Wavefront aligner

Accelerating Pairwise Alignment

Santiago Marco-Sola

1. Pairwise alignment

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

1. Pairwise alignment

- In sequence comparison, pairwise alignment compares two sequences
  1) Determine its similarity (i.e. distance or **score**).
  2) Compute the **alignment** between both.
     - How to convert one sequence into the other applying a series of error events or **alignment operations** that minimize a given cost function or distance function
- Applications
  - Computational biology (e.g. genome biology, protein comparisons, evolutionary studies)
  - Information Retrieval (e.g. diff linux tool, wikipedia, git)
  - Signal processing and sound recognition (e.g. shazam)
  - Others: Image compression, pattern recognition, writing recognition, …

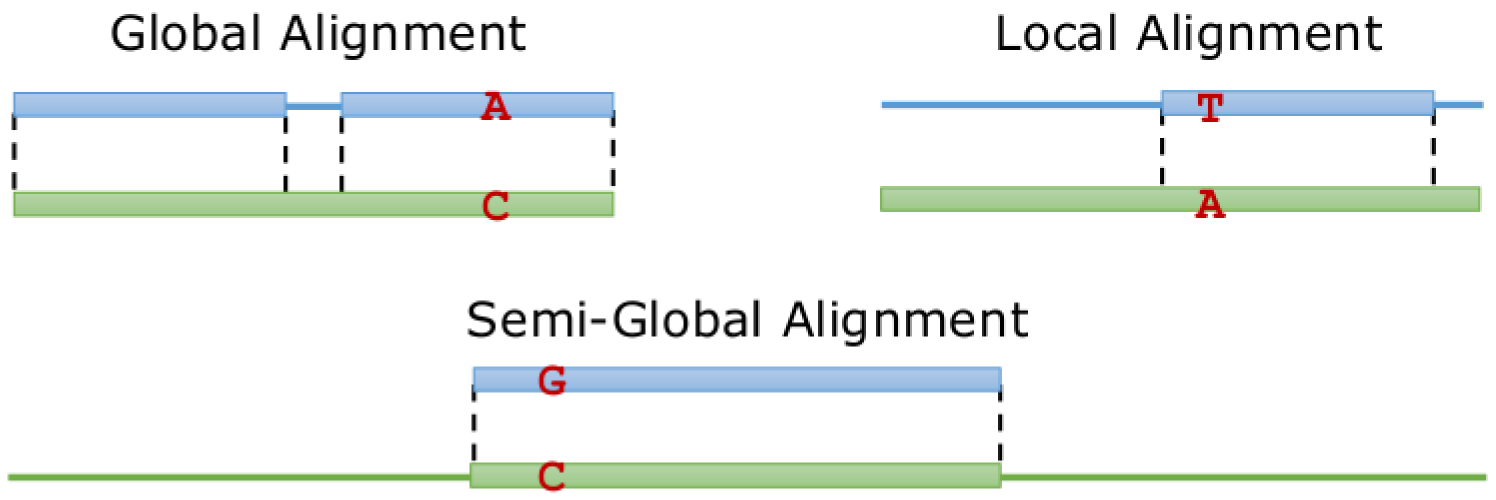
1. Distance metrics

- Some **distance metrics** are most suitable to model each application problem
  - Mismatch Distance (aka Hamming distance or Manhattan distance)
  - Edit Distance (aka Levenshtein distance)
  - Linear-Gap Distance (Needleman-Wunch and Smith-Waterman algorithms)
  - Affine-Gap Distance (Smith-Waterman-Gotoh algorithm)

- Different distance metrics involve different **alignment operations**.
  - Mismatch distance = {mismatches}
  - Levenshtein distance = {mismatches,insertions,deletions}
  - Episodic distance = {insertions}
  - Indel distance (LCS distance) = {insertions,deletions}
  - Damerau–Levenshtein distance = {mismatches,insertions,deletions,transpositions}

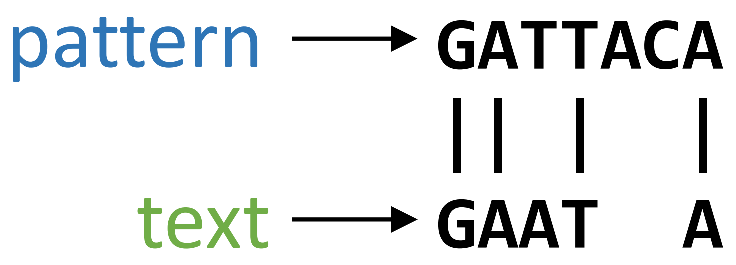
1. Alignment type

- Depending on the application, different types of alignment can be used.
  - **Global alignment**. Aligns both sequences end-to-end.
  - **Local alignment**. Locates a region that aligns between both sequences with highest similarity (i.e. highest score).
  - **Semi-global alignment**. Mostly applied to sequences that are very different in length (e.g. sequence against a genome). Allows **free-end alignment** of the largest sequence.



1. Edit distance

- Optimization problem:
  - Compute the **minimum** number of operations (match, substitution, insertion, and deletion) that transforms one sequence into another.
  - Each edit operation (substitution, single insertion, or single deletion) has unitary cost.

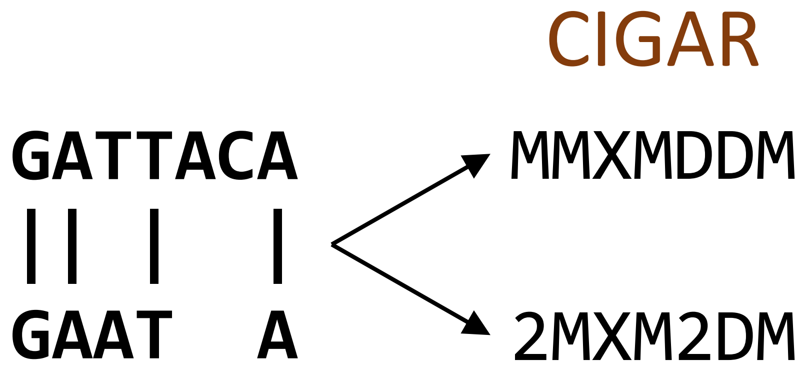


```python
def edit_distance(pattern,text):
    if len(pattern) == 0:
        # Insert all remaining chars
        return len(text)
    elif len(text) == 0:
        # Delete all remaining chars
        return len(pattern)
    else:
        # Find score for match/subtitution
        if pattern[0] == text[0]:
            m_cost = edit_distance(pattern[1:],text[1:])
        else:
            m_cost = edit_distance(pattern[1:],text[1:]) + 1
        # Find score for insertion
        i_cost = edit_distance(pattern[:],text[1:]) + 1
        # Find score for deletion
        d_cost = edit_distance(pattern[1:],text[:]) + 1
        # Find the minimum combination
        return min(m_cost,i_cost,d_cost)

distance = edit_distance("GATTACA","GAATA")
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

1. Edit alignment

- Compute the **actual set of operations** (match, substitution, insertion, and deletion) that minimizes distance/cost function.
  - Multiple solutions possible
  - Algorithms finds the “first-best”



```
def edit_distance(pattern,text):
    if len(pattern) == 0:
        return (len(text),["I"] * len(text))
    elif len(text) == 0:
        return (len(pattern),["D"] * len(pattern))
    else:
        (m_cost,m_cigar) = edit_distance(pattern[1:],text[1:])
        (i_cost,i_cigar) = edit_distance(pattern[:],text[1:])
        (d_cost,d_cigar) = edit_distance(pattern[1:],text[:])
        minimum = min(m_cost,i_cost,d_cost)
        if minimum == m_cost:
            if (pattern[0] == text[0]):
                return (m_cost,["M"] + m_cigar)
            else:
                return (m_cost+1,["X"] + m_cigar)
        elif minimum == i_cost:
            return (i_cost+1,["I"] + i_cigar)
        else:
            return (d_cost+1,["D"] + d_cigar)

distance = edit_distance("GATTACA","GAATA")
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

1. Repeated calls

- Previous algorithm was based on **exploring the whole space of solutions** (alignments), in order to find the one(s) with minimum cost ~ $O(3^n)$
  - There are NOT so many combinations
  - Recursive calls repeat combinations

```
edit_distance('CA', '')
edit_distance('ACA', '')
edit_distance('A', '')
edit_distance('CA', '')
edit_distance('', '')
edit_distance('A', '')
edit_distance('', 'A')
edit_distance('A', 'A')
edit_distance('CA', 'A')
edit_distance('ACA', 'A')
edit_distance('ACA', '')
[...]
```

```
Total Calls    Parameters
   1970        ('A', '')
   1666        ('', 'A')
   1289        ('A', 'A')
   1289        ('', '')
   1002        ('CA', '')
    681        ('CA', 'A')
    462        ('', 'TA')
    450        ('ACA', '')
    377        ('A', 'TA')
    321        ('ACA', 'A')
    231        ('CA', 'TA')
    170        ('TACA', '')
    129        ('TACA', 'A')
    129        ('ACA', 'TA')
[...]
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

1. Memoization

- Use a memorization table (e.g. Hash Table) to store the **partial results** from computing the **edit distance of suffixes**.
  - Stored at the end
  - Checked at the beginning

```python
def edit_distance(pattern,text,calls):
    # Lookup call
    key = pattern+":"+text
    if key in calls:
        return calls[key]
    # Regular algorithm
    if len(pattern) == 0:
        return len(text)
    elif len(text) == 0:
        return len(pattern)
    else:
        if pattern[0] == text[0]:
            m_cost = edit_distance(pattern[1:],text[1:])
        else:
            m_cost = edit_distance(pattern[1:],text[1:]) + 1
        i_cost = edit_distance(pattern[:],text[1:]) + 1
        d_cost = edit_distance(pattern[1:],text[:]) + 1
        minimum = min(m_cost,i_cost,d_cost)
        # Store call
        calls[pattern+":"+text] = minimum
        # Return
        return minimum
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

1. Dynamic programming

- Observe that recursive calls are just computing the edit distance between suffixes of the pattern and suffixes of the text. Therefore we can define:

  δ_e(i,j) -> edit distance between the suffix pattern[i:] and text[j:]

  δ_e(len(pattern),len(text)) -> global alignment distance

- There are only n x m combinations
  - Can be conveniently stored in a n x m matrix (i.e. **Dynamic Programming Table**)

$$\delta_e(i,j) = \begin{cases} \max(i,j) & \text{if } i = 0 \vee j = 0, \\ \min \begin{cases} \delta_e(i-1,j)+1 \\ \delta_e(i,j-1)+1 \\ \delta_e(i-1,j-1)+(v_i \neq w_j) \end{cases} & \text{otherwise} \end{cases}$$

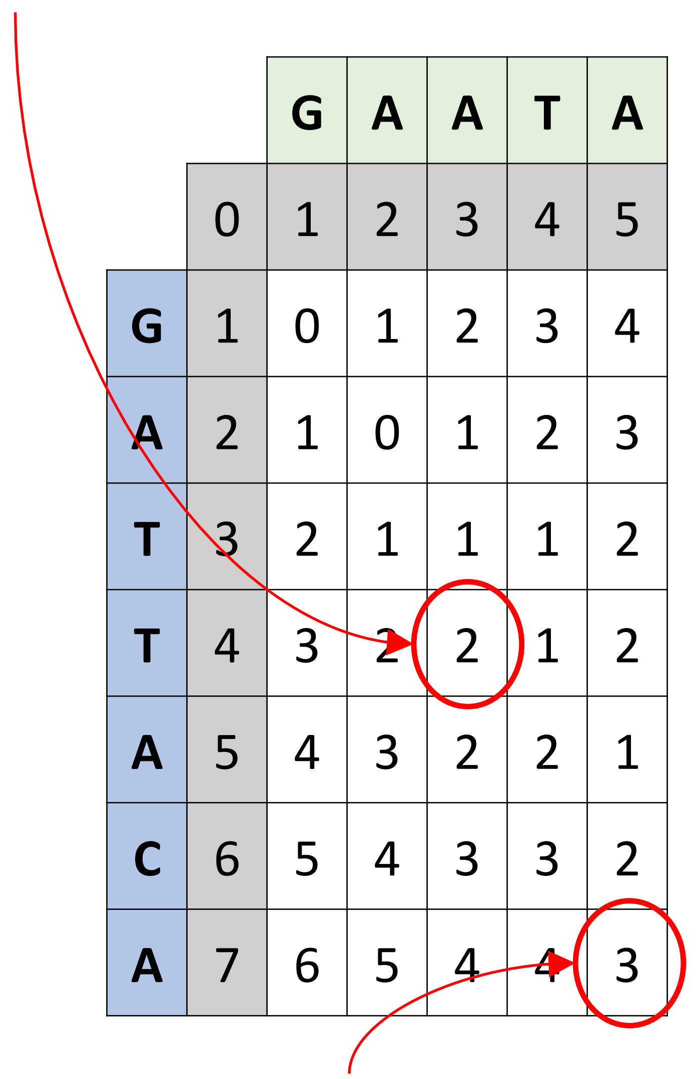
		G	A	A	T	A
	0	1	2	3	4	5
G	1	0	1	2	3	4
A	2	1	0	1	2	3
T	3	2	1	1	1	2
T	4	3	2	2	1	2
A	5	4	3	2	2	1
C	6	5	4	3	3	2
A	7	6	5	4	4	3

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

1. Dynamic programming

- Each cell (i,j) encodes the edit distance between the suffix pattern[0,i-1] and text[0,j-1]
- First row and column contain the **initial conditions**
  - Row -> Cost of deleting leading text
  - Column-> Cost of deleting leading pattern
- Semi-global alignment (ends-free alignment) sets the first row to zero to allow staring the alignment at any point in the text.

edit_distance(‘GATT', ‘GAA') = $\delta_e(4,3)$



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

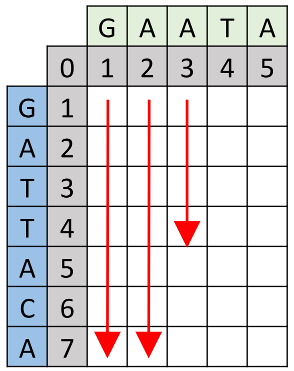
edit_distance(‘GATTACA', ‘GAATA') = $\delta_e(7,5)$

1. DP-Table computation

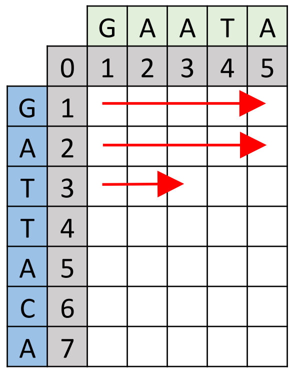
- Each DP-cell depends on other 3 cells. E.g.

  $\delta_e(4,3)$ <- {$\delta_e(4,3)$,$\delta_e(3,3)$,$\delta_e(3,2)$}

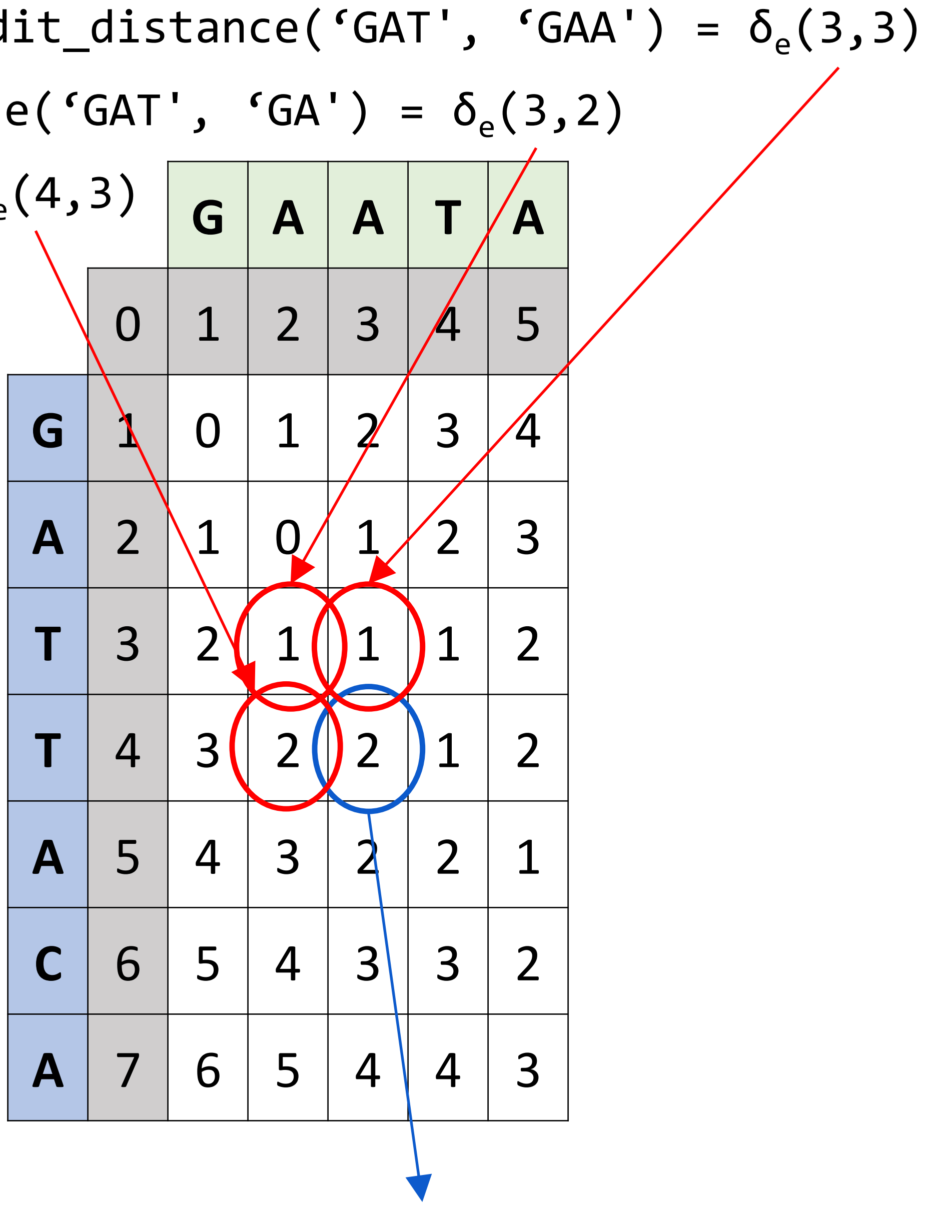
- The DP-table is usually computed row-wise or column-wise, from the left top corner to the right bottom corner (i.e. solution).



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



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



edit_distance('GAT', 'GAA') = $\delta_e(3,3)$

edit_distance('GAT', 'GA') = $\delta_e(3,2)$

edit_distance('GATT', 'GA') = $\delta_e(4,3)$

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

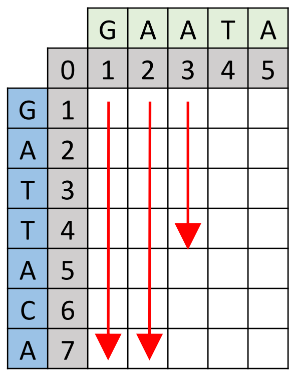
edit_distance('GATT', 'GAA') = $\delta_e(4,3)$

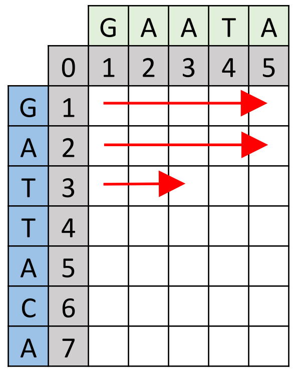
1. DP-Table computation

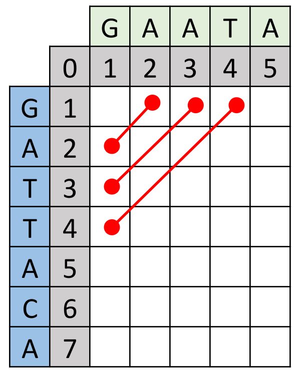
- Each DP-cell depends on other 3 cells. E.g.

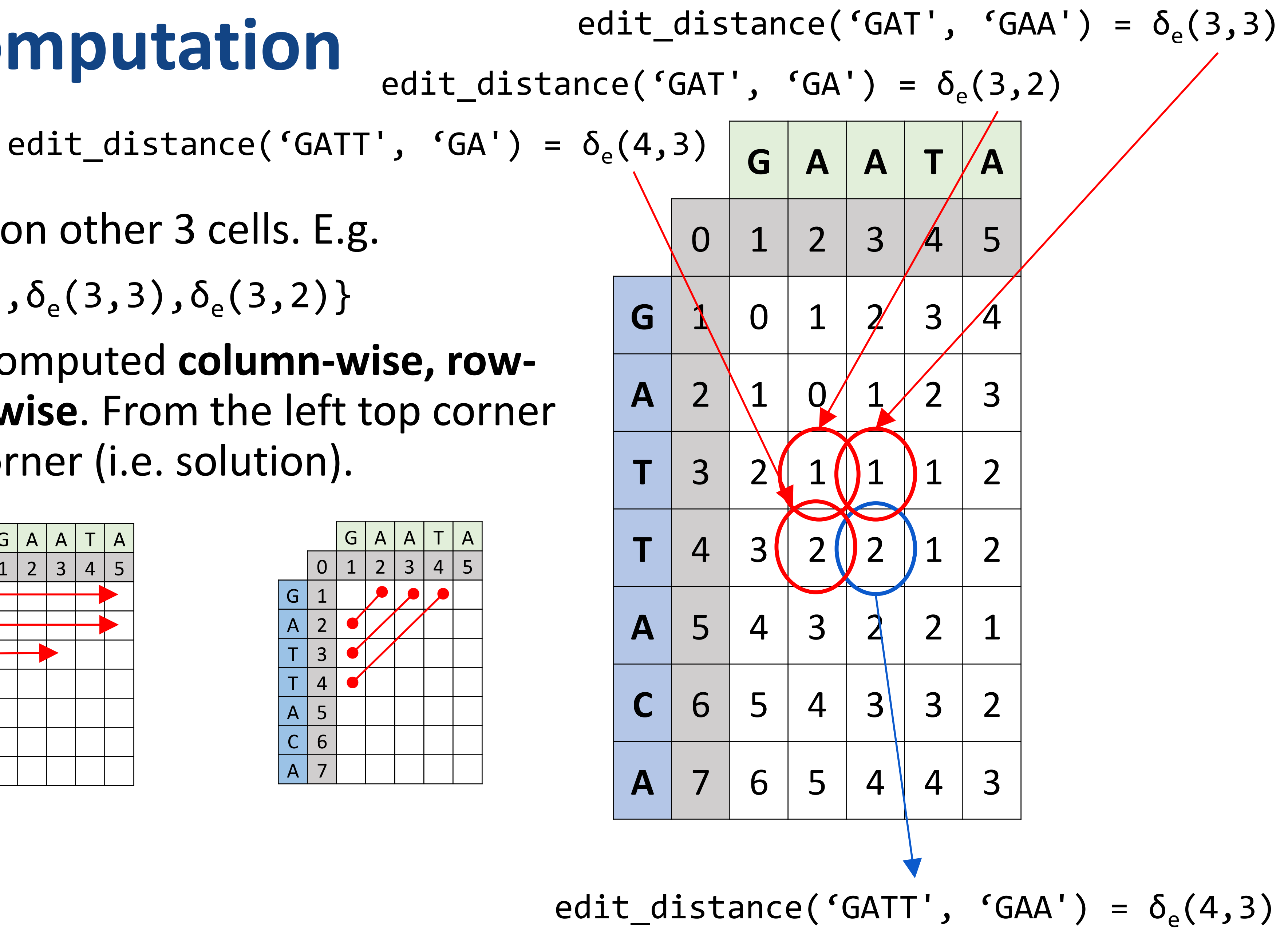
  $\delta_e(4,3) \leftarrow \{\delta_e(4,3), \delta_e(3,3), \delta_e(3,2)\}$

- The DP-table can be computed **column-wise, row-wise** or **antidiagonal-wise**. From the left top corner to the right bottom corner (i.e. solution).









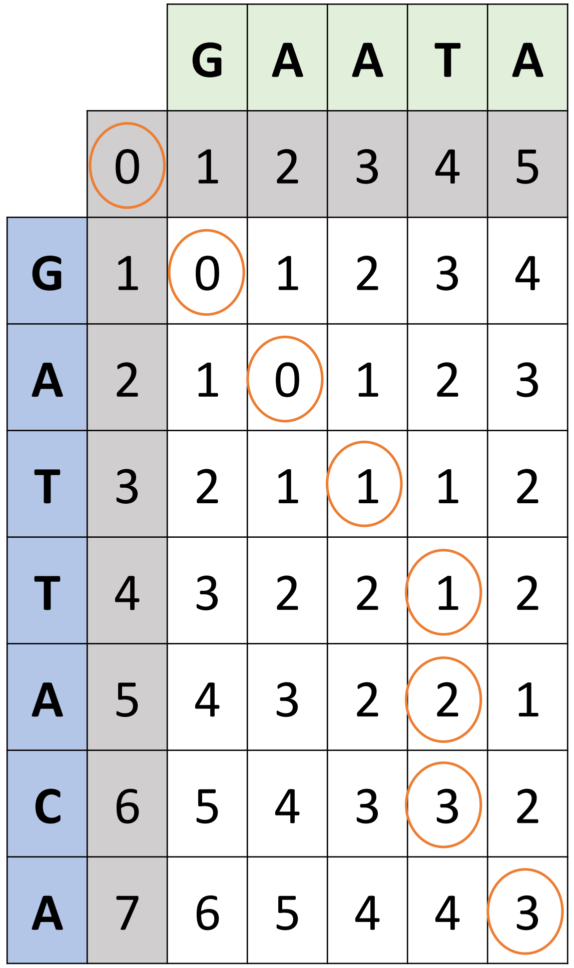
edit_distance('GAT', 'GAA') = δe(3,3)
edit_distance('GAT', 'GA') = δe(3,2)
edit_distance('GATT', 'GA') = δe(4,3)
edit_distance('GATT', 'GAA') = δe(4,3)

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

1. DP-Table Backtrace

- Backtrace traces back the path with minimum cost that lead to the solution (i.e. minimum distance).
  - Linear cost, from the bottom right corner to the upper left corner.
  - Check which of the adjacent cells brought us to that cell
  - Quite a branchy code

```
GATTACA
|| |   |
GAAT   A
```



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

CIGAR MMXMDDM

2. Gap-lineal and gap-affine alignment

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

2. Gap-affine and gap-lineal alignment

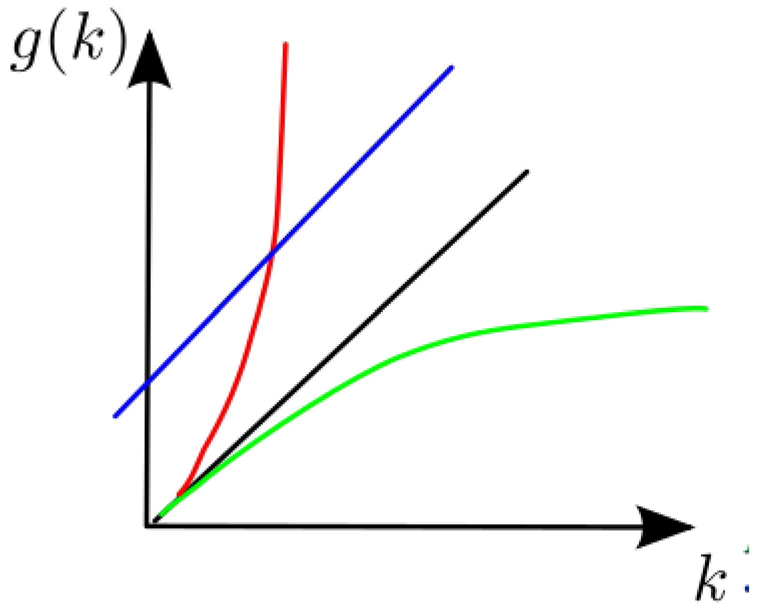
- Instead of unitary cost, associate arbitrary penalties to each alignment operation
  - For example: (M,X,I,D) = (-1,5,4,2)
- Gaps are different in nature - given a fixed number of gaps, a “small number of long gaps” is biologically likelier than a ”big number of small gaps”
  - Gap lineal.
  - Gap affine.

$g(k) = \alpha + \beta k \Rightarrow$ affine, very common

$g(k) = \alpha + \beta k^2$ ϟ

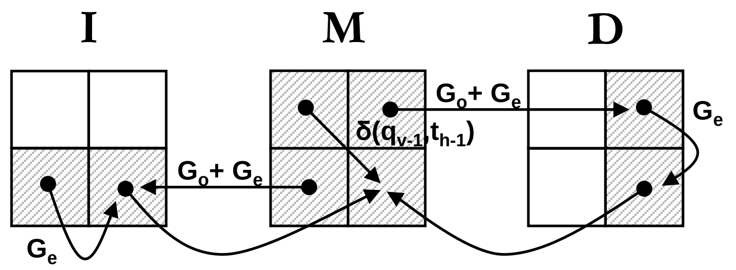
$g(k) = \alpha + \beta \ln(k) b$ ✓

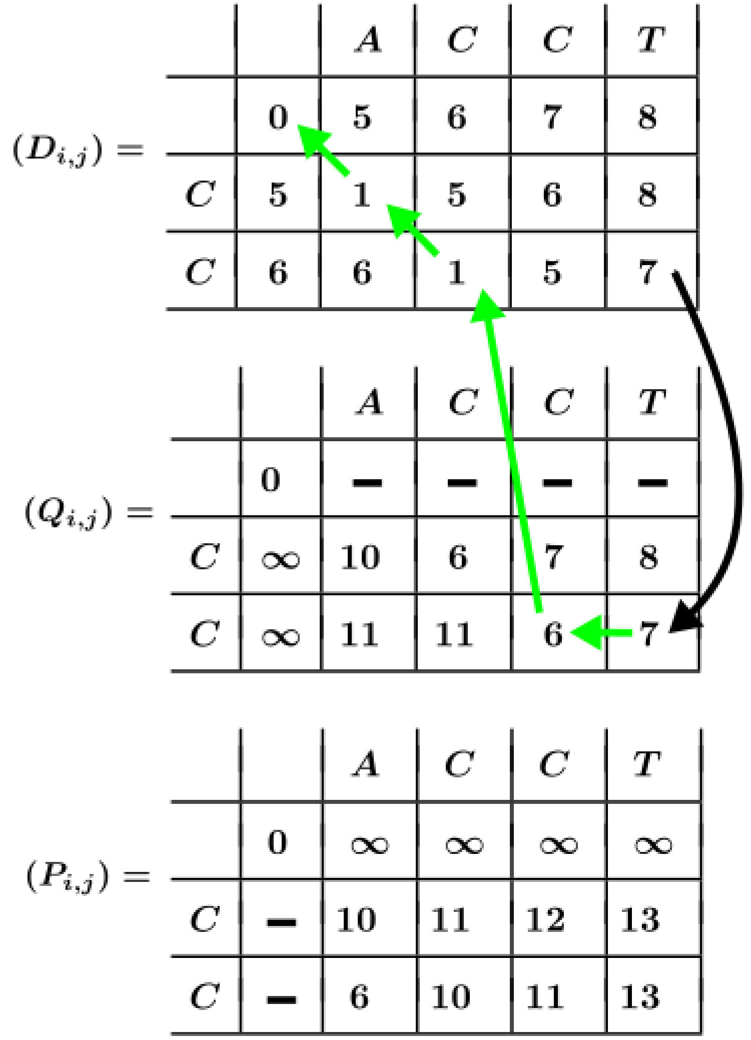
$\Rightarrow$ biologically, the best approximation



2. Gap-affine and gap-lineal alignment

- Large misunderstanding on what each algorithm does
  - Needlman-Wunch: Global alignment with general penalty cost (gap-lineal)
  - Smith-Waterman: Local alignment with general penalty cost (gap-lineal)
  - Smith-Waterman-Gotoh: Local alignment with general penalty cost (gap-affine)





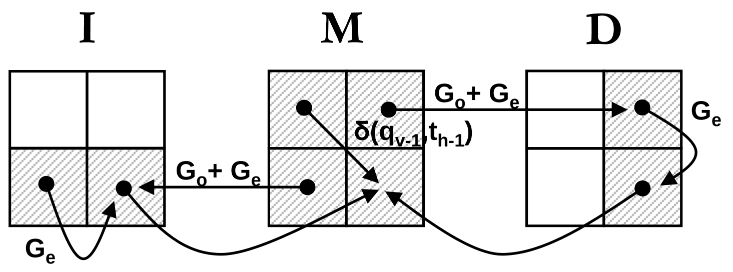
$$(D_{i,j}) = \begin{array}{c|c|c|c|c|c} & & A & C & C & T \\ \hline & 0 & 5 & 6 & 7 & 8 \\ \hline C & 5 & 1 & 5 & 6 & 8 \\ \hline C & 6 & 6 & 1 & 5 & 7 \end{array}$$

$$(Q_{i,j}) = \begin{array}{c|c|c|c|c|c} & & A & C & C & T \\ \hline & 0 & - & - & - & - \\ \hline C & \infty & 10 & 6 & 7 & 8 \\ \hline C & \infty & 11 & 11 & 6 & 7 \end{array}$$

$$(P_{i,j}) = \begin{array}{c|c|c|c|c|c} & & A & C & C & T \\ \hline & 0 & \infty & \infty & \infty & \infty \\ \hline C & - & 10 & 11 & 12 & 13 \\ \hline C & - & 6 & 10 & 11 & 13 \end{array}$$

2. Gap-affine and gap-lineal alignment

- In the end, current SWG (and related) implementations are modified depending on the application context
- In most cases, global and semi-global alignment are implemented with a myriad of heuristics (e.g. banded, cut-offs, drop-offs)
  - It is more efficient to locate regions with high similarity and then apply semi-global or global alignment



3. Wavefront-edit algorithm

BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

3. Wavefront algorithm. Idea (I)

- Diagonals are monotonically increasing
    - For a given cell, if the characters of the text are matching, there is no better outcome as non increasing the value wrt the diagonal value.
    - No need to know (or compute) the vertical/horizontal cells
- **Diagonal Extension**
    - Extending the furthest reaching cells on each diagonal until no match is found
    - Each diagonal is independent from each other
    - Character comparisons can be done in chunks

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

BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

3. Wavefront algorithm. Idea (I)

- Diagonals are monotonically increasing
  - For a given cell, if the characters of the text are matching, there is no better outcome as non increasing the value wrt the diagonal value.
  - No need to know (or compute) the vertical/horizontal cells
- **Diagonal Extension**
  - Extending the furthest reaching cells on each diagonal until no match is found
  - Each diagonal is independent from each other
  - Character comparisons can be done in chunks

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

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

3. Wavefront algorithm. Idea (I)

- Diagonals are monotonically increasing
  - For a given cell, if the characters of the text are matching, there is no better outcome as non increasing the value wrt the diagonal value.
  - No need to know (or compute) the vertical/horizontal cells
- **Diagonal Extension**
  - Extending the furthest reaching cells on each diagonal until no match is found
  - Each diagonal is independent from each other
  - Character comparisons can be done in chunks

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

3. Wavefront algorithm. Idea (II)

- For each cell, vertical and horizontal dependencies increment the value +1
- If characters at that position don’t match, in the worst-case scenario, the outcome will be incremented +1
  - The outcome can never be incrementing more than +1
- We don’t really need to know all the surrounding cells
  - We assume the worst case scenario.
- BUT, from the **furthest reaching diagonal** (current, upper, or lower).
  - We don’t even need to compute all cells to get the alignment

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

3. Wavefront algorithm. Idea (II)

- For each cell, vertical and horizontal dependencies increment the value +1
- If characters at that position don’t match, in the worst-case scenario, the outcome will be incremented +1
  - The outcome can never be incrementing more than +1
- We don’t really need to know all the surrounding cells
  - We assume the worst case scenario.
- BUT, from the **furthest reaching diagonal** (current, upper, or lower).
  - We don’t even need to compute all cells to get the alignment

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

BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

3. Wavefront algorithm. Idea (II)

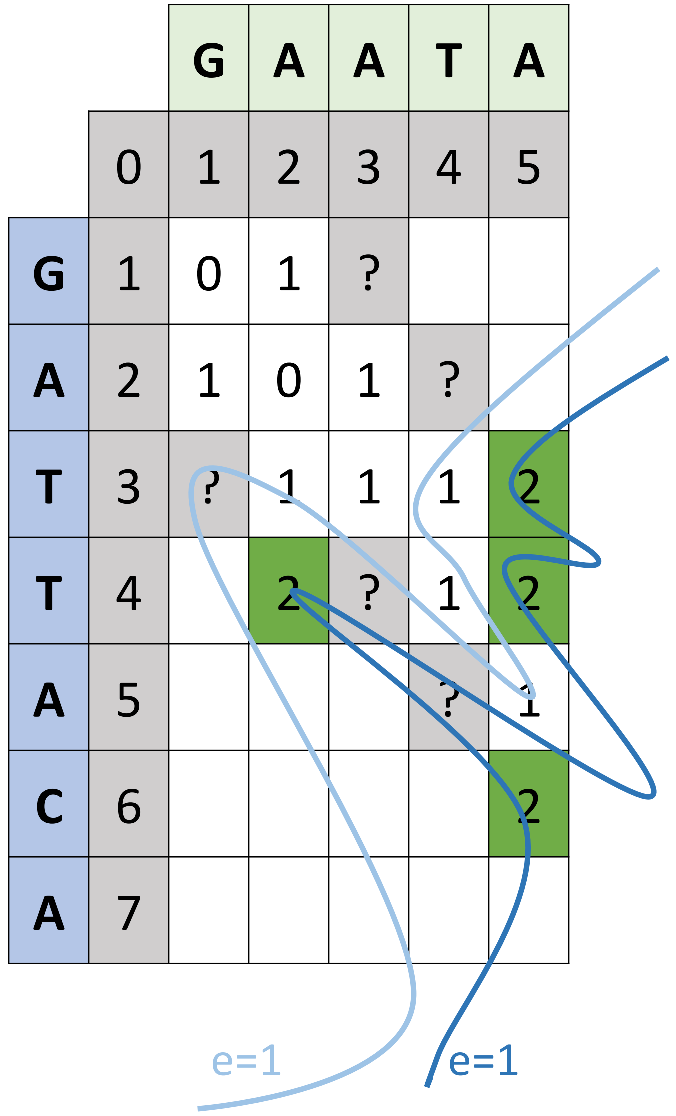
- For each cell, vertical and horizontal dependencies increment the value +1
- If characters at that position don't match, in the worst-case scenario, the outcome will be incremented +1
  - The outcome can never be incrementing more than +1
- We don't really need to know all the surrounding cells
  - We assume the worst case scenario.
- BUT, from the **furthest reaching diagonal** (current, upper, or lower).
  - We don't even need to compute all cells to get the alignment

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

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

3. Wavefront algorithm. Idea (II)

- For each cell, vertical and horizontal dependencies increment the value +1
- If characters at that position don't match, in the worst-case scenario, the outcome will be incremented +1
  - The outcome can never be incrementing more than +1
- We don't really need to know all the surrounding cells
  - We assume the worst case scenario.
- BUT, from the **furthest reaching diagonal** (current, upper, or lower).
  - We don't even need to compute all cells to get the alignment



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

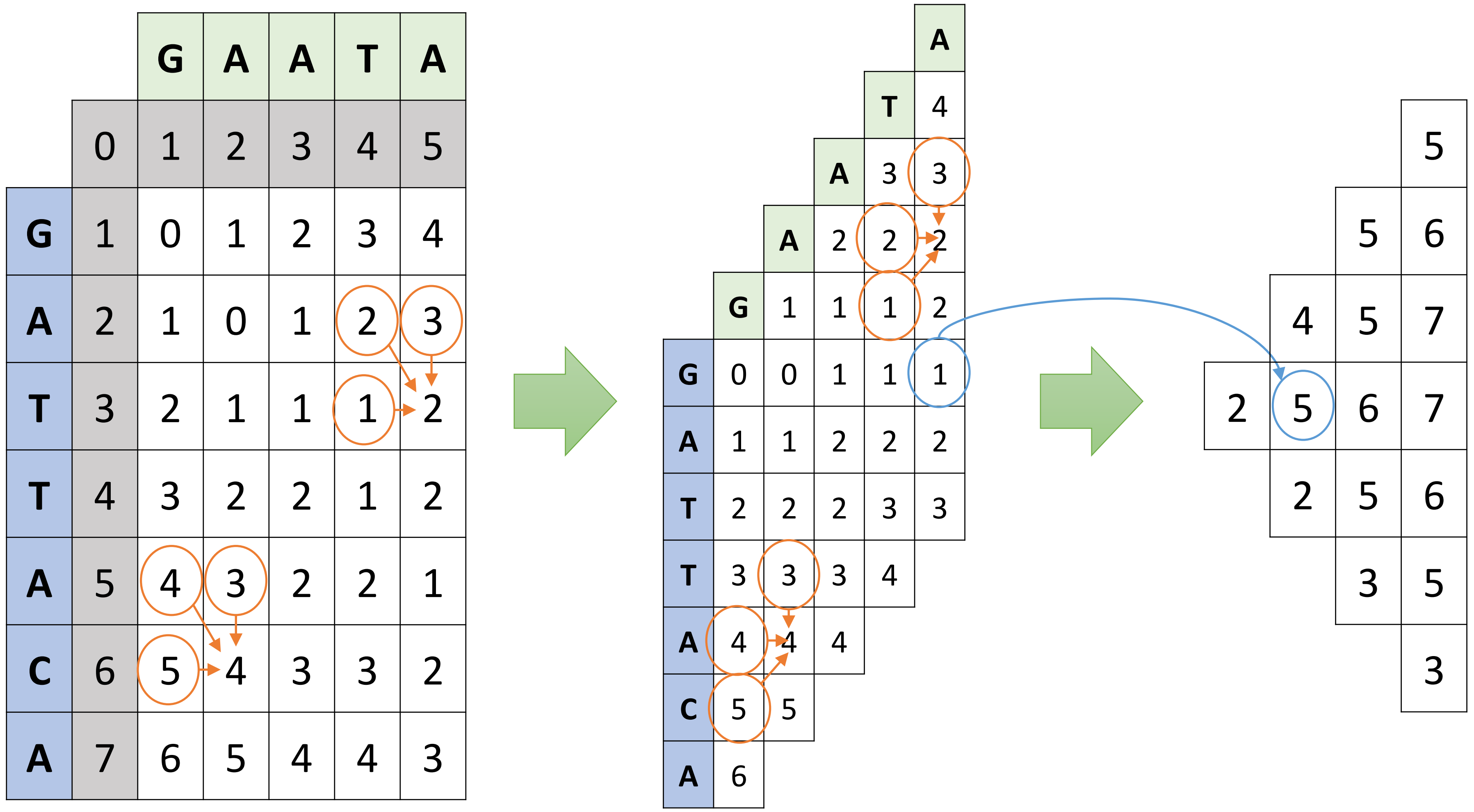
3. WF. Rearrange the layout

1) Align diagonals by shifting up columns

(Dependencies are kept homogeneous)

2) Encode using diagonal offsets

(Instead of scores)



3. WF. Computing wavefronts

For each score:

1) Extend wavefronts matching chars

2) Compute the next wavefront

	0	1	2	3	4	5
k=+5						**A**
k=+4					**T**	4
k=+3				**A**	3	3
k=+2			**A**	2	2	2
k=+1		**G**	1	1	1	2
k = 0	**G**	0	0	1	1	1
k=-1	**A**	1	1	2	2	2
k=-2	**T**	2	2	2	3	3
k=-3	**T**	3	3	3	4	
k=-4	**A**	4	4	4		
k=-5	**C**	5	5			
k=-6	**A**	6				

	d=0	d=1	d=2	d=3
k=+3				
k=+2				
k=+1				
k = 0	0			
k=-1				
k=-2				
k=-3				

3. WF. Computing wavefronts

For each score:

1) Extend wavefronts matching chars

2) Compute the next wavefront

	0	1	2	3	4	5
k=+5						**A**
k=+4					**T**	4
k=+3				**A**	3	3
k=+2			**A**	2	2	2
k=+1		**G**	1	1	1	2
k = 0	**G**	0	0	1	1	1
k=-1	**A**	1	1	2	2	2
k=-2	**T**	2	2	2	3	3
k=-3	**T**	3	3	3	4	
k=-4	**A**	4	4	4		
k=-5	**C**	5	5			
k=-6	**A**	6				

	d=0	d=1	d=2	d=3
k=+3				
k=+2				
k=+1				
k = 0	2			
k=-1				
k=-2				
k=-3				

3. WF. Computing wavefronts

For each score:

1) Extend wavefronts matching chars

2) Compute the next wavefront

	0	1	2	3	4	5
k=+5						**A**
k=+4					**T**	4
k=+3				**A**	3	3
k=+2			**A**	2	2	2
k=+1		**G**	1	1	1	2
k = 0	**G**	0	0	1	1	1
k=-1	**A**	1	1	2	2	2
k=-2	**T**	2	2	2	3	3
k=-3	**T**	3	3	3	4	
k=-4	**A**	4	4	4		
k=-5	**C**	5	5			
k=-6	**A**	6				

	d=0	d=1	d=2	d=3
k=+3				
k=+2				
k=+1		3		
k = 0	2	3		
k=-1		2		
k=-2				
k=-3				

3. WF. Computing wavefronts

For each score:

1) Extend wavefronts matching chars

2) Compute the next wavefront

	0	1	2	3	4	5
k=+5						**A**
k=+4					**T**	4
k=+3				**A**	3	3
k=+2			**A**	2	2	2
k=+1		**G**	1	1	1	2
k = 0	**G**	0	0	1	1	1
k=-1	**A**	1	1	2	2	2
k=-2	**T**	2	2	2	3	3
k=-3	**T**	3	3	3	4	
k=-4	**A**	4	4	4		
k=-5	**C**	5	5			
k=-6	**A**	6				

	d=0	d=1	d=2	d=3
k=+3				
k=+2				
k=+1		4		
k = 0	2	5		
k=-1		2		
k=-2				
k=-3				

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

3. WF. Computing wavefronts

For each score:

1) Extend wavefronts matching chars

2) Compute the next wavefront

	0	1	2	3	4	5
k=+5						**A**
k=+4					**T**	4
k=+3				**A**	3	3
k=+2			**A**	2	2	2
k=+1		**G**	1	1	1	2
k = 0	**G**	0	0	1	1	1
k=-1	**A**	1	1	2	2	2
k=-2	**T**	2	2	2	3	3
k=-3	**T**	3	3	3	4	
k=-4	**A**	4	4	4		
k=-5	**C**	5	5			
k=-6	**A**	6				

	d=0	d=1	d=2	d=3
k=+3				
k=+2			5	
k=+1		4	5	
k = 0	2	5	6	
k=-1		2	5	
k=-2			2	
k=-3				

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

3. WF. Computing wavefronts

For each score:

1) Extend wavefronts matching chars

2) Compute the next wavefront

	0	1	2	3	4	5
k=+5						**A**
k=+4					**T**	4
k=+3				**A**	3	3
k=+2			**A**	2	2	2
k=+1		**G**	1	1	1	2
k = 0	**G**	0	0	1	1	1
k=-1	**A**	1	1	2	2	2
k=-2	**T**	2	2	2	3	3
k=-3	**T**	3	3	3	4	
k=-4	**A**	4	4	4		
k=-5	**C**	5	5			
k=-6	**A**	6				

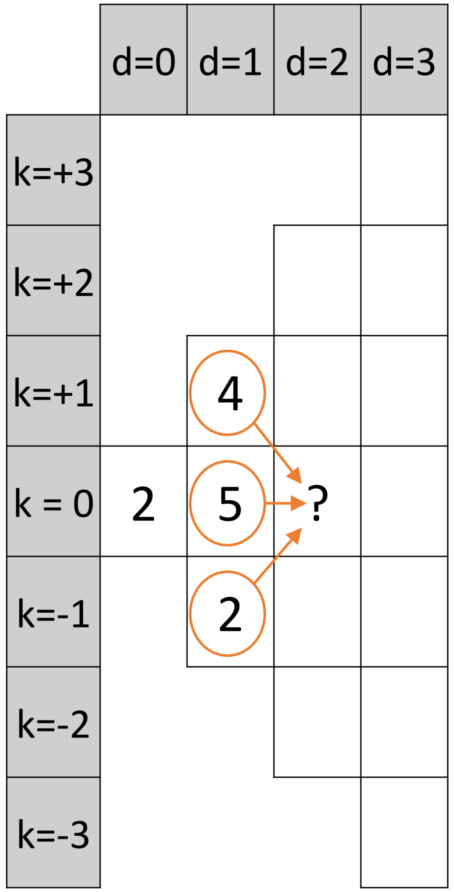
	d=0	d=1	d=2	d=3
k=+3				5
k=+2			5	6
k=+1		4	5	7
k = 0	2	5	6	7
k=-1		2	5	6
k=-2			3	5
k=-3				3

3. WF Implementation

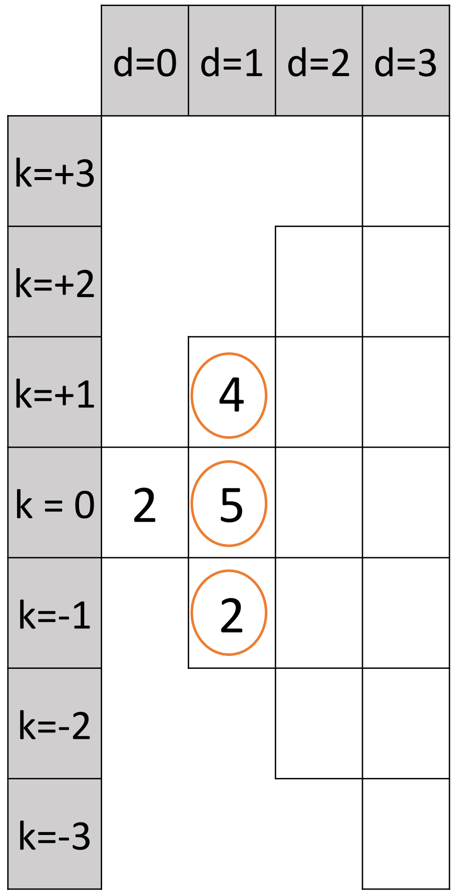
```
void edit_wavefronts_align(
    edit_wavefronts_t* const wavefronts,
    const char* const pattern,const int pattern_length,
    const char* const text,const int text_length) {
  const int max_distance = pattern_length + text_length;
  const int target_k = text_length - pattern_length;

  // Init wavefronts
  wavefronts->wavefronts[0]->offsets[0] = 0;

  // Compute wavefronts for increasing distance
  for (int distance=0;distance<max_distance;++distance) {
    // Extend diagonally each wavefront point
    edit_wavefronts_extend_wavefront(wavefronts,pattern,pattern_length,text,text_length,distance);

    // Exit condition
    if (ABS(target_k) <= distance && wavefronts->wavefronts[distance]->offsets[target_k] == text_length) break;

    // Compute next wavefront starting point
    edit_wavefronts_compute_wavefront(wavefronts,pattern_length,text_length,distance+1);
  }

  // Backtrace wavefronts
  edit_wavefronts_backtrace(wavefronts,pattern,text,target_k,distance);
}
```

3. WF Implementation. Generate wavefront

```
void edit_wavefronts_compute_wavefront(
    edit_wavefronts_t* const wavefronts,
    const int pattern_length,
    const int text_length,
    const int distance) {
  [...]
  // Loop peeling (k=lo-1)
  next_offsets[lo-1] = offsets[lo];
  // Loop peeling (k=lo)
  const ewf_offset_t bottom_upper_del = ((lo+1) <= hi) ? offsets[lo+1] : -1;
  next_offsets[lo] = MAX(offsets[lo]+1,bottom_upper_del);
  // Compute next wavefront starting point
  int k;
  #pragma GCC ivdep
  for (k=lo+1;k<=hi-1;++k) {
    /*
     * const int del = offsets[k+1]; // Upper
     * const int sub = offsets[k] + 1; // Mid
     * const int ins = offsets[k-1] + 1; // Lower
     * next_offsets[k] = MAX(sub,ins,del); // MAX
     */
    const ewf_offset_t max_ins_sub = MAX(offsets[k],offsets[k-1]) + 1;
    next_offsets[k] = MAX(max_ins_sub,offsets[k+1]);
  }
  // Loop peeling (k=hi)
  const ewf_offset_t top_lower_ins = (lo <= (hi-1)) ? offsets[hi-1] : -1;
  next_offsets[hi] = MAX(offsets[hi],top_lower_ins) + 1;
  // Loop peeling (k=hi+1)
  next_offsets[hi+1] = offsets[hi] + 1;
}
```



3. WF Extend wavefront



```
void edit_wavefronts_extend_wavefront_compute(
    edit_wavefronts_t* const wavefronts,
    const char* const pattern,
    const int pattern_length,
    const char* const text,
    const int text_length,
    const int distance) {
  // Parameters
  edit_wavefront_t* const wavefront = wavefronts->wavefronts[distance];
  ewf_offset_t* const offsets = wavefront->offsets;
  const int k_min = wavefront->lo;
  const int k_max = wavefront->hi;
  // Extend diagonally each wavefront point
  int k;
  for (k=k_min;k<=k_max;++k) {
    int v = EWAVEFRONT_V(k,offsets[k]), h = EWAVEFRONT_H(k,offsets[k]);
    while (v<pattern_length && h<text_length && pattern[v++]==text[h++]) {
      ++(offsets[k]);
    }
  }
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

3. WF Extend SIMD

- Compare blocks of 8 chars (64bits)
  - CTZL – Count trailing zeros
- Internal loop is seldom executed
- External loop cannot be vectorized
  - Gather/Scatter operation
  - No SIMD CTZL
    - Though it, can be reformulated using popcount
- Actual BOTTLENECK
  - As generate wavefront can be perfectly vectorized

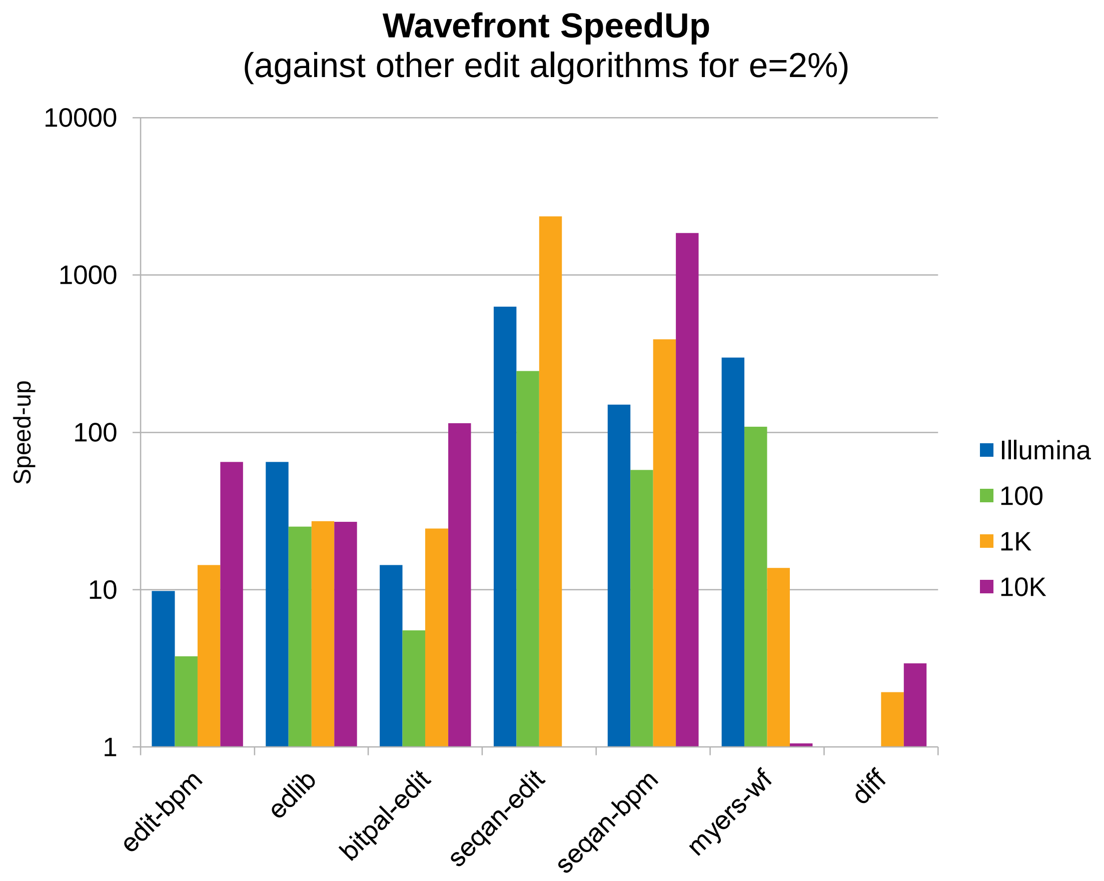
```
void edit_wavefronts_extend_wavefront_compute_packed(
    edit_wavefronts_t* const wavefronts,
    const char* const pattern,
    const int pattern_length,
    const char* const text,
    const int text_length,
    const int distance) {
  edit_wavefront_t* const wavefront = wavefronts->wavefronts[distance];
  ewf_offset_t* const offsets = wavefront->offsets;
  // Extend diagonally each wavefront point
  for (int k=wavefront->lo;k<=wavefront->hi;++k) {
    const ewf_offset_t offset = offsets[k];
    int v = EWAVEFRONT_V(k,offset), h = EWAVEFRONT_H(k,offset);
    // Fetch pattern/text blocks
    uint64_t* pattern_blocks = (uint64_t*)(pattern+v);
    uint64_t* text_blocks = (uint64_t*)(text+h);
    uint64_t pattern_block = *pattern_blocks;
    uint64_t text_block = *text_blocks;
    // Compare 64-bits blocks
    uint64_t cmp = pattern_block ^ text_block;
    while (__builtin_expect(!cmp,0)) {
      offsets[k] += 8; // Increment offset (full block)
      ++pattern_blocks; // Next blocks
      ++text_blocks; // Next blocks
      pattern_block = *pattern_blocks; // Fetch
      text_block = *text_blocks; // Fetch
      cmp = pattern_block ^ text_block; // Compare
    }
    // Count equal characters
    const int equal_right_bits = __builtin_ctzl(cmp);
    const int equal_chars = DIV_FLOOR(equal_right_bits,8);
    // Increment offset
    offsets[k] += equal_chars;
  }
}
```

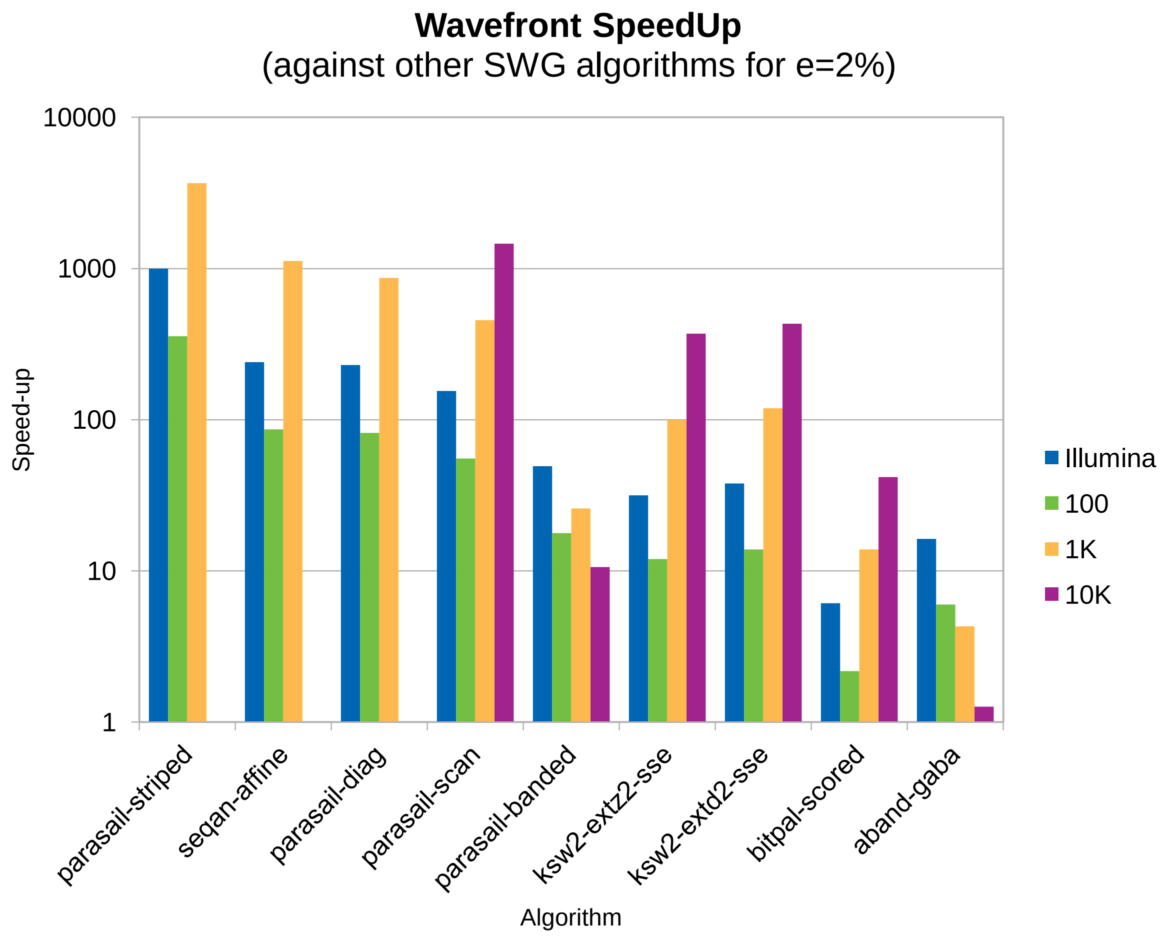
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

4. Benchmark results

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

4. Benchmark results

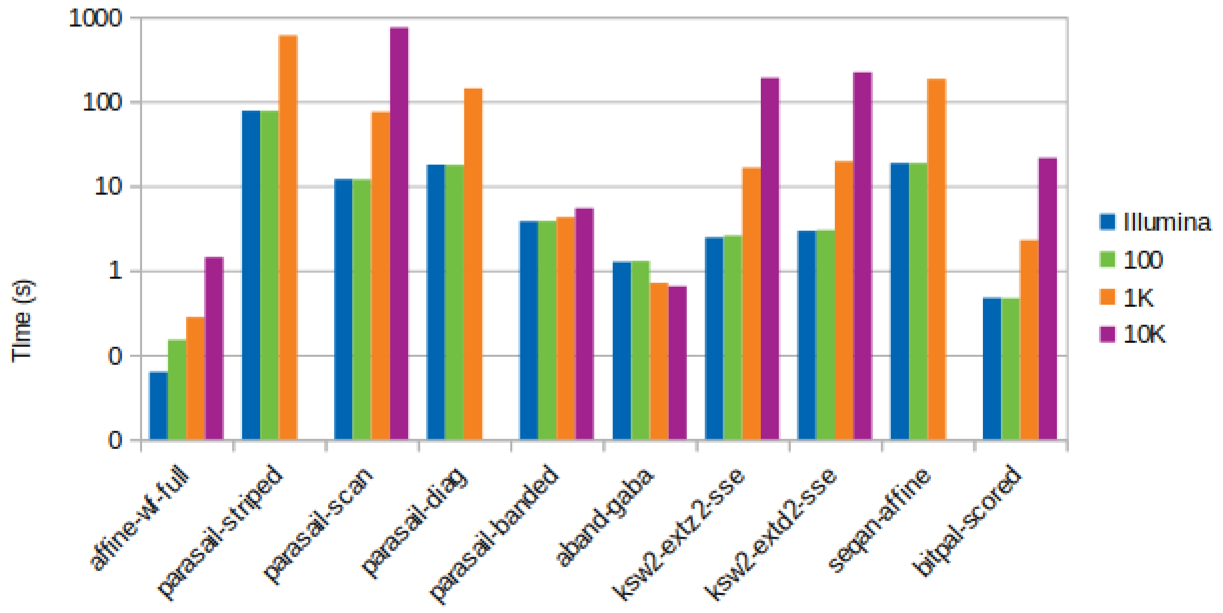




Barcelona Supercomputing Center
Centro Nacional de Supercomputación

4. Benchmark. SWG Scaling

SWG Scaling with the read length (e=2%)



SWG Scaling with the error rate (l=1000nt)

L = 1000nt	e=1%	e=2%	e=5%	e=10%	e=20%
affine-wf-full	0,14	0,28	0,92	2,58	6,92
parasail-striped	603,60	603,60	602,40	597,00	572,40
parasail-scan	75,00	75,00	75,00	75,00	75,00
parasail-diag	142,20	142,20	142,20	142,20	142,20
parasail-banded	4,08	4,26	4,59	4,97	5,43
aband-gaba	0,67	0,71	1,92	0,86	n/a
ksw2-extz2-sse	16,42	16,40	16,53	16,53	16,61
ksw2-extd2-sse	19,57	19,57	19,64	19,73	19,80
seqan-affine	183,00	184,20	197,40	196,80	211,20
bitpal-scored	2,28	2,28	2,30	2,36	2,37

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

5. Wrapping up (some remarks)

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

5. Wavefront Drawbacks (too good to be true?)

- Depends on the error between the patterns. For higher error rates (e > 20%) the algorithm speedups may get reduced
  - Such high error rates are only “somehow” meaningful in protein alignment
  - Most common use cases show high identity (e.g. Illumina ~2%)
- Designed for integer scores (could be extended to floating scores)
  - You can always multiply by 10^n and round scores
- Matching penalty must be zero
  - Benchmarks show that there is rarely any loss in accuracy
- I still haven’t figured out local alignment
  - Can be used for global and semi-global alignment (also, any kind of ends-free alignment)
  - Most tools use ends-free alignment instead of a pure local-alignment

5. Wavefront Leverages

- No heuristics, finds the exact solution.
- Breaks vertical/horizontal dependencies.
  - Result of changing the layout and the order in which the cells are computed (i.e. free of dependencies)
- Both main functions are embarrassingly parallel.
  - Can be automatically vectorized by the compiler for any SIMD ISA supported
  - AFAIK, only SWG implementation AVX512-compliant (with no effort)
- Computes progressively increasing error rates.
  - If the error is 1, then the algorithm perform 1 step -> $O(n \cdot e)$
  - Doesn't need to know error estimations (as opposed to banded algorithms)
  - Allows precise implementation of cut-off techniques
- Encodes diagonal offsets (integer that depend on the patterns length) as opposed to scores (floats that depend on the score/error rate).
  - Alignment_error > $5 \cdot \max\{n,m\}$
  - Can fit more offsets (uint16_t) in a SIMD word (as opposed to float32_t)
- Memory consumption is proportional to the error rate.

5. Wavefront

Opens the way for new and exciting opportunities

Algorithms

FPGAs

GPUs

Hardware design

santiago.marco@bsc.es