Course-ID:	MA-INF 4306
Course:	Lab Development and Application of Data Mining and Learning Systems: Machine Learning
Term:	Summer 2021
Supervisor(s):	Dr. Daniel Trabold

Solving IQ-Number Tests

Luca Scharr

October 11, 2023

Abstract

IQ Tests try to estimate the problem solving skills of a human. One of the most prominent tests that is part of an IQ-Test is the IQ-Number Test, where a subject tries to complete sequences of numbers. While this task seems to be rather simple, it turns out to be computationally challenging to replicate the performance of a human. It even turns out that humans significantly outperform state of the art IQ-Number Test solvers, especially on short sequences. This is the reason why this lab revisits the existing approach of Siebers and Schmid, 2012 and tries to expand upon it. This lab report will show a new approach to IQ-Number Tests in the form of a search-structure and will provide it as a modular python package. Furthermore, this new approach excels at solving short sequences and it can handle multiple masked positions which distinguishes it from the work of Siebers and Schmid, 2012.

1 Introduction

In 2012 Siebers and Schmid published a program to solve IQ sequence tests in Glimm and Krüger, 2012. In "Semi-analytic Natural Number Series Induction", by Siebers and Schmid, 2012, they published a program for the induction of natural number series. They did this by assuming that each series a is from the following structure:

$$a_n = \begin{cases} c_n & \text{for } 0 \leq n < m \\ t_{(n-m) \mod r} & \text{for } n \geq m \end{cases} \tag{1}$$

Here a_n is the n-th position of the series a, c_n is one of the m constants $c_0 \dots c_{m-1}$ and $t_{(n-m) \mod r}$ is one of the r terms $t_0 \dots t_{r-1}$. A term is defined as either a constant number, the value of some predecessor a_{n-j}, the position number n of the current element or the value of some other series b at position $n+j$, where $0 < n \leq m$ and $j \geq -m$. In addition, if t_0 and t_1 are terms, then $t_2 = t_0 \bullet t_1$ is a term, if and only if $\bullet \in \{+, -, \cdot, :, \exp_x(y)\}$. This describes all terms. In addition, there is a condition which makes it possible to find a rule. There needs to be an example of the rule in the series to make it a candidate and at least another example to verify that the given rule is correct. This gives an upper bound on the number of constants m possible for each series, namely $\lfloor \frac{m}{3} \rfloor$.

Given this structure, it becomes apparent that finding the next element of a given series becomes a search-problem. That is: One needs to specify the simplest representation of a given series and generate the next element. In Siebers and Schmid, 2012 this is done as sketched by Figure 1.

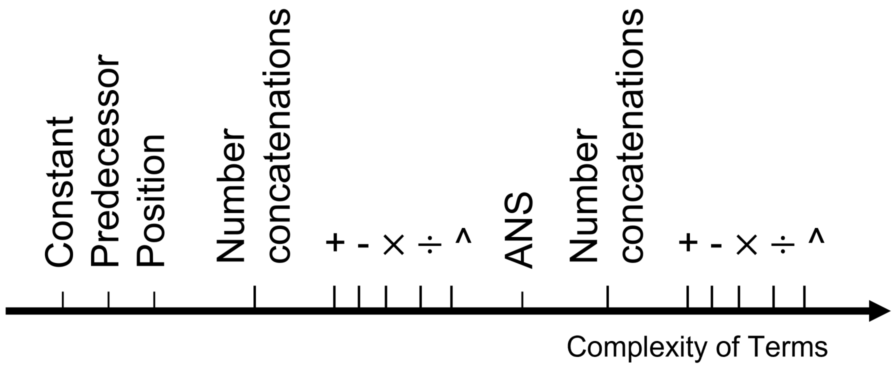


Figure 1: A sketch of the complexity as defined in Siebers and Schmid, 2012. ANS is short for auxiliary number series, which are the series b mentioned in the text. The complexity can be increased infinitely in this way.

In this search-structure one would like to performs breadth first search (BFS) to find a solution. However, this search-structure is infinitely big. Therefore it is impossible to traverse the whole structure. However, as it can be safely assumed that rules below a certain depth (as in Figure 1) wouldn't be found by a human, one can truncate this structure to a finite depth. This makes it possible to employ BFS.

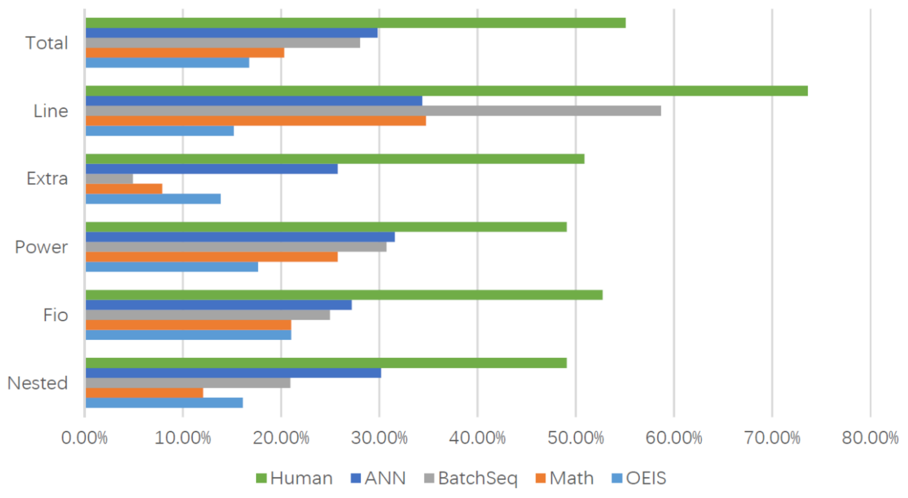


Figure 2: The performance of different IQ-Number Test solvers compared by series type Liu et al., 2019. Notice, that humans consistently outperform all other approaches.

In Liu et al., 2019 different IQ-Number Test solvers are compared. This can be seen in Figure 2. As humans still perform best on all disciplines, it is desirable to increase the performance of IQ-Number Test solvers. Furthermore, there are easy connections which are missed by the program published in Siebers and Schmid, 2012. For example, for this series:

$$9 \quad 16 \quad 25 \quad 36 \quad 49 \quad \_ \quad \rightarrow \quad x(0) = 9 \quad x(n) = (x(n-1)+5)+(n \cdot 2)$$

the expected rule would be $x(n) = (n+3)^2$, but a different rule is returned.

Both of these results together motivate the task of this lab, which consists of these subtasks:

1. Reimplement the program given in Siebers and Schmid, 2012
2. Improve their induction (e.g. for alternating or short series)
3. Add an option to control induced positions per series
4. If possible change the domain from $\mathbb{N}^{\mathbb{N}}$ to $\mathbb{R}^{\mathbb{N}}$

2 My Approach

As the approach in Siebers and Schmid, 2012 is optimised, it wouldn't be reasonable to try to generate better results with the same representation of a series. This and the second part of the task lead to the following questions: How are IQ-Number Tests generated? What defines a possible series, which can be given as a query to an algorithm or a human? Or, when assuming that each position in such series is generated by exactly one rule: What rules are possible?

By analysing the process how I solve IQ-Number Tests, I was able to extract these parts of my strategy:

1. Given a series, look for a rule between equidistant positions in the Series, i.e. find a rule for a sub-series (see (2) where $\sqrt{2}$ and π alternate).
2. Given a series, look for rules which are equidistant, i.e. find a rule for a sub-series of consecutive elements (see (3) where $+2$ and $\cdot 2$ alternate).
3. Search for a series in a special subset of $\mathbb{N}$, such as the prime numbers (see (4) where only the primes at even positions are given).

$$\sqrt{2} \quad \pi \quad \sqrt{2} \quad \pi \quad \sqrt{2} \quad \pi \quad \sqrt{2} \quad \pi \quad \_ \tag{2}$$

$$1 \quad 3 \quad 6 \quad 8 \quad 16 \quad 18 \quad 36 \quad 38 \quad \_ \tag{3}$$

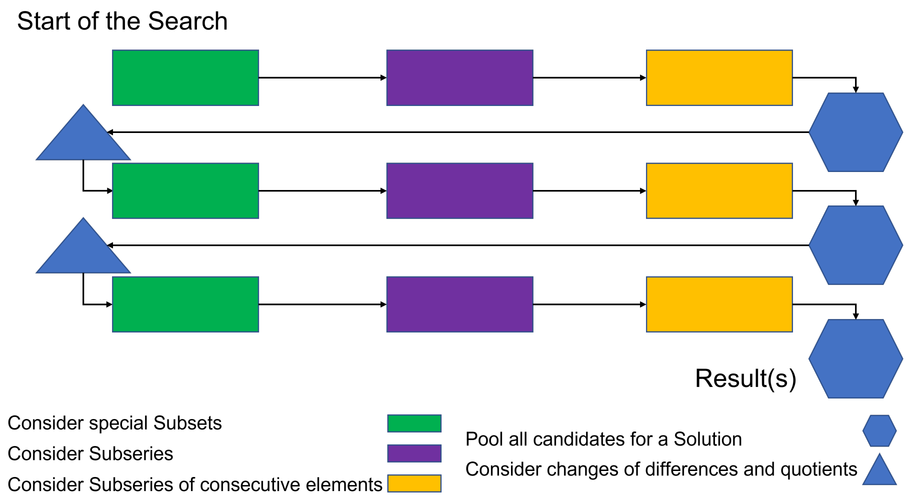
$$1 \quad 3 \quad 7 \quad 13 \quad \_ \tag{4}$$

Now all that remains is to find the set of rules which a human (or I, as there was no time for an extensive study) would usually find. A subset of these rules contains the following:

1. All elements are constant
2. The difference between two elements is constant
3. The quotient between two elements is constant
4. The series is generated by the Fibonacci rule
5. The series of differences or quotients changes according to the rules above
6. The series is a sub-sampling of a special subset of $\mathbb{N}$
7. The positions at which one sub-series changes according to the rules above.

While these rules make it possible to formulate a search-structure, it runs into another problem. The solution to a given query does not need to be unique. For example, the differences between two consecutive elements in series (4) form a simple series, namely $+2\ +4\ +6$. Therefore the next element could either be 19 or 21, depending on the rule which is assumed to generate the given series. Usually it is assumed that there is a solution that can be considered more simple than all other solutions. But even if this would be correct; there is no widely accepted definition for simplicity. Therefore the question arises what simplicity means. A question which, in my opinion, is not so easy to answer.

However, when reformulating the above observations into a search problem, it is possible to define a search-structure that can cope with different approaches to simplicity. This search-structure is sketched in Figure 3.



Start of the Search
Result(s)
Consider special Subsets
Consider Subseries
Consider Subseries of consecutive elements
Pool all candidates for a Solution
Consider changes of differences and quotients

Figure 3: The search-structure following from the set of observations named above. The search starts with the input of a masked series. Each rectangle performs a set of tests. Depending on the colour of the rectangle the tests concern different possible structures in a series. After traversing a layer, the search is terminated if at least one solution to the given series is found, this is indicated by the blue hexagons. If the search inside a layer was no success, then the next layer searches for a structure on the difference and the quotient between consecutive elements. This is indicated by the blue triangles.

Each depicted block performs a set of tests to discover a structure on a given series and supplies the following tests with the information discovered. With the selection and ordering of these tests one implicitly defines simplicity, which allows for the selection of the most appropriate definition of simplicity for a given task.

As the selection of the tests have a significant impact on the search-structure, it is possible to formulate them to fullfill the third and fourth task of this lab.

When considering a given series, I assumed that there exists a groundtruth. This makes it possible to specify the positions to be masked in the groundtruth and return a masked series along with a desired solution. This masked series is the series which is given to the algorithm along with the task to fill a set of positions. All of the above information can be stored inside of an object. This has the advantage that the algorithm can attach rules, which generate parts of the series, to the object. After traversing the search-structure, it is possible that the algorithm found multiple solutions to a query. How one proceeds with these solutions can be decided by the user.

3 Implementation

To make my implementation more accessible, I decided to write a modular Python package which represents the series as numpy arrays that contain floats. In my opinion, it is easy to change the search-structure and supplement further tests. However, as foreign code is not easy to read, the following sections aim to give an overview over the package and aim to answer the question which part of the package to alter for certain supplementations. Furthermore, each routine is commented with the expected input, the output and its purpose. The package can be found here: `https://github.com/luca-scharr/IQ-Number-Test-Solver`.

3.1 Data Class

The algorithm will work with objects of the class *data*, which can be found in *classes.py*. Each object has the following attributes:

1. *pos* stores the positions which still need to be filled. At the start, they decide which positions get masked and should be given as a numpy array containing the positions.
2. *truth* stores the groundtruth. Expects a numpy array containing floats.
3. *series* is the masked series generated from pos and truth.
4. *positive_tests* is a list of strings in which the algorithm will write the positions and the type of test that was positive at these positions.
5. As the search-structure generates different layers, it is necessary that one stores the previous objects from which the current object was generated. This is done in *ancestors* and *gender*.

3.2 Rulegeneration

As previously described, there is a set of rules which get checked (compare Figure 3). They generate the search-structure and ultimately define which solutions are reachable. The search-structure is defined in *searchstructure.py*. The search described in Figure 3 gets performed by *bfs*(...). *perf_layer*(...) performs each layer described in Figure 3. The generation of the next layer is specified in *gen_next*(...).

3.2.1 Elementwise Tests

The tests which search for special subsets of $\mathbb{N}$ are performed elementwise. They are specified in *ele_tests.py* and get performed by *perform_element_tests*(...). The set of elementwise tests consists of the prime test (*is_prime*(...)), the test for squares (*is_square*(...)) and the test for cubes (*is_cube*). The routine *fill_ele_test_space*(...) makes it possible to find dependencies which concern these special subsets of $\mathbb{N}$.

3.2.2 Tests on Subseries

Similar to the elementwise tests, the file which contains the tests regarding (sub)series is named *seq_tests.py*. They get performed by *perform_tests*(...). These tests are for constant elements (*const_test*(...)), a constant difference between elements (*sum_test*(...)), a constant factor between elements (*fac_test*(...)) and a test for Fibonacci's rule (*fib_test*(...)). If they are positive on a (sub)series the result gets filled in by *fill_tests*(...).

3.2.3 Tests between Consecutive Elements

Similar to the tests on (sub)series, the tests between consecutive elements can be found in the file *con_tests.py*. They get performed by the routine *perform_cons_tests*(...), which checks for constant consecutive elements (*const_cons_elem_test*(...)), a constant difference between consecutive elements (*sum_cons_elem_test*(...)), and a constant factor between consecutive elements (*fac_cons_elem_test*(...)). The result gets filled in by *fill_cons_tests*(...).

3.3 Retrieve the Result

The methods that retrieve the result from a final *data* object can be found in *gen_result.py*. As changes of representation while traversing the search-structure can occur, it is necessary to regain the representation in which the series started. This is done in the *back-prop_...*(...) methods, which themselves get combined in *find_root_of_family_tree*(...). This method fills the information gained in a lower layer of the search-structure into the ancestors from a higher layer. Once the top layer is reached, the method returns a solved *data* object.

4 Experiments

The following experiments are conducted without the use of the first block of the search-structure (Figure 3), as this block did not behave as expected.
In an other lab, Maximilian Kernbach, 2021b, tried to find IQ-Number Test Series to measure the performance of IQ-Number Test solvers. His test-sets (described in Figure 4) are used in my experiments to obtain a performance measurement on different sequence types and lengths.

Sequence type	**Construction rule**
Arithmetic	$a_i = a_{i-1} + c$
Geometric	$a_i = f_1 * a_{i-1}$
Arithmetic Geometric	$a_i = f_1 * a_{i-1} + c$
Arithmetic Geometric 2	$a_i = f_1 * a_{i-1} + f_2 * a_{i-2} + c$
Fibonacci	$a_1 = a,\ a_2 = b,\ a_i = fib(n, a_{i-1}, a_{i-2})$ or $a, b \in \mathbb{Z},\ a_i = a_{i-1} + fib(n, a, b)$
Power	$a, b, c \in \mathbb{Z},\ f(a) = (a+b)^2 + c$
Arithmetic Alternating	$a_{i=2k} = a_{i-2} + c_1$ $a_{i=2k-1} = a_{i-2} + c_2$
Arithmetic Irrelevant Number	$a_{i=2k} = a_{i-2} + c$ $a_{i=2k-1} = a_{i-2}$
Arithmetic Incremental	$a_i = a_{i-1} + i * c$

Figure 4: A listing of the sequence types and their construction rules, generated by Kernbach, 2021b.

Each test-set consists of 1.000 test series, each generated by the respective construction rule. Each test series has a unique solution when using the definition of complexity given by Kernbach, 2021b. This, and the fact that the construction rule of "Power" can be expressed by the construction rule of "Arithmetic", is the reason why the sequence type "Power" is not considered in the experiments. As the solver of Siebers and Schmid, 2012 was observed to perform better on longer series, two lengths of series are considered in this experiment.

The shortest series which can be expanded in a reasonable way is of length 3. This is the case, because each generating rule needs to have an example inside of the series to find it and another example to validate the rule. As rules describe the transition between two entries, the length of a shortest reasonable query is 3. However, the construction rule of "Arithmetic Alternating" and "Arithmetic Irrelevant Number" uses two sub-series to generate a query. This affects the shortest possible query and increases its length to 6.

This is why the first experiment considers series of length 6 and demands to find the 7th position. The percentage of correctly filled series achieved by different algorithms is shown in Table 1. Note, that my algorithm outperforms not only the algorithm of Siebers and Schmid, but also all the other considered algorithms.

Table 1: This table shows the precision of the considered algorithms for completing series of length 7 correctly (6 positions given, one to fill). The results are grouped by algorithm (named after the group designing it) and the sequence type contained in the test-set. My algorithm is shown as "Scharr". The results for the other algorithms are from the benchmark by Kernbach, 2021a.

Test-Set	**Siebers and Schmid**	**Wolfram Alpha**	**OEIS**	**Scharr**
Arithmetic	100.0 %	98.0 %	50.1 %	100.0 %
Geometric	100.0 %	100.0 %	19.5 %	100.0 %
Arithmetic Geometric	96.5 %	100.0 %	29.1 %	100.0 %
Arithmetic Geometric 2	10.0 %	15.0 %	4.5 %	42.9 %
Fibonacci	0.0 %	51.5 %	13.4 %	100.0 %
Arithmetic Alternating	3.4 %	11.0 %	0.1 %	100.0 %
Arithmetic & Irrelevant	0.0 %	0.0 %	0.3 %	100.0 %
Arithmetic Incremental	100.0 %	98.0 %	6.8 %	100.0 %

The experiment reveals, that all algorithms considered by Liu et al., 2019 have difficulties with short alternating series and short series that depend on multiple predecessors. My algorithm either solves all series completely or outperforms the other algorithms by a factor of 3, which is a significant margin.

In the second experiment, series of length 10 are given to the algorithms which shall recover the 11th position. The result of this experiment is shown in Table 2.

Table 2: This table shows the precision of the considered algorithms for completing series of length 11 correctly (10 positions given, one to fill). The results are grouped by algorithm (named after the group designing it) and the sequence type contained in the test-set. My algorithm is shown as "Scharr". The results for the other algorithms are from the benchmark by Kernbach, 2021a.

Test-Set	**Siebers and Schmid**	**Wolfram Alpha**	**OEIS**	**Scharr**
Arithmetic Geometric 2	100.0 %	43.0 %	2.2 %	36.6 %
Fibonacci	100.0 %	100.0 %	0.6 %	94.6 %
Arithmetic Alternating	100.0 %	100.0 %	0.0 %	100.0 %
Arithmetic & Irrelevant	99.7 %	100.0 %	0.2 %	99.6 %
Arithmetic Incremental	100.0 %	99.0 %	3.6 %	99.2 %

Interestingly, the performance of the algorithms of WolframAlpha, and Siebers and Schmid increased immensely. The algorithm from Siebers and Schmid solved almost every test-set correctly, which can be explained with the extended length of the test series. The performance of my algorithm dropped with increasing length of the series. This is surprising, as the amount of structural information, that can be extracted from a series, should be higher in a series of greater length. This does not mean that more series stayed unsolved. Instead, the confidence, with which a series got solved, dropped. That is because my algorithm found multiple explanations for a series. Furthermore, the precision of my algorithm is computed in the following way:

$$\text{Precision}(\text{test-set}) = \frac{\text{\# correct solutions}}{\text{\# of all attempts combined}}$$

which values wrong predictions more than correct predictions. Therefore the lower precision might be caused, because the number of total tests performed in my search-structure increases cubical in the length of a query. This could be tackled by shrinking the search-space via a heuristic. While this approach sounds rather complicated in theory, it would not be complicated in practice. It would simply translate into a change to 3 for-loops which would have the further benefit of speeding my algorithm up. Due to time constraints, I was not able to alter my algorithm in time to include an experiment that evaluates this.

In the third experiment, my algorithm got the task of solving series of the length 10 where up to 5 positions got masked uniformly at random. This means that 5 unique indices got drawn uniformly at random form the 10 possible indices per series. The result of this experiment is shown in Table 3.

Table 3: This table shows the precision of my algorithm for completing series of length 10 correctly in relation to the number of masked positions. The results are grouped by the number of positions masked uniformly at random and the sequence type contained in the test-set.

Test-Set	**1**	**2**	**3**	**4**	**5**
Arithmetic	100.0 %	100.0 %	100.0 %	100.0 %	100.0 %
Geometric	100.0 %	99.8 %	99.1 %	98.8 %	96.3 %
Arithmetic Geometric	100.0 %	97.9 %	95.2 %	86.6 %	74.1 %
Arithmetic Geometric 2	31.9 %	27.3 %	19.6 %	17.9 %	15.0 %
Fibonacci	96.5 %	87.2 %	63.9 %	40.0 %	23.5 %
Arithmetic Alternating	99.8 %	99.8 %	89.8 %	66.5 %	42.9 %
Arithmetic & Irrelevant	89.0 %	79.1 %	68.5 %	49.8 %	38.4 %
Arithmetic Incremental	100.0 %	100.0 %	100.0 %	86.4 %	64.6 %

The precision of my algorithm decreases when more of a series is masked. This is to be expected, as the information contained by the masked series decreases. However, the results that are achieved when 5 positions are masked are highly interesting. The sequence types "Arithmetic Alternating" and "Arithmetic Irrelevant Number" should not be solvable, as only 5 positions are unmasked. But as previously described a minimum of 6 unmasked positions should be needed to fill a series correctly.

However, when relaxing the constraint, which induces this minimal number of unmasked positions, to the point where one sub-series can have 2 and the other one 3 entries it would be expected to get a performance of:

$$\frac{\binom{5}{3}\cdot\binom{5}{2}}{\binom{10}{5}} = \frac{25}{63} \approx 39.7\%$$

which reflects the achieved results. This means that my algorithm stays close to the theoretical achievable maximum for at least 4 of the 8 test-sets.

These results cannot be compared to the results of the algorithm of Siebers and Schmid, 2012, as their algorithm is unable to answer this kind of queries.

In Table 4 the time needed to perform each experiment is shown. The first two experiments are indicated by the length of the series that was solved. The parts of the third experiment are indicated by the number of positions masked uniformly at random.

Table 4: This table shows the time needed to perform each experiment discussed above in seconds. The experiments were conducted on an Intel(R) Core(TM) i7-5600U CPU @2.60GHz. It shows that the run-time depends on the sequence length and the depth reached by the algorithm. The complexity of a sequence type impacts the depth reached by the algorithm. The first column shows the test-set, the second and third show the time needed to solve the last position of a sequence of the indicated length. The remaining columns show the time needed to solve a masked sequence of length 10 with up to 5 positions masked uniformly at random.

Test-Set	**7**	**10**	**1**	**2**	**3**	**4**	**5**
Arithmetic	0.09	13.4	13.2	13.3	13.0	13.0	13.0
Geometric	0.09	13.3	13.3	14.0	14.7	18.5	21.9
Arithmetic Geometric	0.11	28.7	29.1	30.7	31.4	33.5	35.6
Arithmetic Geometric 2	0.28	51.6	54.0	55.8	57.2	57.4	61.0
Fibonacci	0.17	25.7	27.0	31.4	38.7	46.0	51.4
Arithmetic Alternating	0.10	13.6	13.1	13.7	19.2	44.9	78.6
Arithmetic & Irrelevant	0.09	13.1	12.9	24.6	34.9	47.7	65.5
Arithmetic Incremental	0.19	40.1	39.7	39.0	39.9	42.8	48.4

These times show that the search-space, traversed by the algorithm, is much smaller for series of the length 7 in comparison to series of the length 10. This is due to the way the second and third block of my search-structure (Figure 3) are implemented. Each of them consider all theoretically possible ways to equidistantly sub-sample the given series. In the case of the third block, this scales cubical in the size of the series. However, one way to sub-sample contradicts many other ways Therefore this expenditure could be reduced by not searching for rules on sub-samples which are redundant or would contradict already found rules. This is not implemented yet, as I did not expect this to a have such a big impact on the run-time. Additionally, it is known that looping in python is slow. Therefore an easily achivable speedup is to perform the looping in a faster language, such as c++. These loops could then call tests implemented in python, which would save the accessibility of this code.

The amount of masked positions per series has an impact on the run-time too. However, the impact is not necessarily big, as visible in the "Arithmetic" test-set. The increased run-time is partly due to the need of finding different rules for different sub-series of a series and partly due to the decreased amount of information stored by each series. This results into a deeper traversal of the search-structure as more positions get masked.

In summary we see that for short sequences of length 7 the algorithm proposed here improves strongly upon the previously existing results of Siebers and Schmid, 2012; WolframAlpha, 2021; OEIS, 2021. It delivers 100.0% precision, where the best competing algorithms do also, and strongly outperforms them, where they do not. This was possible by formulating a new modular search-structure. For longer series of length 11 Siebers and Schmid, 2012; WolframAlpha, 2021 perform preferably, where the proposed algorithm mostly comes close to their performance. However, besides these very positive results, there is room for improvement on longer and more complex series.

5 Future Work

The results form the experiments look promising, but show room for improvement. Some aspects which could be improved were mentioned in the analysis of the experiments. They boil down to three main aspects; Firstly, the implementation of a pruning to the search-structure is needed. This pruning aims to remove tests that are impossible to be positive, as they would contradict test that have been positive already. Secondly, the implementation of a mechanism which selects one solution out of multiple solutions to a series is necessary. This could be a mechanism as simple as majority vote. Lastly, the implementation of tests, that improve upon my ideas, could improve the results achieved in the experiments even further.

The last point has to be considered with caution as the second experiment revealed that it is not only important to find a solution but to have a high confidence that the found solution is the correct solution to the series in the search-space. This may lead to the conjunction of multiple search-structures, each using a subset of all tests available. The optimal number of search-structures and the optimal configuration of tests used per search-structure could then be obtained by the usual training- and validation-set approach. This, however, would need a large set of diverse benchmarks to avoid overfitting. To my understanding, the work of Kernbach, 2021b could be a steppingstone for this.

In conclusion, the performance of my algorithm on the benchmarks considered is excellent, especially for the difficult case of short and alternating IQ-Number Test series. Apart from that, my modular implementation has made the implementation of an IQ-Number Test solver easily accessible.

References

Glimm, Birte and Antonio Krüger (2012). *KI 2012: Advances in Artificial Intelligence: 35th Annual German Conference on AI, Saarbrücken, Germany, September 24-27, 2012, Proceedings*. Vol. 7526. Springer.

Kernbach, Maximilian (2021a). *Benchmark*. Online, last checked on 09. Sep. 2021. URL: https://docs.google.com/spreadsheets/d/1NY_ty8dWDzet3DcqKVGQVRHnx7_T97Cotjfbeiu2IHc/edit#gid=0.

— (2021b). *Topic X: Generating IQ-Number Tests for AI*. In the Lab: Development and Application of Data Mining and Learning Systems: Machine Learning and Data Mining. Presentation on 07. Sep. 2021.

Liu, Yusen et al. (July 2019). "How Well Do Machines Perform on IQ tests: a Comparison Study on a Large-Scale Dataset". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, pp. 6110–6116. DOI: 10.24963/ijcai.2019/846. URL: https://doi.org/10.24963/ijcai.2019/846.

OEIS (2021). Online, last checked on 11. Sep. 2021. URL: https://oeis.org/?language=german.

Siebers and Schmid (2012). "Semi-analytic Natural Number Series Induction". In: pp. 249–252.

WolframAlpha (2021). Online, last checked on 11. Sep. 2021. URL: https://www.wolframalpha.com/.