UNIVERSITY OF ZAGREB, FACULTY OF ELECTRONICS AND COMPUTING

Computing optimal tresholds for q-gram filters

Stipe Kuman, Dino Radaković, Leon Rotim

January 14, 2016

1 INTRODUCTION INTO THE PROBLEM

The focus of this assignment was finding optimal thresholds for q-gram filters using the algorithm described in [1]. The emphasis is on gapped q-grams, discontinuous patterns of q matching positions which can be used to efficiently discard large portions of text prior to using of actual string search algorithms to find matches (filtering), using a metric called q-gram similarity (the number of q-grams shared by two strings). Gapped q-grams were shown to be more efficient than contiguous q-grams in the right conditions, according to [1]. However, computing an optimal threshold for gapped q-gram filters is also more difficult, as opposed to a closed-formula in the case of contiguous q-grams. This project was implemented in C++ (adhering to the C++11 standard) and can be built on the Bio-Linux [2] platform, without any additional dependencies. In the next section we describe the problem, after which we'll describe the original algorithm, alongside our implementation details, followed by the results produced using our implementation.

1.1 PROBLEM DESCRIPTION

First off, we will describe the notation used in this assignment. T is the text file, P is the given pattern and S stands for all the substrings in T that match the given pattern P within a Hamming distance of k (number of non-matching characters). The length of the q-gram filter is q and the threshold is represented by t. Finally, m is the length of P and S.
The goal of the q-gram filter is to reduce the number of potential matches that need to be compared to the given pattern P. The threshold variable t defines the minimum number of q-gram matches between the pattern P and a substring so that the substring would be considered as a potential match. A low value of t will result in too many potential matches, while a too high value may overlook some potential matches which would make the filter lossy.

Only filters that label all the matches as potential matches will be considered in the assignment.
As stated previously, this assignment will focus on gapped q-grams. In gapped q-gram filters the problem is much more complex because a closed form formula has not been found so we have to use an algorithm to find the optimal treshold.
Because we are dealing with gapped q-grams, which we represent with sets, we will have to define a q-shape and some additional terms. The size of a set Q is the number of indexes that it has, while the span is the distance between te lowest and highest member of the set, or in other words *span(Q) = maxQ - minQ + 1* . The *(q,s)-shape* of a q-gram is a set Q with it's size q and span s. For example, if we have a shape Q={0,1,4,6,7} it's size is 5 and it's span is 8 so it is a (5,8)-shape.

2 COMPUTING THE OPTIMAL THRESHOLD

2.1 THE RECURRENCE

A gapped q-gram can be represented by a set of indices denoting matching positions, Q, over a *span* $s = \max_{i \in Q} i - \min_{i \in Q} i$. With this representation in mind, we can define the optimal threshold $t_Q(m,k)$ for a given gapped q-gram Q, on any pattern and its potential match of length m, where the the Hamming distance between the two equals k.
In order to devise an algorithm in terms of optimal substructure, the authors [1] also introduce a third parameter $M \subseteq \{1,2,\ldots,s-1\}$, which represents matches between the two strings at the last $s-1$ positions. As an example, $M = \{1,3,4\}$ with $s = 6$ would represent a match between the strings' characters at positions $(m-4)$, $(m-2)$ and $(m-1)$.

Let $cond(b)$ be 1 if the boolean expression b holds and 0 otherwise. Let $M \oplus x = \{m + x | m \in M\}$ ($M \ominus x$ being the inverse). S The introduced conditional threshold can be computed via the following recurrence:

$$
\begin{aligned}
t_Q(s-1,j,M) &= 0 \\
t_Q(i,j,M) &= \min \begin{cases} t_Q(i-1, j - cond(s-1 \notin M), (M \cup \{0\} \setminus \{s-1\}) \oplus 1) \\ \qquad + cond(Q \subseteq M \cup \{0\}) \\ t_Q(i-1, j - cond(s-1 \notin M), (M \setminus \{s-1\}) \oplus 1) \end{cases}
\end{aligned}
$$

The recurrence above is correct if the following three conditions hold:

1. $s \leq i \leq m$
2. $0 \leq j \leq k$
3. $|M| \geq s - 1 - j$

The first of the given correctness conditions needs to be satisfied because a q-gram of span s wouldn't fit in the given string unless its length were at least as large as the q-gram's span.

The third condition must hold due to the lowest possible number of matches at the last $s-1$ positions with j mismatches in the whole string being $s-1-j$, thus representing the case in which all of the mismatches occur in the last $s-1$ characters of the string.

Another property of the recurrence that's worth mentioning is the case where $j = 0$. Obviously, for two strings to match completely, the q-gram similarity must be maximal. This also implies $M = 1, 2, \ldots, s-1$, and, implicitly, avoiding the second case of the recursion, due to the fact that it generates an invalid set of matches (which violates the third of he given correctness conditions). Therefore, the recurrence will grow in value with $j = 0$, dropping towards the final solution, for given m, k, and M. The recurrence itself can be thought of as a sliding window, going from the end of the original string of length m, towards its beginning, enumerating all possible combinations of mismatches in the process (hence the binary choice at each step).

Once computed, the conditional thresholds can be used to derive the optimal threshold for m and k:

$$t_Q(m,k) = \min_{M \subseteq \{1,2,\ldots,s-1\}, |M| \geq s-1-k} t_Q(m,k,M)$$

The authors claim the impossibility of computing $t_Q(m,k)$ from $t_Q(m-1,k)$ alone [1].

Judging by the branching required to compute each $t_Q(i,j,M)$ recursively, the given recurrence would obviously require asymptotically exponential time to compute. However, it does exhibit an *optimal substructure* – each step in the recurrence (the min-aggregation) takes $\mathcal{O}(1)$ time. The subproblems are also *overlapping*, meaning that a same set of recurrence parameters ought to appear multiple times if the recurrence is solved recursively, in a naïve manner.

2.2 A DYNAMIC PROGRAMMING APPROACH

Exploiting the optimal substructure and overlapping subproblems of the recurrence given in subsection 2.1 leads to a dynamic programming algorithm. Simply put, the algorithm is based on initializing a three-dimensional array with the base case, setting positions $[s-1, j, M]$ to 0 and applying the recurrence in a bottom-up fashion, starting from positions $[s, *, *]$. We represent each set of matches M using a 64-bit integer, due to the maximum feasible (in terms of complexity and computational resources) spans, as stated in the original paper's experiments [1], do not exceed 64. Such a representation allows us to perform set arithmetic efficiently, using bitwise operations, which, when compiled, tend to map to single machine instructions. As an example, '$\oplus$' amounts to a single bit-shifting operation.

When encountering a problem (a set of parameters i, j and M) which doesn't satisfy one of the correctness criteria listed in 2.1, we mark it as incorrect. When one of the subproblems of a problem happens to be marked as incorrect, we ignore it and assign the value based on the other subproblem. When both subpro<blems of a (otherwise correct) problem happen to be marked as incorrect, we mark the problem itself as incorrect, too.

2.3 Asymptotic complexity

The dynamic programming approach significantly reduces asymptotic time complexity by eliminating $\mathcal{O}(2^m)$ as a factor, reducing it to $\mathcal{O}(m)$. Solving a problem based on two subproblems is done in $\mathcal{O}(1)$ with a single comparison of the subproblems' solutions (min). In order to compute $t_Q(m,k)$, one needs to compute $t_Q(m,k,M)$, for all valid values of M.

For given m and k on a q-gram of span s, there are $\sum_{e=0}^{k}\binom{s-1}{e}$ possible arrangements of k mismatches on $(s-1)$ positions (complementary to sets M), which is $\mathcal{O}(2^k)$ [1]. This ultimately leads to asymptotic time complexity of $\mathcal{O}(mk^2 2^k)$, due to our set operations on M (except copying) being $\mathcal{O}(1)$, whereas copying is done in $\mathcal{O}(k)$ time, which accounts for the additional factor $\mathcal{O}(k)$. Different types of sets (possibly purely functional reference-based sets) would account for different, perhaps improved asymptotic time complexities.
We avoid memorizing the whole three-dimensional dynamic programming array, and only keep each i-row and $(i-1)$-row instead, because we ultimately only need to compute the row $t_Q(i,*,*)$. This leads to space complexity of $\mathcal{O}(k^2 2^k)$. The additional $\mathcal{O}(k)$ factor comes from sets used to represent matches (M).

2.4 Pruning method

Even with this reasonably fast dynamic programming algorithm, computing all possible gapped Q-grams with positive threshold for parameters $m = 50$, $k = \{4,5\}$ is still a challenging task. Using brute force method, calculating threshold for all shapes that have $m = 50$ and $k = \{4,5\}$ and leaving only those for which positive threshold is computed, would leave us with a search space of $2^{50} (\approx 10^{15})$ thresholds to compute. Since that kind of search space is clearly unfeasible we use following lemma as stated in [1] to reduce the search space.

Lemma 2.1. *If $Q' \subseteq Q$ then $t_{Q'}(m,k) \geq t_Q(m,k)$*

Using lemma 2.1 we propose a simple pruning technique which we used to speed up computing all Q-shapes with positive thresholds.
We introduce parameter MS as the maximal span size of all Q-grams that will considered. Obviously $MS \leq m$. This parameter is important because its value drastically changes search space.
Let $Set_{current}$ be a set which contains all Q-grams with size q, which respective thresholds had been computed. Next we define two empty sets, Set_{next} and $Set_{forbidden}$, which will contain Q-grams of size $q+1$. After running pseudocode 1 Set_{next} will contain all Q-grams of size $q+1$ that can not be eliminated with lemma 2.1.

[1] the authors provide tighter bounds for both time and space complexity using the sum of binomial coefficients [1]

Algorithm 1 Generating candidate set of Q-grams which may have positive threshold

function generateNextCancidateSet ($Set_{current}, MS$)

Input : $Set_{current}$ - a set of Q-grams of size q, MS - maximal value of span for all considered Q-grams

Output: Set_{next} - set of Q-grams of size q+1 that can't be eliminated using lemma 2.1

```
Set_next ← ∅
Set_forbidden ← ∅
for each Q-gram Q_i ∈ Set_current do
  for i ∈ {2,...,MS} \ Q_i do
    Q'_j = Q_i ∪ i
    if threshold(Q_i) == 0 then
      Set_next ← Set_next \ Q'_j
      Set_forbidden ← Set_forbidden ∪ Q'_j
    else if Q'_j ∉ Set_forbidden then
      Set_next ← Set_next ∪ Q'_j
    end if
  end for
end for
```

return Set_{next}

In more detail if some Q_i has a non positive threshold none of his Q'_j supersets may appear in Set_{next}. We ensure that by explicitly removing all such Q'_j from Set_{next} and adding it to $Set_{forbidden}$. $Set_{forbidden}$ is needed because there can be multiple Q_i with same superset Q'_j and if there is at least one such Q_i with $t_{Q_i} = 0$ we mustn't include Q'_j in Set_{next}.
This rule follows from lemma 2.1 from which we know that $t_{Q'_j}$ must be 0. $Set_{forbidden}$ therefore ensures that some Q-gram Q'_j will not be included in Set_{next} if we earlier determined that its threshold must be 0.
Using this method (Algorithm 1) we can compute thresholds for all Q-grams with positive thresholds with $m = 50$ and $k = \{4,5\}$ as presented in following algorithm 2.

Algorithm 2 Pruning technique for computing all Q-grams with positive threshold

function findAllQgramsWithPositiveThreshold (m, k, MS)

Input: m - size of a pattern, k - number of miss-matches, MS - maximal value of span for all considered Q-grams

```
Set_current ← {Q_start}, where Q_start = {0}
while Set_current ≠ ∅ do
  for each Q_i ∈ Set_current do
    if threshold(Q_i) > 0 then
      add pair (Q_i, threshold(Q_i)) to results
    end if
    Set_current ← generateNextCancidateSet(Set_current, MS)
  end for
```

Using the method above we are basically performing space search where in each iteration $Set_{current}$ contains N Q-gram shapes whose threshold values are to be evaluated and, if positive, submitted to results.
With this simple pruning method we drastically reduced search space by several orders of magnitude. Leaving about 10^8 thresholds to compute in order to acquire all Q-shapes with positive thresholds with $m = 50$ and $k = \{4,5\}$.

2.5 Results and performance

Here we present performance and results obtained using dynamic programming algorithm and pruning method presented in sections 2.2 and 2.4 respectively.
Amount of data that is needed while running dynamic programming recurrence depends on different values of k and $span$. While computing all positive shapes for $m = 50$ and $k = \{4,5\}$ we ran cases $k = 4$ and $k = 5$ separately. So when running computation value of k will be fixed and there will be at most 50 different values of $span$. Before we start computing threshold and running our pruning method we precompute data for all different values of $span$ that will be used in dynamic programming recurrence resulting in drastic time performance gain. Precomputation finishes within 30 seconds (for all values of span). It would be really expensive recomputing this data for each shape when computing its threshold. To show significance of precomputation we generated 1000 random Q-gram examples with span 50. Using naive implementation, without precomputation, it took 3421.21 seconds to compute all 1000 thresholds. Using precomputation all thresholds were computed in 13.78 seconds.
As mentioned in section 2.2 memory usage depends on parameters m an k. For $k = 5$ precomputed data for DP recurrence mentioned earlier sums up to $\approx 1GB$ of RAM which is more then feasible for most PC we have in use today. With $k = 4$ memory usage is considerably smaller using only $\approx 120MB$ of RAM.
To get a better view of time performance we generated random Q-grams shapes and measured time of their respective threshold computation. For each value of $span$ ranging from $span = 10$ up to $span = 50$ we generated 100 random shapes. Time measurements are presented in Table 2.1 and graphically on Figure 2.5.

Span	CPUs			
	1	2	3	4
10	136627.05	126988.41	90977.85	68577.47
11	196651.00	183161.20	128614.36	100547.80
12	285034.06	262230.65	183353.10	143803.90
13	533504.01	368306.46	254942.96	201404.48
14	535976.65	499980.73	353007.54	276948.10
15	699336.82	681061.92	480749.87	372993.07
16	947119.22	910288.81	639657.39	520388.72
17	1237340.15	1203045.88	852631.28	644182.21
18	1611819.83	1532784.16	1079991.32	829194.88
19	2063239.34	2064853.58	1380863.72	1049174.62
20	2407162.88	2491824.53	1839472.54	1307171.31
21	3260196.25	2987138.81	2064527.85	1606242.88
22	3497564.29	3593302.42	2241717.10	1959988.34
23	4324972.57	4422506.20	3122416.46	2377207.61
24	5068956.12	5348155.24	3645893.90	2896774.42
25	5992657.58	6217346.25	4414702.87	3496285.09
26	8976574.14	7492514.17	5405795.41	4268631.13
27	8873869.08	8699772.02	6426091.94	5023584.94
28	10813056.63	10435265.04	7868119.10	6403527.86
29	13002752.35	12339981.24	9297495.81	7423738.08
30	15665078.52	14137244.21	11287121.10	8520270.27
31	17312777.16	12356572.02	12892359.47	11897454.56
32	19742031.53	17242779.55	16276930.99	13006160.85
33	21898560.03	20751728.32	18210394.64	13350733.70
34	25402239.90	21747892.16	18888410.65	15011996.75
35	26162620.93	21964907.21	20115109.16	19169146.94
36	28575425.98	25511574.68	24705952.64	20285299.67
37	32071472.73	29122202.36	25281911.88	23319477.19
38	33983451.96	30944120.06	28064518.11	24578031.54
39	35790107.91	29812912.16	29916421.11	31401271.26
40	36941464.55	33435276.40	26194934.28	28235519.12
41	38408097.05	31553883.27	31017630.10	28979277.53
42	41310449.19	32248354.62	32557746.20	26518729.54
43	41856719.41	36764295.47	33454317.76	32855935.33
44	43877883.72	35791089.10	30272635.14	31065587.07
45	40160153.92	31261790.44	31136853.20	30336145.68
46	38423015.43	33199604.60	28637212.85	22360396.87
47	34861463.42	29926723.66	27191496.25	25294109.14
48	28567696.51	23252670.42	22592821.10	20204953.65
49	22246720.09	19196621.75	15977201.05	15551445.28

Table 2.1: Time needed to compute a single shape with given span (averaged on 100 evaluation) depending on number of used threads in nanoseconds

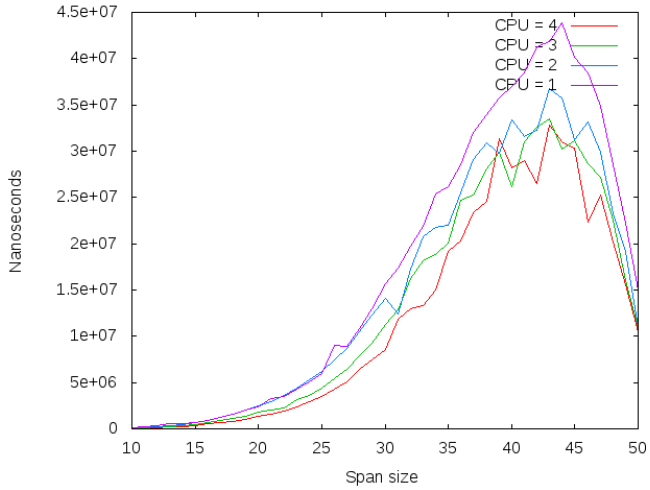

Figure 2.1: Time needed to compute a single shape with given span (averaged on 100 evaluations) depending on number of used threads (CPUs) in nanoseconds

While the implemented dynamic programming algorithm is reasonably fast, the number of thresholds that has to be computed is extremely large (up to 10^8). Therefore we decided to achieve linear factor speed-up using multi-threading. Time performances acquired are presented 2.1.
To reduce search space we first ran computation considering only shapes with $span < 30$. With parameters $m = 50$, $k = 5$ and $MS = 30$ (maximal span of considered shapes) a total of 1588439 shapes with positive threshold were evaluated. Experiments were ran on Intel $i7$ using 4 threads. Complete search time was 15906 seconds (4.4183 hours).
With parameters $m = 50$, $k = 4$ and $MS = 30$ a total of 13104841. Complete search time was 19173.528 seconds (5.3259799 hours).
Results obtained using presented methods are showed in tables 2.2 and 2.3. We note that tables do not contain threshold values for all possible shapes. Even with our effective pruning technique number of thresholds that has to be computed is still large. Due to the imposed deadline and shortage of computational power experiments that were ran with parameters $m = 50$, $k = \{4,5\}$ and $MS = 50$ didn't complete the entire search space. For $k = 4$ only Q-gram shapes with $q \leq 6$ were evaluated and for $k = 5$ only Q-gram shapes with $q \leq 6$ were evalated. Results for complete search space can surely be computed in matter of days.

Span	q												
	2	3	4	5	6	7	8	9	10	11	12	13	14
2	41	-	-	-	-	-	-	-	-	-	-	-	-
3	40	36	-	-	-	-	-	-	-	-	-	-	-
4	39	35	31	-	-	-	-	-	-	-	-	-	-
5	38	34	30	26	-	-	-	-	-	-	-	-	-
6	37	33	29	25	21	-	-	-	-	-	-	-	-
7	36	32	28	24	20	16	-	-	-	-	-	-	-
8	35	31	27	23	19	15	11	-	-	-	-	-	-
9	34	30	26	22	18	14	10	6	-	-	-	-	-
10	33	29	25	21	17	13	9	5	1	-	-	-	-
11	32	28	24	20	17	14	10	7	4	0	-	-	-
12	31	27	23	20	17	13	10	8	5	2	0	-	-
13	30	26	22	19	16	13	10	8	6	3	1	0	-
14	29	25	21	18	15	12	10	8	5	4	2	1	0
15	28	24	20	17	14	12	9	7	5	3	2	1	0
16	27	23	19	16	14	11	9	7	5	3	2	1	0
17	26	22	18	16	13	11	9	7	5	4	2	1	0
18	25	21	17	15	12	10	8	6	5	3	2	1	0
19	24	20	17	14	12	9	7	6	4	3	2	1	1
20	23	19	16	13	11	9	7	5	4	3	2	1	0
21	22	18	15	12	10	8	6	5	3	2	2	1	0
22	21	17	15	12	9	7	6	4	3	2	1	1	0
23	20	17	14	12	9	7	5	4	2	2	1	0	0
24	19	17	15	12	10	7	5	4	2	1	1	0	0
25	20	17	15	12	10	7	5	4	3	2	1	0	0
26	21	17	16	12	11	8	6	4	3	2	1	1	0
27	20	16	16	12	12	8	7	5	4	3	2	1	0
28	19	15	15	11	11	8	7	6	4	3	2	1	1
29	18	14	14	10	10	8	7	5	4	3	2	1	1
30	17	13	13	9	9	7	6	4	4	3	2	1	1
31	16	12	12	9	9	?	?	?	?	?	?	?	?
32	15	11	11	9	7	?	?	?	?	?	?	?	?
33	14	12	10	8	8	?	?	?	?	?	?	?	?
34	13	12	9	8	8	?	?	?	?	?	?	?	?
35	12	12	8	8	8	?	?	?	?	?	?	?	?
36	11	11	7	7	7	?	?	?	?	?	?	?	?
37	10	10	8	6	6	?	?	?	?	?	?	?	?
38	9	9	8	5	5	?	?	?	?	?	?	?	?
39	8	8	8	5	4	?	?	?	?	?	?	?	?
40	7	7	7	5	3	?	?	?	?	?	?	?	?
41	6	6	6	6	4	?	?	?	?	?	?	?	?
42	5	5	5	5	4	?	?	?	?	?	?	?	?
43	4	4	4	4	4	?	?	?	?	?	?	?	?
44	3	3	3	3	3	?	?	?	?	?	?	?	?
45	2	2	2	2	2	?	?	?	?	?	?	?	?
46	1	1	1	1	1	?	?	?	?	?	?	?	?

Table 2.2: Table showing maximal thresholds for Q-gram shapes with regard to size(q) and span of the shape for $m = 50$ and $k = 4$

Span	q										
	2	3	4	5	6	7	8	9	10	11	12
2	39	-	-	-	-	-	-	-	-	-	-
3	38	33	-	-	-	-	-	-	-	-	-
4	37	32	27	-	-	-	-	-	-	-	-
5	36	31	26	21	-	-	-	-	-	-	-
6	35	30	25	20	15	-	-	-	-	-	-
7	34	29	24	19	14	9	-	-	-	-	-
8	33	28	23	18	13	8	3	-	-	-	-
9	32	27	22	18	14	9	5	0	-	-	-
10	31	26	21	18	13	10	6	3	0	-	-
11	30	25	20	16	13	10	7	4	2	0	-
12	29	24	19	16	12	9	7	4	2	0	0
13	28	23	19	15	12	9	6	4	2	1	0
14	27	22	17	14	11	8	6	4	2	1	0
15	26	21	17	13	10	8	5	3	2	1	0
16	25	20	16	13	10	7	5	3	2	1	0
17	24	19	15	12	9	7	5	3	2	1	0
18	23	18	14	11	8	6	4	3	2	1	0
19	22	17	14	11	8	6	4	2	1	1	1
20	21	16	13	10	7	5	3	2	1	1	0
21	20	15	12	9	7	5	3	2	1	0	0
22	19	15	12	9	6	4	2	1	1	0	0
23	18	15	12	9	6	4	2	1	0	0	0
24	18	15	13	9	7	4	2	1	0	0	0
25	19	15	13	9	7	4	3	1	1	0	0
26	20	15	14	9	8	5	3	2	1	0	0
27	19	14	14	9	9	6	4	2	1	1	0
28	18	13	13	8	8	5	4	3	2	1	0
29	17	12	12	8	8	5	5	2	2	1	0
30	16	11	11	7	6	4	4	2	2	1	0
31	15	10	10	7	7	?	?	?	?	?	?
32	14	10	9	7	5	?	?	?	?	?	?
33	13	11	8	6	6	?	?	?	?	?	?
34	12	11	7	6	6	?	?	?	?	?	?
35	11	11	6	6	6	?	?	?	?	?	?
36	10	10	6	5	5	?	?	?	?	?	?
37	9	9	7	4	4	?	?	?	?	?	?
38	8	8	7	4	3	?	?	?	?	?	?
39	7	7	7	4	3	?	?	?	?	?	?
40	6	6	6	4	2	?	?	?	?	?	?
41	5	5	5	5	3	?	?	?	?	?	?
42	4	4	4	4	3	?	?	?	?	?	?
43	3	3	3	3	3	?	?	?	?	?	?
44	2	2	2	2	2	?	?	?	?	?	?
45	1	1	1	1	1	?	?	?	?	?	?

Table 2.3: Table showing maximal thresholds for Q-gram shapes with regard to size(q) and span of the shape for $m = 50$ and $k = 5$

2.6 CONCLUSION

In order to compute optimal threshold for gapped q-grams we used the dynamic programming algorithm described in [1], as well as a pruning technique, as described in in 2.2 and 2.4, respectively. Even though an implementation of the dynamic programming algorithm is a both practical and asymptotic improvement as opposed to the trivial recursive approach, it's still exponential (in terms of time and space complexity) in the number of mismatches between the two strings, which requires significant amount of computational power to compute in reasonable time. Had we employed more machines, we would have evaluated a proportionally larger number of q-grams, thus (hopefully) finding a proportionally larger number of those with non-negative optimal thresholds as well. This is due to the fact that the solution can be (almost) trivially parallelized. The results we've managed to compile so far match exactly with those shown in the authors' original peer-revied paper, meaning our program can safely be used to find q-grams with non-negative thresholds which can be used to perform efficient filtering prior to string searching. We conclude with our conjecture that the greatest improvements in solving this problem can be done by either improving the asymptotic time complexity of the algorithm itself (with both a reasonable constant time overhead and space complexity) or implementing a solution which uses computers' caches in a more efficient way, in order to achieve a significant constant-time speedup.

REFERENCES

[1] Stefan Burkhardt and Juha Kärkkäinen. Better filtering with gapped q-grams. *Fundam. Inf.*, 56(1-2):51–70, January 2003.

[2] Dawn Field, Bela Tiwari, Tim Booth, Stewart Houten, Dan Swan, Nicolas Bertrand, and Milo Thurston. Open software for biologists: from famine to feast. *Nature biotechnology*, 24(7):801–803, 2006.