6 Q-gram filters for ε-matches

This exposition was developed by Clemens Gröpl. It is based on:

- Kim R. Rasmussen, Jens Stoye, Eugene W. Myers: *Efficient q-Gram Filters for Finding All ϵ-Matches over a Given Length*, Journal of Computational Biology, Volume 13, Number 2, 2006, pages 296–308. (Originally presented at GCB 2004 and RECOMB 2005.) [RSM06]

6.1 Motivation

Comparison of large genomic sequences can be speeded up a lot if *filtering techniques* are applied. The key observation is that a local alignment of high sequence similarity must contain at least a few short exact matches.

The idea of using q-grams for fast filtering is not new. A q-gram is a substring of length q. Programs like BLAST use q-grams which occur in both sequences as *seeds* for a local alignment search.

It has also been observed that combining the idea of seeds with a combinatorial argumentation based on some form of the *pigeon hole principle* can be used to discard large parts of the input sequences from further consideration, because they cannot contain a good local alignment.

We can distinguish three kinds of algorithms.

When applied for finding highly similar regions, the classical *exact* algorithms (e. g. Smith-Waterman) will spend most of the time verifying that there is no match between a given pair of regions. The running times (typically the product of sequence lengths) are infeasible for genome size sequences.

Heuristics like BLAST typically employ a q-gram index to locate seeds and perform a verification for the candidate regions located in this way. However, BLAST might fail to recognize an existing match, unless the filtering parameters are set very stringent. Thus one has to trade off sensitivity against speed.

A *filter* is an algorithm that allows us to discard large parts of the input, but is guaranteed not to loose *any* significant match. The trade-off to be considered for filtering algorithms is thus only whether the additional effort is payed off by the saving of time spent for verifications.

In this lecture, we will consider the problem of finding matches of low *error rate* ε and a given *minimum length* n_0.

The cost measure will be the *edit distance* (Levenshtein distance). That is, the distance between two strings is the number of insertions, deletions, and substitutions needed to transform one into the other.

The SWIFT algorithm is an improvement of the QUASAR algorithm by Burkhardt et. al.. Note, however, that QUASAR uses an absolute error threshold rather than an error rate. Using an error rate is more appropriate since the length of a local alignment is not known in advance.

The filter has been successfully applied for the *fragment overlap* computation in sequence assembly and for *BLAST-like searching* in EST sequences.

6.2 Definitions

As usual, let A and B denote strings over a finite alphabet Σ, let $|A|$ be the length of A, let $A[i]$ be the i-th letter of A, and let $A[p..q]$ be the substring starting at position p and ending with position q of A, thus $A[i..i]$ consists of the letter $A[i]$. A substring of length $q > 0$ of A is a *q-gram* of A.

The *(unit cost) edit distance* between strings A and B is the minimum number of edit operations (insertion, deletion, substitution) in an alignment of A and B. It is denoted by $\mathrm{dist}(A, B)$.

The edit distance can be computed by the well-known Needleman-Wunsch algorithm. It computes in $O(|A||B|)$ time an *edit matrix* $E(i, j) := \mathrm{dist}(A[1..i], B[1..j])$. The letter $A[i]$ corresponds to the step from row $i-1$ to i, so it is natural to visualize the letters *between* the rows and columns of the edit matrix, etc..

An *ε-match* is a local alignment for substrings (α, β) with an *error rate* of at most ε. That is, $\mathrm{dist}(\alpha, \beta) \le \varepsilon\left|\beta\right|$. (Note the 'asymmetry' in the definition of error rate.)

The problem can now be formally stated as follows:

> Given a *target* string A and a *query* string B, a *minimum match length* n_0 and a *maximum error rate* $\varepsilon > 0$;
>
> Find all ε-matches (α, β) where α and β are substrings of A and B, respectively, such that
>
> 1. $\left|\beta\right| \ge n_0$ and
> 2. $\mathrm{dist}(\alpha, \beta) \le \lfloor \varepsilon\left|\beta\right| \rfloor$.

6.3 q-gram filters for ε-matches

A *q-hit* is a pair (i, j) of indices such that $A[i..i+q-1] = B[j..j+q-1]$.

The basic idea of the q-gram method is as follows:

1. Find (enumerate) all q-hits between the query and the target strings.
2. Identify regions (in the Cartesian product of the strings) that have "enough" hits.
3. Such candidate regions are then subjected to a closer examination.

The concrete methods differ in the shape and the size of the regions.

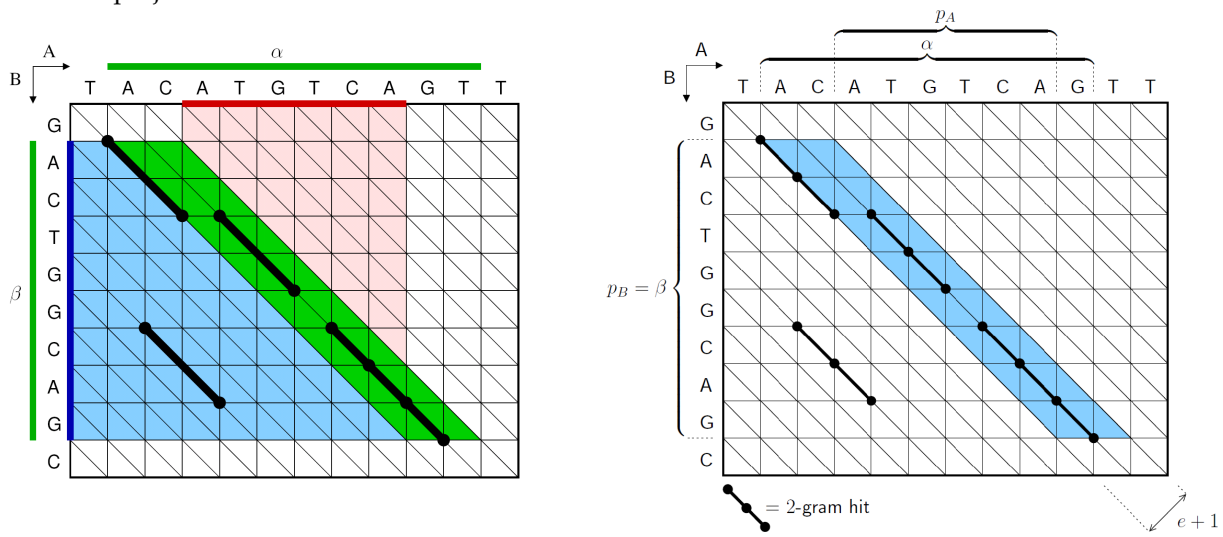
The following lemma relates ε-matches (α, β) to *parallelograms* of the edit matrix. For a moment, we assume that the length of β is known, so that we can work with an absolute bound on the distance.

An *$n \times e$ parallelogram* of the edit matrix consists of entries from $n+1$ consecutive rows and $e+1$ consecutive diagonals.

Lemma 1. Let α and β be substrings of A and B, respectively, and assume that $|\beta| = n$ and $\mathrm{dist}(\alpha, \beta) \le e$. Then there exists an $n \times e$ parallelogram P such that

1. P contains at least $T(n, q, e) := (n+1) - q(e+1)$ q-hits,
2. the B-projection of the parallelogram is $p_B(P) = \beta$,
3. the A-projection $p_A(P)$ of the parallelogram is contained in α.

The A- and B-projections are defined as illustrated below.



The *A-projection* $p_A(P)$ of a parallelogram P is defined as the substring of A between the last column of the first row of P and the first column of the last row of P.

The *B-projection* $p_B(P)$ of a parallelogram P is defined as the substring of B between the first and the last row of P.

(Note: these figures are taken from the RECOMB and GCB version, which uses the transposed matrix of the JCB article.)

Clearly, a q-hit (i, j) corresponds to $q+1$ consecutive entries of the edit matrix along the diagonal $j - i$. A q-hit is *contained* in a parallelogram if its corresponding matrix entries are.

The proof of Lemma 1 is straightforward: Consider the path of an optimal alignment of α and β. At each row except for the last q ones, we have a q-gram unless there is an edit operation among the next q edges. Each edit operation can 'destroy' at most q q-hits.

So the case where $|\beta|$ is fixed was easy. Next we consider ϵ-matches for $|\beta| \ge n_0$. The following lemma is the combinatorial foundation of the SWIFT algorithm.

Lemma 2. Let α and β be substrings of A and B, respectively, and assume that $|\beta| \ge n_0$ and $\mathrm{dist}(\alpha, \beta) \le \varepsilon|\beta|$. Let $U(n, q, \varepsilon) := T(n, q, \lfloor \varepsilon n \rfloor) = (n+1) - q(\lfloor \varepsilon n \rfloor + 1)$ and assume that the q-gram size q and the threshold τ have been chosen such that

$$q < \lceil 1/\varepsilon \rceil \qquad \text{and} \qquad \tau \le \min\left\{U(n_0, q, \varepsilon), U(n_1, q, \varepsilon)\right\},$$

where $n_1 := \left\lceil (\lfloor \varepsilon n_0 \rfloor + 1)/\varepsilon \right\rceil$.

Then there exists a $w \times e$ parallelogram P such that:

1. P contains at least τ q-hits whose projections intersect α and β,
2. $w = (\tau - 1) + q(e + 1)$,
3. $e = \left\lfloor \dfrac{2\tau + q - 3}{1/\varepsilon - q} \right\rfloor$,
4. if $|\beta| \leq w$, then $p_B(P)$ contains β, otherwise β contains $p_B(P)$.

The purpose of Lemma 2 is as follows. Given parameters ε and n_0, we can choose suitable values for q, τ, w, and e using Lemma 2. Then we enumerate all parallelograms P with enough hits according to these parameters. All relevant ε-matches can be found in these regions.

Proof of Lemma 2. The lemma is proven in three steps:

1. Assuming there is an ε-match (α, β) of length $|\beta| = n \geq n_0$, show that there are at least τ q-hits in the surrounding $n \times \lfloor \varepsilon n \rfloor$ parallelogram.
2. Argue that there is a $w \times e$ parallelogram that contains at least τ q-hits, where w and e do not depend on $n \geq n_0$.
3. Determine the dimensions w and e of such a parallelogram.

... details omitted ...

TABLE 1. FILTER PARAMETERS FOR $\epsilon = 0.05$, BY LEMMA 2 AND COROLLARY 1, RESPECTIVELY

	$q = 7$			$q = 9$			$q = 11$		
n_0	*30*	*50*	*100*	*30*	*50*	*100*	*30*	*50*	*100*
w	37	64	128	39	68	136	40	71	133
e	2	4	9	2	4	9	2	4	8
τ	17	30	59	13	24	47	8	17	35

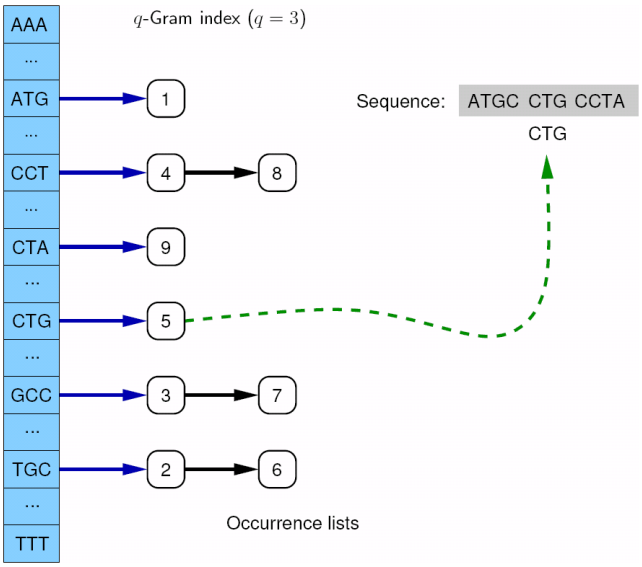
	$q = 11$								
τ	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
n_0	28	29	41	42	43	44	45	46	47
w	39	40	52	53	54	55	67	68	69
e	2	2	3	3	3	3	4	4	4

6.4 Algorithm

The SWIFT algorithm relies on the q-gram filter for ϵ-matches of length n_0 or greater. Using the parameters obtained from Lemma 2, it searches for all $w \times e$ parallelograms which contain a sufficient number of q-grams.

In the preprocessing step, we construct a q-gram index for the target sequence A. The index consists of two tables:

1. The *occurrence table* is a concatenation of the lists $L(G) := \{ i \mid A[i..i + q - 1] = G \}$ for all q-grams $G \in \Sigma^q$ in A.
2. The *lookup table* is an array indexed by the natural encoding of G to base $|\Sigma|$, giving the start of each list in the occurrence table.



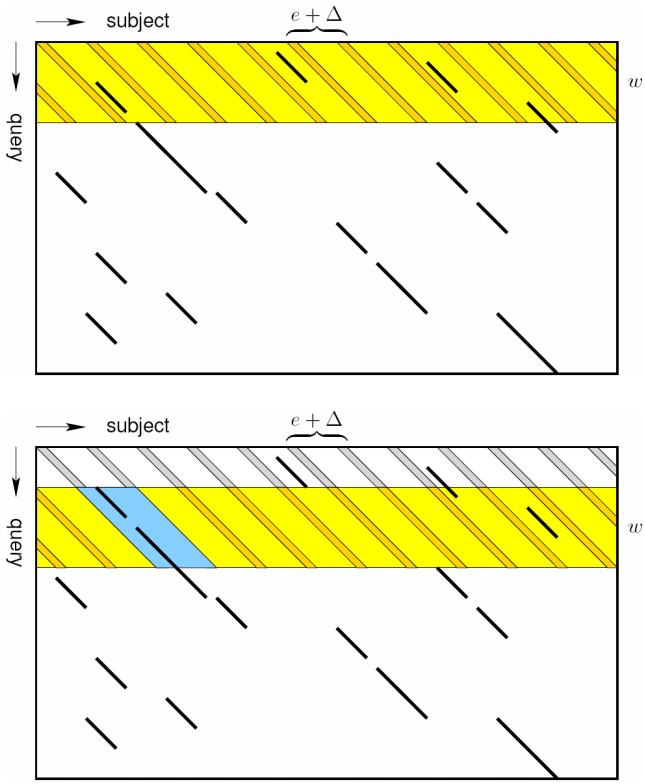
Once the q-gram index is built, the $w \times e$ parallelograms containing τ or more q-hits can be found using a simple sliding window algorithm.

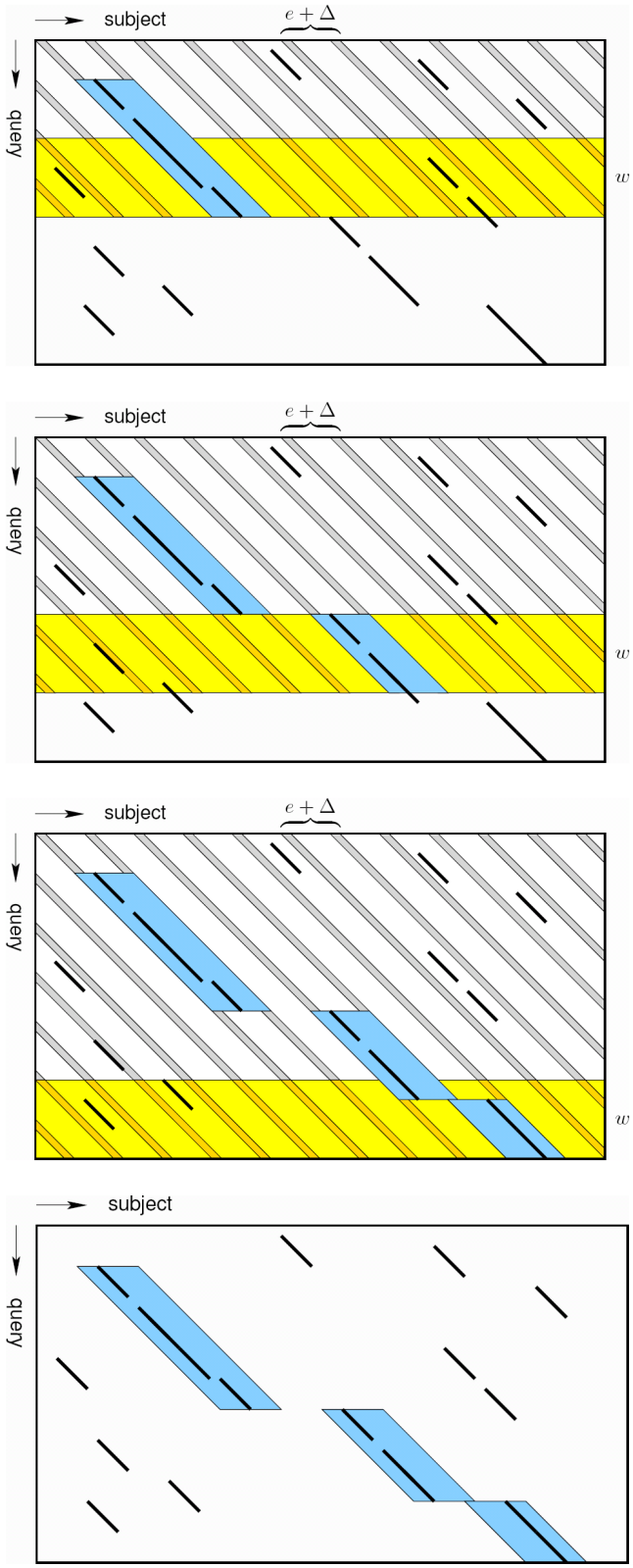
The idea is to split the (fictitious) edit matrix into overlapping *bins* of $e+1$ diagonals. For each bin we count the number of q-hits in the $w \times e$ parallelogram that is the intersection of the diagonals of the corresponding bin and the rows of the sliding window $W_j := B[j..j+q-1]$.

As the sliding window proceeds to W_{j+1}, the bin counters are updated to reflect the changes due to the q-grams leaving and entering the window.

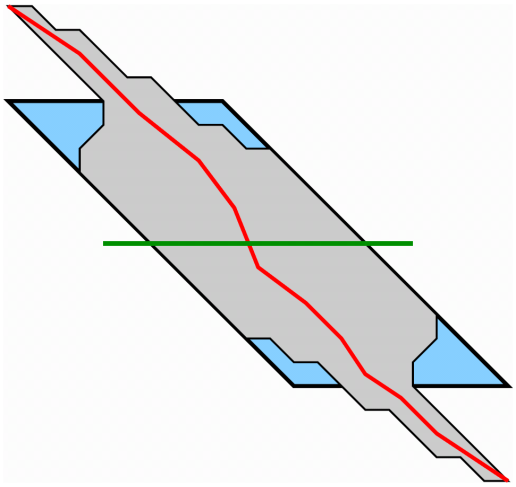
Whenever a bin counter reaches τ, the corresponding parallelogram is reported. Overlapping parallelograms can be merged on the fly.

The space requirement for the bins is reduced by searching for somewhat larger parallelograms of size $w \times (e+\Delta)$. Then each bin counts for $e+\Delta+1$ diagonals, and successive bins overlap by e diagonals. While this will lead to more verifications, it reduces the number of bins which have to be maintained. In practice, Δ is set to a power of 2, and bin indices are computed with fast bit-operations.





Each 'candidate' parallelogram must be checked for the presence of an ε-match. This can be done trivially by dynamic programming. Alternatively, one can use the knowledge about the q-grams in the ε-match to construct an alignment by sparse dynamic programming.



Algorithm 2: Filter for identifying parallelograms for ϵ-matches

Input : Query B; q-gram index I for target A; parameters w, e, τ; and $\Delta = 2^z$
Output: Set of parallelograms P

```
1  Allocate and initialize array of bin records Bins
2  P ← ∅
3  for j ← 0 to |B| − q do
4      G ← B[j, j + q − 1]
5      L(G) ← lookup occurrence list for G in I
6      foreach i ∈ L(G) do
7          d ← |A| + j − i
8          b_0 ← d ≫_bit z
9          b_m ← b_0 mod |Bins|
10         P ← P ∪ UpdateBin(Bins[b_m], j, b_0 ≪_bit z)
11         if (d &_bit (Δ − 1)) < e then
12             b_m ← (b_m + |Bins| − 1) mod |Bins|
13             P ← P ∪ UpdateBin(Bins[b_m], j, (b_0 − 1) ≪_bit z)
14         if (j − e) mod (Δ − 1) = 0 then
15             b_0 ← (j − e) ≫_bit z
16             b_m ← b_0 mod |Bins|
17             /* CheckAndResetBin is similar to lines 3–8 of UpdateBin */
18             P ← P ∪ CheckAndResetBin(Bins[b_m], j, b_0 ≪_bit z)
19 P ← P ∪ { remaining parallelograms in Bins }
```

Algorithm 1: `UpdateBin`(r, j, d)

Input : Bin record r; q-hit position j in sequence B; and offset bin diagonal d.
Output: Empty or singleton parallelogram set P.

```
1  P ← ∅
2  if j − w + q > r.max then
3      if r.count ≥ τ then
4          p.left ← |A| − d
5          p.top ← r.max + q
6          p.bottom ← r.min
7          P ← { p }
8      r.count ← 0
9  if r.count = 0 then
10     r.min ← j
11 if r.max < j then
12     r.max ← j
13     r.count ← r.count + 1
14 return P
```

6.5 Results

TABLE 3. FILTRATION RATIOS AND TIMES FOR EST ALL-AGAINST-ALL COMPARISON

	SWIFT		*QUASAR*			
	Filtration		*Filtration, best ratio*		*Filtration, best time*	
(ϵ, n_0)	*Ratio*	*Time (s)*	*Ratio*	*Time (s)*	*Ratio*	*Time (s)*
(0.05, 50)	$6.5 \cdot 10^{-6}$	6.0	$4.5 \cdot 10^{-4}$	36.1	$2.1 \cdot 10^{-3}$	4.2
(0.04, 30)	$4.5 \cdot 10^{-6}$	5.0	$4.0 \cdot 10^{-4}$	69.0	$3.1 \cdot 10^{-3}$	4.4
(0.05, 30)	$5.4 \cdot 10^{-6}$	6.1	$4.3 \cdot 10^{-4}$	68.5	$3.5 \cdot 10^{-3}$	4.4

TABLE 2. RUNNING TIMES FOR EST ALL-AGAINST-ALL COMPARISON[a]

	SWIFT			*BLAST*	*SSEARCH*
(ϵ, n_0)	(0.05, 50)	(0.04, 30)	(0.05, 30)	—	—
Running time	18 s	29 s	35 s	773 s	8 h

[a]The time for the database formatting and preprocessing in BLAST (3 s) and SWIFT (12 s) is not included.