String Periodicity in the Scope of the ACM Programming Contest

Raphael Arias

Supervisor: Jan Finis
Proseminar "Selected Fun Problems of the ACM Programming Contest"
Chair III Datenbanksysteme
Fakultät für Informatik, Technische Universität München
Email: arias@cs.tum.edu

1 Introduction

This paper is about an interesting problem posed in the Asia Regional Round of the 31st ACM International Collegiate Programming Contest. We will discuss the problem and possible solutions, as well as their efficiency. Throughout the paper we will explore interesting concepts which are related to the solution of the problem.

2 Definitions and Notation

To understand this problem, we have to define some basic concepts first. Throughout this paper we will assume Σ as a finite alphabet. We will be examining strings of characters over this alphabet. Strings will have names (e.g. the string s) and content, the notation being $s =$ "*abcd*". When talking about a string s, we will refer to its length as $|s|$, to a character at its position i as $s[i]$ and to a substring comprising the characters at positions i to j of s as $s[i..j]$.

2.1 Levenshtein Distance

The Levenshtein Distance, also often referred to as the *edit distance* is defined as the minimal number of change operations, that are needed to transform a String s into a String t. The possible change operations are

- insertion of a character,
- deletion of a character and
- replacement of a character.

For instance, we might transform the String "*period*" into "*pearls*" using 1 character insertion ($\epsilon \mapsto a$), 2 character replacements ($io \mapsto ls$) and 1 character deletion ($d \mapsto \epsilon$), thus resulting in a Levenshtein distance of 4.

2.2 Periodicity

Sometimes strings can be periodic, that is, repetitive in some kind of way. This can be very interesting in practical applications, for instance in the field of genetics. In this paper, we introduce two concepts of periodicity.

2.2.1 Exact Period

The notion of an exact period is fairly simple: A string p is an exact period of another string s, under the premise that a natural number n exists, such that $s = p^n$. In this case p^n represents the n-fold concatenation of the string p with itself.
Example: The string "*ol*" is an exact period of the string "*olololol*", because the latter one equals "$(ol)^5$".

2.2.2 k-Approximate Period

A k-approximate period of a string is a similar concept, only that we now allow errors, small modifications of the string p in every repetition of it. Formally we define a k-approximate period of a string s as the string p if there exists a partition of s into $p_1 p_2 \dots p_n$, such that for every p_i the Levenshtein distance to p is less than or equal to k, or formally:

$$(\exists p_1, \dots, p_n \in \Sigma^*)\,[s = p_1 \dots p_n \wedge \forall i \in [n] : \delta(p, p_i) \leq k]\,,$$

where $\delta(p, p_i)$ is the Levenstein distance between p and p_i and $[n] = 0, \dots, n$.

This might become clearer if we look at a simple example. Consider the strings "*abcadcbb*" and "*ab*". We know, that the latter is a k-approximate Period of the former. But how big is k in this case? If we divide s into ["*ab*", "*ca*", "*dc*", "*bb*"] we can calculate the Levenshtein distance between $p =$ "*ab*" and each substring p_i. We get [0, 2, 2, 1], respectively. Since our k is the maximum of these we obtain $k = 2$. However, if we had instead divided s into ["*abc*", "*ad*", "*cb*", "*b*"], we would have obtained [1, 1, 1, 1]. Now we have $k = 1$, which is obviously better. We therefore have to be careful as to how we divide our string. We will examine this more closely in Section 3.2, after introducing the main problem in the following section.

3 The Problem

The problem we are examining is called **Period** and consists in the following: As input we are given two strings, s and p, where $1 \leq |s| \leq 5000$ and $1 \leq |p| \leq 50$ The expected output is the minimal k, such that p is a k-approximative period of s. This corresponds to the following formula:

$$k = \min\{\max\{\delta(p, p_i) \mid i \in \{1, \dots, n\}\}\}$$

Clearly we have two important things to heed here. One of them is finding the correct partition of s, such that k

becomes minimal. The other is calculating the Levenshtein discance δ for each of the substrings p_i. Since the latter is the simpler problem, we will discuss it first.

3.1 The Levenshtein Distance

The concept of the Levenshtein distance is fairly simple and has already been explained. However, there are a few interesting characteristics of the Levenshtein distance that should be pointed out.

3.1.1 Lower Bound

We shall see that there is a lower bound for the Levenshtein distance.

Proposition 1. The distance between two strings s and t is always at least the difference of their lengths.

$$\delta(s,t) \geq ||s| - |t||$$

This is best observed in an example: Consider the Strings "*dist*" and "*distance*". Even though they start exactly the same ("*dist*" is a prefix of "*distance*"), you need at least four character insertions to transform the former into the latter.

3.1.2 Upper Bound

Similarly, there is also an upper bound for the Levenshtein distance between two strings s and t.

Proposition 2. The Levenshtein distance between two strings s and t can never exceed the length of the longer string.

$$\delta(s,t) \leq \max\{|s|,|t|\}$$

To understand this, consider another example: The strings "*truth*" and "*lie*" can be transformed into one another by replacing three characters and either deleting ("*truth*" $\mapsto$ "*lie*") or inserting ("*truth*" $\leftarrow\!\shortmid$ "*lie*") the rest. Even though they have no characters in common, the distance cannot be bigger than this.

3.2 Find All Possible Substrings

This is probably the trickiest part of the problem and before we solve this, we will make an observations that might help us.

We have to realize that there exists an upper bound for the length of each substring p_i.

Proposition 3. Each p_i must have length strictly less than double the length of p.

$$\forall i \in \{1,\ldots,n\} : |p_i| < 2 \cdot |p|$$

Proof. We arrive at this conclusion easily if we assume a partition of s into $p_1 \ldots p_n$ where there is an i, for which $|p_i| = 2 \cdot |p|$. Since trivially $||p_i| - |p|| = |p|$, with Proposition 1 we have $\delta(p_i, p) \geq |p|$ as a lower bound for the distance. If we now split p_i up into two new substrings p_{i_1} and p_{i_2}, both with length $|p|$, with Proposition 2 we obtain this length as an upper bound for $\delta(p_{i_1}, p)$ and $\delta(p_{i_2}, p)$. □

Evidently, we might not have improved, but we certainly have not worsened.

4 Possible Solutions and Their Implementation

In this section we will discuss what the solution to this problem will look like and, evidently, present a practical implementation of the algorithms discovered.

4.1 Brute Force

Brute force solutions are often the most easy to implement - but also the least elegant. For some problems it might make sense to follow this approach, as more elegant solutions might not be obvious or astonishingly complex. However, a lot of problems are not fit to be solved in the brute force manner since that approach is not practical for the given problem size. In the following section we will closely examine the given problem and draw a conclusion concerning the practicality of a brute force approach in this case.

4.1.1 Arguments against this Approach

To answer this question, let us look at the input sizes we are given. As we saw in the problem description in section 3, the size of s and p is bound:

$$1 \leq |s| \leq 5000, \qquad 1 \leq |p| \leq 50.$$

Having this and using Proposition 3, we also have an upper bound for the number of possibilities of dividing s into substrings. However, as this number is not very small, it will not help us much. We therefore construct a lower bound for this number of possibilities to show that they are far too many.

Proposition 4. The number of possibilities to divide a string s of size 5000 into substrings with a maximum length of 99 ($\leq 2 \cdot 50$) is more than $1.2 \cdot 10^{208}$.

Proof. Since there are 5000 characters, there are 4999 places where we could place a separator. If we want to guarantee that no substring is longer than 99 characters, we can just place more than 4900 separators and not consider possibilities with less separators, which would only increase the final result. For the possibilities to place $4901, 4902, \ldots, 4999$ separators we obtain:

$$\sum_{i=4901}^{4999} \binom{4999}{i} \approx 1.2 \cdot 10^{208}$$

□

Note again, that this is a lower bound. This seems to be a sufficiently large number to reconsider our approach to this problem. Bruteforcing this number of possible combinations in order to find the minimal k will just take too long.

4.2 Dynamic Programming

Now that we have seen that brute force is not an option, we have to look for alternatives. There has already been extensive research on string algorithms and some on string periodicity, the most notable in the former topic put forth by Wagner and Fischer.

4.2.1 An Algorithm for the Levenshtein Distance

Fortunately, there is a very efficient algorithm to calculate the Levenshtein distance using dynamic programming. It was introduced by Wagner and Fischer in [2]. The principle is fairly simple. Using the aforementioned example of the two words s = "*period*" and t = "*pearls*" the algorithm proceeds as follows:

		p	e	a	r	l	s
p							
e							
r							
i							
o							
d							

Figure 1: Empty table for $\delta($"*period*","*pearls*"$)$.

1. Build a table of size $|s| + 1 \cdot |t| + 1$ to store the edit distances between all the prefixes of s and t. (See Fig. 1). We will refer to cells of this table and their neighbours using cardinal directions.
2. Initialize the table with the edit distances between every prefix of s and t and the empty string, so
$$\delta(u, \epsilon) \, \forall u \in \Sigma^*. \exists v \in \Sigma^*. s = uv,$$
as well as
$$\delta(\epsilon, u) \, \forall u \in \Sigma^*. \exists v \in \Sigma^*. t = uv.$$
(See Fig. 2).
3. Iterate through the table. For each cell, check whether the indexing characters are equal. If they are, copy the value from the north western neighbour into this cell. Otherwise choose the minimum of the north, north western and western neighbours, increment it by one and write the result.
4. The edit distance between s and t is in the south east corner of the table when the algorithm finishes.

		p	e	a	r	l	s
	0	1	2	3	4	5	6
p	1						
e	2						
r	3						
i	4						
o	5						
d	6						

Figure 2: Initialized table with Levenshtein distances for every prefix and the empty string.

As you can see in Fig. 3, the result is 4, as we already demonstrated in Section 2. For the proof of correctness of this algorithm, see Wagner's and Fischer's paper [2].

4.2.2 A Modification to fit this Problem

Now that we know this algorithm for the Levenshtein distance, let us reconsider our approach to the solution of the overall problem. As we have seen, the algorithm builds a table and fills it with the edit distances between all the prefixes of s and t. So actually, we are only missing the edit distances between the smaller string p and all the suffixes of our (now problem-specific) s:

		p	e	a	r	l	s
	0	1	2	3	4	5	6
p	1	0	1	2	3	4	5
e	2	1	0	1	2	3	4
r	3	2	1	1	1	2	3
i	4	3	2	2	2	2	3
o	5	4	3	3	3	3	3
d	6	5	4	4	4	4	4

Figure 3: Final table showing the results of the algorithm.

$$\delta(p, v) \, \forall v \in \Sigma^*. \exists u \in \Sigma^*. s = uv.$$

So, why do we not simply build such a table for every suffix v? This modification was introduced by Sim et al. [1] and helps us solving the problem efficiently. According to Sim et al.

$$t_i = \min_{0 \leq h < i} \{\max\{t_h, \delta(p, s[h+1, i])\}\},$$

where t_i the minimum value, such that p is a t_i-approximate period of $s[1..i]$ and obviously $1 \leq i \leq |s|$. t_0 is initialized to 0.

4.2.3 Implementation

First we need a slightly modified version of the Levenshtein distance Dynamic Programming algorithm introduced in Section 4.2.1. We modify it so instead of a numeric result it returns the whole distance table built and obtain the function **levenshtein_mod(a,b)** (see Listing 1).

```
1  def levenshtein_mod(a, b):
2      D = [[0]] # distance between empty strings
3      # initialize lists:
4      [D.append([i])  for i, s in enumerate(a, 1)]
5      [D[0].append(j) for j, t in enumerate(b, 1)]
6
7      for j, s in enumerate(b, 1):
8          for i, t in enumerate(a, 1):
9              if s == t:
10                 D[i].append(D[i-1][j-1])
11             else:
12                 D[i].append(
13                     min(
14                         D[i-1][j] + 1,
15                         D[i][j-1] + 1,
16                         D[i-1][j-1] + 1
17                     )
18                 )
19     return D
```

Listing 1: Python implementation of Wagner's and Fischer's Dynamic Programming Algorithm for the edit distance.

The next step is transforming the formula by Sim et al. into an actual implementation. The programming language Python is used for the implementation. The basic algorithm is implemented in the function **solve(s, p)** (see Listing 2).

As you can see in Listing 2, there is a call to the function **distanceMatrix(s, p)**. This function basically calls our

```python
def solve(s, p):
    D = distanceMatrix(s, p)
    t = [0] # t[0] = 0
    for i in xrange(1, len(s)+1):
        cmin = 6000 # current minimum
        for h in xrange(0, i):
            # i is never part of this
            # h+1st table, last line, column i-h+1:
            cmin =  min(
                        cmin,
                        max(t[h], D[h][-1][i-h])
                    )
        t.append(cmin)
    return t[len(s)]
```

Listing 2: Python code implementing the algorithm described by Sim et al.

modified version of the Levenshtein distance function for each suffix of s and combines the results into a 3-dimensional list and returns it (see Listing 3).

```python
def distanceMatrix(s, p):
    D = []
    for i in xrange(len(s)):
        D.append(levenshtein_mod(p, s[i:]))
    return D
```

Listing 3: Python implementation of the function that builds the 3 dimensional list of distances.

4.3 An Alternative Approach: Evolutionary Algorithms

In this section we will introduce an alternative approach to solving this problem and the reasons why, ultimately, this approach is not a viable one.

For this, we will give a (very brief) introduction to the most basic concepts of Evolutionary Algorithms.

4.3.1 The Basic Concepts of Evolutionary Algorithms

As the name suggests, Evolutionary Algorithms are based on the principle of evolution and borrow ideas from it. They are an approach to optimization problems.

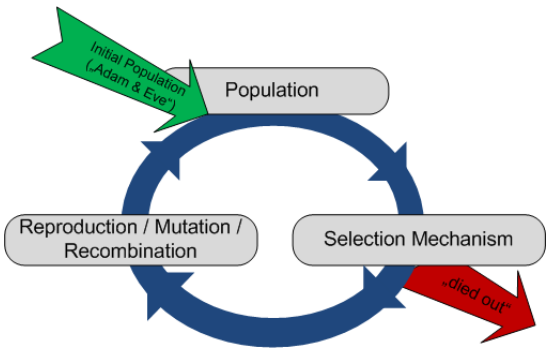

Figure 4: Simplified diagram of an evolutionary algorithm.

The general idea is having a initial *population* of "individuals" (solutions) to a given problem. The algorithm's purpose is to evolve this population into one that contains the optimal solution to a problem. It accomplishes this using a *selection* mechanism, that applies a *fitness function* to the individuals in the population pool. The fitness function measures the fitness of the individual ("how good is the solution for the given problem?") in a way that is specific to the problem and different for each case.

The next step is the most interesting: the evolutionary step itself. This implies the individuals have to be replaced or altered. Like in the biological form of evolution, this can happen by *mutation* or *recombination*.

After the population has changed, the process is repeated a finite number of times. By the end of the evolutionary process, an optimal solution should be part of the population.

4.3.2 Application to this Problem

After introducing the very simplified, basic idea, how do we apply the principle of an evolutionary algorithm to this concrete problem? We need to find a representation of the solutions to our problem that would be viable individuals in the population.

Our individuals in this case are, of course, the partitions of s into substrings, as we need to find the optimal partition for k to be minimal. The initial population pool can be generated by uniformly selecting random partitions of s.

Now we are missing a fitness function and a selection mechanism, as well as the reconfiguration (mutation and/or recombination) of the individuals. The first part of this is, again, quite simple: a solution (partition) P of s into substrings p_i is fit, if the k, such that p is a k-approximate period of s is low.

The selection phase and the mutation mechanism are where matters become more complicated. A possibility for the mutation is the following: In a partition consisting of several substrings, a substring can take or give characters from or to its adjacent substrings. Additionally, a substring can be split into two substrings to increase the amount of substrings. The amount of substrings can also decrease if giving or taking characters from a neighbour results in an empty substring.

For the selection mechanism, we could select members of the population, choosing candidates which have a better fitness with a higher probability. These candidates could be recombined and mutated, forming the next generation.

The problem with the evolutionary approach to this problem is, as one might have noticed, that the algorithm it yields is non-deterministic. There is a great amount of randomness involved and no estimate can be made of how long it takes to find the optimal solution. In fact, one cannot even guarantee that an optimal solution will be found, using this type of algorithm, which makes it unsuitable for an ACM problem. Also, simulating the evolution for a sufficiently large population takes quite long.

The reasons pointed out in the previous paragraph are the ones which led to reject the evolutionary approach as a solution to this problem. Nevertheless the basic principles are interesting to consider in a optimization problem.

5 Complexity Analysis

Of course, the quality of a solution must be verified. Therefore we are going to examine the complexity of the algorithms used with respect to time and space complexity.

5.1 Space Complexity

Space complexity analysis is an asymptotic analysis of the amount of memory space necessary for the execution of an algorithm. In this case, we examine the amount of memory needed to compute the Levenshtein distance and to perform the optimization after that.

5.1.1 Levenshtein Distance

The algorithm presented in Section 4.2.1 for two strings a and b builds a table of size $|a| \cdot |b|$.

5.1.2 Finding the Minimum k

For the solution of the problem we call the Levenshtein Distance algorithm for p each suffix of s. This means, we build one table of size $|p| \cdot |s|$, one of size $|p| \cdot (|s| - 1)$, one of size $|p| \cdot (|s| - 2)$ etc. Therefore we obtain $|s|$ tables with different size corresponding to the following formula. Let $|s| = n$, $|p| = m$

$$\sum_{i=1}^{n} m \cdot i = \frac{1}{2} mn(n+1) \in \mathcal{O}(m \cdot n^2).$$

Additionally we build another list containing the values for t_i with size $n + 1$. Since $m \cdot n^2 + n + 1 \in \mathcal{O}(m \cdot n^2)$, we can safely ignore this in an asymptotic analysis.

5.2 Time Complexity

When speaking about time or runtime complexity, we refer to the asymptotic runtime of the examined algorithms relative to the input size. So in this case, how long will it take us to compute k, once we are given s and p?

5.2.1 Levenshtein Distance

We saw that the table which has to be build in order to check the edit distance between a and b has size $|a| \cdot |b|$. Then each cell in that table is visited and populated with a value. This takes $|a| \cdot |b|$ time.

5.2.2 Finding the Minimum k

As we built a table of size $\mathcal{O}(m \cdot n^2)$ and visit each cell, the runtime for this part is $\mathcal{O}(m \cdot n^2)$ as well. To calculate the t_is, we need some additional time in the loop. For each i we have a loop for $0 \leq h < i$. The amount of iterations therefore is:

$$\sum_{i=1}^{n} i = \frac{1}{2} n(n+1) \in \mathcal{O}(n^2)$$

This does not increase the overall complexity, since

$$\mathcal{O}(m \cdot n^2 + n^2) \subseteq \mathcal{O}(m \cdot n^2).$$

6 Conclusion

As we have seen, there are some solutions that are better suited for this problem than others and why this is the case. The Dynamic Programming approach has runtime benefits over the other solutions and, unlike the evolutionary approach, is deterministic.

Nevertheless the evolutionary algorithm could yield some solutions. With some small improvements they might not even be very bad. There has been some research on String algorithms with evolutionary elements, but that is beyond the scope of this paper.

The Dynamic Programming algorithm discussed here is a very elegant solution to this problem and possibly to other optimization problems with similar constraints.

There are a few improvements to the proposed algorithm, that make it a little more efficient, runtime- and space-complexity-wise. For instance the distance function that builds the tables could be written in a lazy manner, such that cells in the table do not necessarily have to be calculated and stored. They would then be only computed and stored when they are needed.

References

[1] J. S. Sim, C. S. Iliopoulos, K. Park, and W. F. Smyth. Approximate periods of strings. *Theor. Comput. Sci.*, 262(1-2):557–568, July 2001.

[2] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, Jan. 1974.