Towards General Evaluation of Intelligent Systems: Using Semantic Analysis to Improve Environments in the AIQ Test

Ondřej Vadinský(✉)

Department of Information and Knowledge Engineering,
University of Economics, Prague, Czech Republic
ondrej.vadinsky@vse.cz

Abstract. This paper conducted a semantic analysis of environment programs that are used in the Algorithmic Intelligence Quotient test to evaluate the intelligence of agents. The analysis identified several classes of programs that are non-discriminative or contain pointless code adversely affecting the testing process. Extensions of the test were implemented and verified to reduce the proportion of problematic programs thus increasing the suitability of the Algorithmic Intelligence Quotient test as a general artificial intelligence evaluation method.

Keywords: Artificial general intelligence
Evaluating intelligence of artificial systems
Universal Intelligence definition · Algorithmic Intelligence Quotient test

1 Introduction

One of the cardinal questions of *artificial general intelligence* (AGI) [2] is "What is intelligence and how can it be evaluated in an artificial system?" Attempts to answer this question can be traced back to Turing [15], however, it was the work on *C-Test* [4] that first used the Algorithmic Information Theory in a test of intelligence, effectively founding a new area of research, one focused on a *universal evaluation of intelligence* [6].

The question at hand is not only a call for a test, it is also a call for a definition that could serve as a formal foundation for the test. One such definition is the *Universal Intelligence* of Legg and Hutter [11]. Based on a study of a broad variety of definitions, theories, and tests of human, animal, and artificial intelligence given in [10], Legg and Hutter derived the following informal version: "Intelligence measures an agent's ability to achieve goals in a wide range of environments" [11]. They also give its formalization as shown by Eq. 1.

$$\Upsilon(\pi) := \sum_{\mu \in E} 2^{-K(\mu)} V_\mu^\pi, \quad \text{where} \qquad V_\mu^\pi := \mathbb{E}\left(\sum_{i=1}^{\infty} r_i\right) \leq 1 \tag{1}$$

M. Iklé et al. (Eds.): AGI 2018, LNAI 10999, pp. 248–258, 2018.
https://doi.org/10.1007/978-3-319-97676-1_24

The Universal Intelligence Υ of agent π is given by its ability to achieve goals as defined by a value function V_μ^π as maximizing the expected sum of all future rewards (given a history of interactions) over a set E of environments μ weighted by algorithmic probability that uses Kolmogorov complexity K [11].

While the Universal Intelligence definition has several desirable properties, it is not computable [11], and as noted e.g. in [1,5,7,8] it has several other limitations. The *Anytime Intelligence Test* proposal of Hernández-Orallo and Dowe [7] also discusses several aspects that must be considered when converting the uncomputable definition into a practicable intelligence test.

The existing implementation of the Anytime Intelligence test [9] remains rather limited. A more powerful intelligence test called *Algorithmic Intelligence Quotient* (AIQ test) was introduced by Legg and Veness [13]. The AIQ test, as given in Eq. 2, is a computable approximation of Universal Intelligence.

$$\hat{\Upsilon}(\pi) := \frac{1}{N}\sum_{i=1}^{N} \hat{V}_{p_i}^{\pi}, \quad \text{where} \qquad \hat{V}_{p_i}^{\pi} := \frac{1}{k}\sum_{i=1}^{k} r_i \tag{2}$$

The AIQ $\hat{\Upsilon}$ of agent π is given by its ability to achieve goals as defined by an empirical value function $\hat{V}_{p_i}^{\pi}$ as total reward from a single trial of an environment program p_i averaged over a finite sample of N programs sampled according to Solomonoff's Universal Distribution: $M_{\mathcal{U}}(x) := \sum_{p:\mathcal{U}(p)=x*} 2^{-l(p)}$ [13].

The environment programs of the AIQ test are Turing-complete programs, built using a modified *BF language* [14], which compute the current reward and observation from the interaction sequence with the agent. The modified BF language (also referred to as a reference machine) uses 10 instructions [12,13]:

- `+-` increment/decrement respectively the symbol on the working tape,
- `,.` read the agent's action from an input tape and write to the current cell of the work tape/write the current cell of the work tape as a reward (the 1st write) or observation (the remaining writes) to the output tape respectively, and move the respective input or output tape pointer to the right,
- `<>` move the work tape pointer to the left or right,
- `[]` start a loop if the current work cell is non-zero/end the loop respectively,
- `%` write a random symbol to the current work cell,
- `#` end program.

There are only a few limits imposed on the environment programs: 1. the computation of each interaction is limited to 1,000 steps. 2. programs are halted if they try to write more than the set number of reward and observation symbols. 3. read and write instructions are mandatory, reducing the proportion of non-interactive (*passive*) programs. Such environment programs are called *non-discriminative* by [7] since they do not meaningfully contribute to the agent's evaluation. Therefore, any testing effort using such programs is wasted.

This paper, following the suggestion of [16], has two goals: First, an analysis of the environment programs used by the AIQ test will be conducted in Sect. 2 in order to determine the exact extent of the problem of non-discriminative

programs, as well as to identify other potential issues with the programs. Second, in Sect. 3, an attempt will be made to improve the BF program sampler so that the proportion of problematic programs is reduced. The paper will be concluded in Sect. 4 together with the discussion of future work.

2 Semantic Analysis of Environment Programs

Research questions will be stated in Sect. 2.1. Section 2.2 will describe the proposed method called semantic analysis of environment programs. A summary of its results will be given in Sect. 2.3. Section 2.4 will discuss the results briefly.

2.1 Research Questions

As [16] argues, a closer look at at the environment programs used in the AIQ test could allow improved interpretation of the test results, as well as answer some of the concerns raised in [7]. The following questions will be investigated:

- How does chance influence an agent's rewards and observations?
- How do the actions of an agent influence its rewards and observations?
- What are the forms of code that can be considered pointless?

2.2 Method Overview

A method was proposed by [16] that "consists of identifying the semantics of an environment program class and describing its possible syntax in BF language using regular expressions." This section will elaborate on the method so that all the required steps are clear and sufficiently developed.

Semantic Classes. The first step of the method is to specify the *semantic class* in question as a set of environment programs with given semantics. The semantics can be specified rather informally since its formalization will be arrived at in the following steps. An example is: *The agent's reward is always random.*

Syntactic Classes. The second step of the method is to derive one or more *syntactic classes* from the specified semantic class. The syntactic class is a rather formal expression in generalized BF language containing both specific fragments of BF syntax that are required, as well as possibly optional variables for fragments of BF syntax that are to meet given conditions.

An example of one of the syntactic classes for the previous semantic class is the expression `a%p.z#`. Conditions for the variables `apz` should be sufficiently formal that they can be easily converted into regular expressions. For example, fragment `p` can only contain instructions `+-` and can be of a zero length.

Regular Expressions. The third step of the method is to convert the specific syntactic class to one or more *regular expressions*.[1] For the demonstration in this paper, Pearl Compatible Regular Expressions (PCRE) [3] are used as implemented in GNU Grep. An example of one possible regular expression for the (fully specified) previous syntactic class is `^[^\[\.]*%[\+\-]*\..*#`. One drawback of combining BF language with PCRE is that many BF instructions are also meta-characters of PCRE, increasing the need for escaping.

Limits of the Method. The introduced method of semantic analysis is necessarily incomplete, and, depending on the level of detail in the expressions used, also inaccurate. These limits are mainly due to the fact that:

- There are many possible syntactic means to describe any given semantics, making it hard to identify all the syntactic classes for a given semantic class.
- There are syntactic limits of regular expressions that do not always allow for the exact capture of all the conditions of syntactic class variables.

Therefore, the results should be treated as estimates in form: "For at least about x% of environment programs it is likely the case that..." Due to the possibility of nesting, worse estimates are more likely wherever loops are concerned.

Despite the above-stated limits and given adequate effort, the method can be considered to be sufficiently complete with respect to the environment programs sample, as well as sufficiently accurate with regard to the research questions.

2.3 Results

Semantic analysis gives two kinds of results. The first is a detailed specification of semantic and syntactic classes. The second is an estimated proportion of the classes in the BF programs sample. See the Appendix for the full results.

A sample of 200,000 environment programs for the BF reference machine with 5 action symbols (BF 5) generated by the original AIQ test [12] is described in the summary. Other settings were analyzed leading to similar results.

Role of Chance. About 76% of programs in the sample contain the instruction `%` that was added by [13] to the original BF language to enable indeterminism. The following classes are of a special concern for an agent's evaluation:

1. *The agent's reward is always random* as described e.g. by `a%p.z#`, where `a` cannot lead to premature termination, nor can it contain loops that are not closed, nor can it contain the write instruction. Fragment `p` can only contain instructions `+-` and can be of zero length. Fragment `z` can contain any instruction. A simple example is `%.,#`. Such programs are non-discriminative according to [7]. The proportion of this semantic class is about 17%.

[1] If the class is precisely specified, one regular expression should suffice. Increasing the number of regular expressions can, however, improve readability in some cases.

2. *The agent's observation is always random* has a proportion of 5%. This class can be confusing for the agent since it has to ignore unrelated observations.
3. *The agent's reward is almost always random* as given by, e.g. `a%p[q.z]y#`, where `q` can contain instructions `+-[%` and may be of zero length. Fragment `y` must contain at least one write instruction that is guaranteed to be executed. A simple example is `%[.],+.#`. Such environments test the ability of an agent to learn the activity described by `y` even with noisy feedback. This class, however, hinders evaluation since it limits the total achievable reward.
4. *Certain actions lead to random reward* as described e.g. by `a,p[p%p.z]y#`. A simple example is `,+[%.].#`. This class is similar to the previous, however it is the agent that controls the noise in the feedback.

As illustrated above, chance can play different roles in environment programs. The more interesting classes 2, 3, and 4 are, however, rather rare in the samples.

Role of Agent's Action. All environment programs have to contain an instruction that reads an agent's action [12]. There are, however, about 9% of environment programs in which the instruction is only part of pointless code. These cases most likely coincide with the class *agent's reward is always random*.

In the case of a meaningfully processed instruction, there are different roles it can play in the environment:

1. *The agent's reward is always trivially dependent on its action* (like `.,#` or `,+.#`). This occurs in about 34% of cases.
2. *The agent's observation is always trivially dependent on its action* (like `,+.,.#`). This occurs in about 8% of cases.
3. *The agent's reward can be sometimes trivially dependent on its action* (like `+[,.>]%.<#`). This occurs in about 11% of cases.
4. *Certain action activates a certain process* (like `,[>++<[+]]>.#`). This occurs in at most about 50% of cases. These programs can test complex behaviour.

Never-Ending Loops. Some cases of never-ending loops are removed by [12], however, since they do not consider multiple loop levels and more complex forms of syntax, about 2% of programs still contain some form of never-ending loops.

Premature Termination. In order to avoid the halting problem, step and write limits were implemented that can terminate programs prematurely [12]. Semantic analysis can detect some cases when the write limit is exceeded, i.e. the program tries to write more then a set number of reward and observation symbols. 9% of programs are guaranteed to exceed this limit, and a further 22% allow for the possibility of triggering it. These percentages decrease with the increase of the write limit on reference machines with a higher number of observations.

Pointless Code. Some of the randomly generated code of environment programs is necessarily pointless, complicating its analysis and giving it a false

sense of complexity. Only some of the very basic forms of pointless code are removed by [12]. There seem to be two main types of classes:

1. *Part of the program is not executed* as specified by `an#` or `anz#` where `n` is the non-executed code effectively reducing the programs to `a#` or `az#`. This class can be further divided into:
   (a) *Programs with never executed loops* as described e.g. by `a][n]z#`, where `[n]` is pointless. This class has a proportion of about 13%.
   (b) *Programs that are always prematurely terminated due to a write limit* that makes the remaining code pointless. The proportion is about 9%.
2. *Part of the executed program is canceled out by some other part* as typically described by `aprz#` where `p` is made pointless by `r`, reducing the program to `arz#`. Several more specific sub-classes can be identified:
   (a) *Pointless modifications of chance* as described by `a%pz#` where `p` contains a non-zero length combination of `+-`. This class has a proportion of about 36%, and the programs can be reduced to `a%z#`.
   (b) *Code overwritten by action-read or chance* as given by `aq%z#` or `aq,z#`, where `q` contains a non-zero length combination of `+-,%`. An example of such a program is `,.,+%#`. This class has a proportion of about 74%, and the programs can be reduced to `a%z#` or `a,z#` (if the number of action-reads from the original program is kept).
   (c) *Zeroing overwritten by action-read or chance* as described by `a[p]%z#` or `a[p],z#`, where `p` contains a non-zero length combination of `+-%`. With a proportion of about 5% the programs can be reduced to `a%z#` or `a,z#`.
   (d) *Zeroing of chance or action-read* as described by `a%[q]z#` or `a,[q]z#`, where `q` contains a non-zero length combination of `+-,%`. An example of such a program is `,+.,[,+]>%.#`. This class has a proportion of 6%, and the programs can be reduced to `a[+]z#` (in cases of the overwritten action-read, there may be no action-read in `z`).

2.4 Discussion

While necessarily incomplete and only an estimation, it was shown practically that semantic analysis of environment programs produces interesting results. The underlying cause of the identified problems within AIQ test environments seem to lie in the fact that the programs are randomly sampled, thus frequently resulting in pointless code, simple programs, and even non-discriminative programs.

As proposed by [7], a switch to a more suitable reference machine may solve the identified problems. However, as argued by [16], it is also possible to try to reduce the proportion of the problematic programs, and the conducted semantic analysis actually gives the necessary information to do this. Since the resulting program sample can be reused in many tests of many agents, it is worth to invest the effort in making a good sampling procedure to gain efficiency for evaluation.

Applying the results of semantic analysis in an effort to improve the BF sampler will not always be straightforward, since the analysis was designed with *class proportion estimation* in mind. Thus, e.g. all *programs with never executed*

loops can be easily identified by having `][` fragment, however, it is not that simple to match all the possible code of the actual never executed loop with a regular expression. This may not be a problem in case the problematic programs are simply dropped, however, such an approach may significantly prolong the sampling, thus the severity of the identified problem should be also considered.

3 Improving Environment Programs of the AIQ Test

Section 3.1 will introduce an improvement to the AIQ test aimed at reducing the abundance of pointless code in its environments. The improvement that decreases the proportion of non-discriminative programs will be described in Sect. 3.2. See the Appendix for the sources of the extended AIQ test.

3.1 Removing Pointless Code

Since pointless code obfuscates environment programs, its removal should be attempted first as it will facilitate improving the discriminative power.

Implementation. First, the program optimization was changed from one-time code replacement in the original test to a repetitive replacing procedure, which enables multiple optimizations to take place. This behaviour (*SEP-orig*) keeps the original replace patterns, and was made the new default for the BF sampler.

Furthermore, additional replace patterns were added based on the regular expressions that resulted from the semantic analysis. Due to the limits of code replacement and regular expressions, only class 2 pointless code was addressed. Also, some of the conditions had to be made stricter than they had previously been in the case of class proportion estimation. Notably, action-reads were excluded from `q` in 2 (b) and 2 (d) classes, since the current approach cannot enforce the necessary conditions on the `z` fragment. This functionality (*SEP-ext*) can be enabled by `--improved_optimization` switch.

Evaluation. To validate the implemented function, new samples of 200,000 programs for BF 5 reference machine were generated using *SEP-orig* and *SEP-ext*, respectively. Practically no differences between the *SEP-orig* and the original sample were detected using descriptive statistics according to a program length as well as conducting semantic analysis. As for the *SEP-ext* sample, its programs are somewhat shorter than in the original sample. As expected, the proportion of all cases of class 2 pointless code decreased noticeably with 2 (a) decreasing to 2%, 2 (b) to 23%, 2 (c) to 0, and 2 (d) to 5%.

Since the implemented *SEP* methods are code-optimization methods, they can be considered valid if the returned results are comparable to the original test. Validation experiments with the new samples were conducted using the default settings as reported in [16]. These were compared with the results achieved on the original samples using the function also introduced in [16] that saves

intermediate results every 1,000 interactions (*EffEL*). Paired samples t-test did not show significant differences in case of *SEP-orig* when used in short episodes. However, at long episodes, weakly significant negligible differences were shown. Furthermore, in case of *SEP-ext*, strongly significant difference was discovered. For the episode length of 100,000 interactions the difference is 1.5 ± 0.3 between the average of the results in *SEP-ext* and in the *EffEL* experiment, $t(24) = 11.10; p = 6.2 \times 10^{-11}$. See the Appendix for the full validation results.

Discussion. The implemented functionality successfully reduces the proportion of chosen types of pointless code. According to the experimental validation, *SEP-orig* can be considered a valid code-optimization method, however, *SEP-ext* seems to actually change the "quality" of the environment programs used, possibly increasing their discriminativeness. Therefore, *SEP-ext* cannot be considered only a code-optimization method, and its results should not be directly compared to the original test. Nevertheless, its usage can be recommended.

To further reduce the proportion of the pointless code a different approach is needed. There remain the cases where PCRE can be used to identify the program as problematic, but not to select the code to be replaced. There also remain the cases where conditions in the non-replaced parts of the program have to be met.

3.2 Improving Discriminative Power

Now that the abundance of the pointless code is reduced, removal of non-discriminative programs can be attempted.

Implementation. For this improvement, a procedure that classifies sampled environment programs was extended to use the regular expressions resulting from the semantic analysis. Programs of the *agent's observation is always random* class are newly classified as passive which effectively excludes them from the final sample. Some of the conditions had to be made stricter than they were in the case of class proportion estimation. This functionality (*SDP*) can be enabled by `--improved_discriminativeness` switch. Combination with the *SEP-ext* improvement is highly advised, since the variants containing pointless code are not included among the added regular expressions.

Evaluation. To validate the implemented function, a new sample of 200,000 programs for BF 5 reference machine was generated using the *SDP* function in combination with *SEP-ext*. Programs from the new sample are somewhat longer than in the *SEP-ext* sample, yet not as long as in the original sample. As expected, the proportion of the *agent's reward is always random* class decreased to 4%. This was compensated mainly by the increase of proportion of the *agent's reward is always trivially dependent on its action* class to 41%, however, a slight increase was also registered for the more interesting *agent's reward can be sometimes trivially dependent on its action* class. Moreover, the proportion of 2 (b) pointless code class decreased further to 20%.

The implemented *SDP* function is not a code-optimization method, but it is designed to reduce the proportion of non-discriminative environments that artificially decrease the AIQ score of agents by returning random rewards only. Therefore, the method can be considered valid if it enables the agents to score higher AIQ than the original test, thus showing a more reasonable distribution of rewards. A validation experiment with the new sample was conducted using the default settings as reported in [16]. These were compared with the results achieved on the *SEP-ext* samples from the previous experiment. One-sided paired samples t-test did indeed show significant increase in average AIQ between the *SDP* and *SEP-ext* experiments. For an episode length of 100,000 interactions the increase is 8.7 with a confidence interval of $(8.5; \infty)$, $t(24) = 87.16; p = 7.7 \times 10^{-32}$ (shorter episodes resulted in the somewhat lower increases but with similar levels of significance.) See the Appendix for the full validation results.

Discussion. The implemented functionality successfully reduces the proportion of the chosen type of non-discriminative environments. According to the experimental validation, *SDP* can be considered valid. Since the method changes the "quality" of environments, its results should not be directly compared to the original test. However, as it also increases the representativeness of the AIQ score (by testing on a higher number of discriminative environments), the usage of the *SDP* method is highly recommended.

As for the remaining 4% of non-discriminative programs identified by the subsequent analysis, these may result from the way regular expressions are formulated when used to estimate the class proportion. These expressions (unlike those in *SDP* extension) may not precisely capture all the conditions of the semantic class. This remains a viable path for further investigation.

4 Conclusion and Future Work

This paper attempted to analyze the code of environment programs used in the AIQ test of [12,13]. The goal of the analysis was to determine the extent of the problem with non-discriminative environments first noticed in [7], as well as to identify other possible problems with the programs. To address this goal, a method suggested in [16] that is called *semantic analysis of environment programs* was used. The method was elaborated on in this paper, clearly specifying all the necessary steps. It was discovered that non-discriminative environments exist in considerable numbers as well as that rather simple programs occur frequently and some forms of pointless code are prevalent in the programs. These results suggest that *the random sampling of environment programs used by the AIQ test is not very efficient in producing meaningful environment programs.*

Based on the semantic analysis, these problems can be mitigated by using post-processing on sampled programs. *BF programs sampler of the AIQ test was extended so that it can optionally reduce the proportion of chosen types of pointless code as well as non-discriminative programs.* These extensions were successfully verified using followup semantic analysis as well as experimental

validation with the AIQ test in a default setting. The implemented extensions *SEP-ext* and *SDP* should, therefore, be used when testing agents with the AIQ test. The presented results show that *semantic analysis of environment programs is a useful method* even though it is necessarily incomplete and only estimatory in nature.

There are several areas for future work. Since program classes can be identified by the semantic analysis, it is possible to investigate the exact influence of each class on an agent's results. A class of environment programs was identified in which an agent's reward is almost always random. While such a class is discriminative, the achievable average accumulated reward is limited, negatively impacting the AIQ score. Ways of integrating such cases into the overall score should be searched. Some types of pointless code that require a different approach to solve were not addressed in the presented extensions of the BF programs sampler. When the identified problems are addressed, a second round of semantic analysis should be considered as some of the currently infrequent problems may become more prevalent.

Acknowledgements. Computational resources were kindly provided by the CESNET LM2015042 and the CERIT Scientific Cloud LM2015085, provided under the program "Projects of Large Research, Development, and Innovations Infrastructures".

Appendix

Full results of the conducted analyses and experiments are available from: https://github.com/xvado00/IATEP/archive/v1.0.zip.

A tool to conduct the semantic analysis is available from: https://github.com/xvado00/SemAnEP-tool/archive/v1.0.zip.

Full sources of the improved test extending the version presented in [16] are available from: https://github.com/xvado00/AIQ/archive/v1.2.zip.

References

1. Goertzel, B.: Toward a formal characterization of real-world general intelligence. In: Baum, E., Hutter, M., Kitzelmann, E. (eds.) AGI 2010, pp. 19–24. Atlantis Press, Paris (2010)
2. Goertzel, B.: Artificial general intelligence: concept, state of the art, and future prospects. J. Artif. Gen. Intell. **5**(1), 1–48 (2014)
3. Hazel, P.: PCRE - Perl compatible regular expressions (2015). http://pcre.org/
4. Hernandez-Orallo, J.: Beyond the Turing test. J. Log. Lang. Inf. **9**(4), 447–466 (2000)
5. Hernández-Orallo, J.: C-tests revisited: back and forth with complexity. In: Bieger, J., Goertzel, B., Potapov, A. (eds.) AGI 2015. LNCS (LNAI), vol. 9205, pp. 272–282. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21365-1_28
6. Hernández-Orallo, J.: The Measure of All Minds. Cambridge University Press, Cambridge (2017)

7. Hernández-Orallo, J., Dowe, D.L.: Measuring universal intelligence: towards an anytime intelligence test. Artif. Intell. **174**(18), 1508–1539 (2010)
8. Hibbard, B.: Bias and no free lunch in formal measures of intelligence. J. Artif. Gen. Intell. **1**(1), 54–61 (2009)
9. Insa-Cabrera, J., Dowe, D.L., España-Cubillo, S., Hernández-Lloreda, M.V., Hernández-Orallo, J.: Comparing humans and AI agents. In: Schmidhuber, J., Thórisson, K.R., Looks, M. (eds.) AGI 2011. LNCS (LNAI), vol. 6830, pp. 122–132. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22887-2_13
10. Legg, S., Hutter, M.: A collection of definitions of intelligence. In: Goertzel, B., Wang, P. (eds.) Advances in Artificial General Intelligence: Concepts, Architectures and Algorithms, FAIA, vol. 157, pp. 17–24. IOS Press, Amsterdam (2007)
11. Legg, S., Hutter, M.: Universal intelligence: a definition of machine intelligence. Minds Mach. **17**(4), 391–444 (2007)
12. Legg, S., Veness, J.: AIQ: Algorithmic intelligence quotient [source codes] (2011). https://github.com/mathemajician/AIQ
13. Legg, S., Veness, J.: An approximation of the universal intelligence measure. In: Dowe, D.L. (ed.) Algorithmic Probability and Friends. Bayesian Prediction and Artificial Intelligence. LNCS, vol. 7070, pp. 236–249. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-44958-1_18
14. Müller, U.: dev/lang/brainfuck-2.lha in aminet (1993). http://aminet.net/package.php?package=dev/lang/brainfuck-2.lha
15. Turing, A.M.: Computing machinery and intelligence. Mind **59**(236), 433–460 (1950)
16. Vadinský, O.: Towards general evaluation of intelligent systems: lessons learned from reproducing AIQ test results. J. Artif. Gen. Intell. **9**(1), 1–54 (2018)