An Evolutionary Approach to Hard Test Case Generation for Shortest Common Superstring Problem

Maxim Buzdalov
St.Petersburg National Research University
of Information Technologies, Mechanics and Optics
49 Kronverkskiy prosp.
Saint-Petersburg, Russia, 197101
Email: mbuzdalov@gmail.com

Fedor Tsarev
St. Petersburg National Research University
of Information Technologies, Mechanics and Optics
49 Kronverkskiy prosp.
Saint-Petersburg, Russia, 197101
Email: fedor.tsarev@gmail.com

***Abstract*—The *shortest common superstring problem* has important applications in computational biology (e.g. genome assembly) and data compression. This problem is NP-hard, but several heuristic algorithms proved to be efficient for this problem. For example, for the algorithm known as GREEDY it was shown that, if the optimal superstring has the length of N, it produces an answer with length not exceeding $3.5 \cdot N$. However, in practice, no test cases were found where the length of the answer is greater than or equal to $2 \cdot N$. For hard test case generation for such algorithms the traditional approach assumes creating such tests by hand. In this paper, we propose an evolutionary algorithm based framework for hard test case generation. We examine two approaches: single-objective and multi-objective. We introduce new test case quality measures and show that, according to these measures, automatically generated tests are better than any known ones.**

I. Introduction and Problem Description

The shortest common superstring problem [1]–[3] is formulated as follows. Given a set of strings $\{X_1, \ldots, X_n\}$, one needs to find the shortest string S such that for all $1 \le i \le n$ X_i is a substring of S.

This problem arises in a number of applications. For example, it can be used in one of the models of genome assembly [4] which is an important area of bioinformatics. Another application is data compression: for instance, one of the ways of storing sparse matrices while maintaining the constant access speed is to build the shortest common superstring for strings collected by splitting the matrix rows into several pieces [5].

This paper is dedicated to generation of hard test cases for this problem.

A. Definitions

Without loss of generality, we assume that the strings given as input are *substring free*: there are no equal strings in the input, and for any two different strings s and t s is not a substring of t and vice versa.

The *overlap* function $o(x, y)$ for two strings x and y, such that x is not a substring of y and vice versa, is defined as follows. Let the length of the string x be r, the length of y be s. Then

$$o(x, y) = \max\{j : y_i = x_{r-j+i}, 1 \le i \le j\}.$$

We define the concatenation of two strings x and y, which merge their overlapped substrings, as $x \oplus y$. If the overlap of these strings is $k = o(x, y)$, then

$$x \oplus y = x_1 \ldots x_r y_{k+1} y_{k+2} \ldots y_s.$$

B. Exact Algorithm

The shortest common superstring problem is NP-hard [3]. Here we formulate one of the known algorithms of building the exact shortest common superstring.

Let the input strings be $x_1, \ldots, x_n$, where no string is a substring of another one. One of the exact algorithms to solve this problem is formulated as follows:

1) Let $A(S, t)$ be a common superstring for a set of input strings S which ends in an input string t and is the shortest possible.
2) For all x_i, $A(\{x_i\}, x_i) = x_i$.
3) For all S, $t \in S$, $u \notin S$ $A(S \cup \{u\}, u) = \min(A(S \cup \{u\}, u), A(S, t) \oplus u)$.
4) Let $U = \{x_1, \ldots, x_n\}$.
5) The answer is $\min_{1 \le i \le n} A(U, x_i)$.

C. Greedy Algorithm

There are several greedy algorithms for solving the shortest common superstring problem. We analyze the algorithm which is known as GREEDY [1].

Let the input strings be $x_1, \ldots, x_n$, where no string is a substring of another one. The algorithm GREEDY is formulated as follows:

1) $I \leftarrow \{x_1, \ldots, x_n\}$.
2) if $|I| = 1$, then stop. The contents of I is the answer.
3) Choose $x \in I$, $y \in I$ such that $x \ne y$ and $o(x, y)$ is maximal possible.
4) $I \leftarrow I \setminus \{x, y\} \cup \{x \oplus y\}$.

5) Go to 2.

Note that in line 3, in case of several pairs of strings with the maximum overlap, a single pair has to be chosen. In this work, the set I is implemented as a list. From all pairs with overlaps equal to the maximum overlap, the pair with the minimum index of x is chosen. If there are several pairs with the same index of x, the one with the minimum index of y is chosen. The newly built string is *prepended* in line 4.

D. Problem Description

The GREEDY algorithm is shown in [6] to have a theoretical upper bound on the length of the generated common superstring G of $3.5 \cdot OPT$, where OPT is the length of the shortest common superstring. However, no test cases where $|G| \geq 2 \cdot OPT$ were found yet.

We propose an evolutionary algorithm based framework for hard test case generation. The test case space of the problem is analysed using single-objective and multi-objective approaches. In sections III and IV, new quality measures of test cases are introduced, in sections III-C and IV-C the experiment results are presented.

II. P-STRINGS

Test data for the shortest common superstring problem consist of several strings. However, considering strings when analysing the problem is rather inconvenient.

Consider an example of the test data from [2]: $\{c(ab)^k$, $(ba)^k$, $(ab)^k c\}$. This is a *pattern* of good tests: the optimal answer is $O = c(ab)^{k+1}c$, while the greedy algorithm produces $G = c(ab)^k c(ba)^k$. If $k \to \infty$, then $|G|/|O| \to 2$. However, when searching for different testcases while working with the strings rather than patterns, one has to limit the values k for performance reasons. This leads to wrong estimations on the $|G|/|O|$ ratio. For example, if $|G|/|O| = (5 \cdot k + 1)/(2 \cdot k + 2013)$, then for all $k < 4025$ the value will be less than 2, although the asymptotic value is 2.5.

In [7] it is proven that, for a small random perturbation of any given string, the ratio of answers is $1 + o(1)$, which effectively prevents the mutation-only evolutionary algorithms from success (and makes most evolutionary algorithms perform worse). So we need to come up with a solution which does not use strings as is.

Direct processing of test patterns is preferrable because of two reasons: they are much shorter than the corresponding tests, and the asymptotic of the ratio of their lengths can be computed exactly. We present a kind of patterns called *p-strings* which are suitable to run the algorithms atop of them instead of ordinary strings.

A. Definition

A *p-string* $S(x) = a_1^{b_1 + x \cdot c_1} \dots a_n^{b_n + x \cdot c_n}$ is a pattern that for every integer $x > x_0$ yields a string formed by concatenating $(b_1 + x \cdot c_1)$ of characters $a_1, \dots, (b_n + x \cdot c_n)$ of characters a_n. The single element of such a string $a^{b+x\cdot c}$ will be referred to as a *p-character* with the *base* of a and the *power* of $b + x \cdot c$.

The parameter x represents a "very big" or "infinite" positive value. The *length* of such a string is also a function of x equal to $\sum_{i=1}^{n} b_i + x \cdot c_i$. The lengths of the p-strings are compared in the following way: $(b_1 + x \cdot c_1) > (b_2 + x \cdot c_2)$ if $c_1 > c_2$ or $c_1 = c_2$ and $b_1 > b_2$. Note that, for any length $(b + x \cdot c)$, c must be non-negative, and b must be non-negative if $c = 0$ and may be of any value if $c > 0$.

A *canonical form* of a p-string $A(x)$ is a p-string $A'(x)$ which yields the same string for every significantly large integer x and has the minimum number of p-characters. One can show that to construct a canonical form one needs to replace all sequences of consecutive p-characters having the same base $a^{b_1+x\cdot c_1}, \dots, a^{b_k + x \cdot c_k}$ with a single character $a^{(b_1 + \dots + b_k) + x \cdot (c_1 + \dots + c_k)}$.

In the rest of the paper the parameter x is the global parameter for all p-strings under consideration, and we will further omit the (x) part in the description of p-strings.

A *concatenation* of two p-strings a and b for the same parameter is a p-string $c = ab$ in the canonical form which yields a string which is a concatenation of the strings yielded by a and b for every possible value of the parameter. To make a concatenation of two p-strings a and b in the canonical form, one has to write the p-characters of a and then of b, and then build a canonical form of the result. One may show that it is enough to check the last p-character of a and the first p-character of b and merge them if they have equal bases.

B. P-string Overlaps

One of the main concerns in a shortest common superstring problem is an *overlap* $o(s, t)$ of two strings s and t, which is the length of the longest string z such that $s = s'z$ and $t = zt'$ for some s' and t'.

The overlap can be defined for p-strings the same way using the appropriate definitions for the length, the longest string and concatenation. The algorithm for overlap computing should be different, however, because, unlike common strings, it is possible to split a p-character. For example, the overlap of two p-strings $s = a^{1+x}b^{2+3\cdot x}$ and $t = b^{7+2\cdot x}a^5$ is $(7 + 2 \cdot x)$, so that $z = b^{7+2\cdot x}$, $s' = a^{1+x}b^{x-5}$, and $t' = a^5$.

In this paper we use a simple algorithm for calculating the overlap of two p-strings: given two strings s and t, consider every p-character in s, try to align t in the way that the first p-character of their overlap maps to both the first p-character of t and the chosen p-character of s, and, from the succeeded alignments, choose one with the maximal overlap. This algorithm has the complexity of $O(|s| \cdot |t|)$, which is suitable for small p-strings.

C. Shortest Common P-superstring

Most known algorithms for computing the (approximate or exact) shortest common superstring for a set of strings can be generalized to p-strings, as they rely on the overlap operation, concatenation and length comparison only. In particular, both the exact algorithm from Section I-B and the greedy algorithm from Section I-C, which we use to analyse the problem, may be generalized without any additional effort.

III. Evolutionary Approach: Single Objective

The task of generating a hard test case for the greedy algorithm can be formulated in terms of p-strings as the following *optimization problem*: generate a p-string S such that, if $(A \cdot x + B)$ is the length of the solution produced by the GREEDY algorithm and $(a \cdot x + b)$ is the length of the exact shortest common superstring, the ratio A/a is maximum possible.

The search space is a set of fixed-sized lists of p-strings with several restrictions placed on them due to performance concerns: the maximum number of p-characters, the alphabet size, and the limitations on the coefficients in the p-characters.

A. Individual Representation and Evolutionary Operators

The representation of a test case as an individual is straightforward. The individual is a list of p-strings which has a size of M. Each p-string has a positive number of p-characters not exceeding N. For each p-character of these strings $a_i^{b_i + c_i \cdot x}$ the values of $|b_i|$ and $|c_i|$ are limited by L, and the set of possible a_is (the alphabet) has the size of A.

The parameters M, N, L, A are set beforehand and kept constant during the run of an evolutionary algorithm.

A p-character $a^{b+c \cdot x}$ is generated randomly as follows. The base a is selected uniformly from the alphabet, then the quotient c is selected uniformly from the range $[0; L]$, then the quotient b is selected uniformly from the range $[0; L]$ if $c = 0$, or $[-L; L]$ otherwise.

After performing some operation on p-strings, the quotients of their p-characters may exceed L by their absolute value. Such quotients are returned to the allowed range: if a quotient is greater than L, it is made equal to L, and if a quotient is less than $-L$, it is made equal to $-L$.

Generation of a new individuals is performed randomly: M p-strings are generated, each of them receives a uniformly random length from the range from 1 to N. Each p-character of these strings is generated randomly as described above. The p-strings are then converted to their canonical forms.

A mutation is done independently for each p-string in the individual. First, a *type* of mutation is selected at random: `D` with the probability of $\frac{1}{4}$, `I` with the probability of $\frac{1}{4}$, `R` with the probability of $\frac{1}{2}$.

If the type is `D` and the number of p-characters in the string is greater than one, a random p-character from the string is deleted.

If the type is `I` and the number of p-characters is less than N, a random p-character, generated as above, is inserted at a random place in the string, including the position before and after the string.

In all other cases a randomly chosen p-character is mutated. The mutation is done as follows. With the probability of $\frac{1}{2}$ the character is generated randomly. Otherwise, with the probability of $\frac{1}{5}$, the base is replaced by a randomly generated one. Otherwise, the quotients of the power of the character are altered by the random value from the range of $[-1; 1]$, and then the possible out-of-range issues are fixed.

After mutation, the string is turned to the canonical form, and string correction is performed as described above.

There are two variants of the crossover which are applied with equal probability. The first variant of the crossover swaps the p-strings in the individuals as follows. First, the swap length W is chosen from the range $[1, M-2]$. Then, the swap offsets O_1 and O_2 for the individuals are chosen independently from the range $[0, M-W]$. After that, the sublists of length W are swapped in the individuals, the sublist of the first individual is starting at O_1 and of the second individual at O_2 (the indices start with zero).

The second variant does the similar thing to the first one, but with the pairs of strings from the different parents, not with the lists of strings. Namely, for each index in $[0, M-1]$ it selects the p-strings from the parents at this index and swaps the substrings with equal number of p-characters in these strings and sets the results at the same index to the children.

The fitness function is straightforward. For an individual of the form of $[X_1, \ldots, X_M]$, first the substrings of some other strings are removed, and then two superstrings are calculated: the exact shortest common superstring E and the approximation G computed by the GREEDY algorithm. Let the length of E be $a \cdot x + b$, and the length of G be $A \cdot x + B$. Then the fitness value for the considered individual is the ratio A/a.

B. Evolutionary Algorithm

A genetic algorithm is used with the generation size of 100. The following variation of the tournament selection is used:

1) 2^T individuals are selected at random and added to the list X.
2) If $|X| = 1$, then the only individual in the list is returned.
3) The list X is divided into pairs of individuals.
4) From each pair, the best individual is added to X with the probability of 0.9, otherwise the worst individual is added.
5) Go to line 2.

The scheme of the genetic algorithm is as follows:

1) The generation is populated by individuals generated as described in Section III-A.
2) The fitness values of the individuals are computed.
3) If the largest fitness value is greater than or equal to a threshold value Θ, the algorithm is terminated.
4) The top 5 individuals are promoted to the next generation.
5) The rest of the next generation is filled using the following operations:
    a) two individuals are selected by tournament selection with $T = 3$;
    b) they are crossed over;
    c) each of the individuals is mutated with the probability of 0.5.
    d) the resulted individuals are added to the next generation until its size reaches 100.
6) The next generation becomes the current one. Go to line 2.

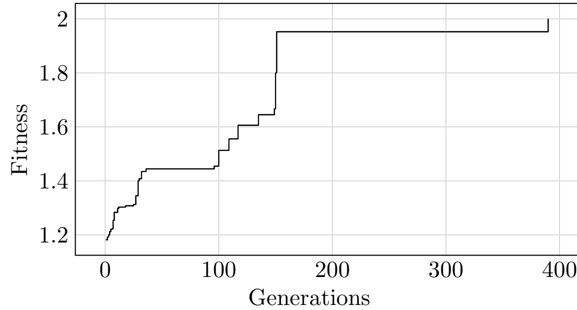

Fig. 1. Example of fitness plot

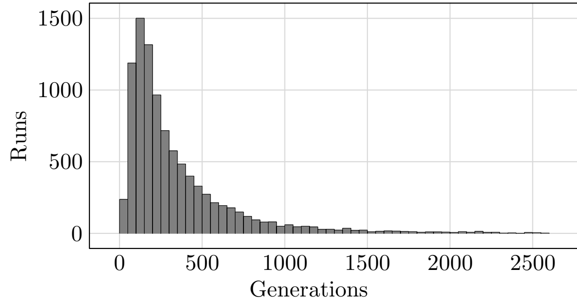

Fig. 2. Density plot for the number of generations needed to reach the optimum

C. Experiments

All the experiments were performed using the following problem parameters: $N = 5$, $M = 5$, $L = 20$, $A = 2$.

The experiments with the fitness threshold $\Theta = 2.0$ show that the genetic algorithm finds the input data with the asymptotic ratio of 2.0 very quickly. In Fig. 1, an example of the fitness plot is given.

As the fitness of 2.0 is reached very quickly, it is possible to gather enough statistic data to estimate the performance. 10000 runs were evaluated, for each of them the number of generations needed to reach the fitness threshold is recorded. In Fig. 2, the best 97% of the runs are shown. The X axis is split into intervals of 100 generations wide each. In the Y axis, the number of runs which fell inside each interval is drawn.

For the best 97% of runs, the mean value of the number of generations is 351.02, and the median is 229. For all runs, these numbers are 739.28 and 237, respectively.

It can be seen that the problem of finding a test with the ratio of lengths equal to 2.0 is an easy task for the genetic algorithm. However, it seems to be impossible for the genetic algorithm to overcome this value.

IV. Evolutionary Approach: Two Objectives

To further explore the problem, we tried applying the multi-objective approach. As shown in numerous sources ([8], [9]), adding new objectives may increase performance or produce better results.

A. Second Objective

Recall that the fitness value, which we define here as the *primary objective*, is computed as follows: if the length of an exact answer to the problem is $(a \cdot x + b)$ and the length of the answer given by the GREEDY algorithm is $(A \cdot x + B)$ then the primary objective is equal to $\alpha = A/a$.

Consider, once again, the example from [2]: $\{c(ab)^k, (ba)^k, (ab)^k c\}$. The exact answer is $O = c(ab)^{k+1}c$, and the greedy answer is $G = c(ab)^k c(ba)^k$. The length of the exact answer is $2 \cdot k + 4$, and the length of the greedy answer is $4 \cdot k + 2$. We can determine not only the ratio of the lengths, which is equal to 2, but also the fact that the greedy answer is *six characters shorter than twice the exact answer*. In fact, no tests are known where the greedy answer is at least twice as long as the exact answer.

We used two different ways of defining the secondary objective:

1) the difference between lengths of the greedy answer and the exact answer times α, which is independent of x:
$$D(a, b, A, B) = A \cdot x + B - \frac{A}{a}(a \cdot x + b) = \frac{aB - Ab}{a}.$$
If $a = 0$, then $D = 0$ as well.
2) the numerator of the fraction above, which is equal to:
$$Z(a, b, A, B) = aB - Ab.$$

For both secondary objectives, the following theorem holds:

Theorem 1: If the value of the secondary objective is positive, then the length of the greedy answer is always bigger than its asymptotic estimation. If the secondary objective is negative, then the greedy answer is always shorter than its estimation. If the secondary objective equals zero, then the greedy answer is always equal to its estimation.

Proof: For the first variant of the secondary objective, the theorem statement follows from the semantics of the objective value. If the greedy answer equals G, the exact answer is E, and the value of the secondary objective is D, then $G = E \cdot \alpha + D$. So if $D > 0$, then the greedy answer is longer than the asymptotic estimation, if $D < 0$, then it is shorter, and if $D = 0$, the greedy answer is equal to its estimation.

For the second variant, consider two cases. In case of $a = 0$, $A = 0$ as well, because the input p-strings are independent of x, so both D and Z are equal to 0 in this case. In case of $a \neq 0$, which implies $a > 0$, the values of D and Z have equal signs. ■

B. Evolutionary Algorithm

In this paper, NSGA-II [10] with run-time complexity improved as in [11] is used as a multi-objective optimization algorithm. The generation size is set to 1000. Crossover, mutation and individual generation are performed as described in Section III-A.

C. Experiments

As the shape of Pareto front for this problem is not known a priori for any of definitions of the secondary objective, we performed several runs of NSGA-II for both definitions until the current Pareto front approximation did not change for some

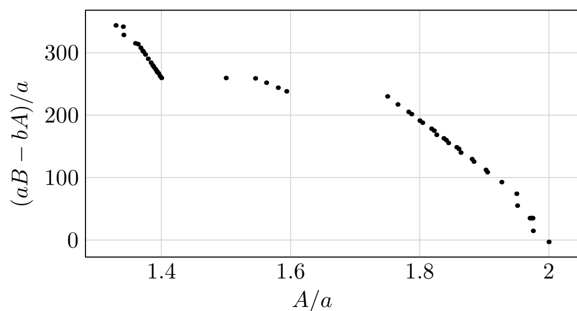

Fig. 3. Pareto front approximation: first variant of second objective

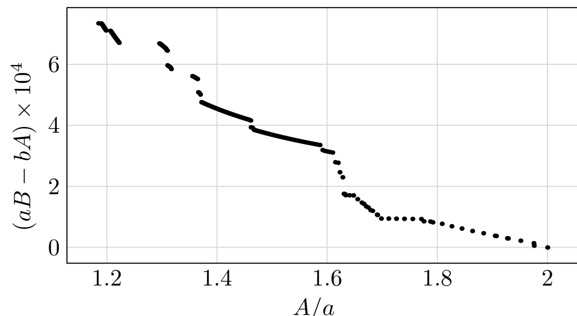

Fig. 4. Pareto front approximation: second variant of second objective

time and built the union of the fronts for each objective. The Pareto front approximation for the first variant is shown on Fig. 3, and for the second version on Fig. 4.

The best value for the first variant, according to the primary objective, has the objective values $(2, -3)$ and produces the following p-strings: $\{a^{20x-20}b^1$, b^1a^{20x-20}, a^{20x-19}, a^{19}, $a^{3x+20}\}$.

One can see that the first three strings determine the answer as the others are their substrings. One can also note that for this test the string order matters: if the greedy algorithm paired the first and the third string first, the answer would be optimal. However, the resolution rule of string pairs, described in Section I-C, makes the first and the second string overlap first, thus producing unoptimal answer. According to the second objective, the length of the greedy answer is three characters less than twice the length of the optimal answer.

The best value for the second variant, according to the primary objective, has the objective values $(2, -42)$ and produces the following strings: $\{b^{11x+3}$, $b^{14x-19}a$, ab^{14x-19}, b^{2x-17}, $b^{14x-18}\}$.

V. Conclusion

In this paper an evolutionary algorithm based hard test case generation framework for the shortest common superstring problem is presented. It uses special string patterns, called p-strings, instead of ordinary strings. Three quality measures for test cases — one primary measure and two secondary measures — were presented, and their optimization was performed by single-objective and multi-objective evolutionary algorithms.

The single-objective experiment shows that it is very easy for the genetic algorithm to reach the string length ratio of 2.0. The impossibility of exceeding this ratio supports the commonly known conjecture that 2.0 is indeed the upper limit for the string length ratio.

From the two-objective experiments we found a test case for which the greedy answer is three characters shorter than twice the length of the optimal answer. This is the best known test case for the shortest common superstring problem according to the quality measures described in the paper. The Pareto front approximations may provide an additional insight to the nature of the problem.

VI. Acknowledgments

The research was supported by Ministry of Education and Science of Russian Federation in the context of Federal Program “Scientific and pedagogical personnel of innovative Russia”.

References

[1] A. Frieze and W. Szpankowski, “Greedy algorithms for the shortest common superstring that are asymptotically optimal,” 1997.

[2] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis, “Linear approximation of shortest superstrings,” *Journal of the ACM*, vol. 41, pp. 630–647, 1994.

[3] J. Gallant, D. Maier, and J. A. Storer, “On finding minimal length superstrings,” *Journal on Computer and System Sciences*, vol. 20, pp. 50–58, 1980.

[4] H.-J. Böckenhauer and D. Bongartz, *Algorithmic Aspects of Bioinformatics (Natural Computing Series)*. Springer, 2007.

[5] S. Skiena, *The Algorithm Design Manual (2. ed.)*. Springer, 2008.

[6] H. Kaplan and N. Shafrir, “The greedy algorithm for shortest superstrings,” *Information Processing Letters*, vol. 93, pp. 13–17, 2005.

[7] B. Ma, “Why greed works for shortest common superstring problem,” *Combinatorial Pattern Matching. Lecture Notes in Computer Science*, vol. 5029, pp. 244–254, 2008.

[8] J. D. Knowles, R. A. Watson, and D. Corne, “Reducing local optima in single-objective problems by multi-objectivization,” in *Proceedings of the First International Conference on Evolutionary Multi-Criterion Optimization*, ser. EMO ’01. London, UK: Springer-Verlag, 2001, pp. 269–283.

[9] M. T. Jensen, “Helper-objectives: Using multi-objective evolutionary algorithms for single-objective optimisation: Evolutionary computation combinatorial optimization,” *Journal of Mathematical Modelling and Algorithms*, vol. 3, no. 4, pp. 323–347, 2004.

[10] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast elitist multi-objective genetic algorithm: NSGA-II,” *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 182–197, 2000.

[11] M. T. Jensen, “Reducing the run-time complexity of multiobjective EAs: The NSGA-II and other algorithms,” *Transactions on Evolutionary Computations*, vol. 7, no. 5, pp. 503–515, 2003.