A Computational Analysis of General Intelligence Tests for Evaluating Cognitive Development

Fernando Martínez-Plumed, Cèsar Ferri, José Hernández-Orallo, María José Ramírez-Quintana
{fmartinez,cferri,jorallo,mramirez}@dsic.upv.es
Universitat Politècnica de València, Spain

Abstract

The progression in several cognitive tests for the same subjects at different ages provides valuable information about their cognitive development. One question that has caught recent interest is whether the same approach can be used to assess the cognitive development of artificial systems. In particular, can we assess whether the 'fluid' or 'crystallised' intelligence of an artificial cognitive system is changing during its cognitive development as a result of acquiring more concepts? In this paper, we address several IQ tests problems (odd-one-out problems, Raven's Progressive Matrices and Thurstone's letter series) with a *general* learning system that is not particularly designed on purpose to solve intelligence tests. The goal is to better understand the role of the basic cognitive operational constructs (such as identity, difference, order, counting, logic, etc.) that are needed to solve these intelligence test problems and serve as a proof-of-concept for evaluation in other developmental problems. From here, we gain some insights into the characteristics and usefulness of these tests and how careful we need to be when applying human test problems to assess the abilities and cognitive development of robots and other artificial cognitive systems.

Keywords: Cognitive development assessment; cognitive tests; general intelligence; task difficulty; cognitive operational constructs; general learning systems; evaluation of artificial systems, inductive programming.

1. Introduction

Humans undergo a cognitive development that starts with important neurological transformations even before birth and lasts during their lifetime. This cognitive development is also accompanied by an uneven variation of a range of cognitive capabilities. In order to assess this, many cognitive tests have been specifically devised for different capabilities and age ranges. In fact, one of the applications of these IQ tests (among many others) is the assessment of children's cognition and learning, in order to spot development problems or to identify specially talented individuals. The notion of 'mental age', for instance, was introduced by one of the fathers of psychometrics, Alfred Binet, to compare retarded children with the normal development of children of their age. Nowadays, the notion of a *single* mental age is more elaborate, as many abilities are known to develop at different age intervals.

This idea of using tests, in the form of a set of tasks or exercises, is also becoming more and more common for the evaluation of *artificial* cognitive systems or architectures. On one hand, we have those who advocate for the use of human cognitive tests for machines directly. This idea is behind the psychometric AI proposal (Bringsjord and Schimanski, 2003; Bringsjord, 2011), where sets of tests for all possible human capabilities should be used to evaluate artificial cognitive systems. This view is consistent with an editorial by Douglas K. Detterman (2011), the editor-in-chief of *Intelligence*, motivated by the success of IBM's program Watson (Ferrucci et al., 2010) on the *Jeopardy!* TV quiz show. Detterman challenged Watson and other artificial systems to pass a battery of human intelligence tests. For instance, *Spaun*, a 2.5-million-neuron model of the brain has been able to "reproduce the largest amount of functionality and behaviour" (Yong, November 2012) by their performance on a "diversity of tasks" (Eliasmith et al., 2012), where some of them are very similar to intelligence test tasks, such as serial working memory, counting, number series, etc. Despite this pullulation and advocacy of intelligence test tasks for evaluating artificial cognitive systems, some criticisms have been raised as whether the resulting scores can be meaningful (Dowe and Hernández-Orallo, 2012), starting with (Sanghi and Dowe, 2003a), who devised a very small and ad-hoc program that was able to score relatively well on many human IQ tests. That does not mean that cognitive tests for humans need to be discarded for artificial systems, but that a proper selection and a careful analysis of results may be required in each case. In fact, some approaches to evaluate *artificial* cognitive systems or architectures have *adapted* existing tests or have been inspired by them, such as the cognitive decathlon (Mueller et al., 2007; Mueller, 2008; Simpson Jr and Twardy, 2008; Calvo-Garzón, 2003), the staged developmental test (Keedwell, 2010), the intelligence tests for robots (Sinapov and Stoytchev, 2010; Schenck, 2013), the universal anytime intelligence tests (Hernández-Orallo and Dowe, 2010), and others (Anderson and Lebiere, 2003; Langley, 2011). This approach is becoming now crucial in artificial intelligence, robotics and cognitive science since the Turing Test (Turing, 1950) has been left as a philosophical rather than practical test (Hernández-Orallo, 2000). Similarly, we also have the more classical evaluation of AI systems using specific testbeds, such as maze problems, games such as chess, pattern recognition problems, robot navigation, etc. However, it is more and more manifest that the success or failure at some specific tasks is not well correlated to the degree of intelligence or cognitive development of a system, a phenomenon that is well illustrated by the problems we can find in CAPTCHAs (von Ahn et al., 2004, 2008). In the end, it is clear that one goal is to solve a specific (application) problem and a very different goal is to devise a cognitive system that can solve many different problems. Interestingly, it is not very likely that such an artificial cognitive system could have a high degree of intelligence from the beginning. In other words, it is *potential* intelligence (Hernández-Orallo and Dowe, 2013) rather than *actual* intelligence what these systems should have originally.

It is then understandable that cognitive science and artificial intelligence are becoming more interested in tests that evaluate general abilities instead of success on a particular set of tasks. In development robotics "the notion of task-independence" (Oudeyer and Kaplan, 2007) is related to "general intelligence": "developmental robots shall not be programmed to achieve a prespecified practical task". The choice of intelligence tests to evaluate cognitive development (and not only cognitive capabilities) is consistent with one observed phenomenon in human intelligence: many of its underlying abilities increase during childhood and youth, stabilise for several decades and then slowly decline while ageing. However, is this really the consequence of a cognitive development or is it a result of some neurophysiological changes in the brain affecting its efficiency? This is linked to the notions of fluid intelligence, which solves problems without the need of previous knowledge, and crystallised intelligence, which combines previously learned constructs for a new problem. In fact, once we distinguish between fluid intelligence and crystallised intelligence, should not fluid intelligence be constant while crystallised intelligence increases through cognitive development?

We can rephrase the previous question more specifically. Are the increasingly better results in IQ tests from infancy to early adulthood explained by the development of new *cognitive operational constructs* that are useful (or even necessary) to solve the tests? Or is it because those taking the tests develop increasing better combinatorial search power? What about artificial systems where the computational power is constant? Is their intelligence expected to remain constant?

In order to shed some light on these questions, we set a parallelism between the concepts of fluid and crystallised intelligence in humans and artificial (cognitive) systems by exploring how a general learning system (without proper cognitive skills) addresses several tasks from IQ tests as a possible way to understand and examine the need of constructs in these tasks and its role in development. With the goal of getting more insight from this experiment, we will use a learning system that uses intelligible ways of expressing background knowledge (to make the required cognitive operational constructs and the generation of representation explicit) as well as the extracted patterns for each exercise. In particular, we will use the system gErl (Martínez-Plumed et al., 2013a,b) —a declarative general learning system that was not devised on purpose for IQ tests— to solve some prototypical intelligence test tasks: odd-one-out problems, Raven's progressive matrices and Thurstone letter series. The presentation of the problems will be set as bare as possible, in order to eliminate the influence of pattern recognition issues that may interfere in our analysis. While gErl is, to our knowledge, the first general system that is able to score average human results in several IQ test tasks, we clearly make the point of how misleading a superficial *score* comparison is. Rather, we use the system to better understand what these IQ test tasks measure and what a system —human or artificial— requires to solve them, and whether an inability can be turned into an ability through development. Therefore, we will pay special attention on what makes some of the instances in each category harder than others: is it because the search space is larger or because they need more cognitive constructs? In other words, we distinguish two previously conflated aspects in item difficulty: the mental operational constructs that each task requires and the combinatorial problem of combining these constructs to find the solution.

It is important to note that gErl is not a cognitive model and it is not inspired by how humans solve problems or how their development takes place. It is orthogonal to them and this is precisely what will allow us to determine those features that are dependent on the problem and not on the system. Also, we do not want to show how good or bad gErl is at solving these IQ test problems. Actually, what we want to show is that, through the use of a general learning tool that uses intelligible patterns, we can better understand their difficulty and the role of the cognitive constructs that are required in the both processes of learning and development of artificial (cognitive) systems. This is possible because of the generality of the system, as other systems that are made on purpose for solving IQ tests may lead to misleading interpretations as they are relatively easy to create with many built-in constructs (Sanghi and Dowe, 2003a; Dowe and Hernández-Orallo, 2012) and may have a strong bias in their scaling of problem difficulty. From this analysis using a general system for several IQ test problems, we will not settle the question of how much important cognitive development is for scoring better in general intelligence tests for natural and artificial cognitive systems, but we will provide a new perspective and procedure to investigate this question. The take-away message is the appropriateness and care of using these and other general intelligence tests to evaluate the mental development of artificial systems.

The paper is organised as follows. Section 2 overviews the use of cognitive tests and their variants (according to age and set of abilities), mostly focusing on the evolution of scores with (mental) age and their relation to human cognitive development. Their application to evaluate artificial cognitive systems or other purposes is also discussed, and the need for a better assessment of their difficulty in terms of the operational constructs and search space that are involved. Furthermore, we briefly describe the general learning system gErl, which will be further explained in Appendix A. Section 3 applies gErl to several odd-one-out problems, Raven's progressive matrices and Thurstone letter series, analyses the required cognitive operational constructs and the complexity of the found patterns. Section 4 closes the paper with an overall analysis of the general findings of the paper and its implications in the issue of cognitive development assessment and the (careful) use of intelligence tests for artificial cognitive systems.

2. General intelligence evaluation during cognitive development

As human capabilities improve with age, many evaluation procedures use several sets of exercises of different types and difficulty in order to accommodate for the subject's expected cognitive development. For instance, Wechsler Intelligence Scale tests (Wechsler, 1958; Kaufman and Lichtenberger, 2006) are nowadays arranged into different batteries (WPPSI, WISC and WAIS), according to age. In this and other tests we have a broad range of abilities, so we can give specific assessments of what components of intelligence are more or less developed in a particular individual, as well as their improvement in time. For instance, the results on verbal tests are expected to improve with the years, as language must be acquired during early childhood and is consolidated for many years even after. However, it is not clear why other abilities, especially the more abstract abilities related to abstract reasoning and inductive inference, have a similar behaviour.

This is closely related to Cattell's concepts of fluid and crystallised intelligence, where the former is the ability of solving problems independently of previously acquired knowledge, while the latter is the ability of correctly finding and applying the given knowledge to a particular problem. Again, it is easy to understand why crystallised intelligence increases with cognitive development (with still moderate increments until the age of 40 or 50, see Fig. 1). Nevertheless, why does fluid intelligence grow until the age of 20? One main hypothesis is that the brain develops until that age, so the increase in fluid intelligence has a neuropsychological explanation (Epstein, 1979). However, another possible explanation is that while the brain reaches its final size and connections very early, some *cognitive constructs* (such as identity, difference, size, order, counting, symmetry, logic, quantification, (re)iteration, recursion, etc.) are also useful in fluid intelligence and need to be acquired during a long period of time. These two hypotheses are also closely linked to the nature-vs-nurture dilemma, as fluid intelligence in adults can be well predicted from early fluid intelligence (Neisser et al., 1996), showing a strong genetic basis, but contrasting with the diminished performance in fluid intelligence tests for children with a poorly-stimulating environment or education. Instead of two alternative hypotheses, the question is rather to establish what the role and relevance are of both an internally-driven cognitive development and an externally-enhanced cognitive development. In fact, the picture becomes more complicated, as it has been shown that fluid intelligence can be increased by cognitive exercising (Sternberg, 2008), although not permanently. The area of autonomous mental development requires, thus, better measurement techniques and devices to determine whether and how systems develop (Oudeyer and Kaplan, 2007), and to tell between what a system has been programmed to do and what abilities the system has really developed. As an interdisciplinary area, the understanding and adaptation of measurement devices from humans to artificial cognitive systems may have an important impact in

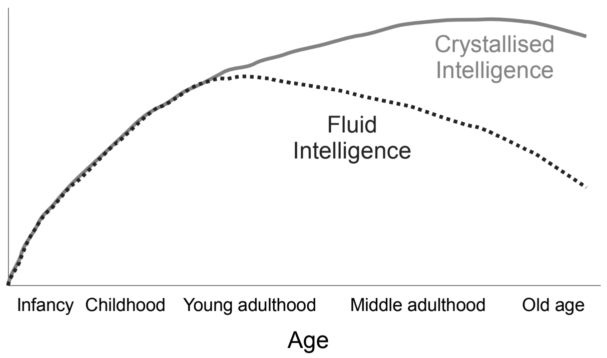

Figure 1: A figurative representation of how fluid and crystallised intelligence "display different life-span developmental trajectories" (Baltes, 1987, Fig.1). Can we expect (and measure) the same evolution of general intelligence for artificial cognitive systems? What is the role of cognitive operational constructs in this evolution?

how the goals and progress in the field are understood, strengthening the connections and cross-citations with other disciplines and a source of new research questions.

While the analysis of both crystallised and fluid intelligence tests would be interesting, crystallised intelligence tests can be more easily contaminated by the existence of predefined structures and task-specific problems. Also, most crystallised intelligence problems are verbal, which limits the application of these tests to systems with a limited or no comprehension of natural language.

Fluid intelligence tests have several advantages for evaluating cognitive development as most of them do not require natural language and also because they are commonly represented in very abstract terms, so it seems possible to analyse and ultimately understand what processes and constructs may be needed to solve them and what their actual difficulty is. If we look at the Wechsler's WPPSI, WISC and WAIS mentioned at the beginning of this section, they usually measure fluid intelligence on the performance scale and crystallised intelligence on the verbal scale. If we focus on non-linguistic abstract problems (usually found under the categories of working memory and perceptual organisation), and exclude memory problems because of lack of a suitable evaluation in artificial systems, we find matrix reasoning tasks, problems about categories and similarity (picture concepts and symbol search) and letter-number sequencing (only present in WISC and WAIS). These tasks usually feature a combination of inductive inference and abstract reasoning and are among those with highest *g* loadings, namely, they highly correlate with the *g* factor[1], so an individual's performance at one type of cognitive task tends to be comparable to their performance at other kinds of cognitive tasks. This is particularly interesting, as the *g* factor accounts for an important fraction of the variation of other cognitive abilities.

From the previous analysis, we will focus on fluid intelligence tests with high *g* loading. In order to make experiments more insightful and easy to replicate we will select problems that are easily accessible, either because they are on the open domain or because they have been well studied by previous works, so the operational constructs that are required and their effect on difficulty are well known. For instance, odd-one-out problems have been chosen as very relevant for cognitive development (Sinapov and Stoytchev, 2010), as they are closely related to the capability of forming categories and distinguishing between them (Griffith et al., 2012). Raven's Progressive Matrices (RPM) (Raven and Court, 1996) have some of the highest *g* loadings of standard cognitive tests. Finally, Thurstone's letter series (Thurstone and Thurstone, 1941) are a well-known instance of extrapolation problems, which are regularly found in most general intelligence tests. The operational constructs involved in these tests are related to fundamental cognitive processes such as inductive inference, abstraction, analogy, etc. Consequently, these tasks allow us to analyse how these cognitive operational constructs are needed from a developmental point of view.

It is important to note, however, that the use of any cognitive test to evaluate cognitive development is based on the assumption that the difficult items in these tests require a more advanced cognitive development than the simpler

[1]The *g* factor (short for "general factor") is a construct developed in *psychometrics* (theory and practice of psychological measurement) that is usually derived from a factor analysis of the results of many tests, and is usually associated with the idea of *general intelligence*.

ones. Otherwise, we could use the adult versions for all. Nonetheless, it is not clear whether this difficulty comes from a higher search power (when looking at the combinatorial possibilities) or it is because some difficult problems require some cognitive operational constructs. With the term ‘cognitive operational construct’ we do not refer to very elaborated linguistic concepts, neither do we refer to knowledge facts, but rather about elementary *operational* concepts such as identity, difference, order, counting, logic, etc. Many of these constructs are specifically addressed in children education curricula.

Circumscribing our study to fluid intelligence tests frames the question in terms of the evolution of the development and evolution of general intelligence, as opposed to the mere acquisition of factual knowledge, from which general intelligence (including crystallised intelligence) should, therefore, not benefit (e.g., “how many protons a hydrogen atom has” is not usually found in a culture-fair test, as could be answered by any idiot savant or a web search engine). This analysis of general intelligence tests removes many other sources of contaminations and renders the question in a more pristine way. Also, it links cognitive development to an increase in general intelligence and suggests a possible path to develop and study how artificial cognitive systems could be created. Similarly to humans, we do not expect a general (artificial) cognitive system to be highly intelligent from its very creation, neither to be highly specialised, but rather to have very low levels of intelligence and being mostly useless in the beginning. In other words, developmental robotics/cognition, cognitive agents and architectures, and other areas of artificial (general) intelligence and cognitive sciences would aim instead at developing potential intelligent systems, whose general intelligence could gradually increase with the interaction with the world, as it happens with humans (up to the limit of the computational resources of the system).

Analysing these questions by administering some tests to humans and by asking them what constructs and patterns they have used and found respectively is a traditional approach for the analysis of human cognitive development. This provides a subjective and anthropomorphic view, and it is also contaminated about many other issues taking place during human development, including physical, biological and social effects. Nonetheless, it is still a useful approach. However, when the goal is to assess the development of artificial cognitive systems, we need a different approach. In fact, there have been other works in the past where IQ tests, development tests or other kinds of cognitive tests are undertaken by artificial systems. The goal of our study is not to evaluate these systems or analyse the different type of tasks involved in IQ tests, but to better understand how the tests work and what they measure. In fact, this analysis of tests, ranging from the seminal papers by Evans (1963, 1965) and his system’s ANALOGY, which was “capable of solving a wide class of the so-called ‘geometric-analogy’ problems frequently encountered on intelligence tests” (Evans, 1965) and today has already been done in Hernández-Orallo et al. (2016), in terms of their main features, achievements, relations, etc.

This collection of systems and approaches dealing with intelligence test tasks are usually defined on purpose to solve these tasks. However, most importantly, none of them has focused on the issue of item difficulty in terms of both the combinatorial search space of each item and the required operational constructs under the context of cognitive development. Item difficulty is crucial in order to set different test scales for different stages of development. Note that it is not very meaningful if we define problem difficulty as the average result of a population of artificial cognitive systems (mimicking what is done with human population, item response theory and IQ normalisation), as the sample of systems would be completely arbitrary. So, in order to assess difficulty we need system generality and intelligibility. More precisely, we need a cognitive tool that meets the following properties:

- The system must have a general purpose. It cannot be defined ad-hoc for intelligence tests, as this may lead to specialisation to one or more tests (Sanghi and Dowe, 2003a), and the results would not be meaningful (Dowe and Hernández-Orallo, 2012). The use of inductive programming systems for cognitive modelling has been recently shown to be effective by Schmid and Kitzelmann (2011).

- Because of the above design generality, the system must *learn* to solve the problems. In other words, it has to be a *learning* system, which must be guided by a rewarding system or the level of success on several examples.

- The system has to be explicit in what constructs and operators must be given as background knowledge when addressing each problem. These operational constructs must also be intelligible.

- The system must produce intelligible solutions, such that their patterns can be compared to those usually extracted by humans, and their structure can be evaluated in terms of size and number of constructs.

The above properties suggest the use of declarative learning systems, i.e., inductive programming systems (Kitzelmann, 2009; Schmid and Kitzelmann, 2011; Flener and Schmid, 2008).

We will use the declarative learning system gErl (Martínez-Plumed et al., 2013a,b) as a cognitive tool that fulfils the above properties. gErl was born as an advocacy of a more general framework for machine learning: a general and declarative rule-based learning system where operators can be (user-)defined and customised according to the problem, data representation and the way the information should be navigated. Roughly speaking, operators perform modifications over any subpart of a rule (arguments, conditions, body, ...) in order to generalise or specialise it and, thus, generating new rules. Since operators affect how the search space needs to be explored, heuristics are learnt as a result of a decision process based on reinforcement learning (Sutton, 1998), where each action is defined as a choice of operator and rule (which operator has to be applied over which rule) guided by an optimality criterion (based on coverage and simplicity) that feeds a rewarding module. The population of rules changes in every learning step of the system (each time an action is performed, new rules can be created). States (current population of rules in the system) and actions are abstractly represented as tuples of descriptive features in a Q matrix (Watkins and Dayan, 1992a) from which a supervised model is learnt and used to obtain the best action to perform for a specific state. The functional programming language Erlang (Virding et al., 1996) is used to represent theories and examples in an understandable way: examples as equations, patterns as rules, and models as sets of rules. In order to define operators, gErl provides some meta-level facilities called meta-operators (which are higher-order functions) that allow the user to define well-known generalisation and specialisation operators. For further information about gErl and the definition of meta-operators, see Appendix A.

We will use gErl in order to analyse several general intelligence problems in sections 3 and 4, discussing on the identification of what makes each instance easy or hard, and the proportion of this difficulty in terms of the pattern size and the constructs that are involved.

3. IQ test problem analysis: Pattern size and required constructs

In this section we analyse three general intelligence tests tasks: odd-one-out, Raven's matrices and letter series. We will use the gErl system as an appropriate cognitive tool to analyse these problems in the context of cognitive development, since gErl satisfies the properties that we identified in the previous section.

3.1. Odd-one-out problems

The odd-one-out problems are focused on geometry and spatial understanding, where the goal is to spot the most dissimilar object from the rest. They were first introduced by Zentall et al. (1974, 1980) as a non-verbal test and they were used for the study of comparative animal learning (e.g., pigeon's intelligence). Some species appear to learn the task readily, slowly or nothing at all. The oddity task has been also used for cross-cultural testing to probe the conceptual primitives of geometry in the Mundurukú, an isolated Amazonian indigenous group, for which Dehaene et al. (2006) designed a visual oddity task. Figure 2 shows three examples of odd-one-out problems of increasing complexity. Typically, the presented items can vary along one dimension (e.g., shape, size, quantity).

3.1.1. Problem representation and constructs

In order to focus on the core of the problem and not on the visual recognition issues we use the abstract representation R-ASCM, introduced by Ruiz (2011). In R-ASCM, for instance, an item composed by two circles and a square is represented as (A, A, B). We will also use the same 35 examples in Ruiz (2011), with the R-ASCM coding, as shown in Table 1. To process the examples with gErl, we represent each set of items as a list of lists[2]. This will be the *lhs* of each example equation. And its corresponding *rhs* will be a number indicating the position of the item that is the odd one. For instance, example number 3 in Table 1 is represented as:

```
ooo([[a,a,a], [a,a,b], [a,a,c]]) -> 1
```

[2]As in most programming languages, constants in Erlang begin with a lower-case letter.

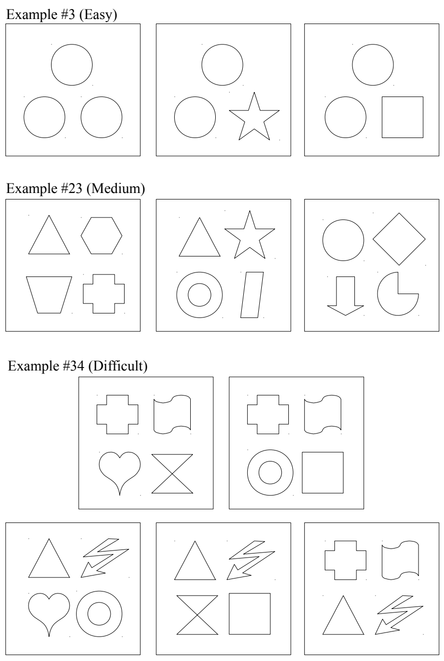

Figure 2: Examples 3, 23 and 34 in Ruiz (2011) (adapted and redrawn), sorted by human-subjective complexity (from easy to difficult).

#Ex	Item 1	Item 2	Item 3	Item 4	Item 5
1	AAA	AAA	ABB		
2	AAA	AAA	BCD		
3	AAA	AAB	AAC		
4	AAA	ABB	ABB		
5	AAA	BBB	ABC		
6	AAA	BCD	EFG		
7	AAA	BBC	CCB		
8	AAB	AAB	ABC		
9	AAB	AAC	DEF		
10	AAB	ABB	EFG		
11	ABC	ABC	ABD		
12	AAB	ABB	ABC		
13	ABC	ADE	FGH		
14	AAAA	BBDE	CCFG		
15	AAAA	AABB	AACC		
16	AAAD	BBEF	CCGH		
17	AABB	AABB	ABCD		
18	AABC	AACD	ABCD		
19	AAAB	BBBD	CCCE		
20	ABCD	ABCD	ABCE		
21	ABCD	ABCE	ABFG		
22	AABC	BBAC	CCAF		
23	ABCD	AEFG	HIJK		
24	AAAA	AAAA	BBBB	BBBB	CCCC
25	AAAD	AAAE	BBBF	BBBG	CCCH
26	AABB	BBCC	AADD	DDCC	EEFF
27	AAEF	BBGH	CCIJ	DDKL	ABCD
28	AAAE	BBBF	CCGH	DDIJ	ABCD
29	AAAE	BBBF	CCGH	DDIJ	AABB
30	AAAB	BBBF	CCGH	DDIJ	AABB
31	AABB	BBCC	AADD	DDCC	AAEE
32	ABCD	BCDE	CDEF	DEFG	FGAB
33	ACDE	AFGH	BIJK	BLMN	OPQR
34	ABEF	ABGH	CDEG	CDFH	ABCD
35	ACDE	AFGH	BIJK	BLMN	ABOP

Table 1: 35 examples of R-ASCM abstract representation (solutions in grey) from Ruiz (2011). For simplicity and space, letters are used here instead of the figural symbols of the real test. Both the items in each example and the coded figures in each item are sorted alphabetically. The first letters of the alphabet correspond to the most frequently used symbols in an item (e.g., A is more frequently used than B, B than C, and C than D).

Next we need to define appropriate operators, both to navigate the structure and to apply local or global changes to the rules. As in Ruiz (2011), we will use some constructs to analyse these lists. In our case, we identify three types of constructs:

- **Differences between items**: As used in Ruiz (2011), we incorporate the notion of difference between items, by adding the function `hamming`, which represents the average Hamming Distance between objects. This measure is calculated for every item inside an example and refers to the average distance of a given item to all the other items within an example. For instance, given the items of the example #3 in Table 1 ($[[a, a, a], [a, a, b], [a, a, c]]$), if we apply this construct over each item we will obtain, for all items, a result equal to 1.
- **Differences inside items**: In addition, we also include another construct about item diversity, which counts the number of different objects inside an item (`diffObj`). For instance, taking the previous example into account, the diversity for the first item is equal to 1 while for the next two items is equal to 2.
- **The notion of an element that is distinct**: Finally, this concept (`distinct`) must actually be given in order to solve these problems, as it is usually explained in the test instructions. It is very related to the previous constructs since, once we have applied any of the previous ones to one example, we have to select the odd one.

Since these functions have to be applied to a list, we define operators op_1 and op_2 with the help of the higher-order function *map*[3] that Erlang provides in order to introduce the two first constructs.

[3]With the syntax `map(Fun, List1) -> List2`, we express that ‘map’ takes a function `Fun` from *As* to *Bs*, and a list of *As* (`List1`) and produces a list of *Bs* (`List2`) by applying the function to every element in the list.

$$op_1 \equiv \quad \mu_{replace}(\overline{3}, f_{hamming})$$
$$op_2 \equiv \quad \mu_{replace}(\overline{3}, f_{diffObj})$$

where $f_{hamming}(\rho) = \texttt{map(hamming,}\rho|_{1.1})$ and $f_{diffObj}(\rho) = \texttt{map(diffObj,}\rho|_{1.1})$, and $\overline{3}$ is a constant function returning the position 3 (namely, the *rhs*) of the input rule over which the operator will be applied.[4] This meta-operator ($\mu_{replace}$) is thus used to define an operator in charge of replacing the *rhs* of the input rule by the defined functions. Since the previous operators return a list with the values for each item in each example, we must let the system apply the function `distinct`, which selects the different item (if exists) in a list. Hence, the operator op_3 can be defined as

$$op_3 \equiv \quad \mu_{replace}(\overline{3}, f_{distinct})$$

where $f_{distinct}(\rho) = \texttt{distinct}(\rho|_3)$.

Finally, we need a way of generalising the examples. This is performed with variables at each possible position.

$$op_4 \equiv \quad \mu_{replace}(pos_{list}, \overline{V_{lists}})$$

where we use now the function $pos_{list}(\rho)$ (which is not a constant) returning all the positions in ρ where we can find a list (1.1, 3.2 and 3.1.2) and $\overline{V_{lists}}$ is a constant function returning a list variable.

The abstract idea of generalisation through the use of variables appears as an operator, but it is not considered a cognitive construct here as it is rather a way of representing that the function `ooo` can be applied to any list.

3.1.2. Results

Table 2 shows the two rules with highest optimality found by gErl. Taking the first rule, 28 of 35 (80%) examples are solved (being on a par with an average human adult). Regarding the second rule, it solves 17 of 35 examples. Note that some examples are covered by both rules. Example number 31 is not covered by any rule because it exhibits other more complex properties, not captured by the constructs. Table 2 also shows the results provided by Ruiz who defines a 2-step clustering algorithm. In the first step, 28 of 35 (80%) examples are solved, exactly the same number that gErl covers with its first rule. The second step consists in taking those examples that were misclassified (*using* the classes of the test set), and recode them using what he calls the *Structural Hamming Distance* (SDM). Note that gErl's approach is more general as we do not have to recode misclassified examples. Also, the `diffObj` construct is much simpler than the SDM. In any case, it is not our goal to compare which system is best (in fact, we mostly follow Ruiz's approach, as we use his representation and the Hamming distance function), but to see what constructs are used in each case.

Example \ Approach		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	Total
gErl	h_1	●	●		●	●		●	●	●	●	●		●				●	●	●	●	●	●	●	●	●	●	●	●	●	●		●	●	●	●	28
	h_2	●	●	●	●		●	●	●	●	●		●		●	●	●	●	●									●	●								17
Ruiz	$1^{st} step$	●	●		●	●		●	●	●	●	●		●				●	●	●	●	●	●	●	●	●	●	●	●	●	●		●	●	●	●	28
	$2^{nd} step$			●									●		●	●	●																				5

Table 2: Results for both gErl and Ruiz's approach (2011) for the problems in Table 1. Filled dots shows those examples from Table 1 solved. h_1 refers to the hypothesis using (among others) the Hamming construct (h_1: $\texttt{ooo}(V_{lists}) \rightarrow \texttt{distinct(map(hamming,}V_{lists}\texttt{)))}$), whereas h_2 refers to the hypothesis using the `diffObj` construct (h_2: $\texttt{ooo}(V_{lists}) \rightarrow \texttt{distinct(map(diffObj,}V_{lists}\texttt{)))}$).

Table 2 is very informative as we can separate those examples (28, actually) that can be solved with the notion of difference between same-length lists (i.e., the Hamming distance, which implies some kind of alignment) and those examples (17, actually) that can just be solved with an internal account of diversity of each item. Finally, only one example requires some other constructs. Apart from this case, we see that the pattern complexity is similar in both

[4]Following the usual representation of (functional) rules as trees, each sub-part (term) of a rule is denoted by a natural number that represents the position of such term in the tree.

categories. At least in gErl, both rules have the same structure and number of constructors. In fact, the number of steps the system takes is similar (227 and 230). As a result, we can say that this is a clear example of problems where two categories can be established by the constructs they require rather than by the search space being significantly higher for one problem or the other. In other words, each category will only be solved when the construct is available (or can be discovered) at a given stage of cognitive development, and does not require any exceptional combinatorial or brain power. Performance differences in humans must be then attributed to the complexity of the problems that can be built. Namely, the more symbols are included in the sets, the more advanced mathematical properties could appear (see, for instance, item 31 in Table 1, which cannot be solved with any of the constructs used). However, the use of mathematical constructs (prime, odd or even numbers, squares, ...) are not measures of fluid intelligence but of crystallised ability, so this mathematical knowledge is beyond the scope of this paper.

3.2. Raven's Progressive Matrices

Raven's Progressive Matrices (RPM) (Raven and Court, 1996) consist of a pattern or a set of items where a missing part or item has to be guessed. The most typical case is a 3×3 grid where a figure is placed at each of the nine positions except the bottom-right cell, which is empty. Eight possible choices ('distractors') to fill in the gap are displayed at the bottom, as illustrated in Fig. 3. There is a logical relation between the figures, which can be seen either horizontally (rows) or vertically (columns).

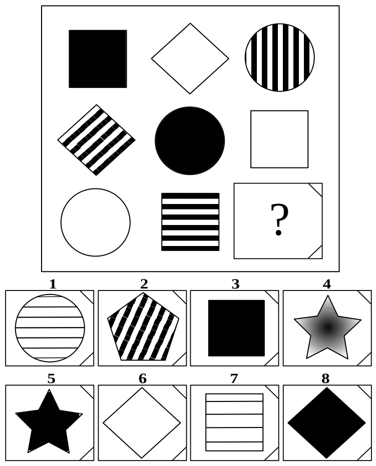

Figure 3: A geometrical reasoning problem similar to Raven's Progressive Matrices. The solution is no. 8 (for copyright reasons, the illustrations in this paper do not depict original RPM problems, but constructed equivalents).

There are three different sets of RPM for participants of different IQ ranges or different abilities (Raven and Court, 1996): the original Standard Progressive Matrices (SPM), the Coloured Progressive Matrices (CPM) for children aged 5 through 11 years-of-age, the elderly, or people with learning difficulties, and the Advanced Progressive Matrices (APM), developed to assess individuals of above-average intelligence. We will just work with SPM, which consist of 5 sets with 12 items in each set —60 items in total. Sets *A* and *B* do not use a 3×3 grid structure, so we will use sets *C*, *D* and *E* as they share the same structure (although the difficulty varies). The items were reconstructed using information from `http://www.raventest.net`.

3.2.1. Problem representation and constructs

Given an example of one kind of Raven's Progressive Matrices (as the one shown in Fig. 3), the task of gErl will be to guess the pattern and, then, to apply it to infer the solution. In other words, gErl tries to build the solution rather than choose among the collection of 8 possible choices (1 solution and 7 distractors) displayed under the matrix (which are not given to gErl). So, in this way, the problem formulation is slightly more difficult than the original one.

In order to represent RPM in gErl, a feature-based coding similar to that of Ragni and Neubert (2012) is used. Additionally, a list-based coding is used to represent cells, rows, and matrices. Every figure inside a cell is abstractly

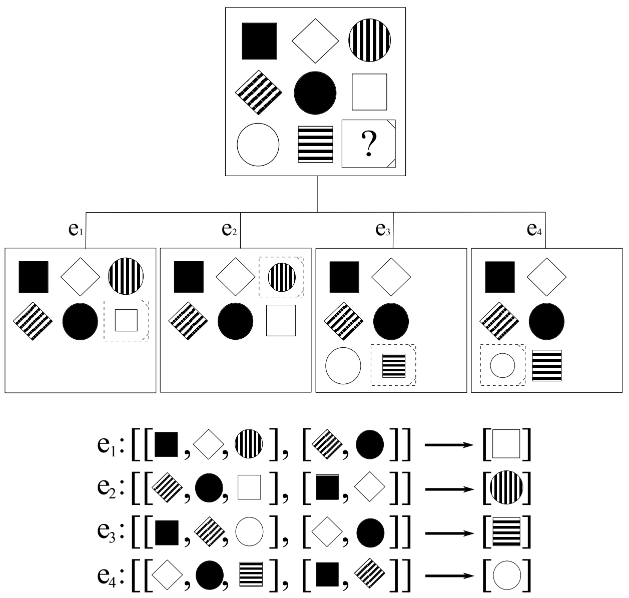

Figure 4: Raven's matrix decomposition example and the list-based representation of the training instances generated (bottom).

represented as a tuple of features:

$$\langle shape, size, quantity, position, type \rangle$$

Every cell is represented as a list of figures. Every row is represented as a lists of cells, and, finally, every Raven's matrix as a list of rows.

Since every single Raven's matrix is a problem itself (each matrix shows a different pattern), we need a way to generate several instances in order to make learning possible, by taking the most information from each matrix. To do that, each matrix is decomposed into several sub-matrices (the number depends on the problem) as we can see in Fig. 4 (top). The last row/column of the original matrix cannot be used to generate any training instance since it contains the gap to be filled in. However, they will be used to create two *test* cases (row and column). Note that the output element must obviously be the same, but the vertical and horizontal patterns may be different. If that happens, the system selects the best program with respect to the row training instances to be applied to the row test instance, and the best program with respect to the column training instances to be applied to the column test instance.

In gErl each training example is written as an equation. For instance, example e_1 from Fig. 4 (bottom) is represented as follows, where none represents the absence of a particular feature:

```
raven([[[<square,big,1,none,black>],[<diamond,big,1,none,white>],[<circle,big,1,none,striped>]],
[[<diamond,big,1,none,striped>],[<circle,big,1,none,black>]]]) -> [<square,big,1,none,white>].
```

To solve RPMs, it is necessary to identify the relations between the objects in a row or column. Carpenter et al. (1990) identified five *relations*:

- **Identity (equality)**: is a particular attribute of different objects remaining constant in a row? For instance, in matrix *a* (Fig. 5), the attribute *shape* (square) remains constant in every row/column. We will use the function `ident`.
- **Ternary distribution (difference)**: is an attribute of the objects in a row/column always differing (3 different values)? For instance, in Fig. 5, the attribute *type* differs in matrix a. We will use the function `dist3` for this.
- **Progression**: are the values of an attribute in an increasing or decreasing sequence in all rows/columns? This rule is seen in matrix *b* (Fig. 5), where the attribute for the object's position turns 45 degrees in each cell. We will use the function `prog` for this.

- **Addition (OR)**: is each object in the third row/column appearing in any of the first two rows/columns? An example is depicted in matrix *c* (Fig. 5). We will use the function `addition` for this.
- **Binary distribution (XOR)**: are there exactly two equal values and one differing value of an attribute in the rows/columns? This rule is seen in matrix *d* in Fig. 5. We will use the function `dist2` for this.

While these *relations* were expressed in terms of solutions, here they are just included as operational constructs and implemented as complex functions in gErl, i.e., each construct goes through each cell (in a complete row/column) looking for a specific descriptive feature and returning a solution for the construct. Since we are only interested in knowing which of those relations are required to solve each RPM problem, and knowing the abstract mental operations they represent, the low-level programming details of these previous functions are beyond the scope of this paper.

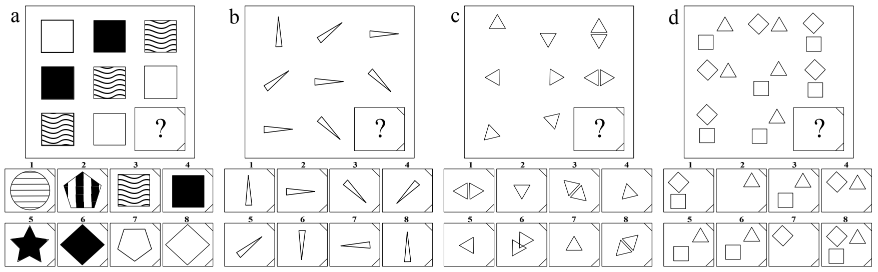

Figure 5: Problems illustrating several *relations* in RPM problems (redrawn and adapted from Ragni and Neubert (2012)). For matrix *a*, the *ternary distribution* is required. The solution is no. 4. Matrix *b* requires *progression*. The solution is no. 6. For matrix *c*, *addition* is required. The solution is no. 8. Matrix *d* requires *binary distribution*. The solution is no. 4.

To solve the selected 36 matrices from SPM, we need a way to apply the five *relations* (functions) to the different attributes of the figures. The meta-operator $\mu_{replace}$ does this perfectly by defining operators following this scheme:

$$\mu_{replace}(pos_F, f_{Rel})$$

where $pos_F(\rho)$ returns all positions where feature F (shape, size, quantity, position or type) appears in the *rhs* of ρ and $f_{Rel}(\rho)$ applies the relation function *Rel* (`ident`, `dist3`, `prog`, `addition`, `dist2`) to $\rho|_p$, $p \in pos_F(\rho)$.

We will obtain as many operators as kinds of attributes multiplied by the number of relations (5 attributes × 5 relations, 25 operators). For instance, the operators that apply `ident` to the five possible attributes will be defined as:

$$\begin{array}{llll} op_1 \equiv & \mu_{replace}(pos_{shape}, f_{id}) & op_2 \equiv & \mu_{replace}(pos_{size}, f_{id}) \\ op_3 \equiv & \mu_{replace}(pos_{quantity}, f_{id}) & op_4 \equiv & \mu_{replace}(pos_{position}, f_{id}) \\ op_5 \equiv & \mu_{replace}(pos_{type}, f_{id}) & & \end{array}$$

where $f_{id}(\rho) = \texttt{ident}(\rho|_p), p \in pos_F$. Below we show an example of application of the operator op_3, which is in charge of applying the relation `ident` over the feature *quantity*, to a rule (where some other operators have been already applied):

$$\begin{array}{rl} op_3(\texttt{raven}(V_{matrix}) \rightarrow & \texttt{[<dist3(shape),ident(Size),1,none,dist3(type)>])} \Rightarrow \\ \texttt{raven}(V_{matrix}) \rightarrow & \texttt{[<dist3(shape),ident(size),ident(quantity),none,dist3(type)>]} \end{array}$$

where V_{matrix} is a matrix variable. This suggests that we also need a generalisation operator for input lists (as we did in the odd-one-out problem): $\mu_{replace}(\overline{1.1}, \overline{V_{matrix}})$. If we apply the previous rule learnt by the system to the test row/column in Fig. 3, the different functions applied over the particular features will be in charge of returning the correct values (e.g. `dist3(shape)` will return `diamond` since the `circle` and `square` shapes have been already used), thus returning the following cell as solution:

`[<diamond,big,1,none,black>]`

which covers the correct solution (no. 8).

Id	Solution	Steps	E^+	$\lvert o_{app}\rvert$	Diff
C01	$\texttt{raven}(V) \rightarrow [\langle\texttt{ident(shape),none,none,none,none}\rangle]$	37	3	2	-2.6
C02	$\texttt{raven}(V) \rightarrow [\langle\texttt{ident(shape),prog(size),none,none,none}\rangle]$	99	4	3	-3.0
C03	$\texttt{raven}(V) \rightarrow [\langle\texttt{ident(shape),prog(size),none,none,none}\rangle]$	99	4	3	-3.3
C04	$\texttt{raven}(V) \rightarrow [\langle\texttt{ident(shape),prog(size),none,none,none}\rangle]$	111	4	3	-2.2
C05	$\texttt{raven}(V) \rightarrow [\langle\texttt{ident(shape),none,prog(quantity),prog(position),none}\rangle]$	131	4	4	-3.5
C06	$\texttt{raven}(V) \rightarrow [\langle\texttt{ident(shape),prog(size),none,none,none}\rangle]$	88	4	3	-1.5
C07	$\texttt{raven}(V) \rightarrow [\langle\texttt{ident(shape),none,none,prog(position),none}\rangle]$	81	4	3	-2.7
C08	$\texttt{raven}(V) \rightarrow [\langle\texttt{ident(shape),none,prog(quantity),none,none}\rangle]$	79	4	3	-0.8
C09	$\texttt{raven}(V) \rightarrow [\langle\texttt{ident(shape),none,none,prog(position),none}\rangle]$	91	4	3	-1.6
C10	$\texttt{raven}(V) \rightarrow [\langle\texttt{ident(shape),none,none,prog(position),none}\rangle]$	91	4	3	-0.5
C11	$\texttt{raven}(V) \rightarrow [\langle\texttt{ident(shape),none,prog(quantity),none,none}\rangle]$	81	4	3	-0.5
C12	$\texttt{raven}(V) \rightarrow [\langle\texttt{ident(shape),none,none,prog(position),none}\rangle]$	83	4	3	1.2
D01	$\texttt{raven}(V) \rightarrow [\langle\texttt{ident(shape),none,none,none,ident(type)}\rangle]$	75	4	3	-2.8
D02	$\texttt{raven}(V) \rightarrow [\langle\texttt{dist3(shape),none,none,none,none}\rangle]$	69	4	2	-2.3
D03	$\texttt{raven}(V) \rightarrow [\langle\texttt{dist3(shape),none,none,none,none}\rangle]$	71	4	2	-2.3
D04	$\texttt{raven}(V) \rightarrow [\langle\texttt{ident(shape),none,none,none,dist3(type)}\rangle]$	94	6	3	-2.1
D05	$\texttt{raven}(V) \rightarrow [\langle\texttt{ident(shape),none,none,none,dist3(type)}\rangle]$	96	6	3	-2.6
D06	$\texttt{raven}(V) \rightarrow [\langle\texttt{ident(shape),none,none,none,dist3(type)}\rangle]$	93	6	3	-2.6
D07	$\texttt{raven}(V) \rightarrow [\langle\texttt{dist3(shape),none,none,none,dist3(type)}\rangle]$	106	6	3	-2.1
D08	$\texttt{raven}(V) \rightarrow [\langle\texttt{dist3(shape),none,none,none,dist3(type)}\rangle]$	91	6	3	-2.0
D09	$\texttt{raven}(V) \rightarrow [\langle\texttt{dist3(shape),none,none,none,dist3(type)}\rangle]$	104	6	3	-1.5
D10	$\texttt{raven}(V) \rightarrow [\langle\texttt{ident(shape),none,none,none,dist3(type)}\rangle]$	93	6	3	-1.4
D11	$\texttt{raven}(V) \rightarrow [\langle\texttt{ident(shape),none,dist3(quantity),none,dist3(type)}\rangle]$	146	6	4	1.1
D12	$\texttt{raven}(V) \rightarrow [\langle\texttt{dist3(shape),none,none,none,dist3(type)}\rangle]$	106	6	3	1.8
E01	$\texttt{raven}(V) \rightarrow [\langle\texttt{addition(shape),none,none,none,none}\rangle]$	61	4	2	-1.5
E02	$\texttt{raven}(V) \rightarrow [\langle\texttt{addition(shape),none,none,none,none}\rangle]$	55	4	2	-1.0
E03	$\texttt{raven}(V) \rightarrow [\langle\texttt{addition(shape),none,none,none,ident(type)}\rangle]$	99	6	3	-1.3
E04	$\texttt{raven}(V) \rightarrow [\langle\texttt{dist2(shape),none,none,none,none}\rangle]$	63	4	2	-0.6
E05	$\texttt{raven}(V) \rightarrow [\langle\texttt{dist2(shape),none,none,none,none}\rangle]$	60	4	2	-0.7
E06	$\texttt{raven}(V) \rightarrow [\langle\texttt{dist2(shape),none,none,none,none}\rangle]$	61	4	2	-0.4
E07	$\texttt{raven}(V) \rightarrow [\langle\texttt{dist2(shape),none,none,none,none}\rangle]$	77	4	2	0.9
E08	$\texttt{raven}(V) \rightarrow [\langle\texttt{dist2(shape),none,none,none,none}\rangle]$	99	4	3	2.9
E09	$\texttt{raven}(V) \rightarrow [\langle\texttt{dist2(shape),none,none,none,dist3(type)}\rangle]$	100	4	3	1.5
E10	$\texttt{raven}(V) \rightarrow [\langle\texttt{dist2(shape),none,none,none,none}\rangle]$	60	4	2	0.6
E11	$\texttt{raven}(V) \rightarrow [\langle\texttt{dist2(shape),none,none,none,none}\rangle]$	65	4	2	0.7
E12	-	-	4	-	1.6

Table 3: Solutions returned, steps needed, number of examples (E^+) that are derived from each problem and the number of different operators $|o_{app}|$ that are applied in order to get the solution in gErl (as a measure of the complexity of the solution) for Raven's SPMs sets C, D & E. The last column, taken from Georgiev (2008), shows the results of the parameter b (difficulty) of an IRT model of human performance on these sets. The last example (E12) is not solved by gErl.

3.2.2. Results

As we see in Table 3, gErl is able to solve 35 out of the 36 problems (12 of 12 in sets C and D, and 11 of 12 in set E). If we take a look at the length and complexity of the solution, we see that the number of steps is usually larger. However, we see no clear correspondence between the size of the pattern and the difficulty humans find on these problems. Hence, the time that gErl requires to solve a problem is explained by the requirement of relations needed to solve it: the more relations needed, the longer it takes to obtain a solution pattern. Otherwise, as we have said, this is not well correlated with the difficulty found by humans, for which the difficulty lies on the complexity of the relation to apply. In fact, some of the examples in set E have a small solution with a small number of steps, but their difficulties for humans are high. However, if we take a look at the constructs we see that most of the examples in set C use the `ident` and the `prog` constructs, yielding easy problems for humans in general. The use of `dist3` does not seem to be a big challenge for most humans either. However, the use of `dist2` seems to be responsible of the higher difficulty for set E, except for the three cases that only use `addition`, which look easier for humans. It seems that the operative constructs play an important role in how difficult these problems are for humans, with `ident` and `prog` being constructs that most individuals have and can use, `dist3` and `addition` being intermediate and `dist2` being a construct that seems to be accessible or be developed by some individuals, in the line of the study carried out by Carpenter et al. (1990), where the authors claim that the problem difficulty not only appears when the number of figures per cell is not constant but also occurs in problems containing a distribution-of-two-values (`dist2`) relation, as well as figure addition (`addition`) and difference (`dist3`). Finally, there are some cases that could be explained

1.	cdcdcdcd_
2.	aaabbbcccdd_
3.	atbataatbat_
4.	abmcdmefmghm_
5.	defgefghfghi_
6.	qxapxbqxa_
7.	aducuaeuabuafua_
8.	mabmbcmcdm_
9.	urtustuttu_
10.	abyabxabwab_
11.	rscdstdetuef_
12.	npaoqapraqsa_
13.	wxaxybyzczadab_
14.	jkqrklrslmst_
15.	pononmnmlmlk_

Figure 6: 15 letter series completion test problems from Simon and Kotovsky (1963).

by a combination of pattern size and constructs, such as the extra difficulty of D11. Also, there are some cases that show a strange result, such as D12, as it has the same pattern as other matrices but the difficulty humans find here is much higher. We have to say that we have not analysed the distractors (how good and plausible the other 7 incorrect given options are, which are given to humans but not to gErl). They may play a contaminating role here as well.

3.3. Letter series completion problems

Thurstone and Thurstone (1941) introduced the letter series completion problems as part of some test batteries. These problems were developed to assess "reasoning ability". The goal of the letter series problems is to identify the following letter in a series (see Fig. 6) from five letter choices. To solve a letter series, an abstract pattern has to be identified that captures the regularity of the sequence. The correct answer can be generated by applying this pattern. As usual in induction problems, there is no generally acceptable concept of correctness. For example, a person might continue a sequence just with some constant letter. Correctness in the context of induction problems typically presupposes that there is one continuation which is most plausible with respect to a short pattern. Typically, problems are carefully constructed in such a way that there is a unique solution. However, there is also research on ambiguous problems, for example in the domain of letter string analogies (Hofstadter, 2008).

3.3.1. Problem representation and constructs

Once again, the first step to deal with Thurstone's letter series problems is to code the examples as equations to be correctly addressed by gErl. Each letter series (*lhs* of the equations) will be coded as a list of characters (or strings). The *rhs* will be the character that follows the series. Below we can see the representation of example 1 in Fig. 6:

$$\mathtt{e}_1 : \mathtt{thurstone}(\text{“}\textit{cdcdcdcd}\text{”}) \to \text{“}\textit{c}\text{”}$$

Since each letter series is a problem itself, we need to provide the system with more than one training instance as we did for the RPM problems. We do that by decomposing each initial example (the input letter series) into several letter series of increasing length. For instance, from $\mathtt{e}_1$ we create the following training instances:

$$\begin{aligned}
&\mathtt{e}_{1,1} : \mathtt{thurstone}(\text{“}\textit{cd}\text{”}) \to \text{“}\textit{c}\text{”}\\
&\mathtt{e}_{1,2} : \mathtt{thurstone}(\text{“}\textit{cdc}\text{”}) \to \text{“}\textit{d}\text{”}\\
&\mathtt{e}_{1,3} : \mathtt{thurstone}(\text{“}\textit{cdcd}\text{”}) \to \text{“}\textit{c}\text{”}\\
&\mathtt{e}_{1,4} : \mathtt{thurstone}(\text{“}\textit{cdcdc}\text{”}) \to \text{“}\textit{d}\text{”}\\
&\mathtt{e}_{1,5} : \mathtt{thurstone}(\text{“}\textit{cdcdcd}\text{”}) \to \text{“}\textit{c}\text{”}\\
&\mathtt{e}_{1,6} : \mathtt{thurstone}(\text{“}\textit{cdcdcdc}\text{”}) \to \text{“}\textit{d}\text{”}\\
&\mathtt{e}_{1,7} : \mathtt{thurstone}(\text{“}\textit{cdcdcdcd}\text{”}) \to \text{“}\textit{c}\text{”}
\end{aligned}$$

In order to determine the set of constructs, we follow the ideas about the basic "subroutines" from (Simon and Kotovsky, 1963; Kotovsky and Simon, 1973) to explain human behaviour in these letter series problem tasks:

- **Sequence**: The problem is a series and needs to be extrapolated: the pattern discovered (symbolic structures built from the vocabulary of such a language) is held in order to continue the generation of the letter series.
- **Letter identity and alphabetical order**: It is assumed that the subjects must know the English alphabet (backward and forward) and are told that it is circular ('a' follows 'z'). That means that they have to know the concepts of letter *identity*, and the *next* and *previous* letters.
- **Periodicity**: Cyclical patterns (or iterations) may be required, e.g., repeat the list *at* to produce *atatatat*.... Finding the length and the positions of where the repetitions start is one of the difficulties of the problem, such as in the series *atbataatbat_*: we can mark it off in segments of length 3 (*ata*, *atb*, *ata* and *at_*). Here we observe that the first and the second position of each segment are occupied, respectively, by an *a* and a *t* which is a symple cycle of *a's* and *t's* (as previously), and the third position is occupied by the cycle *ba ba* ....
- **Composition**: The assembly of two previous components must be needed in order to generate the entire series.

For our purposes, only the first three subroutines will be taken into account (the last ones have to do with implementation aspects of Kotovsky and Simon, 1973). Therefore, we need some operators in order to (a) work with letter sequences (strings), (b) establish inter-letter relations (alphabetical order) and, finally, (c) deal with the periodicity (finding regularly occurring breaks in a given series). As sequences are represented with strings, we need some string operators (a) in gErl:

$$\mu_{replace}(\overline{3}, f_{list_L})$$
$$\mu_{replace}(\overline{3}, f_{list_R})$$

to replace the *rhs* of the rules (position 3) by an application of any of the (Erlang) built-in functions f_{list} that works with lists: `head` (which returns the first element of the list), `tail` (which returns the list without the first element), `last` (which returns the last element of the list) and `init` (which returns the list without the last element). This can be applied either over the input ($\rho|_{1.1}$) letter series string ($f_{list_L}(\rho) = f_{list}(\rho|_{1.1})$) or over the *rhs* ($\rho|_3$) of ρ ($f_{list_R}(\rho) = f_{list}(\rho|_3)$). Note that this latter function allows the system to construct rules whose *rhs* are the result of successive applications of different functions since it takes whatever there is in the *rhs* of ρ as input of the new function to allocate there. This will also make it possible to deal with periodicity.

For handling the alphabetical order (b) we can also use the meta-operator $\mu_{replace}$, instantiated as:

$$\mu_{replace}(\overline{3}, f_{order})$$

where f_{order} is an alphabetical order function, either `previous` or `next` of a specific letter. Note that `next`("*z*") = "*a*" and `previous`("*a*") = "*z*".

Finally, in order to deal with the composition (c), we use the meta-operator $\mu_{condition}$ to generate operators in charge of inserting Boolean conditions (guards) to the rules. In this way, some parts of the sequences are handled differently. In order to distinguish different parts of the sequence we use the position relative to the length of the list. This makes it possible to learn problems with more than one pattern (for instance "abxcdx", where the following letter is "x" if the position of the missing letter is a multiple of 3, and the application of `next` to the last letter, otherwise). So we used the following two conditions.

$$\texttt{length}(L)\ \texttt{mod}\ \varphi = 0$$
$$\texttt{length}(L)\ \texttt{mod}\ \varphi \neq 0$$

where the period $\varphi \in \{2, 3, \dots\}$. Both conditions are defined in the background knowledge as several functions mod_φ and not_mod_φ (respectively), which apply over the input letter sequence parameter in the *lhs* of the rules. Hence, the operators are defined as

$$\mu_{condition}(\overline{2}, mod_\varphi)$$
$$\mu_{condition}(\overline{2}, not\_mod_\varphi)$$

where $mod_{\varphi}(\rho) = \texttt{length}(\rho|_{1.1}) \texttt{ mod } \varphi = 0$ and $not\_mod_{\varphi}(\rho) = \texttt{length}(\rho|_{1.1}) \texttt{ mod } \varphi \neq 0$, generating as many operators as periods φ we have.

As we did for the RPM problems, we also need generalisation operator that will be applied to generalise the input attribute, $\mu_{replace}(\overline{1.1}, \overline{V_{string}})$.

The following example illustrates how the operators are applied to solve the letter series problem $e_{1,3}$:

$$\begin{aligned} op_3(op_2(op_1(\texttt{thurstone}(\text{"}cdcd\text{"}) \rightarrow \text{"}c\text{"}))) &\Rightarrow \\ op_3(op_2(\texttt{thurstone}(V_{string}) \rightarrow \text{"}c\text{"})) &\Rightarrow \\ op_3(\texttt{thurstone}(V_{string}) \rightarrow \texttt{init}(V_{string})) &\Rightarrow \\ \texttt{thurstone}(V_{string}) \rightarrow \texttt{last}(\texttt{init}(V_{string})) & \end{aligned}$$

where

$$\begin{aligned} op_1 &\equiv \mu_{replace}(\overline{1.1}, \overline{V_{String}}) \\ op_2 &\equiv \mu_{replace}(\overline{3}, f_{init_L}) \\ op_3 &\equiv \mu_{replace}(\overline{3}, f_{last_R}) \end{aligned}$$

It is easy to see that the example $\texttt{e}_{1,3}$ follows a regular pattern just alternating the characters "c" and "d", so the last rule obtained $\texttt{thurstone}(V_{string}) \rightarrow \texttt{last}(\texttt{init}(V_{string}))$ returns the right solution whatever the input is (it covers all seven training instances generated from example $\texttt{e}_{1,3}$). For instance, if V_{string} = "*cdcdc*", then $\texttt{init}(V_{string})$ = "*cdcd*" and $\texttt{last}(\texttt{init}(V_{string}))$ = "*d*", which is the correct and general solution for all letter series that follow the same pattern as the previous example.

3.3.2. Results

gErl has been tested on the same 15 problems of the Thurstone Letter Series Completion task (Fig. 6) from Simon and Kotovsky (1963). With the operators that were provided, gErl learns 14 of the 15 test sequences, as shown in Table 4. As we see in the table, the size of the solution (represented by $|o_{app}|$ or their sum when there are two rules) is not related to the difficulty humans find in these exercises (DiffH). In fact, there is no clear trend or correlation between any pair of the three last columns. If we take a look at the constructs, we see that all solutions use string operator constructs, such as `init` and `last`. Only problems 1, 3, 6 do not use the letter order (`next` and `previous`), which may explain why they are easy for humans. The use of composition does not seem to add too much complexity to humans, as problem 4, for instance, is easy for humans, and problems 8 and 10 are not very difficult. Finally, one of the most difficult problems for humans (15) is relatively short and simple in its functional representation. In fact, there seems to be some contamination in the degree of difficulty for humans, depending on which letters in the alphabet are used in each problem. It seems that problems that have patterns involving the first letter of the alphabet are easier for humans than those involving other letters. This possible explanation is not reflected by any of the computer programs used by Simon and Kotovsky, or by the way the problems are presented to gErl.

4. Discussion

In the previous section we have seen how several intelligence test problems are addressed by a general learning system, which uses a declarative, rule-based representation language for examples, patterns and operators. This representation language and the generality of the learning process (which does not have any hard-wired operator) make the complexity of each pattern explicit and the operators that are used for each problem. This provides useful information about the elements that each problem really requires: more computational power (in terms of working memory and combinatorial search) or some basic operational constructs. Many previous works in the literature that have analysed the complexity of intelligence test problems using computational models have focused on the former. Here, we are interested in the latter.

It should be noted that, when we analyse and determine the complexity of intelligence test problems based on problem-dependent characteristics rather than user-dependent ones, we should ensure that we generate and test all possible solution candidates (program outputs in order of their complexities) for a given problem until the minimal one (the shortest) is returned, while keeping the execution time of the solution under some reasonable terms. Actually, we consider a relation between space or length of the solution (L) and its execution time (T) as follows: $LT = L + \log(T)$.

Id	Problem	Solution	Steps	E^+	$\lvert o_{app}\rvert$	DiffH	DiffC
1.	cdcdcdcd_	thurstone(V) $\rightarrow$ last(init(V))	42	5	3	0.03	0.00
2.	aaabbbcccdd_	thurstone(V) $\rightarrow$ next(init(init(V)))	91	9	4	0.08	0.25
3.	atbataatbat_	thurstone(V) $\rightarrow$ last(init(init(init(init(init(V))))))	131	7	7	0.11	0.75
4.	abmcdmefmghm_	thurstone(V) when length(V) mod 3 = 0 $\rightarrow$ last(init(init(V)))	174	8	5	0.13	0.25
		thurstone(V) when length(V) mod 3 $\neq$ 0 $\rightarrow$ next(init(V))	119		4		
5.	defgefghfghi_	thurstone(V) $\rightarrow$ next(init(init(init(V))))	105	8	5	0.40	0.75
6.	qxapxbqxa_	thurstone(V) $\rightarrow$ last(init(init(init(init(init(V))))))	129	7	7	0.29	0.00
7.	aducuaeuabuafua_	-	-	-	-	0.59	1.00
8.	mabmbcmcdm_	thurstone(V) when length(V) mod 3 = 0 $\rightarrow$ last(init(init(V)))	165	8	5	0.27	0.25
		thurstone(V) when length(V) mod 3 $\neq$ 0 $\rightarrow$ next(init(init(V)))	141		5		
9.	turtustuttu_	thurstone(V) when length(V) mod 3 = 0 $\rightarrow$ next(init(init(V)))	192	9	5	0.39	0.25
		thurstone(V) when length(V) mod 3 $\neq$ 0 $\rightarrow$ last(init(init(V)))	185		5		
10.	abyabxabwab_	thurstone(V) when length(V) mod 3 = 0 $\rightarrow$ previous(init(init(V)))	182	9	5	0.24	0.50
		thurstone(V) when length(V) mod 3 $\neq$ 0 $\rightarrow$ last(init(init(V)))	171		5		
11.	rscdstdetuef_	thurstone(V) $\rightarrow$ next(last(init(init(init(V)))))	154	9	6	0.46	0.75
12.	npaoqapraqsa_	thurstone(V) when length(V) mod 3 = 0 $\rightarrow$ last(init(init(V)))	123	8	5	0.40	0.75
		thurstone(V) when length(V) mod 3 $\neq$ 0 $\rightarrow$ next(init(init(V)))	141		5		
13.	wxaxybyzczadab_	thurstone(V) $\rightarrow$ next(init(init(V)))	99	9	4	0.37	0.75
14.	jkqrklrslmst_	thurstone(V) $\rightarrow$ next(init(init(init(V))))	103	9	5	0.32	0.75
15.	pononmnmlmlk_	thurstone(V) $\rightarrow$ previous(init(init(V)))	112	9	5	0.51	0.75

Table 4: Solutions returned by gErl and steps needed to learn the series completion test problems from Simon and Kotovsky (1963). We see that the solutions can extrapolate the sequences indefinitely. Except for problem 7 all extrapolations are correct. We also show the number of examples (E^+) that are derived from each problem and the number of operators $|o_{app}|$ that are applied to get the solution in gErl (as a measure of the complexity of the solution). The last two columns (DiffH and DiffC) show the difficulty (on a scale between 0 and 1) as the proportion of humans that missed the solutions (among the 12+67 cases in Simon and Kotovsky (1963, Table 4)) and among the 4 computer programs in Simon and Kotovsky (1963, Table 3) respectively.

This is closely related to a Levin search (Levin, 1973, 1984) (see Li and Vitányi, 2008 for an overview), which, for a broad class of search problems, can be shown to be optimal with respect to total search time (leaving aside a constant factor independent of the problem size). Despite this strong result, Levin search will fail to solve problems whose solutions all have a high value of LT, which is highly dominated by L. Because of this infeasibility of Levin search, we need to use an approximation. gErl is an approximation of this search. This, of course, might lead to a solution that is not the optimal in terms of LT, thus giving an overestimation of the difficulty of the search problem.

A related issue is whether the solutions are efficient or get more and more efficient after use. This is not related to fluid intelligence but rather to crystallised intelligence. Given a solution or policy to address a problem (e.g., multiplication), the repeated use of the algorithm will lead to 'routinisation', i.e., a phenomenon in humans and non-human animals by which those mechanisms that have been so well mastered and regularly replicated no longer require mindful manipulation. A possible future work would be to analyse that, instead of the optimal solution in terms of LT (which is usually approximated as the shortest solution in the gErl representation), some program transformation techniques could be used to render it more efficient (but semantically equivalent). For instance, gErl may find that the solution for a problem is to sort a sequence of letters. That solution could be expressed with a very short, but inefficient, sorting algorithm. A further improvement —similar to a routinisation— could be to transform the sorting algorithm into another more efficient one.

The problems seen in the previous section feature the following operational constructs (in parenthesis A, B, C referring to odd-one-out, RPM and letter series problems respectively): differences intra and inter items (in A, B), the notion of being distinct (A, B), the concept of identity (A, B, C), the ideas of combination/addition (A, B, C), the sense of sequence and progression (B, C), the notion of order (C) and the notion of periodicity or repetition (C). It seems that if a system lacks many of these concepts, these problems become irresolvable (unless the system can *invent* or *discover* all these concepts). In fact, gErl is not able to solve them when we remove the constructs. On the other hand, if we artificially provide these constructs, even if the system, such as gErl, does not have a real cognitive development or physical embodiment, we can get excellent results.

A very different thing is how much time gErl takes to solve the problem *when it can solve the problem*. For instance, the Pearson correlation coefficient between the number of different operators $|o_{app}|$ (a measure of the complexity of the pattern) and the time gErl takes (in number of steps) is 0.907 for the RPM problems and 0.944 for the letter completion problems[5]. In other words, the time a problem requires does correlate with the combinatorial problem of using the operational constructs, but solving it or not correctly may mostly depend on the constructs. This suggests that many studies about intelligence tests are conflating two components of the difficulty of a problem: whether we have the pieces and how many pieces we need to combine. In fact, in cognitive development we are interested in the set of elements and constructs that evolves with time rather that the computational ability of combining them.

This leads to several observations. If we focus on the evaluation of cognitive development in humans, we see clearly that intelligence tests for small children usually differ in presentation but also in the constructs that are required. For instance, even the easy RPM matrices of the standard series for adult humans may be impossible for children under 6. Note that we are not referring here to psychomotor abilities but pure fluid ability, where the concepts of identity, difference, order, repetition, etc., may still be under development. Naturally, some of these constructs are acquired by interacting with the world and with the use of language, as many of them are not innate. As these cognitive constructs are useful in a wide variety of problems, fluid intelligence (and not only crystallised intelligence) increases when these constructs are developed or learnt.

If we focus on the evaluation of cognitive development in artificial systems, this work confirms that looking at the score of a (non-human) system on a human intelligence test can be very misleading, as already anticipated a decade ago by Sanghi and Dowe (2003b). Unlike the other computer models solving IQ tests analysed by Hernández-Orallo et al. (2016), gErl is the first system (apart from Sanghi and Dowe (2003b)) that is able to deal with more than one IQ test. But, unlike Sanghi and Dowe (2003b), the system has not been designed on purpose for intelligence tests, but rather as a general learning system. In other words, gErl *learns* to solve the problems. For instance, for the RPM problems, gErl had 59/60 hits on sets *C*, *D* & *E*. Since humans who performed well on these sets typically got a perfect score in the easier sets *A* and *B* (Raven and Court, 1996, Table SPM2), we *could* infer that gErl is in the 95th percentile for American adults (IQ: 140), according to the 1993 norms (Raven and Court, 1996). This is not very meaningful, as we know that gErl has some learning abilities, but it is *not* really intelligent. We can always find explanations in the way the problems are represented (where the complex visual representation is simplified), but our results suggest that a better explanation is just to look at the constructs that are provided to the system.

Nonetheless, it has to be said that if we provide many constructs in a huge background knowledge base, the difficulty would then lie in designing the system in such a way that it can choose among them in an efficient way, one of the most challenging problems in AI and machine learning today. This suggests that artificial cognitive systems, if their computational power remains the same (no change of hardware or basic algorithms), should not become faster with time for those problems they were able to solve in previous cognitive stages. In fact, it may still happen that efficiency can be degraded if the system has a much larger knowledge base so that retrieving the appropriate constructs becomes more difficult. This of course assumes that we evaluate the systems without over-training on the particular tasks, as the system can just change its contextual priorities if some kinds of exercises have been presented to the system immediately before the evaluation[6].

In general terms, for both humans and machines, are these (and other) human intelligence tests useful to evaluate cognitive development? As a take-away message, we do think that some of them *can* be useful. For the three kinds of problems we have seen here, the odd-one-out problem will have less interest than the two other problems, as we can only evaluate two particular constructs with the odd-one-out problem, and the constructs are relatively elaborated. In contrast, the other two types of problems (RPM and letter series) have a diversity of constructs. In order to make these intelligence tests more useful we need to create categories about the constructs they use. These categories can be created (less subjectively) by the use of declarative learning systems such as gErl. Also, we have to be very careful to separate success/failure with speed of resolution in the analysis of results, and the evolution of both speed and success for the whole cognitive life of the system. Let us remember that gErl is not a cognitive model and it is not based in how humans learn or develop. When comparing systems (humans, machines or hybrid), we can potentially discover

[5]Since we only have two odd-one-out problem types and actually just two different learning problems, we do not have a meaningful value for the correlation in this case.

[6]It is very difficult to expect that an underdeveloped subject (e.g., a child) could ever guess the solution for some of the problems that require complex operational concepts. This is related to the notion of 'spontaneous overtraining', that is able to get adult performance in preschoolers. It is also relevant to consider the studies about the blocking of some constructs, which can make adults perform like preschoolers (Sirois and Shultz, 2006). One way of obtaining a similar result with artificial systems would be by giving or training the system with the necessary constructs on one case and forbidding (or making forget) the constructs on the other case.

whether they share the same constructs or not, and identify whether the difference in speed and success is given by a higher or lower computational power or by the disposition and better handling of cognitive operational constructs.

There are of course some shortcomings about the use of human intelligence tests for machines, as already pointed out in Dowe and Hernández-Orallo (2012), but most criticisms are related to an anthropocentric choice of the exercises and an anthropocentric determination of their difficulty and scales. We have seen that through the use of general declarative systems where constructs, patterns and examples are explicit and intelligible, we can find non-anthropocentric criteria about what exercises we choose and what constructs and computational effort they require. This indicates that many tasks that have been devised in traditional intelligence tests can be *reused*. This is a compatible alternative to the conception of more principled and better grounded tests based on a mathematical basis (Hernández-Orallo, 2000; Legg and Hutter, 2007; Hernández-Orallo and Dowe, 2010) for both actual (Hernández-Orallo et al., 2014) and potential (Hernández-Orallo and Dowe, 2013) capabilities.

Other limitations are intrinsic to this work. More tests could be analysed and other (preferably declarative) systems could be used to see whether the identification of required constructs is convergent with this work. Also, the ability of acquiring constructs (and consolidating them for future use) has not been analysed in this work and it is a key issue in any cognitive development. While we do not provide any new insight about how this could be done, we think that better understanding and tools for the measuring of cognitive development are crucial for the progress of this field.

In concluding, this work supports the assumption that, even for fluid intelligence tests, the difficult items require a more advanced cognitive development than the simpler ones. This work also encourages the use of general intelligence tests for the evaluation of cognitive development of humans and machines, but with extra care when evaluating the latter. In contrast, this work has raised many doubts about the use for artificial systems of the difficulty gradation and the stage arrangement used for humans, as the pace and sequence of acquisition of constructs may differ dramatically between humans and artificial cognitive systems.

Acknowledgments

This work has been partially supported by the EU (FEDER) and the Spanish MINECO under grants TIN 2015-69175-C4-1-R and TIN 2013-45732-C4-1-P, and by Generalitat Valenciana under grant PROMETEOII/2015/013.

References

Anderson, J., Lebiere, C., 2003. The Newell test for a theory of cognition. Behavioral and Brain Sciences 26 (5), 587–601. 2

Baltes, P. B., 1987. Theoretical propositions of life-span developmental psychology: On the dynamics between growth and decline. Developmental psychology 23 (5), 611. 4

Bringsjord, S., 2011. Psychometric artificial intelligence. Journal of Experimental & Theoretical Artificial Intelligence 23 (3), 271–277. 1

Bringsjord, S., Schimanski, B., 2003. What is artificial intelligence? Psychometric AI as an answer. In: International Joint Conference on Artificial Intelligence. pp. 887–893. 1

Calvo-Garzón, F., 2003. Nonclassical connectionism should enter the decathlon. Behavioral and Brain Sciences 26 (05), 603–604. 2

Carpenter, P. A., Just, M. A., Shell, P., 1990. What one intelligence test measures: A theoretical account of the processing in the raven progressive matrices test. Psychological review 97, 404–431. 10, 12

Dehaene, S., Izard, V., Pica, P., Spelke, E., Jan. 2006. Core Knowledge of Geometry in an Amazonian Indigene Group. Science 311 (5759), 381–384. 6

Detterman, D. K., 2011. A challenge to Watson. Intelligence 39 (2-3), 77 – 78. 1

Dowe, D. L., Hernández-Orallo, J., 2012. IQ tests are not for machines, yet. Intelligence 40 (2), 77–81. 1, 3, 5, 18

Eliasmith, C., Stewart, T. C., Choo, X., Bekolay, T., DeWolf, T., Tang, C., Rasmussen, D., 2012. A large-scale model of the functioning brain. Science 338 (6111), 1202–1205. 1

Epstein, H. T., 1979. Correlated brain and intelligence development in humans. Development and evolution of brain size: Behavioral implications, chapter 6 in Development and Evolution of Brain Size: Behavioral Implications, edited by Martine Hahn, 111–131. 3

Evans, T., 1963. A heuristic program of solving geometric analogy problems. Ph.D. thesis, Mass. Inst. Tech., Cambridge, Mass., U.S.A., also available from AF Cambridge Research Lab, Hanscom AFB, Bedford, Mass., U.S.A.: Data Sciences Lab, Phys and Math Sci Res Paper 64, Project 4641. 5

Evans, T., 1965. A heuristic program to solve geometric-analogy problems. In: Proc. SJCC. Vol. 25. pp. 327–339, vol. 25. 5

Ferrucci, D., Brown, E., Chu-Carroll, J., Fan, J., Gondek, D., Kalyanpur, A. A., Lally, A., Murdock, J., Nyberg, E., Prager, J., et al., 2010. Building Watson: An overview of the DeepQA project. AI Magazine 31 (3), 59–79. 1

Flener, P., Schmid, U., Mar. 2008. An introduction to inductive programming. Artif. Intell. Rev. 29 (1), 45–62. 6

Georgiev, N., 2008. Item analysis of *c*, *d* and *e* series from Raven's standard progressive matrices with item response theory two-parameter logistic model. Europe's Journal of Psychology 4 (3). 12

Griffith, S., Sinapov, J., Sukhoy, V., Stoytchev, A., 2012. A behavior-grounded approach to forming object categories: Separating containers from noncontainers. Autonomous Mental Development, IEEE Transactions on 4 (1), 54–69. 4

Hernández-Orallo, J., 2000. Beyond the Turing Test. J. Logic, Language & Information 9 (4), 447–466. 2, 18

Hernández-Orallo, J., Dowe, D. L., 2010. Measuring universal intelligence: Towards an anytime intelligence test. Artificial Intelligence 174 (18), 1508 – 1539. 2, 18

Hernández-Orallo, J., Dowe, D. L., 2013. On potential cognitive abilities in the machine kingdom. Minds and Machines 23 (2), 179–210. 2, 18

Hernández-Orallo, J., Dowe, D. L., Hernández-Lloreda, M. V., 2014. Universal psychometrics: Measuring cognitive abilities in the machine kingdom. Cognitive Systems Research 27, 5074. 18

Hernández-Orallo, J., Martínez-Plumed, F., Schmid, U., Siebers, M., Dowe, D. L., 2016. Computer models solving intelligence test problems: Progress and implications. Artificial Intelligence 230, 74 – 107.
URL `http://www.sciencedirect.com/science/article/pii/S0004370215001538` 5, 17

Hofstadter, D. R., 2008. Fluid concepts and creative analogies: Computer models of the fundamental mechanisms of thought. Basic Books. 13

Holland, J. H., Booker, L. B., Colombetti, M., Dorigo, M., Goldberg, D. E., Forrest, S., Riolo, R. L., Smith, R. E., Lanzi, P. L., Stolzmann, W., et al., 2000. What is a learning classifier system? In: Learning Classifier Systems, LNCS, vol. 1813. Springer, pp. 3–32. 20

Kaufman, A. S., Lichtenberger, E. O., 2006. Assessing adolescent and adult intelligence. Wiley. 3

Keedwell, E., 2010. Towards a staged developmental intelligence test for machines. In: Towards a Comprehensive Intelligence Test (TCIT): Reconsidering the Turing Test for the 21st Century Symposium. pp. 28–32. 2

Kitzelmann, E., 2009. Inductive programming: A survey of program synthesis techniques. In: International Workshop on Approaches and Applications of Inductive Programming. Springer, pp. 50–73. 6

Kotovsky, K., Simon, H. A., 1973. Empirical tests of a theory of human acquisition of concepts for sequential patterns. Cognitive Psychology 4 (3), 399 – 424. 14

Langley, P., 2011. Artificial intelligence and cognitive systems. AISB Quarterly. 2

Legg, S., Hutter, M., 2007. Universal intelligence: A definition of machine intelligence. Minds and Machines 17 (4), 391–444. 18

Levin, L. A., 1973. Universal sequential search problems. Problemy Peredachi Informatsii 9 (3), 115–116. 16

Levin, L. A., 1984. Randomness conservation inequalities; information and independence in mathematical theories. Information and Control 61 (1), 15–37. 16

Li, M., Vitányi, P. M., 2008. An Introduction to Kolmogorov Complexity and Its Applications, 3rd Edition. Springer Publishing Company. 16

Martínez-Plumed, F., Ferri, C., Hernández-Orallo, J., Ramírez-Quintana, M. J., 2013a. Learning with configurable operators and RL-based heuristics. In: NFMCP. Vol. 7765 of LNCS. Springer-Verlag, pp. 1–16. 2, 6, 20

Martínez-Plumed, F., Ferri, C., Hernández-Orallo, J., Ramírez-Quintana, M. J., 2013b. On the definition of a general learning system with user-defined operators. CoRR abs/1311.4235. 2, 6, 20

Mueller, S., 2008. Is the Turing Test still relevant? a plan for developing the cognitive decathlon to test intelligent embodied behavior. In: 19th Midwest Artificial Intelligence and Cognitive Science Conference, MAICS. pp. 8–15. 2

Mueller, S., Jones, M., Minnery, B., Hiland, J. M., 2007. The BICA cognitive decathlon: A test suite for biologically-inspired cognitive agents. In: Proceedings of Behavior Representation in Modeling and Simulation Conference, Norfolk. pp. 8–15. 2

Neisser, U., Boodoo, G., Bouchard Jr, T. J., Boykin, A. W., Brody, N., Ceci, S. J., Halpern, D. F., Loehlin, J. C., Perloff, R., Sternberg, R. J., et al., 1996. Intelligence: Knowns and unknowns. American psychologist 51 (2), 77. 3

Oudeyer, P.-Y., Kaplan, F., 2007. Dialog: How can we assess open-ended development? The Newsletter of the Autonomous Mental Development Technical Committee 4 (1), 1–3. 2, 3

Ragni, M., Neubert, S., 2012. Solving ravens IQ-tests: An AI and cognitive modeling approach. In: ECAI. IOS Press, pp. 666–671. 9, 11

Raven, J., Court, J., 1996. Manual for Raven's Progressive Matrices and Vocabulary Scales: Standard progressive matrices. Manual for Raven's Progressive Matrices and Vocabulary Scales. Oxford Psychologists Press. 4, 9, 17

Ruiz, P. E., 2011. Building and solving odd-one-out classification problems: A systematic approach. Intelligence 39 (5), 342–350. 6, 7, 8

Sanghi, P., Dowe, D. L., 2003a. A computer program capable of passing IQ tests. In: 4th Intl. Conf. on Cognitive Science (ICCS'03), Sydney. pp. 570–575. 1, 3, 5

Sanghi, P., Dowe, D. L., 13-17 July 2003b. A computer program capable of passing I.Q. tests. In: Slezak, P. P. (Ed.), Proc. of the Joint International Conference on Cognitive Science, 4th ICCS International Conference on Cognitive Science & 7th ASCS Australasian Society for Cognitive Science (ICCS/ASCS-2003). Sydney, NSW, Australia, pp. 570–575. 17

Schenck, C., 2013. Intelligence tests for robots: Solving perceptual reasoning tasks with a humanoid robot. Master's thesis, Iowa State University. 2

Schmid, U., Kitzelmann, E., 2011. Inductive rule learning on the knowledge level. Cognitive Systems Research 12 (34), 237 – 248, special Issue on Complex Cognition. 5, 6

Simon, H. A., Kotovsky, K., 1963. Human acquisition of concepts for sequential patterns. Psychological Review 70 (6), 534. 13, 14, 15, 16

Simpson Jr, R., Twardy, C., 2008. Refining the cognitive decathlon. In: Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems. ACM, pp. 124–131. 2

Sinapov, J., Stoytchev, A., 2010. The odd one out task: Toward an intelligence test for robots. In: Development and Learning (ICDL), 2010 IEEE 9th International Conference on. IEEE, pp. 126–131. 2, 4

Sirois, S., Shultz, T. R., 2006. Preschoolers out of adults: Discriminative learning with a cognitive load. The quarterly Journal of experimental psychology 59 (8), 1357–1377. 17

Sternberg, R. J., 2008. Increasing fluid intelligence is possible after all. Proceedings of the National Academy of Sciences 105 (19), 6791–6792. 3

Sutton, R., 1998. Reinforcement Learning: An Introduction. MIT Press. 6

Sutton, R. S., Barto, A. G., 1998. Reinforcement learning: An introduction. MIT press. 22

Thurstone, L., Thurstone, T., 1941. Factorial studies of intelligence. Psychometrika monograph supplements. The University of Chicago press. 4, 13

Turing, A. M., 1950. Computing machinery and intelligence. Mind 59, 433–460. 2

Virding, R., Wikström, C., Williams, M., 1996. Concurrent programming in ERLANG (2nd ed.). Prentice Hall International (UK) Ltd., Hertfordshire, UK,. 6, 20

von Ahn, L., Blum, M., Langford, J., 2004. Telling humans and computers apart automatically. Communications of the ACM 47 (2), 56–60. 2

von Ahn, L., Maurer, B., McMillen, C., Abraham, D., Blum, M., 2008. RECAPTCHA: Human-based character recognition via web security measures. Science 321 (5895), 1465. 2

Wallace, C. S., Boulton, D. M., 1968. An information measure for classification. Computer Journal 11 (2), 185–194. 20

Watkins, C., Dayan, P., 1992a. Q-learning. Machine Learning 8, 279–292. 6

Watkins, C., Dayan, P., 1992b. Q-learning. Machine Learning 8, 279–292. 22

Wechsler, D., 1958. The Measurement and Appraisal of Adult Intelligence . Williams & Wilkins Co. 3

Yong, E., November 2012. A large-scale model of the functioning brain. Nature 29. 1

Zentall, T., Hogan, D., Edwards, C., 1980. Oddity learning in the pigeon: Effect of negative instances, correction, and number of incorrect alternatives. Animal Learning & Behavior 8 (4), 621–629. 6

Zentall, T., Hogan, D., Holder, J., 1974. Comparison of two oddity tasks with pigeons. Learning and Motivation 5 (1), 106 – 117. 6

A. The gErl System

The gErl system (Martínez-Plumed et al., 2013a,b) is a highly general and declarative learning system that can be configured with different (possibly user-defined) operators depending on the problem, data representation and the way the information should be navigated. In what follows, we will make a short account of how gErl works (for more information about the system, see Martínez-Plumed et al. (2013b)).

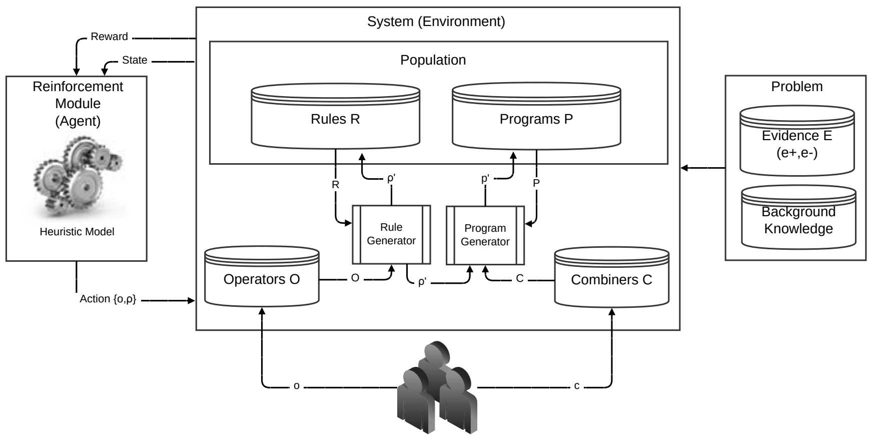

Figure 7: gErl's system architecture

A.1. Architecture and rule representation

Fig. 7 shows gErl's architecture, which works with populations of rules (expressed as unconditional / conditional equations) and programs (sets of rules) in the functional programming language Erlang, Virding et al. (1996). Rules and programs *evolve* as in an evolutionary programming setting or a learning classifier system (Holland et al., 2000). Operators are applied to rules for generating new rules, which are then combined with existing programs. With appropriate operators, using a complexity and compression (Minimum Message Length, MML) principle (Wallace and Boulton, 1968) as the main component of the optimality criterion (which feeds a rewarding module), and using a reinforcement learning-based heuristic (where the application of an operator over a rule is seen as a decision problem fed by the optimality criterion) many complex problems can be solved. gErl has been applied to some inductive programming problems involving deep structures and recursion.

A subset of the functional programming language Erlang is used as knowledge representation language to represent theories and examples in an understandable way: examples as equations, patterns as rules, models as sets of rules, and operators. The advantages of the use of the same knowledge representation language has been previously shown by the fields of ILP, IFP and IFLP. Hence, this a flexible language with powerful features for defining operators and

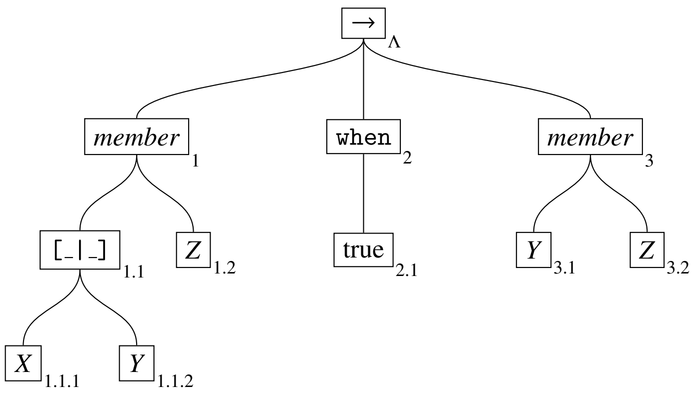

Figure 8: Position tree of the rule $member([X|Y], Z)$ `when` $true \rightarrow member(Y, Z)$. Positions are placed at the bottom-right of each term. The position tree of a rule ρ matches exactly with the Erlang's*Abstract Syntax Tree* (AST): each node of the tree denotes a component occurring in the source code.

able to represent all other elements (theories and examples) in an understandable way. Below we get a more formal account of its notation.

Let Σ be a set of *function symbols* together with their arity and $\mathcal{X}$ a countably set of *variables*, then $\mathcal{T}(\Sigma, \mathcal{X})$ denotes the set of *terms* built from Σ and $\mathcal{X}$. Following Erlang syntax, constant terms begin with a lower-case letter and variables begin with an upper-case letter. $\mathcal{R}$ denotes the space of all (conditional) functional rules ρ expressed as l [when G] $\rightarrow r$, where terms l and r are the left hand side (*lhs*) and the right hand side (*rhs*) of ρ, respectively, and $G = \{g_1, g_2, \ldots g_m \mid m \geq 0\}$) is a set of conditions or Boolean expressions called guards. $\mathcal{P}$ denotes the space of all possible functional programs formed by sets of rules $\rho \in \mathcal{R}$. An example e is a ground rule $l \rightarrow r$ being r in normal form with both l and r are ground. We say that e is covered by a program ω (denoted by $\omega \models e$) if l and r have the same normal form with respect to ω. A program $\omega \in \mathcal{P}$ is a solution of a learning problem defined by a set of positive examples E^+, a (possibly empty) set of negative examples E^- and a background theory K if it covers all positive examples, $K \cup \omega \models E^+$ (completeness), and does not cover any negative example, $K \cup \omega \not\models E^-$ (consistency).

We represent rules as labelled trees we called *position trees*. A *position* $p \in \mathcal{P}os$ is a (possibly empty) sequence of natural numbers that denotes a subtree in the position tree, where Λ (the empty sequence) denotes the entire tree. The subpart of rule ρ at position $p \in Pos(\rho)$ is denoted as $\rho|_p$. Fig. 8 shows the position tree of the rule ρ: $member([X|Y], Z)$ `when` $true \rightarrow member(Y, Z)$. As we can see, $\rho|_{1.1}$ is the term $[X|Y]$ and $\rho|_{3.2}$ is the term Z.

The definition of customised operators is one of the key concepts of gErl. An operator $o \in O$ is a function $o : \mathcal{R} \rightarrow 2^{\mathcal{R}}$. Roughly speaking, operators perform modifications over any of the subparts of a rule in order to generalise or specialise it. For defining (user) operators, the system is equipped with meta-level facilities called *meta-operators*, which are higher-order functions that take as input a (set of) position(s) (as given by its position tree) and a term, and return an operator. gErl provides the following two meta-operators:

1. $\mu_{replace}(pos, f)$ defines an operator that replaces the subparts $\rho|_p, p \in pos(\rho)$ by $f(\rho)$. Notice that this meta-operator can be used to define both generalisation and specialisation operators (depending on whether $f(\rho)$ is more general/specific than $\rho|_p$). Let us see an example: given an instance ρ_1: `adult(20) -> true`, and the operator defined as $op_1 \equiv \mu_{replace}(\overline{1.1}, X)$ whose goal is to replace the first argument in the *lhs* of a rule by a variable, the result of its application over the previous instance will be $op_1(e_1) \Rightarrow$ `adult(X) -> true`.
2. $\mu_{condition}(pos, f)$ defines an operator that inserts a Boolean condition $f(\rho)$ in all the positions $p \in pos(\rho)$. As gErl only allows the insertion of Boolean conditions in the guard of the rules, *pos* is a constant function that always returns position 2 (which denotes the conditions of a rule). Notice that this meta-operator can only be used to define specialisation operators. Let us see an example: given the previous generated rule ρ_2: `adult(X) -> true`, and the operator $op_2 \equiv \mu_{condition}(\overline{2}, g)$ where $g(\rho) = \rho|_{1.1} > 18$ which goal is to add a new guard, its application over the previous rule will be $op_2(e_2) \Rightarrow$ `adult(X) when X > 18 -> true`.

Note that, to simplify notation, any constant function $fun(\cdots) = const$ is denoted as $\overline{const}$.

A.2. RL-based heuristics

The freedom given to the user concerning the definition of their own operators implies the impossibility of defining specific heuristics to explore the search space. This means that heuristics must be overhauled as the problem of deciding the operator that must be used (over a rule) at each particular state of the learning process. A reinforcement learning (RL) (Sutton and Barto, 1998) approach is used instead, where the population of rules at each step of the learning process can be seen as the *state* of the system and the selection of the tuple operator and rule can be seen as the *action*.

As we can see in Fig. 7, gErl works with a set of rules $R \subseteq \mathcal{R}$, a set of programs $P \subseteq \mathcal{P}$ and a set of operators $O \subseteq \mathcal{O}$. Initially, R is populated with the positive evidence E^+ and P is populated with as many unitary programs as rules there are in R. As the process progresses, new rules and programs will be generated. First, the *Rule Generator* module (Fig. 7) takes the operator o and the rule ρ returned by the *Reinforcement Learning Module* (policy) and generates a new rule ρ' which is added to R. Then, the *Program Generator* module combines ρ' (if appropriate) with an existing program ω to create a new program ω' (which is added to P). Formally, we define a state at each iteration or step t of the system as a tuple $\sigma_t = \langle R, P\rangle$ that represents the population of rules and programs in t. An action is a tuple $\langle o, \rho\rangle$ with $o \in O$ and $\rho \in R$ that represents the operator o to be applied to the rule ρ. Our decision problem is a four-tuple $\langle \mathcal{S}, \mathcal{A}, \tau, \psi\rangle$ where: $\mathcal{S}$ is the state space; $\mathcal{A}$ is a finite actions space ($\mathcal{A} = \mathcal{O} \times \mathcal{R}$); $\tau : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is a transition function between states and $\psi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function. At each step t, the system is in a state $s_t \in \mathcal{S}$ and the reinforcement learning policy π selects an action $a_t \in \mathcal{A}$ to execute. As a result, the state changes into $s_{t+1} = \tau(s_t, a_t)$, and the policy receives a reward $\psi_t = \psi(s_t, a_t)$. Hence, the aim of the decision process is to find a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maximises:

$$V^\pi(s_t) = \sum_{i=0}^{\infty} \gamma^i \psi_{t+i}$$

for all s_t, where $\gamma \in [0, 1]$ is the *discount parameter*, which determines the importance of the future rewards ($\gamma = 0$ only considers current rewards, while $\gamma = 1$ strives for a high long-term reward).

Given the probably infinite number of states and rules, the abstraction of both of them is necessary, where $\mathcal{S}$ and $\mathcal{A}$ are replaced by feature-based abstraction $\dot{\mathcal{S}}$ and $\dot{\mathcal{A}}$ respectively (tuples of features). Then, the idea is to replace the *state-value* function $Q(s, a)$ of the Q-learning (Watkins and Dayan, 1992b) (which returns quality *values*, $q \in \mathbb{R}$) by a supervised model $Q_M : \dot{\mathcal{S}} \times \dot{\mathcal{A}} \to \mathbb{R}$ that calculates the q value for each state and action, using their abstractions. gErl uses linear regression (by default, but other regression techniques could be used) for generating Q_M, which is retrained periodically from $Q(s, a)$. Then, Q_M is used to obtain the best action a_t for the state s_t as follows:

$$a_t = \arg\max_{\dot{a} \in \dot{\mathcal{A}}} \{Q_M(\dot{s}_t, \dot{a})\}$$

In order to train the model, we use a 'matrix' Q (which is actually a table), whose rows are in $\dot{\mathcal{S}} \times \dot{\mathcal{A}} \times \mathbb{R}$ where $\mathbb{R}$ is a real value for q. Abusing notation, we will denote by $Q[\dot{s}, \dot{a}]$ the value of q in the row of Q for that state $\dot{s}$ and action $\dot{a}$. At each step, Q is updated using the following formula:

$$Q[\dot{s}_t, \dot{a}_t] \leftarrow \alpha \left[w_{t+1} + \gamma \max_{a_{t+1}} Q_M(\dot{s}_{t+1}, \dot{a}_{t+1}) \right] + (1 - \alpha) Q[\dot{s}_t, \dot{a}]$$

where the *learning rate* α ($\alpha \in [0, 1]$) determines to what extent the newly acquired information will be given more or less relevance over the old information. The default parameters are $\alpha = 0.5$ and $\gamma = 0.5$.