Analysis of Eolang efficiency Stage IV

HSE Team
hsalekh@hse.ru
HSE
Moscow,Russia

Abstract

EO programming language is a new object-oriented programming language that has been going through many phases of development in a research and development project. In this phase, we compare Eolang efficiency to C++ and Java; detect differences; identify pros and cons by SWOT; define metrics to compare, create benchmark tests, and write test cases as well as automate testing and bench-marking to run for different EO compiler versions. The main goal is to analyze Eolang efficiency.

Keywords: SWOT, efficiency, metrics, benchmark, test cases

1 Introduction

This report includes metrics analysis, SWOT analysis, benchmark description and results. Benchmarks and tests are available at GitHub repository and integrated into release pipeline in the form of Continuous Integration (CI).

2 Comparison Metrics

2.1 Criteria for comparison

There are many criteria important to comparing or evaluating general purpose programming languages:

1. **Simplicity of language constructs**.
   The simplicity of a language design[2] include such measurable aspects as the minimality of required concepts and the integrity and consistency of its structures. Simpilicity here relates to ease of programming. Simple is beautiful is the golden mantra in programming [7]. While efficiency and performance are major factors, simplicity and maintenance cost wins over them in many use cases. These become a deciding factor while choosing a programming language, exploring features in a language or even deciding on standard coding practices within an organization.
   Binary result: yes/no.
2. **Readability**.
   Readability refers to the ease with which codes can be read and understood [4].This relates to maintainability, an important factor as many programs greatly outlive their expected lifetimes.
   Binary result: yes/no.
3. **Compilation speed**.
   This metric identify the program total execution time(m.s.ms) [1]. This can comparatively help determine how long it takes to run an algorithm in the different languages.
   Result: 1..5, 5-high speed,1-low speed
4. **Memory usage**.
   This metric identify the total memory that were used due to program performing(Kbyte). Some programming languages may be efficient in memory consumption while others are not. This criteria is important for clarify that difference between EO, C++ and Java.
   Result: 1..5, 5-low usage,1-high usage.
5. **LOC (Lines of code)**.
   The total amount of LOC that were used to develop the algorithm. This metric is used to measure the size and complexity of a software project. It is measured by counting the number of lines in text of the program's source code. LOC can be used to predict the amount of effort that would be required to develop a program, as well as estimate programming

productivity or maintainability once the software is developed.
Result:1..5, 5-high amount,1-low amount.

6. **Semantic model**.
Semantic model is designed to capture more of the meaning of an application environment than is possible with contemporary database models. An SM specification describes a database in terms of the kinds of entities that exist in the application environment, the classifications and groupings of those entities, and the structural interconnections among them. SM provides a collection of high-level modeling primitives[2] to capture the semantics of an application environment. Does the language include convenient semantic model?
Binary result: yes/no.

In order to compare Eolang efficiency to C++ and Java and detect the differences, we select a number of algorithms to implement in all three languages. The following algorithms were chosen:

1. Prim's algorithm
2. Dijkstra's algorithm
3. Kruskal's algorithm
4. Ford-fulkerson's algorithm

The justification for choosing these specific algorithms is that graph algorithms are usually complex, contain loops and recursive calls, and therefore, you can test them and compare,via different criteria.

In subsections: Testing results, it is needed to designate the definitions:

1. vNum-the number of edges
2. Time-the time of program execution
3. Memory-the total memory used while program execution

2.2 Prim's algorithm

Prim's algorithm is an algorithm for constructing the minimum spanning tree of a connected weighted undirected graph.

The input of the algorithm is a connected weighted undirected graph, which is represented by a sequence of integers separated by spaces. Each group of 3 numbers describes an arc of the graph: the first two numbers are adjacent vertices, the third is the weight of the edge.

Ex. 0 2 4 2 4 1 4 1 5 1 3 2 3 0 7

First, a random vertex is selected. The edge of the minimum weight incident to the selected vertex is searched for. This is the first edge of the spanning tree. Next, an edge of the graph of the minimum weight is added to the tree in which only one of the vertices belongs to the tree. When the number of tree edges is equal to the number of nodes in the graph minus one (the number of nodes in the graph and the tree are equal), the algorithm ends.

The result of the work of programs implementing the algorithm is a sequence of edges of the minimal spanning tree: Ex. (2 4 - 1) (0 2 - 4) (4 1 - 5) (1 3 - 2)

Time Complexity: O((V + E) log V)

2.2.1 Testing results:

See Fig. 1. for test results of [15]

2.3 Dijkstra's algorithm

Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a SPT (shortest path tree) with a given source as a root. We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source [3]. Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm

1. Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3. While sptSet doesn't include all vertices

```
========== Prim's algorithm ===========
| C++:                                 |
----------------------------------------
| vNum | Time       | Memory(Kbyte)   |
----------------------------------------
| 10   | 0:00.00    |            3256 |
----------------------------------------
| 30   | 0:00.00    |            3312 |
----------------------------------------
| 50   | 0:00.01    |            3356 |
----------------------------------------
| Java:                                |
----------------------------------------
| vNum | Time       | Memory(Kbyte)   |
----------------------------------------
| 10   | 0:00.05    |           36176 |
----------------------------------------
| 30   | 0:00.12    |           36516 |
----------------------------------------
| 50   | 0:00.09    |           37848 |
----------------------------------------
| EO:                                  |
----------------------------------------
| vNum | Time       | Memory(Kbyte)   |
----------------------------------------
| 10   | 0:00.33    |          121636 |
----------------------------------------
| 30   | 0:03.55    |          601572 |
----------------------------------------
| 50   | 0:20.52    |          812628 |
----------------------------------------
```

Figure 1. Prim's algorithm test results

a) Pick a vertex u which is not there in sptSet and has a minimum distance value.
b) Include u to sptSet.
c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Time Complexity: O(E log V)

2.3.1 Testing results: See Fig. 2. for the test result of [11]

2.4 Kruskal's algorithm

The Kruskal's algorithm is used to find the minimum spanning tree of a graph [6]. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree. A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

To find the MST using the Kruskal's algorithm:

1. Sort all the edges in non-decreasing order of their weight.

```
======== Dijkstra's algorithm =========
| C++:                                 |
----------------------------------------
| vNum | Time       | Memory(Kbyte)    |
----------------------------------------
| 10   | 0:00.00    |            3296  |
----------------------------------------
| 30   | 0:00.00    |            3368  |
----------------------------------------
| 50   | 0:00.00    |            3364  |
----------------------------------------
| Java:                                |
----------------------------------------
| vNum | Time       | Memory(Kbyte)    |
----------------------------------------
| 10   | 0:00.09    |           37592  |
----------------------------------------
| 30   | 0:00.09    |           35860  |
----------------------------------------
| 50   | 0:00.13    |           36788  |
----------------------------------------
| EO:                                  |
----------------------------------------
| vNum | Time       | Memory(Kbyte)    |
----------------------------------------
| 10   | 0:00.15    |           51972  |
----------------------------------------
| 30   | 0:00.43    |          112568  |
----------------------------------------
| 50   | 0:00.48    |          127216  |
----------------------------------------
```

Figure 2. Dijkstra's algorithm Test results

2. (Union-Find algorithm) Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step no. 2 until there are (V-1) edges in the spanning tree.

Time Complexity: O(E log V)

2.4.1 Testing results: See Fig. 3. for test of results of [14]

2.5 Ford-Falkerson Algorithm

Given a graph which represents a flow network where every edge has a capacity. Also given two vertices source 's' and sink 't' in the graph, find the maximum possible flow from s to t with following constraints: a) Flow on an edge does not exceed the given capacity of the edge. b) Incoming flow is equal to outgoing flow for every vertex except s and t [5].

2.5.1 Implementation. Let us first define the concept of Residual Graph which is needed for understanding the implementation.

Residual Graph of a flow network is a graph which indicates additional possible flow. If there is a path from source to sink in residual graph, then it is possible to add flow. Every edge of a residual graph has a value called residual capacity

```
========= Kruskal's algorithm =========
| C++:                                |
---------------------------------------
| vNum | Time       | Memory(Kbyte)   |
---------------------------------------
| 10   | 0:00.00    |            3344 |
---------------------------------------
| 30   | 0:00.00    |            3288 |
---------------------------------------
| 50   | 0:00.00    |            3356 |
---------------------------------------
| Java:                               |
---------------------------------------
| vNum | Time       | Memory(Kbyte)   |
---------------------------------------
| 10   | 0:00.09    |           39576 |
---------------------------------------
| 30   | 0:00.19    |           39700 |
---------------------------------------
| 50   | 0:00.18    |           39900 |
---------------------------------------
| EO:                                 |
---------------------------------------
| vNum | Time       | Memory(Kbyte)   |
---------------------------------------
| 10   | 0:00.25    |           67488 |
---------------------------------------
| 30   | 0:01.02    |          266308 |
---------------------------------------
| 50   | 0:03.02    |          809048 |
---------------------------------------
```

Figure 3. Kruskal's algorithm test results)

which is equal to original capacity of the edge minus current flow. Residual capacity is basically the current capacity of the edge. Let us now talk about implementation details. Residual capacity is 0 if there is no edge between two vertices of residual graph. We can initialize the residual graph as original graph as there is no initial flow and initially residual capacity is equal to original capacity. To find an augmenting path, we can either do a BFS or DFS of the residual graph. We have used DFS in below implementation. Using DFS, we can find out if there is a path from source to sink.

The important thing is, we need to update residual capacities in the residual graph. We subtract path flow from all edges along the path and we add path flow along the reverse edges We need to add path flow along reverse edges because may later need to send flow in reverse direction.

2.5.2 Time Complexity: Time complexity of the above algorithm is O(max_flow E). We run a loop while there is an augmenting path. In worst case, we may add 1 unit flow in every iteration. Therefore the time complexity becomes O(max_flow E).

2.5.3 Testing results: See Fig. 4. for test of results of [13]

```
====== Ford-Fulkerson algorithm =======
| C++:                                |
---------------------------------------
| vNum | Time       | Memory(Kbyte)   |
---------------------------------------
| 10   | 0:00.00    |            3456 |
---------------------------------------
| 30   | 0:00.01    |            3372 |
---------------------------------------
| 50   | 0:00.00    |            3424 |
---------------------------------------
| Java:                               |
---------------------------------------
| vNum | Time       | Memory(Kbyte)   |
---------------------------------------
| 10   | 0:00.03    |           34280 |
---------------------------------------
| 30   | 0:00.04    |           34424 |
---------------------------------------
| 50   | 0:00.09    |           35740 |
---------------------------------------
| EO:                                 |
---------------------------------------
| vNum | Time       | Memory(Kbyte)   |
---------------------------------------
| 10   | 0:05.92    |         1319204 |
---------------------------------------
| 30   | 0:05.71    |          468860 |
---------------------------------------
| 50   | 1:57.33    |          469332 |
---------------------------------------
```

Figure 4. Ford Falkerson's algorithm test results

2.6 Comparison Metrics - *Algorithms*

Language	Metric	Grade
C++	Simplicity	yes
Java	Simplicity	yes
EO	Simplicity	no
C++	Readability	yes
Java	Readability	yes
EO	Readability	no
C++	Compilation speed	5(time:3 ms.)
Java	Compilation speed	4(time:1.18 s.)
EO	Compilation speed	1(time:30 s.)
C++	Memory usage	5(memory:30 mb.)
Java	Memory usage	3(memory:340 mb.)
EO	Memory usage	1(memory:2.96 gb.)
C++	LOC	5 (350 LOC)
Java	LOC	3 (500 LOC)
EO	LOC	1 (770 LOC)
C++	Semantic model	yes
Java	Semantic model	yes
EO	Semantic model	yes

3 SWOT Analysis

The criteria are equally important because they affect the development cost and effort required over the lifetime of the program, and also affect the usefulness and quality of the developed program. It is important to note that there are several points that are difficult to compare for several reasons, as far as various programming languages are concerned. One of the controversial points is that Eolang is generally positioned as a language intended for static code analysis, which, however, is not explicitly advertised and there is no special emphasis on this in the current project. However, this leads to the fact that initially, it is essentially a subject-oriented and not a universal language that does not allow effectively displaying not only the styles of writing programs but also having a much smaller set of expressive means for describing algorithms and data.

3.1 STRENGTHS

Here, in general, a fairly simple semantic model of the language can be noted, which is due to the initially laid down idea of forming an "elegant" programming style. Based on this, the program contains only objects, the semantics of which allows them to be used as actions (directives) that provide both a description of the functionality of algorithms and the structuring of data. This allows you to form a fairly compact **semantic model**, which is the strength of the language.

The solutions proposed in the language increase the reliability of the generated code, albeit often at the expense of efficiency. But for the main target of analysis and reliability improvement, this is not a significant factor.

When compared to C ++ and Java languages, it can be noted that in these languages there are many unreliable constructs for programming. The languages themselves have many redundant and overlapping constructs, which often do not allow generating unambiguous and reliable code.

Compactness, extensibility and openness can be used to describe the strength of Eolang.

3.2 WEAKNESSES

One of the weaknesses is that the limited capabilities of the language do not allow it to be used in many subject areas in comparison with C ++ and Java. That is, where high performance computing is required.

Another point to mention is **lack of tools** that provide support for parallel and distributed computing, which is currently used in one form or another in almost all modern programming languages.

And not enough attention is paid to the formation of the type system. Using a typeless solution can turn out to be unreliable in many situations, which, in turn, may lead to difficulties associated with static code analysis. In addition, in some cases, in order to increase control over data when writing programs in Eolang, additional constructions will have to be introduced to model data types and explicitly check them either during static analysis or at runtime.

For C++ and Java, static typing is used to control the data at compile time. In addition, these languages support dynamic typing due to Object-Oriented polymorphism and the possibility of dynamic type checking at runtime.

Currently, the EO community is limited. There are not many conferences, local meetups, forums, Facebook groups, open source projects based on the language and people willing to help.

Also, as mentioned in section 2.6, Eolang lacks **simplicity**. The EO language is barely simple to read and write and has a quite steep learning curve for new programmers. In comparison to the other languages in context, corresponding code in Eolang takes consumes more lines **(LOC)**. The comparison made in section 2.6 shows that Eolang has minimal efficiency in terms of **memory consumption** and **compilation speed**. Thus, Eolang consumes a lot of resources and increasing becomes complex as the code base gets large. These add to the weaknesses of the language.

Lastly, conceptual incompleteness. The Eolang concept or idea is not fully or completely described.

3.3 OPPORTUNITIES

Most likely it is prudent to start with the possibilities, since they determine the specifics of the language. The limitations by means of the OO paradigm and recursive computations based on the absence of object mutability possibly simplifies the code for static analysis, but at the same time significantly reduces the number of effective techniques used in real programming. When creating algorithms, you often have to write longer and more inefficient code, which is difficult to further optimize when reduced to a real executor. At the same time, as the practice of using functional programming languages shows, the use of similar techniques increases the reliability of programs and ensures the formation of controlled code.

When comparing with C ++ and Java, it is enough to note here that both proposed languages are universal and include tools for writing programs that allows one to choose between reliable and efficient programming. In principle, it is possible to list these tools, emphasizing what is not in Eolang, emphasizing that this significantly expands the possibilities of programming, but often at the expense of the reliability of the code.

Also the platform independence of Eolang provides an opportunity to potentially interoperate the language with many other languages and use existing libraries.

Additionally, there is potential **formalizability of semantics**, presence of formal calculus of objects, convergence of object and functional paradigms, potentially short and reliable code.

3.4 THREATS

Among the main threats, we could considered the use of the language not for its main purpose, which can lead to writing code that will be less expressive than the code written in C ++ and Java. At the same time, attempts to model the constructs of these languages in Eolang can lead to more cumbersome and less reliable code. In particular, explicit modeling of data types will require validation at runtime, or may lead to the development of additional analysis programs to isolate and match data types during the static analysis phase.

It may also be worth noting the problem associated with the lack of type control when entering data, when the incoming data in the presence of a **typeless** language model will be difficult to control. There are similar threats in other languages, but the presence of a static type system or explicit dynamic typing allows the ability to control the input and transformation of data directly using language constructs without additional modeling. Also the typless nature of the language drives it more in the direction of functional approach [9] rather than OOP, as intended. It would be threatening to use Eolang as a system programming language since it lacks strong typing, compared to many system programming languages which rather are strongly typed to help manage complexity [8].

4 Continuous Integration Testing on GitHub

As one of results of this stage, we propose a semi-autonomous testing solution integrated into the GitHub Actions. Besides, the testing solution is based on YAML configurations, Python , and JSON. YAML is used to configure GitHub Actions Pipelines. Python is the main language used to implement the semi-autonomous testing framework. JSON is used to configure tests to run them with the proposed testing framework.

Fig. 5. shows the idea behind the proposed testing framework. The framework takes the following steps during its work:

1. GitHub Actions pipelines are triggered and activated. There are two pipelines, one for each tested compilers (CQFN and HSE).
2. The pipeline runs python/retrieve-tests-matrix.py to build the tests matrix. Pipeline matrices allow us to dynamically configure the testing grid in GitHub actions. This means that one YAML configuration file can produce arbitrary number of jobs at runtime. To build the tests matrix, the framework recursively traverses the eo-tests directory and collects test.json files. Each test.json contains the following metadata: a name, a description, a type, an expected result, an activity flag. After the tests metadata is collected, GitHub Actions initialize one job per each test.json.
3. Inside each test's job, GitHub Actions runs python/run-test.py to run the corresponding test.
4. There are three types of tests in total: compilation tests, runtime tests, and compilation time tests. To run tests, four environments are prepared in /environments, one per each test type and compiler (HSE or CQFN). The testing framework copies program files into the corresponding environment and performs testing by running the environment and comparing the expected result to the actual result.

In addition to the test framework, we prepared seventy tests that check different aspect of the language, including essential operations of abstraction, application, decoration and dataization. The HSE compiler passed 44 out of 70 tests, while the CQFN compiler passed only 15. The HSE compiler failed at the tests that check language features not yet supported by the compiler (these are partial application, named attribute binding, exclamation mark syntax, decoratee and parent reference in application syntax). The CQFN tests failed because of incompatibility of the tests with the compiler and the testing system itself, mostly. To conclude, we proposed the semi-autonomous testing solution that allows EO maintainers to add tests written in EO (both compilation-time and run-time tests) in a flexible manner through incorporation of JSON configuration files. In addition to the framework, we prepared 70 test samples. Testing showed that CQFN and HSE compilers differ in their supported language and runtime features. In future, we plan to synchronize code bases of compilers and address issues found through the testing.The proposed framework is delivered as a GitHub repository[10].

5 Conclusion

Time and memory efficiency tests show the huge differences between EO and C++/Java.

The ratio in Memory usage between EO and C++, is 200.This means that EO programs used 200 times more memory than C++ analogues.

The several Memory ratio between EO and Java, is 20.It is 10 times less than EO/C++ ratio but still, not efficient.

The Time ratio between EO and C++, is 180.This means that EO programs needs 180 times more time to execute than C++ analogues.

The several time ratio between EO and Java, is 10.It is 18 times less than EO/C++ ratio.

It can be estimated that the more complex structure or algorithm is implemented in EO, the less Runtime efficiency is. Therefore, the higher the complexity of the algorithm is, the more complex the structure of EO code gets, and the more difficult it becomes to read. Thus, the resulting code loses readability. This further leads to debugging problems,

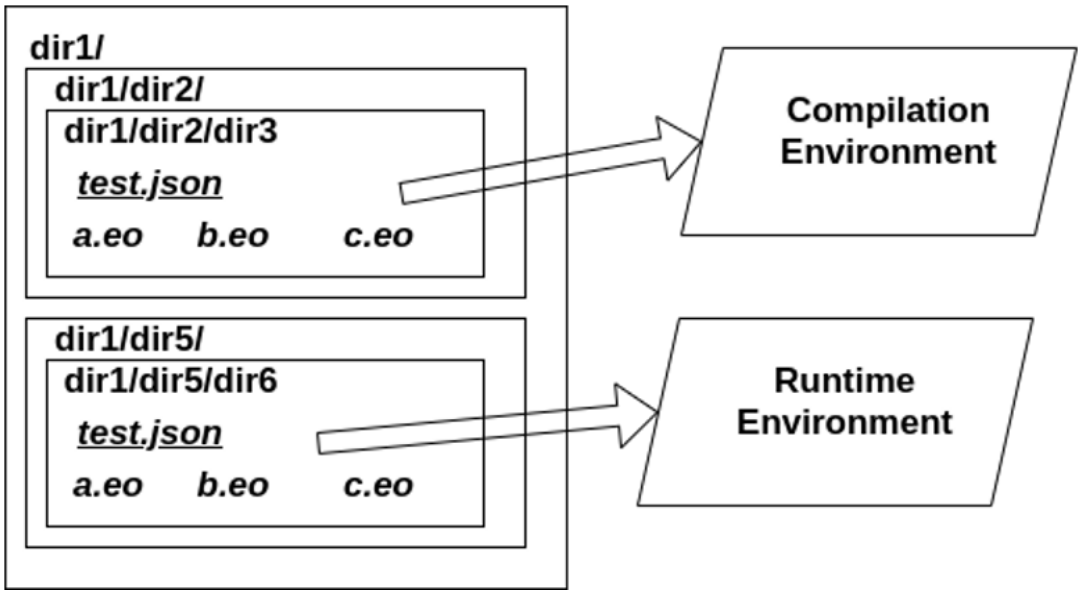

Figure 5. CI model

as it becomes more difficult to track and identify bugs or problems in a large amount of EO code. Code is available at [12]

References

[1] [n.d.]. Comparing Programming Languages. https://www.cs.ucf.edu/~leavens/ComS541Fall98/hw-pages/comparing/

[2] Yaofei Chen and Rose Dios. 2006. An Empirical Study of Programming Language Trends. *IEEE Xplore* 8 (2006). https://doi.org/10.29322/IJSRP.8.12.2018.p8441

[3] G. Deepa, Priyank Kumar, A. Manimaran, K. Rajakumar, and V. Krishnamoorthy. 2018. Dijkstra Algorithm Application: Shortest Distance between Buildings. *International Journal of Engineering & Technology* 7, 4.10 (Oct. 2018), 974–976. https://doi.org/10.14419/ijet.v7i4.10.26638 Number: 4.10.

[4] Mark Kraeling. 2013. Chapter 7 - Embedded Software Programming and Implementation Guidelines. In *Software Engineering for Embedded Systems*, Robert Oshana and Mark Kraeling (Eds.). Newnes, Oxford, 183–204. https://doi.org/10.1016/B978-0-12-415917-4.00007-4

[5] Myint Than Kyi and Lin Lin Naing. 2018. Application of Ford-Fulkerson Algorithm to Maximum Flow in Water Distribution Pipeline Network. *International Journal of Scientific and Research Publications (IJSRP)* 8, 12 (Dec. 2018). https://doi.org/10.29322/IJSRP.8.12.2018.p8441

[6] Haiming Li, Qiyang Xia, and Yong Wang. 2017. Research and Improvement of Kruskal Algorithm. *Journal of Computer and Communications* 5, 12 (Sept. 2017), 63–69. https://doi.org/10.4236/jcc.2017.512007 Number: 12 Publisher: Scientific Research Publishing.

[7] Mitendra Mahto On. [n.d.]. Code simplicity: A language independent perspective | Hacker Noon. https://hackernoon.com/code-simplicity-a-language-independent-perspective-756cf031c913

[8] J.K. Ousterhout. 1998. Scripting: higher level programming for the 21st Century. *Computer* 31, 3 (1998), 23–30. https://doi.org/10.1109/2.660187

[9] John C. Reynolds. 1970. GEDANKEN—a Simple Typeless Language Based on the Principle of Completeness and the Reference Concept. *Commun. ACM* 13, 5 (May 1970), 308–319. https://doi.org/10.1145/362349.362364

[10] HSE Team. 2021. CI/CD. Retrieved August 28, 2021 from https://github.com/HSE-Eolang/hse-eo-tests.

[11] HSE Team. 2021. Dijkstra. Retrieved August 28, 2021 from https://github.com/HSE-Eolang/eo_graphs/tree/master/src/main/eo/dijkstra

[12] HSE Team. 2021. Eo Graphs. Retrieved August 28, 2021 from https://github.com/HSE-Eolang/eo_graphs

[13] HSE Team. 2021. Ford Falkerson. Retrieved August 28, 2021 from https://github.com/HSE-Eolang/eo_graphs/tree/master/src/main/eo/fordfalkerson

[14] HSE Team. 2021. Kruskal. Retrieved August 28, 2021 from https://github.com/HSE-Eolang/eo_graphs/blob/master/src/main/eo/kruskal.eo

[15] HSE Team. 2021. Prim. Retrieved August 28, 2021 from https://github.com/HSE-Eolang/eo_graphs/blob/master/src/main/eo/prim.eo