sigma \cup \{\underbrace{-}_{\text{gap}}\}$, s.t.

1. deleting all gaps from the first row yields s
2. deleting all gaps from the second row yields t
3. no column consists of two gaps

Scoring alignments

scoring function

- score of a column: match (same char), mismatch (diff. chars), gap
- score of $\mathcal{A}$ = sum of column scores

Ex.

match	mismatch	gap
2	-1	-1

-ACCT	ACCT	ACCT	ACC-T	---ACCT
CA--T	-CAT	CAT-	--CAT	CAT----

Scoring alignments

scoring function

- score of a column: match (same char), mismatch (diff. chars), gap
- score of $\mathcal{A}$ = sum of column scores

Ex.

match	mismatch	gap
2	-1	-1

-ACCT	ACCT	ACCT	ACC-T	---ACCT
CA--T	-CAT	CAT-	--CAT	CAT----
1	2	-4	1	-7

Scoring alignments

scoring function

- score of a column: match (same char), mismatch (diff. chars), gap
- score of $\mathcal{A}$ = sum of column scores

Ex.

match	mismatch	gap
2	-1	-1

`-ACCT`	`ACCT`	`ACCT`	`ACC-T`	`---ACCT`
`CA--T`	`-CAT`	`CAT-`	`--CAT`	`CAT----`
1	2	-4	1	-7

N.B.: Remember that these values depend on the scoring function!

Scoring alignments

So acc. to our scoring function, alignment 2 is the best (of the five)!

-ACCT CA--T	ACCT -CAT	ACCT CAT-	ACC-T --CAT	---ACCT CAT----
1	2	-4	1	-7

But is it best possible?

Optimal alignments

Def.

An optimal alignment of s and t is an alignment $\mathcal{A}$ with maximum score, i.e. an alignment $\mathcal{A}$ s.t.

$$score(\mathcal{A}) = \max\{score(\mathcal{A}') \ : \ \mathcal{A}' \text{ is an alignment of } s \text{ and } t\}$$

Optimal alignments

Def.

An optimal alignment of s and t is an alignment $\mathcal{A}$ with maximum score, i.e. an alignment $\mathcal{A}$ s.t.

$$score(\mathcal{A}) = \max\{score(\mathcal{A}') \ : \ \mathcal{A}' \text{ is an alignment of } s \text{ and } t\}$$

Def.

Given $s, t \in \Sigma^*$ and scoring function f, the similarity of s and t, is

$$\begin{aligned} sim(s,t) &= \text{score of an optimal alignment} \\ &= \max\{score(\mathcal{A}) \ : \ \mathcal{A} \text{ is an alignment of } s \text{ and } t\} \end{aligned}$$

Our computational problem: Global alignment

Problem variant 1

Input: Two strings s, t over alphabet Σ, scoring function f.

Output: $sim(s, t)$.

Problem variant 2

Input: Two strings s, t over alphabet Σ, scoring function f.

Output: An optimal alignment of s and t.

N.B.: In variant 1, we want only a number, we are not interested in an optimal alignment itself.

Our computational problem: Global alignment

For now, let's concentrate on Variant 1 (i.e. only $sim(s, t)$ is sought).

Global alignment

Input: Two strings s, t over alphabet Σ, scoring function f.

Output: $sim(s, t)$.

Our computational problem: Global alignment

For now, let's concentrate on Variant 1 (i.e. only $sim(s, t)$ is sought).

Global alignment

Input: Two strings s, t over alphabet Σ, scoring function f.

Output: $sim(s, t)$.

We will see two algorithms for this problem.

Exhaustive search

Algorithm 1: Exhaustive search

1. consider every possible alignment of s and t
2. for each of these, compute its score
3. output the maximum of the scores computed

Algorithm *Exhaustive search for global alignment*
Input: strings s, t, with $|s| = n, |t| = m$; scoring function f
Output: value $sim(s, t)$

```
1.   int max = (n + m)g;                          //g is the cost of a gap
2.   for each alignment A of s and t (in some order)
3.        do if score(A) > max
4.             then max ← score(A);
5.   return max;
```

Note:

1. The variable *max* is needed for storing the highest score so far seen.
2. The initial value of *max* is the score of *some* alignment of s, t (which one?)

Number of alignments

List all alignments of $s = \texttt{AC}$ and $t = \texttt{GA}$.

Number of alignments

List all alignments of $s = \texttt{AC}$ and $t = \texttt{GA}$.

You should have got these 13 al's:

```
-AC     A-C     --AC     A--C     -A-C
GA-     GA-     GA--     -GA-     G-A-

 AC      A-C      -AC
 GA      -GA      G-A

AC-     AC-     AC--     -AC-     A-C-
-GA     G-A     --GA     G--A     -G-A
```

Number of alignments

Question

How many alignments are there in general for two strings s and t?

Number of alignments

Question

How many alignments are there in general for two strings s and t?

Observation

The number of alignments depends only on the length of s and t.

Number of alignments

Question

How many alignments are there in general for two strings s and t?

Observation

The number of alignments depends only on the length of s and t.

Def.

Let $N(n, m) =$ number of al's of two strings of length n and m.

We know:

- $N(2, 2) = 13$
- $N(1, 1) =$

Number of alignments

Question

How many alignments are there in general for two strings s and t?

Observation

The number of alignments depends only on the length of s and t.

Def.

Let $N(n, m)$ = number of al's of two strings of length n and m.

We know:

- $N(2, 2) = 13$
- $N(1, 1) = 3$
- $N(n, 0) =$

Number of alignments

Question

How many alignments are there in general for two strings s and t?

Observation

The number of alignments depends only on the length of s and t.

Def.

Let $N(n, m)$ = number of al's of two strings of length n and m.

We know:

- $N(2, 2) = 13$
- $N(1, 1) = 3$
- $N(n, 0) = 1,\ N(0, m) = 1$

Number of alignments

Question

How many alignments are there in general for two strings s and t?

Observation

The number of alignments depends only on the length of s and t.

Def.

Let $N(n, m)$ = number of al's of two strings of length n and m.

We know:

- $N(2, 2) = 13$
- $N(1, 1) = 3$
- $N(n, 0) = 1,\ N(0, m) = 1$
- we set: $N(0, 0) = 1$ (empty alignment)

Number of alignments

$N(n, m)$	0	1	2	3	4	5
0	1	1	1	1	1	1
1	1	3				
2	1		13			
3	1					
4	1					
5	1					

Number of alignments

Look at the last column of the alignments:

```
-AC   A-C   --AC   A--C   -A-C
GA-   GA-   GA--   -GA-   G-A-

 AC    A-C   -AC
 GA    -GA   G-A

AC-   AC-   AC--   -AC-   A-C-
-GA   G-A   --GA   G--A   -G-A
```

Number of alignments

We have a recursive formula:

- $N(n, 0) = N(0, m) = 1$ for $n, m \geq 0$
- and for $n, m > 0$:

$$N(n, m) = N(n - 1, m) + N(n - 1, m - 1) + N(n, m - 1)$$

Number of alignments

$N(n,m)$	0	1	2	3	4	5
0	1	1	1	1	1	1
1	1	3				
2	1		13			
3	1					
4	1					
5	1					

Number of alignments

$N(n,m)$	0	1	2	3	4	5
0	1	1	1	1	1	1
1	1	3	5			
2	1		13			
3	1					
4	1					
5	1					

Number of alignments

$N(n,m)$	0	1	2	3	4	5
0	1	1	1	1	1	1
1	1	3	5			
2	1	5	13			
3	1					
4	1					
5	1					

Number of alignments

$N(n,m)$	0	1	2	3	4	5
0	1	1	1	1	1	1
1	1	3	5	7		
2	1	5	13			
3	1					
4	1					
5	1					

Number of alignments

$N(n,m)$	0	1	2	3	4	5
0	1	1	1	1	1	1
1	1	3	5	7		
2	1	5	13	25		
3	1					
4	1					
5	1					

Number of alignments

$N(n,m)$	0	1	2	3	4	5
0	1	1	1	1	1	1
1	1	3	5	7	9	11
2	1	5	13	25	41	61
3	1	7	25	63	129	231
4	1	9	41	129	321	681
5	1	11	61	231	681	1683

Number of alignments

Let's look at the case $n = m$:

n	0	1	2	3	4	5	...	1000
$N(n, n)$	1	3	13	63	321	1683	...	$\approx 10^{767}$

Number of alignments

Let's look at the case $n = m$:

n	0	1	2	3	4	5	...	1000
$N(n, n)$	1	3	13	63	321	1683	...	$\approx 10^{767}$

In fact, it can be shown that $N(n, n)$ grows exponentially.

Number of alignments

Let's look at the case $n = m$:

n	0	1	2	3	4	5	...	1000
$N(n, n)$	1	3	13	63	321	1683	...	$\approx 10^{767}$

In fact, it can be shown that $N(n, n)$ grows exponentially.

Running time of exhaustive search:

For any al. $\mathcal{A}$, we have $\max(n, m) \leq |\mathcal{A}| \leq (n + m)$, thus:

$$N(n, m) \cdot \max(n, m) \leq \text{no. of steps of algo.} \leq N(n, m) \cdot (n + m)$$

Therefore, it has exponential running time: too slow!

A Dynamic Programming Algorithm

Dynamic Programming

- is a class of algorithms (like greedy, divide and conquer, ...)
- applicable when solution can be constructed from solutions of subproblems
- subproblem solutions re-used several times
- uses a matrix ("DP-table") for storing subproblem solutions

Smaller subproblems

Crucial idea

If $\mathcal{A}$ is an optimal alignment, then $\mathcal{B}$, the same alignment without the last column, is also optimal.

Proof

By contradiction (see board).

Smaller subproblems

Crucial idea

If $\mathcal{A}$ is an optimal alignment, then $\mathcal{B}$, the same alignment without the last column, is also optimal.

Proof

By contradiction (see board).

So we will compute the scores of optimal alignments of all pairs of prefixes of s and t, and construct an optimal alignment from that!

The DP-table

Algorithm 2: Needleman-Wunsch algorithm for global alignment

- construct a DP-table D of size $(n+1) \times (m+1)$ s.t.

$$D(i,j) = sim(s_1 \dots s_i, t_1 \dots t_j)$$

  (We will see in a moment how!)
- return $D(n,m)$

Constructing solutions from smaller subproblems

Look at an alignment of s and t. There are 3 cases:

1. last column is $\binom{\mathtt{s_n}}{-}$
2. last column is $\binom{\mathtt{s_n}}{\mathtt{t_m}}$
3. last column is $\binom{-}{\mathtt{t_m}}$

Constructing solutions from smaller subproblems

Look at an alignment of s and t. There are 3 cases:

1. last column is $\binom{\mathtt{s_n}}{-}$
2. last column is $\binom{\mathtt{s_n}}{\mathtt{t_m}}$
3. last column is $\binom{-}{\mathtt{t_m}}$

Recall that if $\mathcal{A}$ is optimal, then so is $\mathcal{B} = (\mathcal{A}$ without last column)!

Constructing solutions from smaller subproblems

Look at an alignment of s and t. There are 3 cases:

1. last column is $\binom{\mathtt{s_n}}{-}$
2. last column is $\binom{\mathtt{s_n}}{\mathtt{t_m}}$
3. last column is $\binom{-}{\mathtt{t_m}}$

Recall that if $\mathcal{A}$ is optimal, then so is $\mathcal{B} = (\mathcal{A}$ without last column)!

- in case 1, $\mathcal{B}$ is an opt. al. of $\mathtt{s_1 ... s_{n-1}}$ and $\mathtt{t_1 ... t_m}$
- in case 2, $\mathcal{B}$ is an opt. al. of $\mathtt{s_1 ... s_{n-1}}$ and $\mathtt{t_1 ... t_{m-1}}$
- in case 3, $\mathcal{B}$ is an opt. al. of $\mathtt{s_1 ... s_n}$ and $\mathtt{t_1 ... t_{m-1}}$

Constructing solutions from smaller subproblems

So to compute $sim(s, t) = D(n, m)$, we need to know

- $sim(\mathtt{s_1...s_{n-1}}, \mathtt{t_1...t_m})$
- $sim(\mathtt{s_1...s_{n-1}}, \mathtt{t_1...t_{m-1}})$
- $sim(\mathtt{s_1...s_n}, \mathtt{t_1...t_{m-1}})$

and add the score of the last column!

Constructing solutions from smaller subproblems

So to compute $sim(s, t) = D(n, m)$, we need to know

- $sim(\mathtt{s_1 ... s_{n-1}}, \mathtt{t_1 ... t_m}) = D(n-1, m)$
- $sim(\mathtt{s_1 ... s_{n-1}}, \mathtt{t_1 ... t_{m-1}}) = D(n-1, m-1)$
- $sim(\mathtt{s_1 ... s_n}, \mathtt{t_1 ... t_{m-1}}) = D(n, m-1)$

and add the score of the last column!

Constructing solutions from smaller subproblems

So to compute $sim(s, t) = D(n, m)$, we need to know

- $sim(\mathtt{s_1 ... s_{n-1}}, \mathtt{t_1 ... t_m}) = D(n-1, m)$
- $sim(\mathtt{s_1 ... s_{n-1}}, \mathtt{t_1 ... t_{m-1}}) = D(n-1, m-1)$
- $sim(\mathtt{s_1 ... s_n}, \mathtt{t_1 ... t_{m-1}}) = D(n, m-1)$

and add the score of the last column!

$$D(n,m) = \max \begin{cases} D(n-1,m) + gap \\ D(n-1,m-1) + \begin{cases} match & \text{if } s_n = t_m \\ mismatch & \text{if } s_n \neq t_m \end{cases} \\ D(n,m-1) + gap \end{cases}$$

Constructing solutions from smaller subproblems

Now we can compute all entries of D:

- $D(i,0) = i \cdot gap \quad$ for $i \geq 0$
- $D(0,j) = j \cdot gap \quad$ for $j \geq 0$
- recursion (for $i,j > 0$):

$$D(i,j) = \max \begin{cases} D(i-1,j) + gap \\ D(i-1,j-1) + \begin{cases} match & \text{if } s_i = t_j \\ mismatch & \text{if } s_i \neq t_j \end{cases} \\ D(i,j-1) + gap \end{cases}$$

Recall $s =$ ACCT, $t =$ CAT

match: 2, mismatch: -1, gap: -1

$D(i,j)$			C	A	T
		0	1	2	3
	0	0	-1	-2	-3
A	1	-1			
C	2	-2			
C	3	-3			
T	4	-4			

Recall $s =$ ACCT, $t =$ CAT

match: 2, mismatch: -1, gap: -1

$D(i,j)$			C	A	T
		0	1	2	3
	0	0	-1	-2	-3
A	1	-1	-1		
C	2	-2			
C	3	-3			
T	4	-4			

$D(1,1) = \max\{-1-1, 0-1, -1-1\} = -1$

Recall $s =$ ACCT, $t =$ CAT

match: 2, mismatch: -1, gap: -1

$D(i,j)$			C	A	T
		0	1	2	3
	0	0	-1	-2	-3
A	1	-1	-1	1	
C	2	-2			
C	3	-3			
T	4	-4			

$D(1,1) = \max\{-1-1, 0-1, -1-1\} = -1$ $D(1,2) = \max\{-2-1, -1+2, -1-1\} = 1$

$s = \texttt{ACCT}$, $t = \texttt{CAT}$ match: 2, mismatch: -1, gap: -1

$D(i,j)$			C	A	T
		0	1	2	3
	0	0	−1	−2	−3
A	1	−1	−1	1	0
C	2	−2	1	0	0
C	3	−3	0	0	1
T	4	−4	−1	−1	2

Needleman-Wunsch DP algorithm for global alignment

Variant which outputs $sim(s, t)$ only.

Algorithm *DP algorithm for global alignment*
Input: strings s, t, with $|s| = n, |t| = m$; scoring function f
Output: value $sim(s, t)$

1. **for** $j = 0$ to m **do** $D(0, j) \leftarrow j \cdot g$;
2. **for** $i = 1$ to n **do** $D(i, 0) \leftarrow i \cdot g$;
3. **for** $i = 1$ to n **do**
4. **for** $j = 1$ to m **do**
5. $$D(i,j) \leftarrow \max \begin{cases} D(i-1,j) + g \\ D(i-1,j-1) + f(s_i, t_j) \\ D(i,j-1) + g \end{cases}$$
6. **return** $D(n, m)$;

Needleman-Wunsch DP algorithm for global alignment

- Algorithm first introduced by Needleman & Wunsch (1970).
- Different orders of computation are possible: necessary to compute $D(i-1,j)$, $D(i-1,j-1)$, and $D(i,j-1)$ before $D(i,j)$
- Time: $O(n \cdot m)$
  (initialize first row and column in constant time, for the remaining $n \cdot m$ cells, we have 3 lookups and additions, so a constant number of operations)
- Space: $O(n \cdot m)$
  (matrix of size $(n+1)(m+1)$)
- for $n = m$, we get time and space $O(n^2)$, hence this is called a quadratic (time and space) algorithm
- Space-saving variant exists (later)

Finding an optimal alignment

Recall Variant 2: not only $sim(s, t)$, but also an optimal alignment.

Backtrace in DP-table

- possibility 1: find correct path, redoing computation (more time)
- possibility 2: compute backtracing table during main algorithm (more space)

Analysis

- poss. 1: time: up to 3 operations per column of alignment computed, so $O(\text{length of alignment}) = O(n + m)$, or $O(n)$ if $n = m$; space: only additional space for the output alignment: $O(n + m)$
- poss. 2: time: one operation per column of alignment, so $O(n + m)$; space: additional $O(n \cdot m)$ space for matrix containing traceback pointers

Finding an optimal alignment

N.B.

1. Typically we want only **one** optimal alignment
2. Order of computation matters for output!

Re 1:

There could be an exponential number of optimal alignments, see $s = AAAA \cdots AAA = A^{2n}$, $t = A^n$, then every alignment of length $2n$ (i.e. aligning each character of t with some character of s, and aligning the remaining n characters of s with gaps) is optimal. But there are $\binom{2n}{n} \geq 2^n$ such alignments.

Algorithm *Backtracing in DP-table (without traceback pointers)*
Input: strings s, t with $|s| = n, |t| = m$; scoring function f; DP-table
Output: an optimal alignment $\mathcal{A}$ of $sim(s, t)$

1. $i \leftarrow n$; $j \leftarrow m$; $\mathcal{A} \leftarrow$ empty alignment;
2. **while** $(i > 0$ and $j > 0)$
3. **do if** $D(i, j) = D(i-1, j) + g$
4. **then** $\mathcal{A} \leftarrow \binom{s_i}{-}\mathcal{A}$;
5. $i \leftarrow i - 1$;
6. **else if** $D(i, j) = D(i-1, j-1) + f(s_i, t_j)$
7. **then** $\mathcal{A} \leftarrow \binom{s_i}{t_j}\mathcal{A}$;
8. $i \leftarrow i - 1; j \leftarrow j - 1$;
9. **else** $\mathcal{A} \leftarrow \binom{-}{t_j}\mathcal{A}$;
10. $j \leftarrow j - 1$;
11. **if** $i > 0$ **then** $\mathcal{A} \leftarrow \binom{s_1 \ldots s_i}{- \ldots -}\mathcal{A}$;
12. **if** $j > 0$ **then** $\mathcal{A} \leftarrow \binom{- \ldots -}{t_1 \ldots t_j}\mathcal{A}$;
13. **return** $\mathcal{A}$;

Space-saving variant

- For computing row i, we only need row $i - 1$

Space-saving variant

- For computing row i, we only need row $i-1$
- after having finished computing row i, we never need row $i-1$ again

Space-saving variant

- For computing row i, we only need row $i-1$
- after having finished computing row i, we never need row $i-1$ again
- so we can overwrite row $i-1$ after having finished row i

Space-saving variant

- For computing row i, we only need row $i-1$
- after having finished computing row i, we never need row $i-1$ again
- so we can overwrite row $i-1$ after having finished row i
- Altogether, at any given time, we only need the current row and the previous row.

Space-saving variant

- For computing row i, we only need row $i-1$
- after having finished computing row i, we never need row $i-1$ again
- so we can overwrite row $i-1$ after having finished row i
- Altogether, at any given time, we only need the current row and the previous row.
- The same could be done with two columns instead of two rows.

Space-saving variant

- For computing row i, we only need row $i-1$
- after having finished computing row i, we never need row $i-1$ again
- so we can overwrite row $i-1$ after having finished row i
- Altogether, at any given time, we only need the current row and the previous row.
- The same could be done with two columns instead of two rows.
- Space: $O(\min(n,m))$, for $n = m$: $O(n)$

Space-saving variant

- For computing row i, we only need row $i-1$
- after having finished computing row i, we never need row $i-1$ again
- so we can overwrite row $i-1$ after having finished row i
- Altogether, at any given time, we only need the current row and the previous row.
- The same could be done with two columns instead of two rows.
- Space: $O(\min(n,m))$, for $n=m$: $O(n)$
- Time: $O(nm)$ (resp. $O(n^2)$), since we still need to compute all $(n+1)(m+1)$ entries

Space-saving variant

- For computing row i, we only need row $i-1$
- after having finished computing row i, we never need row $i-1$ again
- so we can overwrite row $i-1$ after having finished row i
- Altogether, at any given time, we only need the current row and the previous row.
- The same could be done with two columns instead of two rows.
- Space: $O(\min(n,m))$, for $n=m$: $O(n)$
- Time: $O(nm)$ (resp. $O(n^2)$), since we still need to compute all $(n+1)(m+1)$ entries
- This variant does not allow to compute an optimal alignment! (i.e. does not solve variant 2 of the problem)

Local alignment

Local alignment

- Often what we are interested in are so-called regions of high similarity in the two input strings, i.e. substrings which are similar, and not how similar the entire two strings are.
- So we want to find substrings s' of s, and t' of t s.t.

$$sim(s', t') = \max\{sim(u, v) \ : \ u \text{ substring of } s, v \text{ substring of } t\}.$$

- Typically here we also want to know all such pairs of substrings themselves and their alignment, not only their similarity value.

Smith-Waterman DP algorithm for local alignment

- Smith-Waterman DP-algorithm (1981).
- Algorithm similar to NW-algorithm for global alignment.
- Crucial points:
  1. for each pair of indices i, j, compute the highest score of an alignment of any substring u ending in position i of s with any substring v ending in position j of t
  2. the empty string is always a substring (in every position), and score of empty alignment $= 0$
  3. so all entries ≥ 0
  4. for the final output: find the maximum over all entries of the matrix
- Now we maximize like before and over 0:
  $L(i,j) = \max\{L(i-1,j)+g, L(i-1,j-1)+f(s_i,t_j), L(i,j-1)+g, 0\}$

Smith-Waterman DP algorithm for local alignment

Algorithm *DP algorithm for local alignment*
Input: strings s, t, with $|s| = n, |t| = m$; scoring function f
Output: value max

1. **for** $j = 0$ to m **do** $L(0, j) \leftarrow 0$;
2. **for** $i = 1$ to n **do** $L(i, 0) \leftarrow 0$;
3. **for** $i = 1$ to n **do**
4. **for** $j = 1$ to m **do**
5. $$L(i,j) \leftarrow \max \begin{cases} L(i-1,j) + g \\ L(i-1,j-1) + f(s_i, t_j) \\ L(i,j-1) + g \\ 0 \end{cases}$$
6. **return** $max = \max\{L(i,j) \ : \ 0 \leq i \leq n, 0 \leq j \leq m\}$;

Question: How do we compute *max* in line 6.?

Smith-Waterman DP algorithm for local alignment

Finding all optimal local alignments

- Find all occurrences of $\max\{L(i,j) : 0 \leq i \leq n, 0 \leq j \leq m\}$
- from each, backtrace until reaching a 0

Analysis

- $O(nm)$ time and space for computing matrix L
- $O(K)$ time for finding all optimal local alignments, where $K = \sum_{\mathcal{A} \text{ opt. local al.}} |\mathcal{A}|$ is the sum of the lengths of the optimal local alignments, i.e. the output size.