Implementation and Evaluation of Algorithms with ALGator

Tomaž Dobravec
Faculty of Computer and Information Science
University of Ljubljana, Slovenia
E-mail: tomaz.dobravec@fri.uni-lj.si

Keywords: automatic algorithm testing, quality evaluation, empirical analysis

Received: October 30, 2018

In this paper we present an automatic algorithm evaluation system called ALGATOR*, which was developed to facilitate the algorithm design and evaluation process. The system enables unbiased tests of the correctness of the algorithm's results on given test cases and comparisons of the quality of implemented algorithms for solving various kinds of problems (e.g. sorting data, matrix multiplication, traveler salesman problem, shortest path problem, and the like). Within the* ALGATOR *one can define a problem by specifying the problem descriptors, test sets with corresponding test cases, input parameters and output indicators, algorithm specifications and criteria for measuring the quality of algorithms. When a user of the system submits an algorithm for solving a given problem,* ALGATOR *automatically executes this algorithm on predefined tests, measures the quality indicators and prepares the results to be compared with the results of other algorithms in the system. The* ALGATOR *is meant to be used by algorithm developers to perform independent quality tests for their solutions.*

Povzetek: V tem delu je predstavljen sistem ALGATOR*, ki omogoča izvajanje implementiranih algoritmov na vnaprej pripravljenih testnih primerih ter ocenjevanje pravilnosti in kvalitete izvajanja. Da bi omogočili uporabo pri reševanju čim večjega nabora problemov, smo sistem zasnovali tako, da so njegovi gradniki (problem, algoritem, testna množica, testni primer) definirani na abstrakten način. Pred uporabo* ALGATOR*ja za reševanje nalog konkretnega problema, administrator projekta konkretizira abstraktne entitete, kasnejši uporabniki (razvijalci algoritmov) pa konkretizirajo le svoje izdelke (algoritme). Izvedba testiranja poteka povsem samodejno, implementirani algoritmi se poženejo na vseh testnih množicah, rezultati testiranja (časovni in drugi indikatorji) pa se zapišejo v bazo in se kasneje lahko uporabijo za prikaz v obliki grafikonov in tabel ali za analizo in primerjavo z rezultati drugih algoritmov.*

1 Introduction

As written by Aho et al. in [1], *"Efforts must be made to ensure that promising algorithms discovered by the theory community are implemented, tested and refined to the point where they can be usefully applied in practice. [...] to increase the impact of theory on key application areas"*. It is very important to develop good algorithms and theoretically prove their efficiency. On the other hand, if these algorithms do not perform well in practice, all the theoretical work is not much more than "art for art's sake".

From this point of view, algorithm evaluation is a very important part of an algorithm design process. To facilitate the implementation, execution and evaluation of the algorithms and make this part of the algorithm design process as simple as possible, the ALGATOR system was developed. It can be used to execute an algorithm implementation on the given predefined sets of test cases and to analyze various indicators of the execution. Within every project of the system user can define the problem to be solved, sets of test cases, parameters of the input and indicators of the output data and the criteria for the algorithm quality evaluation. When a project is defined, any number of algorithm implementations (programs) can be added. When requested, system executes all the implemented algorithms, checks the correctness (on a given set of tests) and compares the quality of the results. Using the ALGATOR user can add additional quality criteria, draw graphs and perform evaluations and comparisons of defined algorithms.

1.1 A typical usage of the system

ALGATOR is a complex system in which users can perform various kinds of tasks - from technically demanding (e.g., defining the properties of a project) to rather straightforward and simple tasks (e.g., using charts presented on a web page to compare the quality of two algorithms). To keep the hard job for the experts and to simplify the usage for the others, ALGATOR uses different user roles and privileges, namely, a system administrator, a project administrator, a researcher and a guest. There are many different possible ways of using the ALGATOR system but the main common and the most perspective is the one presented in the following. A system administrator prepares the system by providing the hardware, installing ALGATOR software packages, and publishing the internet address of the installed system. A project administrator adds a new project and defines all the project's properties. When the project is completely defined and declared as public, ALGATOR automatically generates an internet subpage with the project presentation and usage guide sections. The project administrator adds some state of the art algorithms for solving the problem of the project, which will be used as a reference for the evaluation process (i.e. the results of the algorithms added by researchers will be compared with the results of these referential algorithms). According to the rules, presented at the project's website, a researcher adds a new algorithm. ALGATOR will automatically run the new algorithm on predefined tests. The researcher then checks the correctness and compares the results of his algorithm with the results of the other algorithms defined in the project. The researcher can also decide to make the algorithm public (by default, the algorithms are private and can only be seen by the author). A guest of the system lists the results and prints the graphs and other data produced by ALGATOR. A guest can also perform some actions (like customization of the presentation) that do not alter the project configuration. At any time a guest can register to the system and contribute as a project administrator or as a researcher.

1.2 Related work

ALGATOR was created to fill the gap in the area of algorithm evaluation process since, as far as we know, there is no similar tool available that would offer all of ALGATOR's capabilities. Even though there are some tools available on the web that allow analysis of program execution, they have been developed with different goals, therefore their support in the area of algorithm analysis is not possible or it is too cumbersome. On the web, we can find the following tools that partly cover the ALGATOR's functionality.

Code Profilers (JProfiler [12], JMeter (`jmeter.apache.org`), Netbeans Profiler (`profiler.netbeans.org`) and the like) can follow and record a program execution indicators and they are able to display various information about the usage of computer resources (like time and memory). Using the recorded data, a user can detect possible code errors and memory leaks and perform various code optimization. Although code profilers offer a number of different tests and measurements, their use is not appropriate for accurate analysis of algorithms, as they do not allow to tailor the sets of test cases and to perform analysis based on user-defined measurements and output indicators.

Algorithms presentation pages (like [5]) systematically present various computer problems and practical examples, grouped into categories (e.g. searching and sorting problems, mathematical problems, graph problems, string problems, etc.). Besides the presentation of a wider field for each domain, the pages usually present at least one solution (i.e. algorithm) for each problem and offer the possibility to upload user-defined implementations. The emphasis of this kind of pages is mainly on educational asspects and on presentation of problem descriptions therefore they do not allow the independent analysis of the algorithms using the user criteria nor do they offer the tools to display custom-defined graphs or other statistical information.

Problem-domain dedicated pages present a specific problem domain (like [2] for mixed integer programming problem and [3] for the traveler selsman problem) and they offer tools to compare different solutions. Although these pages are not configurable (for example, user can use only pre-defined measurements and result indicators), they are ideal for testing a specific solutions. The main drawback comparing to ALGATOR is that each page covers only one problem domain while ALGATOR aims to cover as many areas and problems domains as possible and to present the results in a uniform way for all of them.

Educational coding platforms (like Sphere online judge (`www.spoj.com`), CodeChef (`www.codechef.com`), CodeForces (`codeforces.com`), CodeFights (`codefights.com`) and the like) introduce different problems with exactly defined input and expected output. After a user of such a platform uploads its own solution, the system performs its execution, evaluation and ranking according to regularity and efficiency. The essential difference between ALGATOR and these environments is that they offer only several predefined basic algorithm quality criteria (like, does an algorithm return a correct solution or not, or, how many of the given tests were solved correctly, and the like). In addition, the purpose of these platforms is focused on the process of learning programming skills and not in finding an optimal solutions for specific problems, which results in insufficient result presentation (e.g., it is not possible to compare different solutions using charts, plots or tables). For the purpose, for which these pages exist (i.e. to help their users to learn programming), they serve very good and they can be of a great help, nevertheless for the purpose of comparing different algorithms to solve the same problem they lack some crucial functionalities (like fully configurable input, output and result-presentation layer of the project) which are supported by ALGATOR.

2 Project definition

The main task of the ALGATOR's project administrator is to provide the configuration files and to implement corresponding Java or C++ interfaces. Besides the definition of the output format (where the sequence of the parameters and indicators in output file is described), the test cases, the test sets and the algorithm structure has to be defined precisely.

The test cases and the test sets

A test case in ALGATOR execution environment is defined by a subclass of the `TestCase` class, which contains data structures to hold the input and output (result) data. Since these data structures are project-specific (i.e. each problem needs data of its own type) the project administrator has to implement the `[Project]Input` and the `[Project]Output` classes and prepare the corresponding data structures. For example, in the data-sorting problem, the `SortInput` class could be defined as presented in Listings 1.

Listings 1.
An input of the sorting problem

```
public class SortInput extends AbstractInput{
  // An array of data to be sorted
  public int [] arrayToSort;
}
```

A test set contains one or more test cases and it is a minimal execution unit. A test set is defined by a single text file in which every line defines one test case. The format of these lines is project-specific and it is defined by a project administrator. If required, additional files can be used to specify the test cases. Again, the syntax and the semantics of the content of these files is defined by the project administrator. Listings 2 presents an example of the text file defining five test cases for the data-sorting problem.

Listings 2.
Examples of test cases for the sorting problem

```
test1:10:INLINE:3 5 1 8 6 3 8 9 0 6
test2:10000:RND
test3:20000:SORTED
test4:30000:INVERSE
test5:50000:FILE:numbers.txt:16534
```

To iterate through the text file associated with a given test set, ALGATOR uses the methods of the `DefaultTestSetIterator` class. For each line read from the text file the `getTestCase()` method of the `[Project]TestCase` class is called. This method parses the input line and creates a set of parameters that describe the test case. Using these parameters it calls the `generateTestCase()` method which creates the instance of the test case. Since the representation of test cases is project-specific, the project administrator has to provide the correct implementation of the `getTestCase()` and the `generateTestCase()` methods. All the other methods are general and they can be used without modification. A part of an implementation of the `generateTestCase()` method is presented in Listings 3.

Listings 3.
A part of the `generateTestCase()` method for the sorting problem

```
@Override
public SortingTestCase generateTestCase
                (Variables inputParameters) {
  int probSize  = inputParameters.
          getVariable("N").getIntValue();
  String group = inputParameters.
          getVariable("Group").getStringValue();

  // prepare an array of integers  ...
  int [] array = new int[probSize];
  // ... and fill table according to group
  switch (group) {
    case "RND":
      Random rnd = new Random();
      for (int i = 0; i < probSize; i++)
        array[i] = Math.abs(rnd.nextInt());
      break;
    case "SORTED":
      for (int i = 0; i < probSize; i++)
        array[i] = i;
      break;
    // ...
  }

  // create a test case ...
  SortingTestCase sortingTestCase =
    new SortingTestCase();
  sortingTestCase.setInput(
    new SortingInput(array));
  int [] expectedResultArray =
    getSortedArray(array);
  sortingTestCase.setExpectedOutput(
    new SortingOutput(expectedResultArray));
  // ... and return
  return sortingTestCase;
}
```

Algorithms

The "heart" of each project are the implemented algorithms. Each algorithm is represented by a subclass of the `AbsAlgorithm` class with the following methods:

`ErrorStatus init(TestCase test)`. This method takes care of the input of the algorithm; it reads the test case and prepares the data. To enable fast algorithm execution all expensive initial tasks have to be done in this method. When this method is done all the required algorithm's input data has to be prepared in a proper format.

`void run()`. In this method the `execute()` method is called. The parameters of the `execute()` method are project-specific and are provided by the project administrator. ALGATOR takes the time of the execution of the `run()` method as an algorithm execution time therefore nothing else than the `execute()` method call should be placed in the `run()` method (see Listings 4).

`ParameterSet done()`. This method collects all the parameters and indicators of the execution and prepares them in the form suitable to be written into the output file.

The `AbsAlgorithm` class is abstract and the project administrator has to provide the `[Project]AbsAlgorithm` subclass with the above

Listings 4.
A simple implementation of the `run()` method

```
public void run() {
  result = execute(sortTestCase.getInput());
}
```

mentioned methods implemented. Besides he has to declare fields for input data (in these fields the input data obtained from the test case will be stored during the execution of the `init()` method) and the `abstract execute()` method with appropriate number and type of parameters. The task of the researcher is to implement a subclass of `[Project]AbsAlgorithm` and implement the `execute()` method. In other words, all the "dirty job" of preparing data and collecting the results is done by the project administrator. The researcher who wants to provide an algorithm only has to implement one method which returns a proper result. In the case of data-sorting problem, an algorithm only needs to sort the array of data; a very simple (but technically correct) algorithm for sorting data is listed in Listings 5.

Listings 5.
A simple implementation of the algorithm

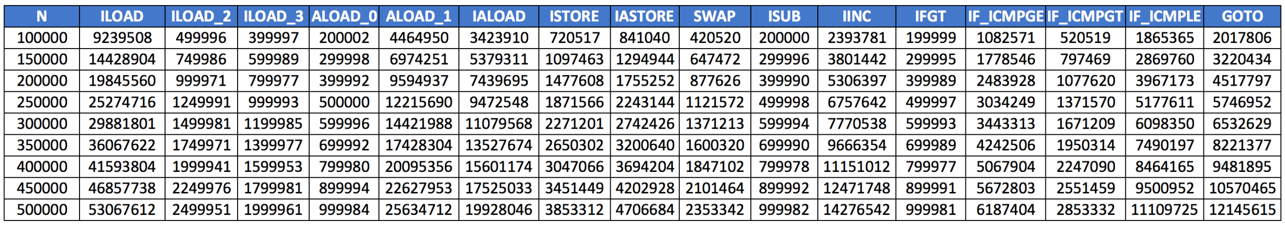
```
public class JavaSortAlgorithm
            extends SortAbsAlgorithm {

  SortOutput execute(SortInput input) {
    SortOutput result = new SortOutput();
    java.util.Arrays.sort(input.arrayToSort);
    result.sortedArray = input.arrayToSort;
    return result;
  }
}
```

3 Indicators of the algorithm

Since ALGATOR was designed to be used for various kinds of problems, the criteria for measuring the quality of algorithms are not defined as a part of the system but they have to be defined by the project administrator. The current version of the system enables measurements of three different kinds of indicators: a) the indicators to measure the speed and the quality of the algorithm (the so called EM indicators), b) the project-specific counters to count the usage of the parts of the algorithm's program code (the so called CNT indicators), and c) the counters of the Java byte code usage (the so called JVM indicators). These indicators are calculated with independent measurements that are performed as separated tasks so they do not interfere with one another. For example: when ALGATOR measures time, the CNT and JVM indicators are disabled. To perform the JVM measurements a dedicated Java virtual machine is used.

The EM measurements.
These measurements are used to measure the time and other project-specific indicators. All measurements of the time are performed automatically. To provide as accurate time indicators as possible ALGATOR tries to reduce the influence of the uncontrolled computer activities (e.g. sudden increase of a system resource usage) by running each algorithm several times. The system measures the first, the best, the worst and the average time of the execution. The project administrator only needs to specify the phases of algorithm execution (e.g. the pre-processing phase, the main phase, the post-processing phase, ...) and to select which of the time indicators are to be presented as the result of execution.
The project-specific indicators are defined by the project administrator. They can be presented as a string or as a number. For example, for exact algorithms, the value of an indicator could be "OK" (is the algorithm produced the correct result) or "NOK" (if the result of the algorithm is not correct). For approximation algorithms the value of an indicator could be the quality of the result (i.e. the quotient of the correct result and the result of the algorithm).
ALGATOR produces the values of the EM indicators for each `(algorithm, test_case)` pair by performing the following steps: a) load the test case and create its project-specific representation, b) load the algorithm (by using the Java reflection), c) read the values of the test case specific parameters, d) run the algorithm and measure its time consumption, e) read the values of the time indicators, f) determine the values of the project-specific indicators, g) writes all the parameters and the indicators of the execution to the output file. Since the time indicators are natively plugged into the ALGATOR system, the step f) of the above procedure is the only step that has to be configured by the project administrator (everything else is done automatically by ALGATOR). To configure a project-specific indicator the administrator has to provide its description in a configuration file (defining indicators type and possible values) and a program code to determine its value using the test case parameters and the algorithm's result. For example, in the Sorting problem the administrator provides a correctness indicator by a code as presented in Listings 6. In this code (which is a part of a `done()` method that is invoked just after the execution of the algorithm) the `isArraySorted()` method returns `true` if the input array is sorted. The name (`"Correctness"`) and the type (`String`) of the indicator has to be defined in a configuration file.

The CNT measurements.
The CNT measurements are used to count the usage of the parts of the program code. This option is used to analyze the usage of a certain system resource or to count the usage of the selected type of commands on the programming language level. Using this one can, for example, measure how many times the memory allocation functions were executed during the algorithm execution and the amount of



N	ILOAD	ILOAD_2	ILOAD_3	ALOAD_0	ALOAD_1	IALOAD	ISTORE	IASTORE	SWAP	ISUB	IINC	IFGT	IF_ICMPGE	IF_ICMPGT	IF_ICMPLE	GOTO
100000	9239508	499996	399997	200002	4464950	3423910	720517	841040	420520	200000	2393781	199999	1082571	520519	1865365	2017806
150000	14428904	749986	599989	299998	6974251	5379311	1097463	1294944	647472	299996	3801442	299995	1778546	797469	2869760	3220434
200000	19845560	999971	799977	399992	9594937	7439695	1477608	1755252	877626	399990	5306397	399989	2483928	1077620	3967173	4517797
250000	25274716	1249991	999993	500000	12215690	9472548	1871566	2243144	1121572	499998	6757642	499997	3034249	1371570	5177611	5746952
300000	29881801	1499981	1199985	599996	14421988	11079568	2271201	2742426	1371213	599994	7770538	599993	3443313	1671209	6098350	6532629
350000	36067622	1749971	1399977	699992	17428304	13527674	2650302	3200640	1600320	699990	9666354	699989	4242506	1950314	7490197	8221377
400000	41593804	1999941	1599953	799980	20095356	15601174	3047066	3694204	1847102	799978	11151012	799977	5067904	2247090	8464165	9481895
450000	46857738	2249976	1799981	899994	22627953	17525033	3451449	4202928	2101464	899992	12471748	899991	5672803	2551459	9500952	10570465
500000	53067612	2499951	1999961	999984	25634712	19928046	3853312	4706684	2353342	999982	14276542	999981	6187404	2853332	11109725	12145615

Figure 1: The number of Java bytecode instructions used by Hoare's Quicksort algorithm while sorting integer arrays with $\mathbf{N} = \mathbf{100.000} \ldots \mathbf{500.000}$ elements.

Listings 6.
Checking for the correctness of the algorithm

```
boolean checkOK = isArraySorted(testArray);
EIndicator checkPar = new EIndicator(
  "Correctness",  checkOK ? "OK" : "NOK");
indicators.addIndicator(checkPar);
```

the memory allocated by these calls. One can also use CNT measurements to detect which part of the algorithm is most frequently used. For example, if the problem in concern would be data-sorting, using the CNT measurements one could count the number of comparisons, the number of swaps of elements and the number of recursive function calls (which are the measures that can predict the algorithm execution behavior [11]). To facilitate the CNT measurement in the project, the project administrator has to define the names and the meaning of the counters and the researchers have to tag the appropriate places in their code. Everything else is done automatically by ALGATOR.
For example, if we want to count the number of swaps and the number of comparisons performed by a sorting algorithm, we would define two counters (namely, the `"CMP"` and the `"SWAP"` counter) in a configuration of a sorting problem. Additionally, to ensure a correct value of these counters, the source code of algorithms should be tagged, so that every code line in which a comparison of two elements appears would be accompanied by at tag line `//@count{CMP,1}`, and a code line in which a swap of two elements is invoked by `//@count{SWAP,1}`. An example of tagged code for the BubbleSort algorithm is presented in Listings 7.

Listings 7.
A tagged source code of the BubbleSort algorithm used to count the number of comparisons and swaps.

```
public void execute(int[] data) {
  for (int i=0; i<data.length; i++)
    for (int j=0; j<data.length-1; j++) {
      //@COUNT{CMP, 1}
      if (data[j] > data[j+1]) {
        //@COUNT{SWAP, 1}
        swap(data, i, j);
      }
    }
  }
}
```

When ALGATOR is asked to provide the counter values, it replaces all the tags with a Java code (e.g., it replaces a tag `//@count{CMP,1}` with a Java code `Counters.add("CMP", 1);`), recompiles the source and runs the algorithm. When the algorithm stops, ALGATOR collects the values of the counters and writes them to an appropriate output file.

The JVM measurements.
An algorithm written in the Java programming language compiles into Java byte code. An interesting option offered by ALGATOR is the ability to count how many times each byte code instruction was used during the execution of an algorithm on a given test case. To facilitate this option ALGATOR uses a dedicated Java virtual machine `VMEP` which was developed as a part of ALGATOR project [9]. This virtual machine [10] extends an open source virtual machine JamVM [8] and supports counting of the usage of each bytecode instruction used during the algorithm execution. When ALGATOR is asked to provide JVM statistics, it executes the algorithm in the VMEP and stores the bytecode–usage counters that it returns. In [7] Lambert and Power indicated that the frequency of the usage of each byte code instruction can be used to predict the execution time. Even though ALGATOR's ability to count the byte code instructions usage is quite young, we expect that the data produced by the JVM measurements could be useful not only for the quantitative but also the substantive analysis of the algorithms.

As an usage example of the VMEP's indicators let us consider the Hoare's implementation of a sorting algorithm (i.e. a Quicksort algorithm in which the partitioning phase uses one pivot to split a given array into two subarrays [6]). It is known that this algorithm performs $\mathcal{O}(n \;\log n)$ steps on average to sort a given array of n elements. Running this algorithm in ALGator leads to interesting conclusions as presented in the following. To sort an array of integers the Hoare's algorithm uses only 16 (out of 202 possible) Java bytecode instructions, namely, `ILOAD`, `ILOAD_2`, `ILOAD_3`, `ALOAD_0`, `ALOAD_1`, `IALOAD`, `ISTORE`, `IASTORE`, `SWAP`, `ISUB`, `IINC`, `IFGT`, `IF_ICMPGE`, `IF_ICMPGT`, `IF_ICMPLE`, and `GOTO` (see Figure 1).

Among these instructions the most common used are the `ILOAD` (32%), `ALOAD_1` (15%), `IALOAD` (12%), `IINC` (8%) and `GOTO` (6%) instructions (an average usage of the other instructions is less than 5%). The overall number of all instructions used is $57,8 * n \log n$, with a relative error (for $n = 100.000 \ldots 1.000.000$) less than 3,5%. Which means, for example, that for n=100.000, the algorithm will perform approximately 29 millions of java bytecode instructions to sort an array. As a consequence, knowing only the size of the input array, one can predict the number of required instructions very accurately. On the other hand, this shallow analysis can not be used to predict the execution time of the algorithm due to a weak relation between the number of used instructions and the time consumption. Using the results obtained on an Intel(R) Core(TM) i7-6700 CPU computer running at 3.40GHz an average quotient between the number of instructions and the execution time (in microseconds) is 3949 (with a relative error 17.3% for $n = 100.000 \ldots 1.000.000$), which means that on average the Java virtual machine performs around 4000 instructions per microsecond (i.e. 0.25 nano second per instruction). For more accurate analysis of the relation between the number of instructions and the execution time, we would need to distinguish slow and fast instructions [7] with a special attention being paid to instructions that can fetch the data from a non-cached memory.

Listings 8.
An example of the ALGATOR's query

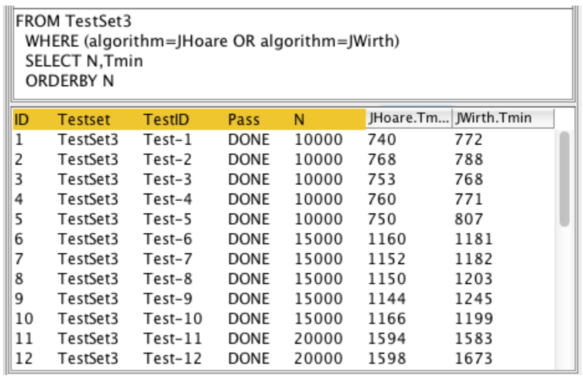
```
queryF1C1 = FROM TestSet0
  WHERE (algorithm=*) AND ComputerID=F1.C1
  SELECT Tmin AS A1;
queryF2C1 = FROM TestSet0
  WHERE (algorithm=*) AND ComputerID=F2.C1
  SELECT Tmin AS A2;
FROM queryF1C1, queryF2C1
  WHERE (algorithm=JHoare)
  SELECT N, A1/A2 AS Q
```

4 Analyzing the results

As a result of the algorithm execution ALGATOR produces text output files. For each tuple (algorithm, test set, measurement) one file is created; each line in this file contains parameters and indicators of one test case.

The data in the output line is separated by semicolons (CSV format). For efficient work with this data ALGATOR provides the analyzer with its own query language and with the visualization module for presenting data as graphs. For example, to get the minimal execution times for algorithms named JHoare and JWirth on the test set called TestSet3, a user can run query as depicted in Figure 2.

ALGATOR query language is a powerful tool that enables all sorts of data manipulation. An example of a complex query to calculate the quotient of minimal



```
FROM TestSet3
  WHERE (algorithm=JHoare OR algorithm=JWirth)
  SELECT N,Tmin
  ORDERBY N
```

ID	Testset	TestID	Pass	N	JHoare.Tm...	JWirth.Tmin
1	TestSet3	Test-1	DONE	10000	740	772
2	TestSet3	Test-2	DONE	10000	768	788
3	TestSet3	Test-3	DONE	10000	753	768
4	TestSet3	Test-4	DONE	10000	760	771
5	TestSet3	Test-5	DONE	10000	750	807
6	TestSet3	Test-6	DONE	15000	1160	1181
7	TestSet3	Test-7	DONE	15000	1152	1182
8	TestSet3	Test-8	DONE	15000	1150	1203
9	TestSet3	Test-9	DONE	15000	1144	1245
10	TestSet3	Test-10	DONE	15000	1166	1199
11	TestSet3	Test-11	DONE	20000	1594	1583
12	TestSet3	Test-12	DONE	20000	1598	1673

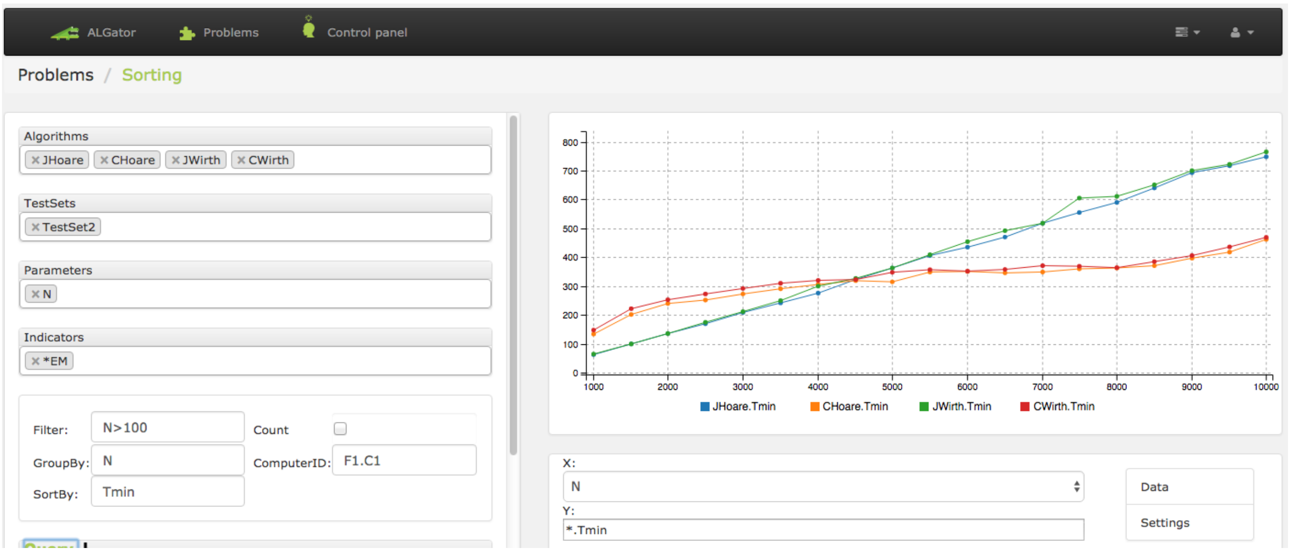
Figure 2: An example of data query with result.

times for the JHoare algorithm running on two different computers (F1.C1 and F2.C1) is presented in Listings 8. Note that the ALGATOR system might contain several computers that are able to execute the algorithms (such a computer in the system is called an execution engine). Each execution engine has its name, which comprises of a name of a computer family and a unique name of the computer inside this family (e.g. in the name F1.C1 the F1 represents the family and C1 the computer name). One family contains computers with equal hardware configuration. To provide comparable results, the algorithms of a given problem are usually run on the same computer (or at least on computers of the same family). Nevertheless, a researcher might additionally run algorithms on other computers and compare the execution results as presented in the query in Listings 8. If the computers used in this test have different hardware configurations, the results of such a comparison might reveal the influence of the particular hardware to the algorithms' behaviour.

The results of the execution can be analysed in one of the ALGATOR's visualization modules (one is implemented as a web and the other as a standalone application). In these modules a user can design queries (to produce arrays of numerical results) and draw charts as depicted in Figure 3.

5 Conclusion

The execution part of ALGATOR [4] was developed in both Java and C++ programming languages, therefore the algorithms to be tested could be implemented in one of these two languages. Measuring the exact execution time of the algorithms written in Java is a challenging task since the system can only measure real time and because there is no way to eliminate the side effects of the Java virtual machine's background tasks (e.g. garbage collection). To overcome this problem, ALGATOR executes each algorithm several times and reports the first, the minimal, the maximal and the average time of execution. Comparing and analyzing these times one can detect the influence of



Algorithms: ×JHoare ×CHoare ×JWirth ×CWirth
TestSets: ×TestSet2
Parameters: ×N
Indicators: ×*EM
Filter: N>100
Count: ☐
GroupBy: N
ComputerID: F1.C1
SortBy: Tmin
X: N
Y: *.Tmin

Figure 3: The visualization module of ALGATOR.

the execution environment to the overall execution time. In many cases this influence is negligible. Having the Java implementation of the algorithm also has some benefits. Namely, ALGATOR counts and generates the statistics of the usage of the Java byte code instructions. As stated in [7] these statistics provide enough information to be used for the platform independent timing of the algorithms. Our preliminary tests indicate a high correlation between the number of used Java byte code instructions (multiplied by the corresponding weight depending on the type of instruction) and the execution time. The ability to implement the algorithms in both (Java and C++) programming languages, enables the researchers to compare the execution time of both and to estimate the impact of the programming language. ALGATOR is a testing environment, which aims to make the testing process as easy as possible for both, the project administrators and for the researchers. We tired to minimize the effort that has to be used to prepare the project and the algorithm and we think that this goal was achieved. The biggest challenge for the project administrator is to prepare adequate test cases and to write several lines of Java of C++ code (in an average case not more that about 100 lines of code), while the researcher has to write only a few lines of code to call the existing Java or C++ implementation of the algorithm. All the other tasks needed to execute the algorithm and to produce the desired indicators are performed automatically by ALGATOR, therefore the researchers can focus on the analyses of the results. Furthermore, ALGATOR uses the same test cases for all the algorithms of the project, therefore the researchers can not tailor the tests to be optimal for their implementations, which makes the results of the evaluation fair and reliable.

References

[1] A. V. Aho, D. S. Johnson, R. M. Karp, S. R. Kosaraju, C. C. McGeoch, C. H. Papadimitriou, and P. Pevzner. Emerging opportunities for theoretical computer science. *SIGACT News*, 28(3):65–74, 1997.

[2] D. Applegate, W. Cook, S. Dash, and M. Mevenkamp. QSopt linear programming solver. www.math.uwaterloo.ca/ bico/qsopt, 2011.

[3] W. Cook. Concorde TSP solver. www.math.uwaterloo.ca/tsp/concorde, 2015.

[4] T. Dobravec. ALGator - an open source automatic algorithm evaluation system. https://github.com/ALGatorDevel/Algator, 2018.

[5] G. F. Geeks. A computer science portal for geeks. www.geeksforgeeks.org/fundamentals-of-algorithms, 2018.

[6] C. A. R. Hoare. Quicksort. *Comput. J.*, 5:10–15, 1962.

[7] J. M. Lambert and J. F. Power. Platform independent timing of Java virtual machine bytecode instructions. *Electronic Notes in Theoretical Computer Science*, 220:79–113, 2008.

[8] R. Lougher. JamVM - an open source Java virtual machine. jamvm.sourceforge.net, 2014.

[9] J. Nikolaj. Java virtual machine for counting the Java bytecode usage (original title: Predelava javanskega navideznega stroja za štetje ukazov zložne kode, language: Slovene). *University of Ljubljana, Faculty of Computer and Information Science*, 2014.

[10] J. Nikolaj. A source code of VMEP virtual machine. github.com/nikolai5slo/jamvm, 2014.

[11] R. Segedwick. The analysis of Quicksort programs. *Acta Informatica*, 7:327–355, 1977.

[12] E. Technologies. The definitive guide to JProfiler (ebook). resources.ej-technologies.com/jprofiler/help/doc/JProfiler.pdf, 2018.