A complexity based approach for solving Hofstadter's analogies

Pierre-Alexandre Murena[1], Jean-Louis Dessalles[1], and Antoine Cornuéjols[2]

[1] Télécom ParisTech - Université Paris Saclay,
46 rue Barrault, 75013 Paris, France,
last@telecom-paristech.fr
[2] UMR MIA-Paris
AgroParisTech, INRA, Université Paris Saclay,
5 rue Claude Bernard, 75005 Paris
antoine.cornuejols@agroparistech.fr

Abstract. Analogical reasoning is a central problem both for human cognition and for artificial learning. Many aspects of this problem remain unsolved, though, and analogical reasoning is still a difficult task for machines. In this paper, we consider the problem of analogical reasoning and assume that the relevance of a solution can be measured by the complexity of the analogy. This hypothesis is tested in a basic alphanumeric micro-world. In order to compute complexity, we present specifications for a prototype language used to describe analogies. A few elementary operators for this language are exposed, and their complexity is discussed both from a theoretical and practical point of view. We expose several alternative definitions of relevance in analogical reasoning and show how they are related to complexity.

Keywords: Analogy, Complexity, Relevance

1 Introduction

Analogical reasoning is a fundamental ability of human mind which consists in establishing a mapping between two domains based on common representations. Analogies are involved in particular in the use of metaphors, humour [11] and in scientific research [4]. It is also the key ability measured in IQ tests [16]. Although it is perceived as a very basic and natural task by human beings, transferring this ability to computers remains a challenging task, whether for detecting, understanding, evaluating or producing analogies. A typical analogy can be expressed as follows: *'b' is to 'a' what 'd' is to 'c'*, which will be written **a : b :: c : d**. This problem involves two *domains*, called *source domain* and *target domain.* The analogy is based on the pairing of the transformation **a : b** in the source domain and the transformation **c : d** in the target domain. Several models have been developed so far to cope with analogical reasoning, but they are based on complex modelings and huge computing power, which is not plausible from a cognitive point of view. For example, softwares such as Copycat [8] and its successor Metacat [14] explore the possible mappings between the two involved problems (source and target problems).

The question of relevance is central in analogical reasoning in the sense that it defines the quality of the considered mappings. Because infinitely-many common properties can be found between two objects, a relevance measure has to be found to disqualify properties of little interest [6]. Moreover, several criteria may be considered to measure relevance of a mapping: the number of common properties [18], the abstraction level of the shared properties, structural alignment [5], pragmatic centrality [10] or representational distortion [7].

Inspired by some previous works [3], [15], [1], we consider in this paper that relevance in analogical reasoning can be measured by description length and Kolmogorov complexity (which is its formal equivalent). We propose the principles for a new generative language which can be used to describe analogical problems. Although it is presented in the domain of Hofstadter's analogies (i.e. analogical problems in alphanumerical domain), its principles are general and could be used in several other contexts. The idea of such a language is similar to the idea developed by [17] in the context of sequence continuation. This language offers a strict general framework and offers a cognitively plausible and generative description of analogies.

2 Representation bias for Hofstadter's micro-world

2.1 Presentation of Hofstadter's problem and its variant

In order to study general properties of proportional analogy, Douglas Hofstadter introduced a micro-world made up of letter-strings [9]. The choice of such a micro-world is justified by its simplicity and the wide variety of typical analogical problems it covers. The base domain of Hofstadter's micro-world is the alphabet, in which letters are considered as Platonic objects, hence as abstract entities. Elementary universal concepts are defined relatively to strings of letters, such as *first*, *last*, *successor* and *predecessor*. To this domain is added a base of semantic constructs defined by Hofstadter: copy-groups, successor-groups and predecessor-groups [8]. The typical problem considered by Hofstadter in this micro-world is the

We consider a slightly modified version of Hofstadter's problem. Our modifications correspond to an extension of the micro-world.

First, we consider an additional base alphabet: the number alphabet. This alphabet adds an infinite number of elements to the problem but does not make the base problem more complicated. Furthermore, this addition encourages the use of user-defined base structures and raises the issue of transfer between different domains. In particular, the analogy equation **ABC : ABD :: 123 : x** seems very basic for a human mind while it corresponds to a change of representation from the world of letters to the world of numbers. Besides, the use of other base alphabets can be justified by some prior knowledge of the users: for instance, it can be thought that the problem **ABC : ABD :: QWE : x** will admit a simple solution for any system familiar with the English keyboard layout.

Secondly, we consider a mapping from numbers to any base alphabet. This operation was discarded by Hofstadter's rules but seems important to us. The problem **ABC : ABD :: ABBCCC : x** relies on a such a mapping: the string **ABBCCC** is naturally described as "*n-th letter of the alphabet repeated n times for* $n \in \{1, 2, 3\}$".

The third major difference between our approach and Hofstadter's original works lies in the consideration of descriptive groups. While Hofstadter's approach is merely descriptive, we adopt a generative formalism in which the way strings were formed is taken into account. The static description of *copy-groups*, *successor-groups* or *predecessor-groups* is replaced in our framework by methods such as *copy*, *succession* or *predecession.*

2.2 Complexity-based description

In this paper, we will focus on the resolution of analogy equations of the form **A : B :: C : x** where **x** is unknown. We submit that the solution of such an equation is given by $\mathbf{x} = \arg\min_{\mathbf{x}} C(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{x})$ where the function C corresponds to the (minimum) description length for the four terms. Such a hypothesis is related to the well-known philosophical principle of Occam's razor stating the best choice is the shortest.

A strict definition for the description length is offered by algorithmic theory of information with Kolmogorov complexity [13]. Basically, the complexity $C_{\mathcal{M}}(x)$ of a string x corresponds to the length (in bits) of the shortest program on a Universal Turing Machine (UTM) $\mathcal{M}$ that produces x.

In practice, this quantity is not calculable, hence only upper-bounds are used to estimate the complexity of an object. An upper-bound corresponds to a restricted choice of programs or equivalently to the choice of a limited Turing Machine. In this paper, we consider a particular machine by defining an elementary language. The language we develop is an *ad hoc* construction encoding a theory of the domain of interest.

We do not consider here *prefix codes*, ie. decodable codes in which no code word can be the prefix of another code word. To cope with decoding, we consider that the code is space-delimited, which means that costless delimiters are present in it. This idea is in use in the Morse code for example. Morse code encodes letters by sequences of dashes and dots (ie. with a binary alphabet). A full word is given by a succession of letters separated by short breaks. These breaks are not part of the Morse code but are used to indicate the transition from one letter to another. In such contexts, the delimiters are supposed to be processed by the physical layer of the system, hence to ensure a uniquely decodable code while having no influence on complexity.

2.3 A generative language

Based on the specifications listed above, we develop a prototype language designed to produce and solve analogies. We present here the global characteristics of our language.

As mentioned, a major difference between our perspective and Hofstadter's works is the generative point of view. Largely inspired by Leyton's theory of shapes [12], we consider a description of the process generating analogies rather than a description of the analogies themselves. Any string will result from a transformation of the base alphabet: for instance, **ABCDE** is perceived as the sequence of the first five letters in the alphabet and **ZYX** as the sequence of the first three letters in the reversed alphabet.

In order to integrate this sequential transformation of an original string, we consider that the machine has access to a one-dimensional discrete tape. At each time step, the machine writes on this tape or modifies the previously written string. Thus, the base operation consists in copying the alphabet onto the tape. Thus, the generative procedure consists in a sequence of operations read from left to right and separated by commas. The operations are applied one by one and refer to understandable manipulations. Even if any operation may be incorporated to the language, we will consider here only a restricted set of predefined transformations, called operators $\{\mathcal{O}_1, \mathcal{O}_2, \dots\}$. The complexity of an operator is independent of the operation it performs. An upper bound of this complexity is the rank of the operator in the list of operators. For instance, the complexity of operator $\mathcal{O}_1$ is equal to 0, no matter how complex the corresponding operation actually is.

Besides, the instruction `next_block` is used to move to the next term in the analogy definition. For the analogy **A : B :: C : D**, the order of the blocks is **A**, **B**, **C** and **D**.

The core of the language is the use of a triple memory: a long-term domain memory, a long-term operator memory and a short-term memory. The long-term domain memory stores all accessible domain descriptions, including the alphabets that are accessible to the system. This memory will be denoted by $\mathcal{M}_d = \{\mathcal{A}_1, \mathcal{A}_2, \dots\}$ where the $\mathcal{A}_i$ designate the alphabets. The long-term operation memory stores the repertoire of all applicable operators and is denoted by $\mathcal{M}_o$. Both long-term memories contain prior knowledge and cannot be modified by the machine. All memory modifications are done in the short-term memory which stores *concepts* to be reused in the description. Here, concepts can be either strings or system-defined operators. The short-term memory is designated by $\mathcal{M}_s$.

Using the ideas exposed above, we design a set of rules defining a sketch of grammar for the generative language. The rules presented here are general and do not describe the available operators. A list of elementary operators will be presented and discussed later.

1. A program is encoded as a list of predicates separated by commas. Instructions are read from left to right: this order coincides with the execution order.
2. The program uses a one-dimensional and infinite tape. Intermediate results are written on the tape. Each instruction modifies the content of the tape, either by adding new elements or by correcting previous characters.

3. The base element of a string is called a *group*. A group is recursively defined as a concatenation of groups. The minimal group is made up of one letter. The whole string written on the tape corresponds to one group.
4. Instructions generate groups, either by replacing the group on which they apply or by concatenating the result to it.
5. By default, operators apply to the whole string. To apply the operator to one precise group only, the instruction is declared inside another special instruction called `group`. The `group` instruction can be seen as a way to change the scope.
6. Operators apply to the preceding group and are specified with at most one single parameter. If no parameter is given, a default parameter is used.
7. A string can be put into short-term memory by means of the special instruction `let`. The short-term memory can be accessed with the key instruction `mem`.
8. Operators can be put into short-term memory and accessed respectively with the key words `let` and `mem`. In the declaration, the parameter is indicated by the character `?` and may be used at several places in the instruction.
9. The instruction `next_block` is used to move to the next term in the analogy definition. For the analogy **A : B :: C : D**, the order of the blocks is **A**, **B**, **C** and **D**.

2.4 Basic operators

The list of operators available for the language determines the bias of the machine. The more operators are given to the system, the more sophisticated the obtained expressions can be.

The most basic set of programs is empty: it corresponds to a system capable of giving letters one by one only. Such a system is sufficient in some contexts. Consider for example the real problem of learning declension in a language. In order to learn a declension, students learn by heart a single example and transfer the acquired knowledge to new words. This corresponds for instance to the analogy **rosa : rosam :: vita : vitam** for a simple Latin declension. This analogy is encoded by the following code:

```
let('r','o','s','a'), let('v','i','t','a'),
  let(?, next_block, ?, 'm'),
  mem, 0, mem, 2, next_block, mem, 0, mem, 1;
```

This program has to be interpreted as follows: In the first line, the groups `'rosa'` and `'vita'` are put in short-term memory. The second line defines a new operator which displays the argument, switches to the next block, displays the argument again and finally adds the character `'m'`. The third line retrieves the just-defined operation and applies it successively to the two words, also retrieved from memory.

In order to build effective descriptions for more complex systems, additional operators can be defined.

Two basic operations can be considered as a generative equivalent of Hofstadter's *copy-groups* and *successor-groups*: `copy` and `sequence`.

The operator `copy` repeats the group of interest a given number of times. The parameter of the operator is the number of copies and has 2 as default value. For instance, the instruction `‘a’, copy, 4;` outputs `aaaa`, and the instruction `group(‘a’, ‘b’), copy, 2;` outputs `abab`.

The operator `sequence` outputs the sequence of the first n elements of the group, where n is the parameter. The default value for the parameter is 1. For instance, the instruction `alphabet, sequence, 3;` outputs `abc`. The elements selected by the operator correspond to subgroups of the total group, not necessarily to actual characters. For example, the instruction

```
group(‘a’, ‘b’), group(‘c’, ‘d’), group(‘e’, ‘f’), sequence, 2;
```

outputs `abcd`.

The operator `sequence` alone is not as general as Hofstadter's successor-groups: For example, it cannot describe the sequence `ijk`. In order to cope with this difficulty, we introduce the `shift` operator. Given with parameter n, the operator shifts the subgroups in the subgroup of n steps. The shift is not circular, but a circular version of it may be defined if needed. For example, the instruction `alphabet, shift, 3;` outputs `defg...yz`, while the instruction `alphabet, shift_circular, 3;` would output `defg..yzabc`.

The operator `map` applies an operator (specified as parameter) to all subgroups in the group of interest. The parameter is an operator specified with its parameter (if needed). For example the instruction

```
alphabet, sequence, 3, map, copy, 2;
```

applies the repetition with parameter 2 to the first three letters of the alphabet, hence outputs `aabbcc`.

The operator `reverse` is used to reverse the order of elements in a group. This operator does not have a parameter. For example the output for instruction `alphabet, sequence, 3, reverse;` will be `cba`.

To these writing operators, we have to add another class of operators, which will be designated in the following as *pointing operators*. Unlike previously described operators, pointing operators are used to extract subgroups on which the following operator will apply. By default, an operator applies to the whole string in its scope. However, in some cases, an operation is needed on several subgroups inside the total group, hence operators are needed to point toward desired subgroups. We propose two pointing operators: `find` and `last`.

The operator `find` searches all occurrences of a group `g` inside the group of interest. The group `g` is given as a parameter. For instance, instruction `‘a’,‘b’,‘a’, find, ‘a’, copy, 2;` will output `aabaa`.

The operator `last` (specified with no parameter) selects the last subgroup. For instance, instruction `alphabet, sequence, 4, last, copy, 3;` will output `abcddd`.

A summed-up list of these operators is given in table 1.

Name	Description	Example
`copy`	Repeats the group a given number of times. Equivalent of Hofstadter's *copy-group*.	`‘a’, copy, 4;` outputs `aaaa`
`sequence`	Outputs the sequence of the first n elements of the group. Equivalent of Hofstadter's *copy-group*.	`alphabet, sequence, 3;` outputs `abc`
`shift`	Shifts the subgroups of n positions.	`alphabet, shift, 3;` outputs `defg...yz`
`shift_circular`	Circular version of the `shift` operator	`alphabet, shift_circular, 3;` outputs `defg...yzabc`
`reverse`	Reverses the order of elements in a group.	`alphabet,sequence,3,reverse;` outputs `cba`
`find`	Searches all occurrences of a group given as parameter.	`‘a’,‘b’,‘a’,find,‘a’,copy,2;` outputs `aabaa`
`last`	Selects last group	`‘a’,‘b’,‘a’, last, copy, 2;` outputs `abaa`

Table 1. Example of operators used by the language.

2.5 Using memory

The strength of the proposed language lies in its use of a triple memory to access elements of different nature: a long-term domain memory $\mathcal{M}_d$ storing domain descriptions (e.g. alphabets), a long-term operator description $\mathcal{M}_o$ storing system procedures to modify objects, and a short-term memory storing temporary elements. Managing memory is of major importance when it comes to producing programs of minimal length.

The access to elements in long-term memories $\mathcal{M}_d$ and $\mathcal{M}_o$ is hidden in the language for simplicity purpose, but it cannot be ignored. The designation of support alphabets (`alphabet`, `numbers`, `utf8`, `qwerty-keyboard`...), hence of the domain, and the designation of operators (`copy`, `sequence`, `find`...) are treated as proper nouns to encapsulate an access to an ordered memory. The rank of entities in memory is a characteristics of the machine and cannot be changed.

The user is in charge of the management of short-term memory. Entities (operators or strings) are stored in memory with the `let` meta-operator and accessed with the `mem` meta-operator. For example, the instruction `let(‘a’)` will store the generation of `a` but the string is not written on the band. It will be written only when invoked from memory. The short-term memory is organized as a stack (hence last-in first-out): the parameter given to the `mem` operator is the depth of the element in the stack.

Using short-term memory is not compulsory to describe a string: the language syntax does not prevent from repeating identical instructions. However, in a context of finding a minimal description (which is the purpose of our framework), using memory is an important way to pool identical entities.

3 Relevance of a solution

3.1 From language to code

The principles outlined in previous section form a simplified grammar for our generative language. They are not sufficient yet to calculate the complexity of an analogy. The missing step is the formation of a binary code from an instruction.

The basic idea we use to obtain an efficient code consists in using a positional code in lists. This code associates element 0 to the blank symbol, `0` to element 1 and increments of 1 bit at for each element (`0`, `1`, `00`, `01`, `10`...). Using this code, the complexity of the n-th element of a list is $\lceil \log_2 n \rceil$.

Table 2. Positional code in a list and corresponding complexity

Index	1	2	3	4	5	6	7	8	9	10	11	12
Code		`0`	`1`	`00`	`01`	`10`	`11`	`000`	`001`	`010`	`011`	`100`
Complexity	0	1	1	2	2	2	2	3	3	3	3	3

The global description of the language is organized as a list of lists: a word is designated by the path inside the sequence of lists. For instance, the code for the character `d` corresponds to the code of domain memory (`1`), alphabet (`0`) and `d` (`01`), hence `1,0,01`. The code is not self-delimited: the delimiter is the comma symbol and can delimit a blank symbol. For instance, the number 2 is encoded by `1,,00`. Because a language word corresponds necessarily to a tree leaf, the code is uniquely decodable.

The complexity of an instruction is determined from the corresponding code. We propose to consider that the complexity corresponds directly to the number of bits required in the code. For instance, the complexity of the character 2 will be the number of bits in `1,0,0`, hence $C(2) = 3$. The same reasoning is applied to any instruction, including complex instructions describing complete analogies.

The ordering suggested in figure 1 is entirely arbitrary, except for the blank element the purpose of which is to avoid specifying unnecessary parameters, and thus has to count for a low complexity. How to build an ordering in practice is a fundamental question which is not answered practically in this paper. A way to build a cognitively plausible language encoding would consist in evaluating the ordering based on human experiments. Such experiments will have to be made in future research.

3.2 Relevance of a description

Several acceptable instructions can generate a given string. For example, the string `abc` can be produced either by `alphabet, sequence, 3;` (instruction 1) or `'a','b','c';` (instruction 2) or `alphabet, sequence, 2, 'c';` (instruction 3). These three instructions do not seem equally satisfying from a human

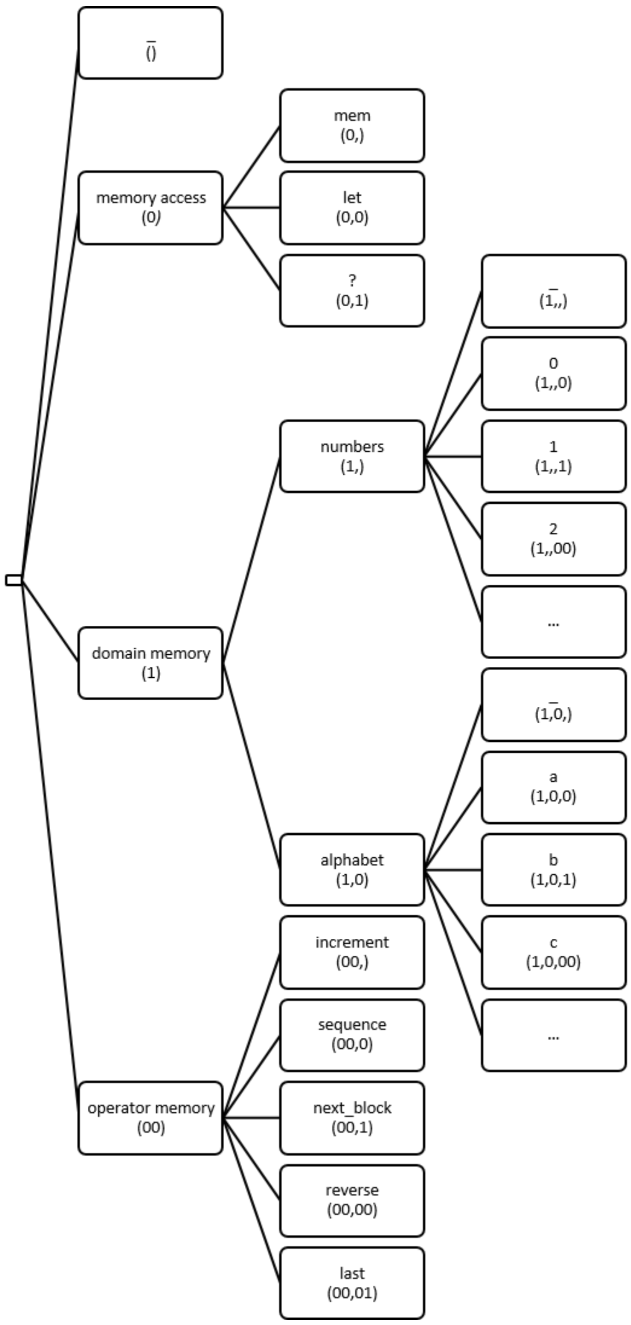

Fig. 1. Instruction tree used for the code. Each element is indicated with its name and the corresponding code (written in parenthesis).

point of view. We submit that the difference in terms of relevance can be quantified by their description length. Using a specific code description, the description lengths for the three previous instructions are respectively $DL_1 = 8$, $DL_2 = 10$ and $DL_3 = 12$. In this example, it is clear that the instruction with minimal description length corresponds to the most relevant description of the string. As a first step of our reasoning, we state that *the most relevant generative description of a string is the description of minimal description length.* An upper-bound for the Kolmogorov complexity of a string is defined as the description length of the most relevant instruction which outputs the string of interest. Despite the huge restriction applied to a general UTM by the choice of our language, the complexity remains non computable: its computation requires an exploration of all instructions producing the string, hence of an infinite space. Several solutions can be adopted in order to build the optimal program. First, greedy approaches would impose a research bias by the mean of a locally optimal exploration of the space of programs. Additionally to this guided exploration of the space of programs, a resource-bounded research can be considered [2].

3.3 Relevance of a solution for an analogy equation

Using the version of Kolmogorov complexity obtained by our system as described above, it is possible to apply the minimum complexity decision rule.

In order to evaluate the way human beings react to analogy problems, we proposed an experiment with several Hofstadters analogy problems.

Participants were 68 (36 female), ages 16-72, from various social and educational backgrounds. Each participant was given a series of analogies. The series were in the same order for all participants, and some questions were repeated several times in the experiment. All analogies had in common the source transformation **ABC : ABD**. The main results are presented in Table 3.

The results presented in Table 3 confirm that in most cases the most chosen solution corresponds to a minimum of cognitive complexity. The complexity is calculated here using our small language and the coding rules exposed earlier. Its limits are visible with the two examples **ABC : ABD :: 135 : x** and **ABC : ABD :: 147 : x**. In these examples, the language fails at describing the progression of the sequence "two by two" (1-3-5-7) or "three by three" (1-4-7-10) which would decrease the overall complexity.

However, despite the simplicity of the language used to calculate the complexity, it is noticeable that the most frequent solution adopted by the users corresponds a complexity drop. This property is not verified with only two problems: for the problem **ABC : ABD :: 122333 : x**, the large value of the complexity in the most frequent case is due to the limitations of the language which fails at providing a compact description of the complete analogy because of a too rigid grammar. In the case of the analogy **ABC : ABD :: XYZ : x**, adding the circularity constraint has a cost in the language, while it seems to be a natural operation for human beings.

The experiment also reveals a major weakness of our modeling: The descriptions provided by our language are static and do not depend on the environment.

Problem	Solution	Propostion	Complexity
IJK	IJL	93%	37
	IJD	2.9%	38
BCA	BCB	49%	42
	BDA	43%	46
AABABC	AABABD	74%	33
	AACABD	12%	46
IJKLM	IJKLN	62%	40
	IJLLM	15%	41
123	124	96%	27
	123	3%	31
KJI	KJJ	37%	43
	LJI	32%	46
IJK	IJL	93%	37
	IJD	2.9%	38
135	136	63%	35
	137	8.9%	37
BCD	BCE	81%	35
	BDE	5.9%	44
IJJKKK	IJJLLL	40%	52
	IJJKKL	25%	53
XYZ	XYA	85%	40
	IJD	4.4%	34
122333	122444	40%	56
	122334	31%	49
RSSTTT	RSSUUU	41%	54
	RSSTTU	31%	55
IJJKKK	IJJLLL	41%	52
	IJD	28%	53
AABABC	AABABD	72%	33
	AACABD	12%	46
MRRJJJ	MRRJJK	28%	64
	MRRKKK	19%	65
147	148	69%	36
	1410	10%	38

Table 3. Main results of the survey. For each problem, only the two main solutions are presented, with their frequency and the corresponding complexity.

On the contrary, the variations of the average answering time and the changes in the answers (when a same problem is repeated at several places) indicates clearly that having faced similar structures in the past helps in solving a new analogy. Finally, the relative relevance of two solutions is not necessarily sufficient to explain human preference in this matter, though. For instance, on the first problem, a large majority of people choose the **IJL** answer despite the small complexity difference. This possible divergence is related to research biases which are not taken into account in our approach. This effect is particularly visible with the more difficult analogy equation **ABC : ABD :: AABABC : x**. Very few humans notice the structure **A-AB-ABC**, hence the corresponding solution **x** = **AABABCD**. However, the structure **A-AB-ABC** is perceived as more relevant when presented.

We have shown that complexity offers a criterion to compare two given solutions to an analogy equation. This sole property is not sufficient in practice to obtain an analogy solver. Since the space of solutions is infinite, additional hypotheses must be considered in order to restrict the exploration space.

3.4 Toward an effective analogy solver

In order to develop an efficient automatic analogy solver based on complexity minimization, two issues have to be overcome: how to effectively compute the complexity of a complete analogy, and how to explore the space of solutions.

Several phases are described in Hofstadter's Copycat program [8]: syntactic scanning, semantic phase, rule generation, world mapping, rule slipping, rule execution and closure checking. All these phases can be transposed directly in a complexity minimization framework. We propose to examine them with the example **ABC : ABD :: IJK : x**. We will present code structures for each phase and show that the phases can be described in terms of memory.

Syntactic scanning and semantic phase Syntactic scanning examines immediate syntactic connections inside all strings. For instance, successions or repetitions are targeted during this phase. This approach offers a bottom-up exploration of the description space. Unlike Copycat's approach (separating syntactic and semantic description), we propose to merge the two phases in a first description of the analogy. In our example, we obtain the following description:

```
alphabet, sequence, 3, next_block, alphabet, sequence, 2, 'd',
next_block, group(alphabet, shift, 8, sequence, 3)
```

Rule generation Rule generation focuses on the first domain of the analogy (hence **ABC : ABD**) and aims at factorizing it. The purpose is to make a transformation appear during factorization. Here, the idea is to factorize the common structure **ABC** or **AB** and to propose a transformation for both of them. The factorization is made using memory.

```
// Factorization of ABC
let(alphabet, sequence, 3), mem,, next_block, mem,, last, increment;

// Factorization of AB
let(alphabet, sequence, 2), mem,, ‘c’, next_block, mem,, ‘d’;
```

Once the structure is memorized, the transformation is stored into a second memory instance:

```
let(...), let(mem,, next_block, mem,, last, increment), mem,;
```

World mapping and rule slipping World mapping is a crucial step in analogical reasoning: it consists in unifying both domains by finding a correlation between them. In our example, the correlation has to be found in the expression of `alphabet, sequence, 3` and `alphabet, shift, 8, sequence, 3`. In this case, the correlation can be established using the instruction `shift, 0` that corresponds to the identity operator (mapping a sequence to itself). A factorization can then be proposed:

```
let(alphabet, shift, ?, sequence, 3)
```

This factorization leads to slight changes in previous code definitions. Such modifications correspond to the rule slipping phase. The modifications are stored in memory, in order to produce a general description method available both for source and target.

```
let(alphabet, shift, ? sequence, 3),
  let(mem,, ?, next_block, mem,, ?, last, increment);
```

Rule execution The final instructions are obtained through the following steps:

```
let(alphabet, shift, ? sequence, 3),  // Structure definition
  let(mem,, ?, next_block, mem,, ?, last, increment);  // Rule
  mem,,, next_block, mem,, 8;
```

Executing this instructions, we obtain analogy **ABC : ABD :: IJK : IJL**. The way the system generates each step is an open research problem and will have to be solved in order to obtain an actual analogy solver. Because this solver will rely on storing factorized structures in memory, the found solution will coincide with a local minimum of complexity. No guarantee can be offered in any way that this local minimum corresponds to a global minimum.

4 Conclusion

In this paper, we proposed to interpret analogical reasoning as a complexity minimization problem and to solve an analogy equation by taking the solution minimizing total complexity. Our approach relies on a restricted Turing machine: we proposed basic rules defining a small language adapted to Hofstadter's analogies. The language has been chosen to be generative (hence consistent with Leyton's theory of shapes) and not self-delimited (which allows compression with unspecified parameters). We gave general principles governing such a language. The system is flexible in the choice of the operations that can be involved for the description of an analogy. This language is associated to a code directly used in the computation of complexity. We use this code to measure the relative relevance of descriptions for a same string and the global relevance of a solution to an analogy. We used this code to measure the complexity of several analogies and noticed that the minimum complexity solution corresponds in most cases to the most frequent solution given by human beings.

Although the considered case might seem restrictive, our approach applies on a wider range of problems. Humans often justify their analogies with a semantic description. We consider our developed language as such. Similar languages can be developed for other analogies. Several issues remain open. A future research would be to develop a system able to generate descriptions automatically, hence to solve analogy equations automatically. The question of the performance evaluation of an analogy solver remains open: our framework measures only the relevance of a single solution. Some work has to be done to offer either a theoretical measure of the global quality for an analogy solver or an experimental validation of its efficiency. Finally, a real investigation on an extension of this language to other domains is needed in order to conclude on its actual generalization properties.

Acknowledgments. This research is supported by the program Futur & Ruptures (Institut Mines Télécom).

References

1. Bayoudh, M., Prade, H., Richard, G.: Evaluation of analogical proportions through Kolmogorov complexity. Knowledge-Based Systems 29, 20–30 (2012)
2. Buhrman, H., Fortnow, L., Laplante, S.: Resource-Bounded Kolmogorov Complexity Revisited. SIAM J. Comput. 31(3), 887–905 (2001)
3. Cornuéjols, A., Ales-Bianchetti, J.: Analogy and induction : which (missing) link? In: Workshop "Advances in Analogy Research : Integration of Theory and Data from Cognitive, Computational and Neural Sciences". Sofia, Bulgaria (1998)
4. Dunbar, K.: Designing for science: Implications from everyday, classroom, and professional settings. chap. What scientific thinking reveals about the nature of cognition., pp. 115–140. Mahwah, NJ, US: Lawrence Erlbaum Associates Publishers (2001)
5. Gentner, D., Markman, A.B.: Structure mapping in analogy and similarity. American psychologist 52(1), 45 (1997)
6. Goodman, N.: Problems and Projects. Indianapolis: Bobbs-Merrill (1972)
7. Hodgetts, C.J., Hahn, U., Chater, N.: Transformation and alignment in similarity. Cognition 113(1), 62–79 (2009)

8. Hofstadter, D.: The Copycat Project: An Experiment in Nondeterminism and Creative Analogies. AI Memo 755, Artificial Intelligence Laboratory, Massachusetts Institute of Technology (1984)
9. Hofstadter, D., Mitchell, M.: Fluid concepts and creative analogies. chap. The Copycat Project: A Model of Mental Fluidity and Analogy-making, pp. 205–267. Basic Books, Inc., New York, NY, USA (1995)
10. Holyoak, K.J., Thagard, P.: Analogical Mapping by Constraint Satisfaction. Cognitive Science 13(3), 295–355 (1989)
11. Holyoak, K., Holyoak, K., Thagard, P.: Mental Leaps: Analogy in Creative Thought. A Bradford book, Bradford Books (1996)
12. Leyton, M.: A Generative Theory of Shape. Springer (2001)
13. Li, M., Vitanyi, P.M.: An Introduction to Kolmogorov Complexity and Its Applications. Springer Publishing Company, Incorporated, 3 edn. (2008)
14. Marshall, J.B.: Metacat: a self-watching cognitive architecture for analogy-making and high-level perception. In: In Proceedings of the 24th Annual Conference of the Cognitive Science Society. Citeseer (1999)
15. Prade, H., Richard, G.: Testing Analogical Proportions with Google using Kolmogorov Information Theory. In: FLAIRS Conference (2009)
16. Ragni, M., Neubert, S.: Solving Raven's IQ-tests: an AI and cognitive modeling approach. In: Proceedings of the 20th European Conference on Artificial Intelligence. pp. 666–671. IOS Press (2012)
17. Strannegård, C., Nizamani, A.R., Sjöberg, A., Engström, F.: Bounded Kolmogorov Complexity Based on Cognitive Models, pp. 130–139. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
18. Tversky, A.: Features of similarity. Psychological review 84(4), 327 (1977)