AOBA: An Online Benchmark tool for Algorithms in stringology

Ryu Wakimoto, Satoshi Kobayashi, Yuki Igarashi, Davaajav Jargalsaikhan, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara

Graduate School of Information Sciences, Tohoku University, Japan
{ryu_wakimoto, satoshi_kobayashi, yuki_igarashi, davaajav_jargalsaikhan}@shino.ecei.tohoku.ac.jp
{diptarama, ryoshinaka, ayumis}@tohoku.ac.jp

Abstract. Experimental performance of an algorithm often has a big gap from its theoretical complexity analysis. It is necessary to understand the characteristics of the experimental performances of different algorithms for the same problem to select an appropriate algorithm depending on available computing resources and properties of input data. In this paper, we present AOBA, an integrated online platform for testing, evaluating, analyzing and comparing different algorithms for various types of string processing problems. New algorithms can be submitted to AOBA and will be evaluated with many test data. All experiments work on the web: everyone can use AOBA without any local environment for experiments. AOBA also includes a powerful in-browser visualizer for analyzing and comparing performances of algorithms.

Keywords: string processing · experimental evaluation · online evaluation tool

1 Introduction

String processing is one of the most important research fields in computer science. Various kinds of data including natural language sentences, biological sequences, and other types of sequences of symbols are strings in nature. Even numerical sequences can be seen as strings and moreover any kinds of data handled by a computer are represented as strings of bits in the end. So efficient string processing algorithms and their implementations can make a huge impact on the wide range of applications. It is important to understand and evaluate the performances of string processing algorithms fairly and correctly.

It is common for theoretical and experimental evaluations of an algorithm to diverge. For instance, Cantone and Faro [3] proposed two algorithms, Quite-Naive and Tailed-Substring, that solve the exact matching problem with constant

This research was partially supported by JSPS KAKENHI Grant Numbers JP15H05706 and JP19K20208.

space. The time complexities of these algorithms are same as the naive one, but they experimentally run much faster. Also, the Knuth-Morris-Pratt (KMP) algorithm [11] is well known as an algorithm that has a truly smaller time complexity than the naive one, but the KMP algorithm works slower in practice.

In this regard, several performance evaluation tools have been developed so far. Hume and Sunday [8] presented a testing framework for algorithms that solve the exact matching problem. The goal of the exact matching problem is to find all occurrences of a given pattern string inside a given text. Faro et al. [5] presented SMART[1], which is designed for developing, testing, comparing and evaluating exact matching algorithms. It contains implementations of 86 algorithms by C programming language and a wide corpus of 12 texts, including natural language texts, genome sequences, protein sequences, and random texts. A lot of known algorithms can be tested and compared with these texts and implementations. A new algorithm can also be added to experiment easily. Results of experiments can be output in various formats. SMART also provides a graphical user interface (SMARTGUI). It allows to run experiments and show real-time report of the experiments. These tools [5, 8] are among the most widely used testing frameworks. However, they are exclusive to exact matching algorithms only. In addition, local environments are needed to use them.

In this paper, we introduce AOBA (An Online Benchmark tool for Algorithms in stringology), an integrated platform on the web for testing and evaluating string processing algorithms. AOBA is not limited to the exact matching problem. It can handle various types of string processing problems including pattern matching, pattern mining, string compression, and so on. In the abstract, it can handle problems that have a unique solution text for an input text.

This paper is organized as follows. In Section 2, we give a brief description of AOBA. In Section 3, we describe the backend in detail and discuss its security. In Section 4, we provide experimental results on selected algorithms evaluated with AOBA. Finally, in Section 5, we summarize our work.

2 AOBA in Short

AOBA is an integrated web platform for string processing algorithms. Without any local environments, users can easily test their algorithms and compare their performance with other algorithms submitted to AOBA. AOBA is available at `http://aoba.iss.is.tohoku.ac.jp`. A demonstration video is also available at `https://youtu.be/d1Z5-LBLLJI`. Below are the key features of AOBA:

- complete support on the web, no need for a local environment;
- compatibility with various types of string processing problems;
- integrated datasets and evaluation environments;
- support for adding new problems and datasets through user requests;
- fair and secure evaluation by a sandbox environment;
- powerful visualizer for comparing performances of different algorithms.

[1] `https://smart-tool.github.io/smart/`

In order to test and evaluate their algorithm implementations on AOBA, users should make a *submission*. A submission should contain source files and build/run scripts that instruct the system how to compile the source files and run the program. Each submission should target only one *problem* (for instance, the exact matching problem). Each problem has multiple *test cases*, each consisting of two text files: the input and the expected output.

Once a submission has been made, AOBA compiles the source files according to the build script. After that, AOBA runs the program on the input text and compares the actual output from the program with the expected output for all test cases. The execution time and the maximum memory usage are measured during the execution. We call these measurement results the *performance* of the algorithm. Detailed descriptions of the evaluation flow will be given in Section 2.1. All compilations and executions are done in a sandbox environment whose specifications are defined for each problem.

We have also implemented an in-browser visualizer to analyze performances of algorithms. The performance of the submission is plotted immediately after execution. Detailed descriptions of the visualizer will be given in Section 2.2. Users can also request to add new problems and test cases to AOBA. Currently, approval from administrators is needed to publish a new problem.

2.1 Submitting an Algorithm

This section describes the workflow of an algorithm testing and evaluation in AOBA. A submission is processed in two steps, *pre-evaluation* and *evaluation*. In both steps, the submitted source code will be compiled with the build script. Then, the input data text is fed into the algorithm through the standard input. The compiled code is executed with the run script, and the actual output is verified for correctness by comparing with the expected output.

When the submission is made, it is immediately proceeded to the pre-evaluation step. The pre-evaluation step verifies the correctness of the submission by running it on a small number of test cases. If any error is found in this step, the submission is disallowed to proceed to the next step. Once the pre-evaluation has successfully completed, the submission can proceed to the evaluation step upon the user's request.

The evaluation step measures the performance, i.e., execution time and max memory usage, of the submission by running the program on test cases. Each test case used in the evaluation step has metadata representing characteristics of the test case in addition to the input and the output texts. The number of parameters constituting the metadata, the semantics and the type of each parameter are defined for each problem. For example, a test case of the exact matching problem has three integer parameters: text length n, pattern length m and alphabet size σ. At any time during the evaluation step, one can check the evaluation progress and the evaluation results with the visualizer.

Note that we have set some limitations on the number of executions on each step to provide our computational resources equally for every user.

2.2 The Performance Visualizer

The evaluation result is given as multi-dimensional data corresponding to the multiple parameters associated with test cases. AOBA is equipped with an in-browser visualizer that helps understanding the intricate result. By aggregating some axes, it shows the result in a 2D or 3D plot image, depending on the user's preference. Figure 1 shows a screenshot of the visualizer interface, which consists of the setting area and the plot area. The setting area holds settings about axes and their parameters. This visualizer is reactive: the plot is updated immediately whenever any changes to the settings are made.

The visualizer can simultaneously plot evaluation results from different submissions for the same problem. Figure 1 compares the performance of the Horspool algorithm [7] (HOR) against those of the Boyer-Moore [2] (BM) and Shift-And [1] (SA) algorithms for the exact string matching problem. One can observe that the execution time of HOR is short on long patterns over large alphabets similarly to BM. In contrast, that of SA has little dependency on those parameters. AOBA also provides a heat map plot for the submissions. A heat map plot focuses on the distribution of algorithms that have the fastest execution time or most efficient memory usage. Figure 3 shows an example of a heat map plot.

The visualizer is available on both of the submission page and the problem page in the web site. The one on the submission page shows the performance of that specific submission. On the problem page, it shows the best performances of submissions that have the fastest execution time or the most efficient memory usage, among all submissions for the problem.

3 Backend for Fair and Secure Execution

AOBA needs to safely execute untrusted source codes submitted by users. At the same time, measured performances must be accurate. In this section, we describe the secure execution environment called sandbox and the accurate performance measurement method to realize it. We also discuss security based on possible attacks. Finally, we describe the whole system architecture.

3.1 A sandbox environment

A *sandbox* is an evaluation environment for safe program execution. Sandbox environments have widely been studied in the computer security field. Existing sandboxes are mainly realized by the following methods [15–17], among which AOBA uses the container-based method.

Container-based The Linux kernel provides features for grouping processes together and allocating computational resources for each group. The one called `namespace` allocates operating-system-level resources, like process ID, interprocess communication, network, file system processes. On the other hand, `cgroups` restricts the physical resources that can be handled by the process in the group such as CPU and memory. Processes can be abstracted

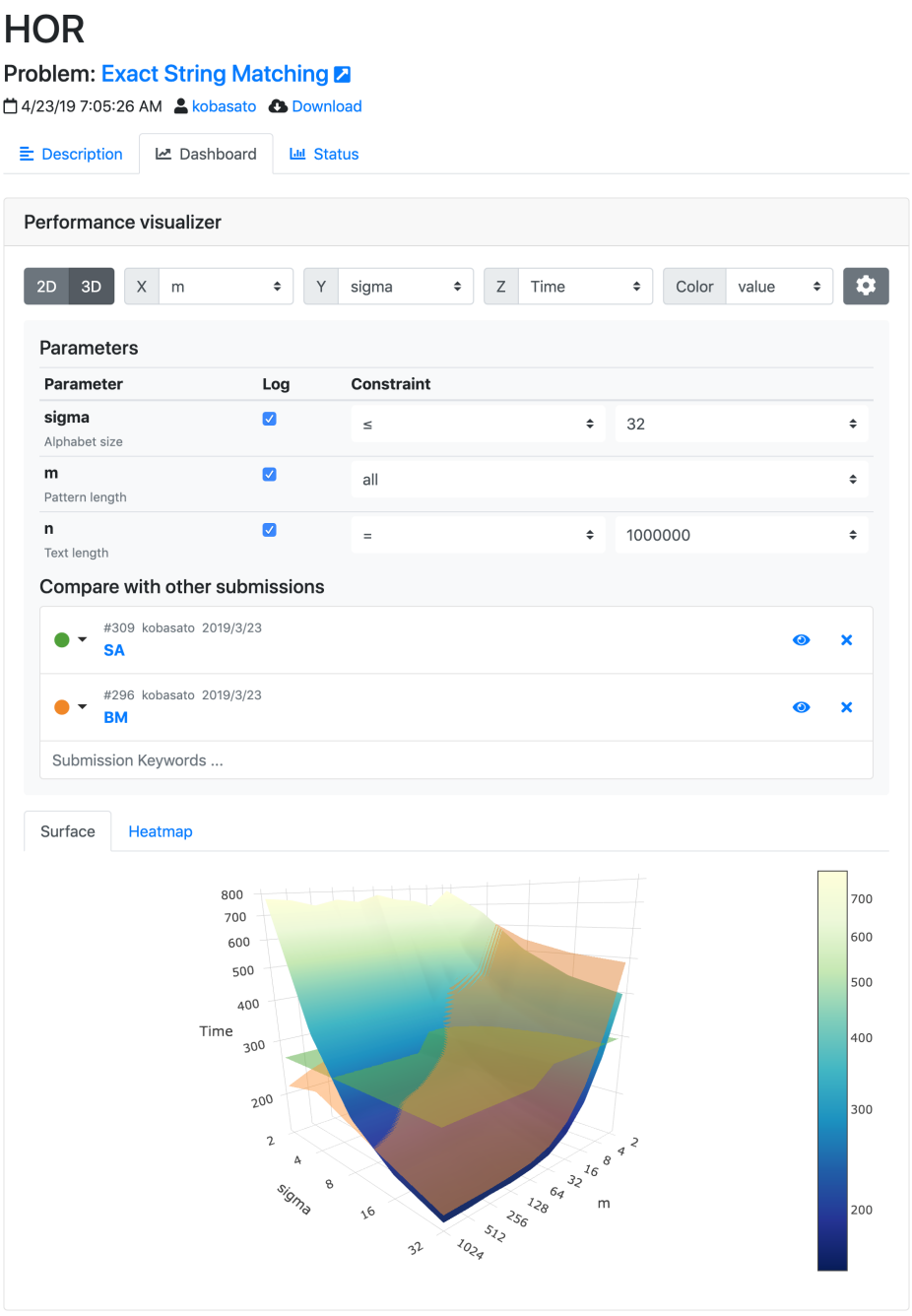

Fig. 1. A screenshot of the visualizer comparing the execution time (ms) of the Horspool (HOR) algorithm with the Boyer-Moore (BM) and Shift-And (SA) algorithms for the exact matching problem. The performances of HOR, SA and BM algorithms are plotted as gradation, green and orange surfaces, respectively.

from the environment in which they actually run. This type of abstraction method is called a *container*. Containers are much lighter and faster than virtual machines, since containers virtualize at the operating system level, unlike the virtual machines that virtualize hardware-level resources.

ptrace-based A system call `ptrace` is used for monitoring and controlling resources consumed by running processes. By using this in conjunction with `setrlimit` and `seccomp` system calls, an untrusted source code can be executed on limited resources.

LSM-based Linux Security Modules (LSM) is a framework that enables the many different access control models to be implemented as loadable kernel modules. Since it is loaded into the kernel and the processes are controlled from the kernel, the overhead is less than that of `ptrace`.

In addition to security, it is also important not to interfere with the performance of the process, so that the original execution time is measured as accurately as possible. Merry and Bruce [14] pointed out the `ptrace`-based methods had huge overhead in the case of frequent I/O calls and interprocess communication. While the LSM-based methods have less overhead than the `ptrace`-based method, they have high development cost, because they require to build a kernel module and hooking it properly to the kernel. On the other hand, as mentioned above, container-based methods are relatively easy to build. Moreover, containers operate extremely fast, since they are largely realized using functions supported by the kernel space.

In fact, Mareš and Blackham [13] revealed that the performance of the container-based method is as good as the performance of the LSM-based methods. From the facts above, AOBA uses the container-based method for implementing such environments. Specifically, AOBA uses Docker[2] for building the container-based sandbox environment. Docker is open-source software that provides secure and light-weight containers. Containers are created from *images* that specify their precise contents. Furthermore, the initial state of the file system provided by Docker can be set by specifying a definition file called Dockerfile. This allows us to manage compilers and libraries to be used.

To provide secure and fast IO access AOBA creates a `tmpfs` volume and mounts it to the Docker container. Since `tmpfs` is stored in the main memory instead of the hard disk, it ensures fast access to the data. When there is not enough space for the `tmpfs` volume in the main memory, the data access time will be slow. As a workaround, AOBA allocates memory needed for the container data before the execution, assuming the total size of data does not exceed the total main memory size.

3.2 Accurate Measurement

A measured performance may differ from the ideal value by mainly two factors: an error due to the unstable measurement and a malicious program that attempts

[2] `https://www.docker.com/`

to rewrite the measurement result. In this section, we give a detailed description of measuring execution time and memory usage on AOBA to prevent unstable measurement. We discuss its security in Section 3.3.

Measure execution time While measuring it outside a container is the safest way, a measured result contains time required for the API calls between the internal system and a container. This overhead may cause unstable results. In fact, we confirmed that an error of up to 200ms occurs on our environment. Instead, AOBA measures execution time directly inside a container by the `time` command. This method improves the accuracy but has a greater security risk. We will explain how to deal with it in Section 3.3.

Measure memory usage Generally, memory usage of a single process can be measured by the `ps` command and so on. However, this method cannot be used because AOBA allows multi-processing. Therefore, memory usage is measured by total memory usage over a whole container for multi-processing. More specifically, its value equals to `max_usage_in_bytes` in the `memory` subsystem of `cgroups`. The value measured by this method contains extra memory usage due to other processes such as `time` command. However, this error is constant and negligibly small for sufficiently large problems.

These measurement methods realize accurate measurement. Nevertheless, outliers may be measured depending on system conditions, so a measurement for a test case is repeated. Under the current setting, the program is repeatedly run three times. Among the runs, the one with the fastest execution time is reported as the final result.

3.3 Possible Attacks

Various kinds of attacks to the system are possible in both of program compilation and execution. Tochev and Bogdanov [15] provided a comprehensive description of possible attacks on judge systems. Attacks that they pointed out can be classified into the following three types.

Embedding answers into the program The simplest attack is embedding all possible answers to a source code. This can be prevented by preparing a sufficiently complex problem and limiting the submission file size. A more aggressive attack is possible: embedding test cases that are obtained illegally. Therefore, it is needed to ensure test cases are inaccessible from users.

Putting a heavy load on the system to make it unstable Submitting excessive file, Denial of Service (DoS) attacks during compilation and execution, excessive disk/memory usage, etc. fall under this type. Attacks of this type can be prevented by limiting available resources such as disk/memory usage and execution time.

Access something that should not be accessed Accessing forbidden files such as the answer of a test case is one of this type of attacks. As a more critical example, it is possible to break a sandbox environment. The security of containers and permissions should be maintained properly because most of these attacks depend on their vulnerabilities.

To protect AOBA from these types of attacks, we limit memory/disk usage, submission size, compilation/execution time, and access rights.

In addition to the above, various attacks can be considered because AOBA also allows multi-threading and multi-processing. For example, a process that remains running after its parent process has finished, called an *orphan process*, may cause some attacks that break a sandbox. A *fork bomb* attack is also possible. It is a DoS attack that creates processes infinitely to slow down the system. Each container has a process id space provided by `namespace`. A container kills all remaining processes in this space when it terminates. It ensures orphan processes in a container cannot break it. Available resources from a container are limited by `cgroups`, thus DoS attacks due to multi-processing like a fork bomb do not affect the entire system.

Moreover, attacks that rewrite the performance fraudulently should be considered because AOBA attaches importance to the reliability of measured performances. Memory usage is measured outside and throughout a container, so its value reliable as long as the security of containers is maintained.

A measurement of an execution time has more security threats compared to that of memory usage because it is done in a directory inside the container. For example, it is a possible attack that the main process of a submitted program creates an orphan process and terminates itself immediately to make its execution time less. After that, an orphan process calculates and outputs an answer. A more aggressive attack kills the process of the `time` command during the execution and writes fake output to tamper with execution time. To protect from such attacks, the `time` command for the measurement is executed by the `root` user, then the user is switched to a general user called the `guest` user that executes a submitted program. This method ensures that it cannot disturb a process of the `time` command. Outputs from the `time` command and a submitted program are redirected to specific files in a container by the `root` user. The `guest` user for the execution has no access rights to these files. Therefore, orphan processes cannot overwrite such outputs. Only a process whose execution time is measured by the `time` command can write outputs. In this way, the security of an execution time measurement is guaranteed.

3.4 System Architecture

The whole system of AOBA consists of the following five components.

Web provides an interface that allows users to interact with AOBA.
Web-Internal dispatches jobs related to a submission to appropriate components, manages and checks the status of other components.
Database stores data used in Web and Web-Internal.
Storage stores large files such as submitted source files and test cases.
Executor executes and evaluates submitted files in a sandbox environment.

AOBA evaluates submissions as follows. Web component receives submissions via the web, then Web-Internal component creates a *job* for each submission

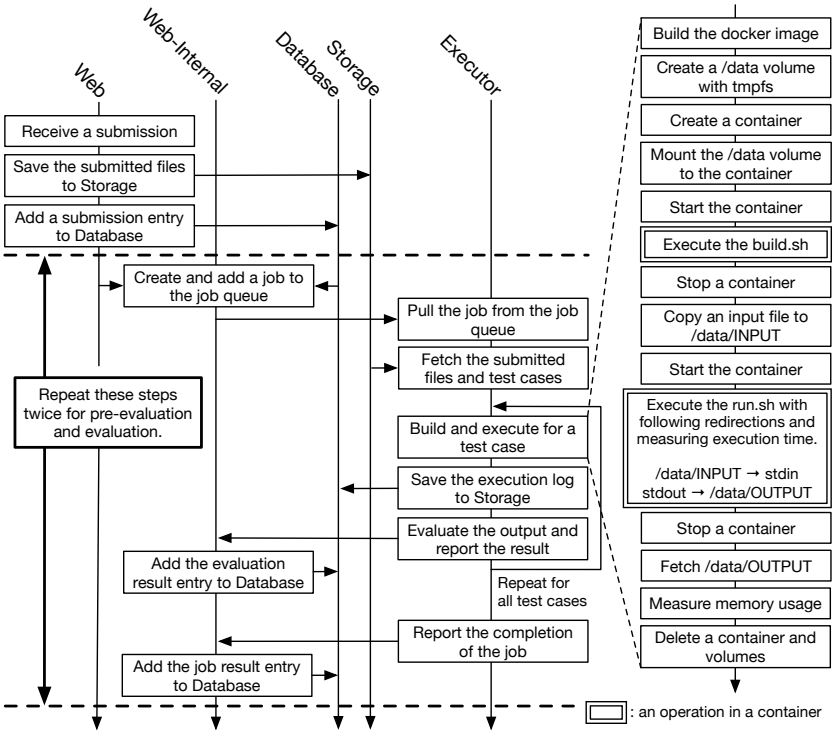

Fig. 2. The whole evaluation flow of a submission in AOBA.

and pushes it to the job queue for the pre-evaluation step. A job is an execution unit of the Executor; the collection of information necessary for evaluating the submission such as the ID of a submission and test cases. Then the Executor pulls the job from the job queue and evaluates it. If the submission passes the pre-evaluation, Web-Internal component creates another job for the evaluation step. The Executor evaluates it as well as the pre-evaluation step. Figure 2 shows the detailed evaluation flow.

All of these components except Web-Internal can be distributed to multiple hosts. Moreover, the Executor components are not hosted on the same machine with the other components for measurement stability.

4 Demonstrations

In this section, we show experimental results about several string problems as examples to demonstrate testing and analyzing algorithms in AOBA. In all experiments, an executor is hosted in a PC with Xeon E3-1220 V2, Ubuntu 18.04, Docker 18.09.0, 8GB RAM. In the sandbox, available resources are limited to 1 physical core, 1GB RAM.

Table 1. Evaluated exact matching algorithms.

Name	Description	Name	Description
BM	Boyer-Moore	KMP	Knuth-Morris-Pratt
HOR	Horspool	EBOM	Extended Backward-Oracle-Matching
NSN	Not-So-Naive	FJS	Franek-Jennings-Smyth
QS	Quick-Search	HASHq	Wu-Manber for Single Pattern Matching
RAITA	Raita	FSBNDM	Forward Simplified BNDM
SA	Shift-And	KBNDM	Factorized BNDM
TW	Two-Way	LBNDM	Long patterns BNDM
TVSBS	Thathoo et al.	SBNDM-BMH	Simplified BNDM with Horspool Shift
RCOL	Reverse-Colussi	SBNDM2	Simplified BNDM with loop-unrolling
TS	Tailed-Substring	SSEF	Streaming SIMD Extensions Filter

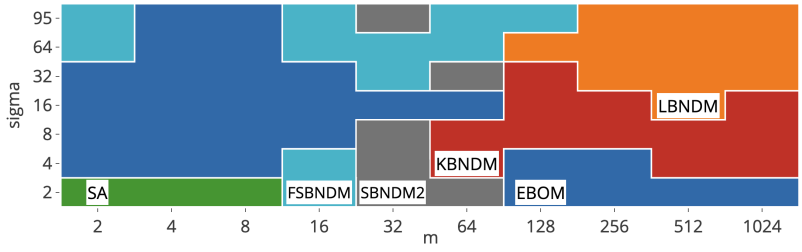

Fig. 3. Distribution of exact matching algorithms with the fastest execution time

First, we show evaluation results for the exact matching problem on AOBA to demonstrate examining the performance distribution of many algorithms. The exact string matching consists in finding all occurrences of a given pattern string P in a given string T. A lot of exact matching algorithms are researched and proposed [4]. We evaluated twenty algorithms shown in Table 1. All implementations are migrated from SMART [5][3]. In this experiment, a text T with σ symbols was generated randomly and the pattern P was picked randomly from T for $\sigma = 2, 4, 8, 16, 32, 64, 96$. The lengths of the text and pattern are $n = 10^6$ and $m = 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024$, respectively.

Figure 3 is the heat map plot of the visualizer that shows the distribution of algorithms that has the fastest execution time against parameters m and σ. Based on the obtained knowledge of the properties of algorithms, the most suitable algorithm can be selected for a target application.

Another problem that AOBA is equipped with is the order preserving pattern matching problem (OPPM) [10]. SMART does not have the functionality to test this problem. OPPM considers relative order of elements in numerical strings. For example, $(5, 3, 8)$ is *order-isomorphic* to $(4, 1, 12)$, since the second element is the smallest and the last element is the largest. On the other hand, $(5, 3, 8)$ is not order-isomorphic to $(8, 2, 6)$, where the relative orders of the first and the

[3] `http://www.dmi.unict.it/~faro/smart/algorithms.php`

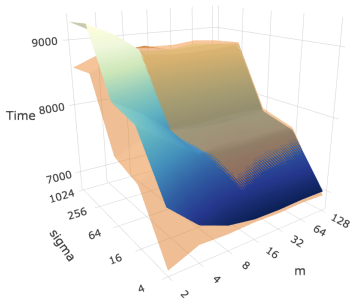

Fig. 4. The surface plot shows the execution time (ms) comparison of OP-KMP (orange) and OP-DS (blue).

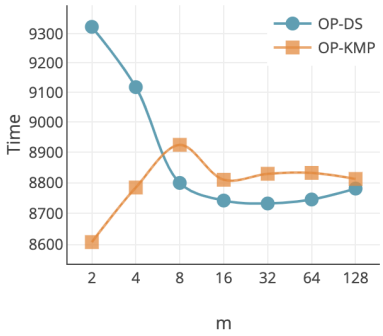

Fig. 5. The execution time (ms) comparison of OP-KMP and OP-DS where the alphabet size is fixed ($\sigma = 256$).

last elements are reversed in those two strings. OPPM consists in finding all the substrings of an input text T which are order-isomorphic to an input pattern P.

We let AOBA compare the KMP-based algorithm (OP-KMP) [10] and the duel-and-sweep based algorithm (OP-DS) [9] for OPPM. Figure 4 shows the execution time comparison of the algorithms against pattern length m and alphabet size σ by a 3D plot. These algorithms have almost the same linear dependency on the alphabet size σ, but it shows different dependency on the pattern length m. The 2D plot in Figure 5 focuses on the execution time dependency on m, where the alphabet size is fixed ($\sigma = 256$). OP-DS is the algorithm based on pruning pattern positions. The authors for OP-DS claim that it makes fewer comparisons between the pattern and the text than OP-KMP. Actually, OP-DS works slightly faster than OP-KMP in large pattern length ($m \geq 8$). In contrast, for small pattern length ($m \leq 4$), OP-KMP works much faster. It implies the pruning in OP-DS does not work well in such a situation.

5 Concluding Remarks

We introduced an integrated evaluation web platform AOBA designed for testing and analyzing the performance of algorithms for diverse types of string processing problems. We are expecting that the platform will help researchers to study efficient algorithms and implementations. In fact, some of the authors, Kobayashi et al. [12], have recently succeeded in discovering faster variants of the FJS algorithm [6] in the process of developing AOBA.

Still, the datasets and problems that currently AOBA provides may not be sufficiently many. We will continue enhancing AOBA, while users' contributions are very much welcome. We are also considering releasing an on-premise version.

A whole execution of an evaluation takes much time as the number of test cases increases. In addition, AOBA reports an algorithm's performance based on many multiple runs to make results more accurate but it also takes time. Cur-

rently, AOBA has an insufficient number of executor components. Some problems take several hours to evaluate a single submission. This can be solved by scaling executor components. However, more strategic solutions can also be considered. For instance, the performance may be approximated by using some test cases instead of all of them.

References

1. Baeza-Yates, R., Gonnet, G.: A new approach to text searching. ACM SIGIR Forum **23**, 168–175 (1988). https://doi.org/10.1145/75335.75352
2. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Communications of the ACM **20**(10), 762–772 (1977)
3. Cantone, D., Faro, S.: Searching for a substring with constant extra-space complexity. In: Third International Conference on Fun with algorithms. pp. 118–131 (2004)
4. Faro, S., Lecroq, T.: The exact online string matching problem: A review of the most recent results. ACM Computing Surveys **45**(2), 13 (2013)
5. Faro, S., Lecroq, T., Borzi, S., di Mauro, S., Maggio, A.: The string matching algorithms research tool. In: Proceedings of the Prague Stringology Conference 2016. pp. 99–111 (2016)
6. Franek, F., Jennings, C.G., Smyth, W.F.: A simple fast hybrid pattern-matching algorithm. Journal of Discrete Algorithms **5**(4), 682–695 (2007)
7. Horspool, R.N.: Practical fast searching in strings. Software: Practice and Experience **10**(6), 501–506 (1980)
8. Hume, A., Sunday, D.: Fast string searching. Software: Practice and Experience **21**(11), 1221–1248 (1991)
9. Jargalsaikhan, D., Ueki, Y., Yoshinaka, R., Shinohara, A., et al.: Duel and sweep algorithm for order-preserving pattern matching. In: International Conference on Current Trends in Theory and Practice of Informatics. pp. 624–635. Springer (2018)
10. Kim, J., Eades, P., Fleischer, R., Hong, S.H., Iliopoulos, C.S., Park, K., Puglisi, S.J., Tokuyama, T.: Order-preserving matching. Theoretical Computer Science **525**, 68–79 (2014)
11. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM journal on computing **6**(2), 323–350 (1977)
12. Kobayashi, S., Hendrian, D., Yoshinaka, R., Shinohara, A.: An improvement of the Franek-Jennings-Smyth pattern matching algorithm. In: Proceedings of the Prague Stringology Conference 2019. pp. 56–68 (2019)
13. Mareš, M., Blackham, B.: A new contest sandbox. Olympiads in Informatics **6**, 100–109 (2012)
14. Merry, B.: Performance analysis of sandboxes for reactive tasks. Olympiads in Informatics **4**, 87–94 (2010)
15. Tochev, T., Bogdanov, T.: Validating the security and stability of the grader for a programming contest system. Olympiads in Informatics **4** (2010)
16. Wasik, S., Antczak, M., Laskowski, A., Sternal, T., et al.: A survey on online judge systems and their applications. ACM Computing Surveys **51**(3), 1–34 (2018)
17. Yi, C., Feng, S., Gong, Z.: A comparison of sandbox technologies used in online judge systems. In: Applied Mechanics and Materials. vol. 490, pp. 1201–1204 (2014)