Speeding-Up the Dynamic Programming Procedure for the Edit Distance of Two Strings

Giuseppe Lancia[1(✉)] and Marcello Dalpasso[2]

[1] Dipartimento di Scienze Matematiche, Informatiche e Fisiche, University of Udine, Via delle Scienze 206, 33100 Udine, Italy
giuseppe.lancia@uniud.it

[2] Dipartimento di Ingegneria dell'Informazione, University of Padova, Via Gradenigo 6/A, 35131 Padova, Italy
marcello.dalpasso@unipd.it

Abstract. We describe a way to compute the edit distance of two strings without having to fill the whole dynamic programming (DP) matrix, through a sequence of increasing guesses on the edit distance. If the strings share a certain degree of similarity, the edit distance can be quite smaller than the value of non-optimal solutions, and a large fraction (up to 80–90%) of the DP matrix cells do not need to be computed. Including the method's overhead, this translates into a speedup factor from 3× up to 30× in the time needed to find the optimal solution for strings of length about 20,000.

1 Problem and Notation

Let Σ be an alphabet and s', s'' be two strings over Σ. We can always turn s' into s'' through a sequence of three basic operations:

- *Deletion* of a symbol σ of s': cost $\texttt{del}(\sigma)$.
- *Insertion* of a symbol τ of s'' into s': cost $\texttt{ins}(\tau)$.
- *Substitution* of a symbol σ of s' with a symbol $\tau \neq \sigma$ of s'': cost $\texttt{sub}(\sigma, \tau)$.

The cost of the sequence is the sum of the costs of the individual operations. The *Edit Distance* problem calls for computing a sequence of operations of minimum cost, called the *edit distance* of s' and s'', here $d(s', s'')$. We assume the costs are positive (so that $d(s', s'') = 0 \Rightarrow s' = s''$) and $s' \neq s''$, so that $d(s', s'') > 0$. For convenience, we define $\texttt{sub}(\sigma, \sigma) := 0$ for all σ.

Usually the costs are represented in the form of a *substitution matrix*, i.e., a square matrix of order $|\Sigma| + 1$, with 0's on the diagonal. The last row and column of the substitution matrix are associated to an extra symbol '-', called *gap*, and contain the costs of insertions and deletions.

Some substitution matrices (here called *simple*) have the property that all insertions and deletions have the same cost (the *indel* cost, IND) and all substitutions have the same cost (SUB). For instance the standard substitution matrix

G. Anderst-Kotsis et al. (Eds.): DEXA 2019 Workshops, CCIS 1062, pp. 59–66, 2019.
https://doi.org/10.1007/978-3-030-27684-3_9

for the edit distance (called `ones.mat` here) is simple, with $\texttt{IND} = \texttt{SUB} = 1$. Another popular simple substitution matrix is used for DNA comparison and has $\texttt{SUB} = 15$ and $\texttt{IND} = 10$: we call this matrix `dna.mat`. In general, in a substitution matrix we can assume that $\texttt{sub}(\sigma, \tau) < \texttt{del}(\sigma) + \texttt{ins}(\tau)$ (otherwise no substitution would ever be made since it can be better mimicked by a deletion followed by insertion). For a simple matrix this becomes $\texttt{SUB} < 2\,\texttt{IND}$.

The edit distance problem, first investigated by Levenshtein in [5] (and therefore also known as *Levenshtein distance*), is a classic of string-related problems [3], with main applications in the field of bioinformatics [4]. In this context, Σ is either the alphabet of the 4 nucleotides or of the 20 amino acids, and the problem is solved to determine the similarity between two genomic sequences (i.e., to *align* them in the best possible way). It is effectively solved via a $\Theta(nm)$ dynamic programming (DP) procedure [2,6], where $n = |s'|$ and $m = |s''|$.

The DP procedure is a two-level nested `for` cycle which fills a table $\texttt{P}[\cdot, \cdot]$ of $n + 1$ rows and $m + 1$ columns. At the end, the value of a generic entry $\texttt{P}[i, j]$ with $0 \le i \le n$ and $0 \le j \le m$ is equal to the edit distance between the prefixes $s'[1, \ldots, i]$ and $s''[1, \ldots, j]$. The standard code is like this (we assume that $\texttt{P}[x, y]$ returns $+\infty$ if either $x < 0$ or $y < 0$):

$\texttt{P}[0,0]{:=}0$
for $j := 1$ **to** m **do** $\texttt{P}[0, j] := \texttt{P}[0, j-1] + \texttt{ins}(s''[j])$
for $i := 1$ **to** n **do**
  for $j := 0$ **to** m **do**
    $\texttt{P[i,j]} := \min\{\texttt{P}[i, j-1] + \texttt{ins}(s''[j]),\ \texttt{P}[i-1, j] + \texttt{del}(s'[i]),$
    $\texttt{P}[i-1, j-1] + \texttt{sub}(s'[i], s''[j])\}$

Hence, $d(s', s'') = \texttt{P}[n, m]$. The corresponding optimal sequence of operations can be retrieved, for instance, by starting at the cell $v := (n, m)$ and determining which cell, among the candidates $(n, m-1)$, $(n-1, m-1)$ and $(n-1, m)$, is responsible for the value of v; then, set v to be such a cell and proceed backtracking in the matrix until $v = (0, 0)$. We say that the cells thus touched are on an *optimal path* P^* from $(0, 0)$ to (n, m).

Our Goal. In retrospective, once $\texttt{P}[\cdot, \cdot]$ has been filled and we backtrack along the optimal path, we see that there are some cells of $\texttt{P}[\cdot, \cdot]$ whose value is so large that they could have never been on an optimal path but they have been computed nonetheless. Ideally, we would have liked to fill $\texttt{P}[\cdot, \cdot]$ only in the cells of P^*, but the knowledge of these cells needs the knowledge of some other cells, adjacent to them, and these in turn need some other cells, etc., so that it might look impossible to avoid computing the value of some of the $(n+1)(m+1)$ cells, since we cannot exclude that any particular cell could belong to P^* or have an effect on P^*.

Indeed, as we will show, this is not the case and it is possible to determine a subset of cells (a sort of "stripe" $\mathcal{S}$ from $(0, 0)$ to (n, m)) which contains P^* and whose cells can be evaluated without having to evaluate any cell out of $\mathcal{S}$. The total work to find P^* would then depend on $|\mathcal{S}|$ rather than being $\Theta(nm)$. The smaller $|\mathcal{S}|$, the better.

In this paper we outline an iterative procedure to determine such an $\mathcal{S}$, starting from a tentative small stripe and progressively increasing it until it can be proved that it is large enough: as shown in the results' section, the proposed approach outperforms DP if the strings are similar enough. We must underline that a recent result [1] shows that, under a strong conjecture similar to the $P \neq NP$ belief, no algorithm of worst-case complexity $O((nm)^t)$ with $t < 1$ is likely to exist for computing the edit distance. This however does not affect our result, which is to show that it is possible to have $t < 1$ in the *best*-case (while for Dynamic Programming best and worst case take the same time), or to have the same complexity as DP but with a better multiplicative constant.

2 Guesses and Stripes

If we reverse s' and s'', obtaining r' and r'', it is obvious that $d(s', s'') = d(r', r'')$. Assume $\mathtt{P}^r[\cdot,\cdot]$ is the DP matrix for $d(r', r'')$. Then, in the same way as $\mathtt{P}[i,j]$ represents the edit distance between a length-i prefix of s' and a length-j prefix of s'', we have that $\mathtt{P}^r[h,q]$ represents the edit distance between a length-h suffix of s' and a length-q suffix of s''.

To have a consistent indexing between the two matrices, let $\mathtt{S}[i,j] := \mathtt{P}^r[n-i, m-j]$. This way $\mathtt{P}[i,j] = d(s'[1,\ldots,i], s''[1,\ldots,j])$ and $\mathtt{S}[i,j]$ denotes the best cost of completing the transformation of s' into s'', turning the suffix $s'[i+1,\ldots,n]$ into the suffix $s''[j+1,\ldots,m]$. In particular, $\mathtt{P}[i,j] + \mathtt{S}[i,j]$ denotes *the optimal cost for turning* s' *into* s'' *given that the prefix* $s'[1,\ldots,i]$ *gets turned into* $s''[1,\ldots,j]$.

Our strategy will aim at calculating only a subset of cells of $\mathtt{P}[\cdot,\cdot]$ but, in order to do so, we will also need to calculate a subset of cells of $\mathtt{S}[\cdot,\cdot]$.

Let $w_0 := (0,0)$ and $w_* := (n,m)$ be the upper left and the lower right cell, respectively. We say that two cells $a = (i,j)$ and $b = (u,v)$, with $a \neq b$, are *consecutive* (or *adjacent*) if $0 \leq u - i \leq 1$ and $0 \leq v - j \leq 1$. For two consecutive cells a and b we define a transition cost $\gamma(a,b)$ as follows: $\gamma((i,j),(i,j+1)) := \mathtt{ins}(s''[j+1])$; $\gamma((i,j),(i+1,j)) := \mathtt{del}(s'[i+1])$; and $\gamma((i,j),(i+1,j+1)) := \mathtt{sub}(s'[i+1], s''[j+1])$.

A *path* is a sequence $(v_0,\ldots,v_k)$ of cells such that $v_0 = w_0$, $v_k = w_*$ and, for each $t = 0,\ldots,k-1$, v_t and v_{t+1} are consecutive. A path has length (or cost) $\sum_{t=0}^{k-1} \gamma(v_t, v_{t+1})$. Let OPT be the value of a shortest path, i.e., $\mathrm{OPT} := \mathtt{P}[w_*]$ (or equivalently, $\mathrm{OPT} := \mathtt{S}[w_0]$): we are seeking to determine OPT.

Our approach will require to make a sequence of guesses of the value of OPT, until we guess right. Given a certain guess $\tau \in \mathbb{R}$ let us define two sets of cells: $\mathcal{M}^0(\tau) := \{v : \mathtt{P}[v] \leq \tau/2\}$ (the top matrix part) and $\mathcal{M}^*(\tau) := \{v : \mathtt{S}[v] \leq \tau/2\}$ (the bottom matrix part).

Claim 1. *Let* $\tau \in \mathbb{R}$. *If* $\tau \geq OPT$ *then there exist consecutive cells* $v \in \mathcal{M}^0(\tau)$ *and* $u \in \mathcal{M}^*(\tau)$ *such that* $\mathtt{P}[v] + \gamma(v,u) + \mathtt{S}[u] = OPT$.

Proof. Let $P^* = (x_0,\ldots,x_t)$ be the optimal path, where $x_0 = w_0$ and $x_t = w_*$. If $x_t \in \mathcal{M}^0(\tau)$ then $\mathrm{OPT} \leq \tau/2$. In this case, each $x_i \in \mathcal{M}^0(\tau)$ and also each $x_i \in$

$\mathcal{M}^*(\tau)$ so the claim is satisfied by taking $v = x_j$ and $u = x_{j+1}$ for any $j \leq t-1$. Otherwise, let j be the largest index $\leq t-1$ such that $x_j \in \mathcal{M}^0(\tau)$. Notice that this implies that $\mathtt{P}[x_{j+1}] > \tau/2$. Now, if it were $\mathtt{S}[x_{j+1}] > \tau/2$ we would have the contradiction $\mathrm{OPT} = \mathtt{P}[x_{j+1}] + \mathtt{S}[x_{j+1}] > \tau \geq \mathrm{OPT}$. Therefore, $x_{j+1} \in \mathcal{M}^*(\tau)$. By setting $v := x_j$ and $u := x_{j+1}$ we have $\mathrm{OPT} = \mathtt{P}[v] + \gamma(v,u) + \mathtt{S}[u]$.

Given τ, we call *kissing pair* any pair of consecutive cells v and u such that $v \in \mathcal{M}^0(\tau)$ and $u \in \mathcal{M}^*(\tau)$. If kissing pairs exist, we say that $\mathcal{M}^0(\tau)$ and $\mathcal{M}^*(\tau)$ *kiss*, otherwise they are *apart*. Let $\mu(\tau)$ be the minimum value of $\mathtt{P}[v] + \gamma(v,u) + \mathtt{S}[u]$ over all kissing pairs (v,u). The following test gives a sufficient condition for a guess to be too small.

Claim 2. *Let $\tau \in \mathbb{R}$. If $\mathcal{M}^0(\tau)$ and $\mathcal{M}^*(\tau)$ are apart then $OPT > \tau$.*

Let r', c' be the largest row and column touched by $\mathcal{M}^0(\tau)$ and r'', c'' the smallest row and column touched by $\mathcal{M}^*(\tau)$.

Claim 3. *Let $\tau \in \mathbb{R}$. If $(r'' - r' > 1) \wedge (c'' - c' > 1)$ then $OPT > \tau$.*

The following lemma describes an optimality condition for a guess τ:

Lemma 1. *Let $\tau \in \mathbb{R}$. If $\mu(\tau) \leq \tau$, then $OPT = \mu(\tau)$.*

Proof. $\mu(\tau) \leq \tau$ implies that $\mathcal{M}^0(\tau)$ and $\mathcal{M}^*(\tau)$ kiss, then there is a path of length $\mu(\tau)$, so that $\mathrm{OPT} \leq \mu(\tau)$. Assume $\mathrm{OPT} < \mu(\tau)$. This implies $\mathrm{OPT} < \tau$. By Claim 1, there is a kissing pair (v,u) such that $\mu(\tau) \leq \mathtt{P}[v] + \gamma(v,u) + \mathtt{S}[u] = \mathrm{OPT} < \mu(\tau)$, which is a contradiction.

Given a set X of cells, let $\mathcal{S}(X)$ be the minimum set of consecutive cells, over the various rows, which contains all X. That is, if (i,a) is the first cell of X appearing in row i, and (i,b) is the last, then all cells $\{(i,a),(i,a+1),\ldots,(i,b)\}$ are in $\mathcal{S}(X)$ and this is true for all rows i containing elements of X. To build $\mathcal{M}^0(\tau)$ we need to compute $\mathtt{P}[v]$ for all $v \in \mathcal{S}(\mathcal{M}^0(\tau))$. We will show how to compute $\{\mathtt{P}[v] : v \in \mathcal{S}(\mathcal{M}^0(\tau))\}$ in time $O(|\mathcal{S}(\mathcal{M}^0(\tau))|)$: similar considerations hold for $\mathcal{M}^*(\tau)$. The sets $\mathcal{M}^0(\tau)$ and $\mathcal{M}^*(\tau)$ are similar to some diagonal "stripes" of cells: $\mathcal{M}^0(\tau)$ goes down diagonally from the upper-left corner while $\mathcal{M}^*(\tau)$ grows diagonally from the lower-right corner. Let us call $\mathrm{PART}(\tau) := \mathcal{M}^0(\tau) \cup \mathcal{M}^*(\tau)$ this partial DP matrix.

3 The Overall Procedure

For each guess τ, our procedure actually compute only the cells belonging to $\mathrm{PART}(\tau)$. Lemma 1 implies that we would like to make the smallest possible guess which triggers the condition $\mu(\tau) \leq \tau$. By Claim 1, we could use as guess an upper bound $\mathtt{UB}$ for OPT. Then, we would compute $\mathrm{PART}(\tau)$ (this can be done in time $O(|\mathrm{PART}(\tau)|)$ and find the kissing pairs (in the same time complexity), obtaining OPT. Unfortunately, $O(|\mathrm{PART}(\tau)|)$ is significantly smaller than $\Theta(nm)$ only if the upper bound is really tight (ideally, $\mathtt{UB} \simeq \mathrm{OPT}$) and it

should be computed extremely fast to be competitive with DP: no such quick and strong bound is known for the edit distance.

We therefore proceed "bottom-up", starting with a "small", optimistic guess $\mathtt{LB} \leq OPT$ (a lower bound, or even $\tau = 0$) and then make a sequence of adjustments, increasing the guess until it is large enough to trigger the condition of Lemma 1:

1. Set $k := 0$, $\tau^0 := 0$ and compute PART(0)
2. **repeat**
3. $k := k + 1$
4. Increase the guess: $\tau^k := \mathtt{LB} + k\Delta$
5. Compute PART(τ^k) from PART(τ^{k-1}) by left/right extending each strip
6. Find the best kissing pair (v, u) in PART(τ^k)
7. Set $\mu(\tau^k) := \mathtt{P}[v] + \gamma(v, u) + \mathtt{S}[u]$ ($\mu(\tau^k) := +\infty$ if there are no kissing pairs)
8. **until** $\mu(\tau^k) \leq \tau^k$

Step 1 is straightforward: PART(0) consists of two diagonals of 0s, one starting at w_0 and being as long as the longest common prefix of s' and s'', the other one starting at w_* and being as long as their longest common suffix.

Step 5 is incremental and needs some hints. We focus on updating $\mathcal{M}^0(\tau^{k-1})$ into $\mathcal{M}^0(\tau^k)$ (updating $\mathcal{M}^*$ is similar). We can assume, inductively on k, that for each row i we know the index $\alpha_{k-1}(i)$ of the first element such that $\mathtt{P}[i, j] \leq \tau^{k-1}/2$ and the index $\omega_{k-1}(i)$ of the last element such that $\mathtt{P}[i, j] \leq \tau^{k-1}/2$ (if there is no such element in row i, then $\alpha_{k-1}(i) := m + 1$).

In row 0 $\alpha_k(0) := \alpha_{k-1}(0) = 0$, so we only extend row 0 on its right. Starting at $j := \omega_{k-1}(0) + 1$ we compute all elements $\mathtt{P}[0, j]$ and stop as soon as $\mathtt{P}[0, j] > \tau^k/2$, setting $\omega_k(0) := j - 1$. Then, proceeding inductively on i, assume we have already extended the intervals at rows $0, \ldots, i - 1$ and are working on row i. We first extend the interval to the left of $\alpha_{k-1}(i)$. Notice that in the columns $0 \leq j < \alpha_k(i - 1)$ of row i there can be no entry of value $\leq \tau^k/2$, or there would have been also one in row $i - 1$, contradicting the definition of $\alpha_k(i - 1)$. Therefore, we start at $j := \alpha_k(i - 1)$ and compute the entries $\mathtt{P}[i, j]$ from the adjacent entries $[i, j-1]$, $[i-1, j]$ and $[i-1, j-1]$ (clearly using only those whose value $\mathtt{P}[\,]$ is known). We stop as soon as $j = \alpha_{k-1}(i)$ (or, if $\alpha_{k-1}(i) = m + 1$, we stop at $j = \omega_k(i - 1) + 1$). We set $\alpha_k(i)$ to be the first j found such that $\mathtt{P}[i, j] \leq \tau^k/2$. Now we extend the strip to its right. Starting at $j := \omega_{k-1}(i) + 1$ we keep computing $\mathtt{P}[i, j]$ from the known adjacent cells. We stop as soon as all the adjacent cells are not in $\mathcal{M}^0(\tau^k)$ and set $\omega_k(i) := j - 1$.

Notice how in step 6 the time needed to find the kissing pairs is bounded by $O(\min\{|\mathcal{M}^0(\tau^k)|, |\mathcal{M}^*(\tau^k)|\})$ rather than by $O(|\mathcal{M}^0(\tau^k)| \times |\mathcal{M}^*(\tau^k)|)$. Indeed, even if each pair (v, u) is made of an element $v \in \mathcal{M}^0(\tau^k)$ and another $u \in \mathcal{M}^*(\tau^k)$, to find the kissing pairs it is sufficient to scan all the elements of the smallest set, and, for each of them, look at the adjacent cells (3 at most) to see if they belong to the other set. This test can be done in $O(1)$ time. For example, to check if $(x, y) \in \mathcal{M}^0(\tau^k)$ we first check if $\alpha_k(x) \leq y \leq \omega_k(x)$. If that is the case, $\mathtt{P}[x, y]$ is known and we check if it is $\leq \tau^k/2$.

Lower Bound and Guess Increment. The procedure we have outlined would work for any lower bound LB and any increment $\Delta > 0$ (indeed, as soon as τ^k becomes $\geq OPT$, by Claim 1 we find the optimal path), but its effectiveness depends on both these parameters. The best fine-tuning should be subject of further investigations, but we already found a quite good combination.

First we describe the lower bound used, which holds for all simple substitution matrices. Denote by s^L the longest between s' and s'' and by s^l the shortest. For each character $\sigma \in \Sigma$ and string $y \in \Sigma^*$, let $n_\sigma(y)$ be the number of occurrences of σ in y. For each symbol $\sigma \in \Sigma$ we define $\texttt{excess}(\sigma) = \max\{n_\sigma(s^l) - n_\sigma(s^L), 0\}$. Notice how the largest number of characters σ of s^l which could be possibly matched to identical characters in s^L is $n_\sigma(s^l) - \texttt{excess}(\sigma)$.

Claim 4. *The following is a valid lower bound to OPT, computed in time $O(n+m)$:*

$$LB = \left(\sum_{\sigma} excess(\sigma) \right) \times SUB + \left(|s^L| - |s^l| \right) \times IND$$

The proof that this is indeed a bound is omitted for space reasons. In our experiments we have noticed that this bound is quite strong when s' and s'' share a good deal of similarity.

In order to decide the step Δ with which we increase the guess, we opted to make this step proportional to the starting bound. By some tuning (not reported for space reasons) we determined that $\Delta := \texttt{LB}/3$ results overall in an effective procedure which terminates after a small number of iterations.

4 Computational Experiments and Conclusions

To assess the effectiveness of the proposed method we ran some experiments using an Intel® Core™ i7-7700 8CPU under Linux Ubuntu, equipped with 16 GB RAM at 3.6 GHz clock. The programs were implemented in C and compiled under `gcc 5.4.0`.

The problem instances were generated at random with a procedure based on two probabilities p_d and p_m, a base string length L, and the alphabet size. In each random instance, s' has length L and is randomly generated within the alphabet, while s'' is obtained by modifying s' as follows: each original character is deleted with probability p_d, then, if not deleted, it is mutated (randomly within the alphabet) with probability p_m. In the experiments reported here, we always used $p_d = p_m =: p$, leaving to further investigation the sensitivity to differing parameters. Clearly, lower values of p lead to more similar strings.

Table 1 reports the experiments run to compare the effectiveness of our method to the standard DP implementation. As it can be seen, the speedup is strongly dependant on p, i.e., on the string similarity: the more similar are the strings, the more effective our method is. However, even with $p = 0.2$ (i.e., the strings are quite dissimilar, differing approximately by 20% both in length and in contents), the proposed method saves about half time over DP, while it achieves an average speedup factor of 35× when $p = 0.01$.

Table 1. Comparison between our method and DP. Times are in seconds and the speedup is shown along with the filled percentage of the matrix. The alphabet size is 4.

		dna.mat				ones.mat				
String size		10000	20000	30000	40000	10000	20000	30000	40000	Average
$p = 0.01$	DP time	2.068	8.304	18.200	32.380	2.060	8.160	18.048	33.032	
	Our time	0.088	0.236	0.460	0.928	0.068	0.216	0.444	0.884	
	Speedup	23×	35×	40×	35×	30×	38×	41×	37×	**35**×
	Filled Perc.	1.8%	1.4%	1.2%	1.2%	1.3%	1.2%	1.1%	1.1%	1.3%
$p = 0.05$	DP time	2.020	8.048	18.480	32.484	2.080	8.220	19.032	32.480	
	Our time	0.352	1.164	2.848	5.944	0.236	0.868	2.040	4.076	
	Speedup	6×	7×	6×	5×	9×	9×	9×	8×	**8**×
	Filled Perc.	8.5%	7.4%	7.9%	8.3%	5.7%	5.1%	5.4%	5.7%	6.7%
$p = 0.1$	DP time	1.992	8.492	17.792	32.088	2.088	7.896	18.060	31.580	
	Our time	0.552	2.244	5.140	9.236	0.500	2.016	4.276	8.196	
	Speedup	4×	4×	3×	3×	4×	4×	4×	4×	**4**×
	Filled Perc.	13.9%	14.2%	13.5%	13.7%	12.3%	12.4%	11.9%	12.0%	13.0%
$p = 0.2$	DP time	1.772	7.548	16.092	28.524	1.780	7.104	15.900	32.976	
	Our time	1.140	4.528	9.912	18.864	0.996	3.908	8.260	15.456	
	Speedup	2×	2×	2×	2×	2×	2×	2×	2×	**2**×
	Filled Perc.	30.5%	29.1%	28.4%	28.9%	24.8%	23.8%	23.3%	23.6%	26.6%

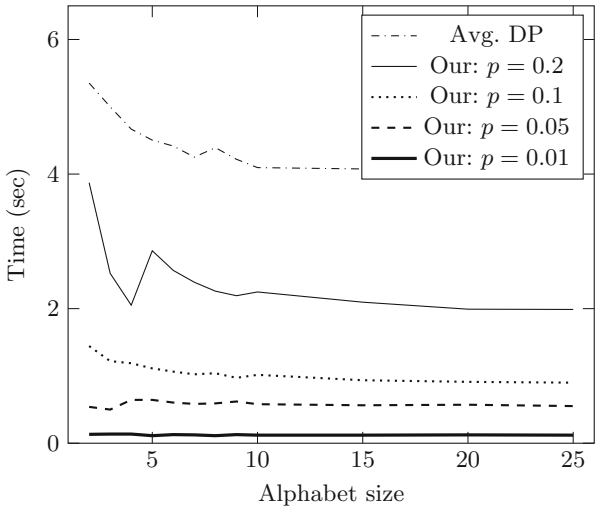

Fig. 1. Sensitivity of time performance towards the alphabet size as well as the mutation probability of strings, with base size equal to 15000 characters. DP is rather independent of p, thus we report an average time.

Another preliminary analysis, shown in Fig. 1, regards the sensitivity of the proposed method to the alphabet size. As can be seen, the performance are generally getting better with alphabets of increasing size, with a clear dependance on p.

5 Conclusions and Future Work

This preliminary extended abstract shows a promising approach to calculating the edit distance between two strings, leaving as future work the analysis of its performance with different lower bounds (or no lower bound at all), different τ increments as well writing the most effective code to readily identify and handle the kissing pairs.

References

1. Backurs, A., Indyk, P.: Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). SIAM J. Comput. **47**(3), 1087–1097 (2018)
2. Gotoh, O.: An improved algorithm for matching biological sequences. J. Mol. Biol. **162**(3), 705–708 (1982)
3. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, 534 p. Cambridge University Press (1997)
4. Jones, N.C., Pevzner, P.A.: An Introduction to Bioinformatics Algorithms, 456 p. MIT Press (2004)
5. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Doklady Akademii Nauk SSSR **163**(4), 845–848 (1965)
6. Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol. **48**(3), 443–453 (1970)