Solving Hofstadter's Analogies using Structural Information Theory

Geerten Rijsdijk[1] and Giovanni Sileno[1]

[1]Informatics Institute, University of Amsterdam, Amsterdam
geerten.rijsdijk@student.uva.nl, g.sileno@uva.nl

Abstract. Analogies are common part of human life; our ability to handle them is critical in problem solving, humor, metaphors and argumentation. This paper introduces a method to solve string-based (symbolic) analogies based on hybrid inferential process integrating Structural Information Theory—a framework used to predict phenomena of perceptual organization—with some metric-based processing. Results are discussed against two empirical experiments, one of which conducted along this work, together with the development of a Python version of the SIT encoding algorithm PISA.

Keywords: Analogical reasoning · Symbolic Analogies · Compression · Structural Information Theory · Complexity

1 Introduction

Analogies are common part of human life; our ability to handle them is critical in problem solving, humor, metaphors and argumentation [8]. In psychology, analogy is seen as the process of understanding new information by means of structural similarities with previously acquired information [10], and analogical reasoning is one of the predominantly measured abilities on IQ tests. Because of their importance to cognition, analogies have interested researchers in the field of artificial intelligence. Systems for the computation of analogies have been created since the '60s for many different purposes such as solving puzzles based on objects in images [3], obtaining information by inference [9], understanding the development in analogical reasoning in children [16], or even as support in suggesting specialized care for patients with dementia [21]. Recent contributions in natural language processing [14] have suggested that analogical inference can be directly performed as vector operations on word vectors (e.g. $\texttt{Paris} \approx \texttt{France} + \texttt{Berlin} - \texttt{Germany}$). However, even if some machine learning methods have proven to be unexpectedly good at reproducing some of these inferences, the overall results are not yet conclusive [18]. The centrality of analogies in human reasoning motivates to continue the effort to find a better understanding of their underlying mechanisms.

The aim of this paper is to offer an alternative solution to string-based (or symbolic) analogies as those proposed by Hofstadter [8]. The contribution is a hybrid inferential process integrating *Structural Information Theory* (SIT), introduced to predict phenomena of perceptual organization, with some metric-based processing depending on the atomic components of the input. Thanks to an anonymous reviewer we discovered that such an application of SIT has been explored before [1], with a investigation on the algebraic properties of SIT extended with domain-dependent operators (e.g. *succ* to produce consecutive symbols). However, that work was presented before the creation of the minimal encoding algorithm PISA used in the present research to conduct our experiments. Even if preliminary, the results of the method we propose go beyond the state of art both in terms of the types of analogy it can deal with and its speed. Additionally, we report on the development of a Python version of the SIT encoding algorithm PISA, modified to consider various methods to compute (descriptive) complexity. The code used for this work is publicly available.[1]

The paper proceeds as follows: in the remainder of this section, Hofstadter's analogies and Structural Information Theory are briefly introduced. Section 2 outlines the analogy solving algorithm, and presents in detail the different components. Section 3 briefly discusses the Python implementation of PISA. In section 4, the algorithm is evaluated on two datasets, and its performance is compared to that of *Metacat*. The paper ends with a discussion and a conclusion.

1.1 Hofstadter's Analogies

Schematically, an analogy can often be expressed as "A is to B what C is to D" (also known as *proportional analogy*). In order to model and perform simple but relevant experiments on analogical reasoning, Douglas Hofstadter proposed a micro-world for analogy-making at the end of the '80s [15]. In this microworld, the objects used for the analogies are strings of letters. An example of such an analogy is:

$$ABC{:}ABD{::}BCD{:}?$$

which should be read as: "ABC is to ABD like BCD is to ?". The answer commonly given by respondents to this test is BCE.

In order to predict human answers, Hofstadter created a computer program called Copycat [15]. To complete a given analogy, the program works with "agents", which gradually build up structures representing the understanding of the problem, eventually reaching a solution. Later the Copycat program was improved to Metacat [13], which adds a memory, allowing the program to prevent itself from performing actions it has previously tried. *Metacat*, which was last updated in 2016[2], represents plausibly the state of the art of algorithms available for this problem.

1.2 Structural Information Theory

Structural Information Theory, or SIT, is a theory about perception with roots in *Gestalt* psychology. Central to SIT is the *simplicity principle*, in practice a

[1] http://github.com/GeertenRijsdijk/SIT_analogies

[2] http://science.slc.edu/~jmarshall/metacat/

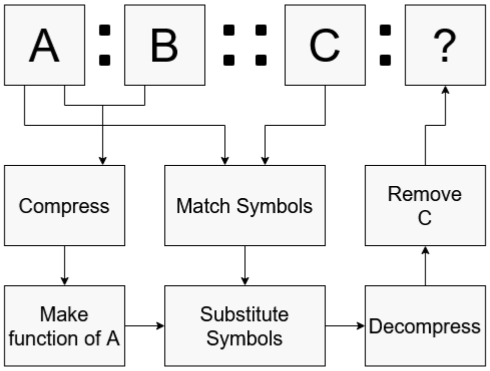

Fig. 1. Outline of the process used to answer an analogy of the form A:B::C:?.

formalization of Occam's razor: the simplest explanation for data is likely to be the correct one [11]. SIT has been empirically validated in several cognitive experiments with human participants [11,6].

SIT proposes to map the application of the simplicity principle to the Minimal Encoding Problem using the SIT language: given a string, use regularities to find an encoding with as little *complexity* as possible [11]. Such encoding can also be seen as a *compression*, since it may greatly decrease the amount of memory needed to store strings. There are three regularities/operators considered in the SIT coding language: *iteration* (I-form), *symmetry* (S-form) and *alternation* (A-form). They are defined in the following way:

I-form: $n * (\bar{y})$ $\Rightarrow yyy...y$ (n times, n $\geq$ 2)
S-form: $S[(\bar{x_0})(\bar{x_1})...(\bar{x_n}), (\bar{p})]$ $\Rightarrow x_0x_1...x_npx_n...x_1x_0$
S-form: $S[(\bar{x_0})(\bar{x_1})...(\bar{x_n})]$ $\Rightarrow x_0x_1...x_nx_n...x_1x_0$
A-form: $<(\bar{y})>/<(\bar{x_0})(\bar{x_1})...(\bar{x_n})>$ $\Rightarrow yx_0yx_1...yx_n$
A-form: $<(\bar{x_0})(\bar{x_1})...(\bar{x_n})>/<(\bar{y})>$ $\Rightarrow x_0yx_1...yx_ny$

(The p in a S-form is an optional element, called *pivot.*) These regularities have been proven to be the only ones which are *holographic* and *transparent* [5]. Holographic means invariant under growth, e.g. a repetition of n symbols can have that same symbol added to it forever, and will remain a repetition. Transparent means that the arguments of a regularity occur linearly in the original string it encodes. With alternations, the arguments can occur non-consecutively, they still occur in the same linear order.

2 Analogy Solving

Analogies rely on a perceived underlying structures (see e.g. Structural Mapping Theory [4]). Both Copycat [15] and Metacat [13] are based on the idea that there are structures on one side of the analogy that need to be replicated on the other side. Their "agents" are functionally meant to identify the structures to be mapped. At functional level, however, what these agents do is nothing else than *compressing* the symbolic input, and this means that other compressors may work as well.

Figure 1 outlines the higher-level process that was followed to produce an analogical inference. In an analogy of form A:B::C:?, it is expected that there exists a certain structure in the left-hand side A:B, which can be extracted by means of some compression method. By applying this same structure to the partially available right-hand side C, decompressing the resulting code and taking away the part C, a possible answer to the analogy can be found.

2.1 Structural compression

The SIT encoding defines a way—empirically validated on perceptual experiments—to compress strings and is therefore a plausible candidate for describing regularities emerging from the input. Indeed, handling strings—here seen as ordered lists of characters, not as words from some language—seems to be based primarily on perceptual mechanisms rather than on semantics.

Applying SIT on analogical inference, we decided to extract the structure of A:B focusing on the concatenated string A+B. The idea of compressing A and B together, rather than separately, is inspired by the technique used in the famous paper by Li and Vitanji on the *Similarity Metric* [12], in which concatenations a.o. of DNA-sequences of two species were compressed to find a measure of their similarity. In our case, we used for compression the PISA algorithm (Parameter load plus ISA-rules) [7], a minimal coding algorithm proposed specifically for SIT [7]. For example, in the problem ABC:ABD::IJK:?, the minimal code (or compression) of the concatenated left-hand side ABC:ABD would be $S[(AB),(C)]D$ or $<(AB)><(C)(D)>$. PISA is currently the most optimized algorithm for performing this task, being only weakly exponential. Along with this work, a Python implementation of this algorithm is presented.

2.2 Generating symbols from symbols

To apply the structure extracted from the left-hand side of the analogy (namely from A+B) to the right-hand side (C+?), this structure needs to be defined only as a function of symbols in A, as only the first part of the right hand is known. The symbols in B need to be generated from the symbols in A, and for this some invertible function capturing an adequate relationship between the atomic symbols is needed. The SIT coding language cannot do this; the only relationship this language considers is the *identity* relationship: denoting an atomic component of perception with a or z is completely arbitrary; the only assumption is that all a or z map to the same type of atomic component.

In contrast, Hofstadter's analogies seem to rely on some metrical information. For instance, d is the most natural answer to the analogy a:b::c:?, obtained by *contrasting* the b and a (as objects in an alphabet) and applying the output of this operation on c. The role of contrast in concept construction, similarity and description generation is indeed central [2,20]. For alphabetic characters as in Hofstadter's analogies, contrast between symbols can be simply defined as a directional distance between the positions of those symbols in the alphabet. So, for example, the distance between b and a is 1, and the distance between a and e is -4. This allow us to rewrite symbols in B as one of the symbol in A and a directional distance. A challenge that arises with this approach is *which symbol the distance should be calculated from*. We attempted multiple approaches to this problem, some of which were more useful in different situations than others.

Previous symbol strategy The simplest way of deciding which symbol to calculate distance from is to choose the *previous symbol*. For example, in the analogy a:bcd::i:?, the left part $abcd$ can be described as $a(\$+1)(\$+1)(\$+1)$, where \$ refers here to the last symbol used in the code. This same structure can be applied to obtain the plausible (see section 4) right-hand side i:jkl.

Last new symbol strategy There are however analogies where this approach fails. Take the analogy aba:aca::ada:?, in which $abaaca$ gets encoded as $S[(a),(b)]$ $S[(a),(c)]$. When distances are applied, the code becomes $S[(a),(b)]$ $S[(a),((\$+2))]$. Now, when the symbol b is substituted by the symbol d, the decompressed code becomes $adaaca$, resulting in solution aca, whereas aea is a much more plausible answer. An approach solving this issue is choosing the *last new symbol* in the organization extracted using SIT. In the previous example, the last new symbol is b, the mapping would then result in $S[(a),(b)]$ $S[(a),((\$\$+1))]$, where \$\$ is the last new symbol used in the organization. Substituting b with d would result in $S[(a),(d)]$ $S[(a),((\$\$+1))]$, decompressing into the expected $adaaea$.

Same position strategy These approaches do not use the actual position of symbols in the input string, but this does seem to play a role at times. Consider for instance ae:bd::cc:?. Here, a plausible answer would be db, where the change applied to the string is an increase by 1 for the first element, and a decrease by 1 for the second. In a case like this, a position-based approach would be useful, resulting in code $ae(a+1)(e-1)$ or $ae(\$\$\$+1)(\$\$\$-1)$, where \$\$\$ is the object in A in the same position of the object in B. However, this approach cannot be used when parts A, B and C of the analogy have different lengths.

2.3 Symbol Substitution

Once the compression of A+B has been defined using only symbols present in A, a substitution (or replacement) of the symbols in A with the symbols in C can be performed. In order to do so, A and C need to be represented in ways that allow the mapping of their components. Here too, different strategies have been tried, selecting the best depending on the case.

Representation as Strings The simplest way to represent A and C is in their input form: strings, i.e. ordered lists of characters, in which each symbol counts as one element. However, there are cases in which this method is not applicable, as for instance when A and C do not share the same length.

Representation by Compression The number of elements in the representation can be reduced by compressing the string and seeing it as a list of symbols and *highest level operators*. For example, $ijjkkk$ can be compressed into $i2{*}(j)3{*}(k)$, which is then split into i, $2{*}(j)$ and $3{*}(k)$. When a highest level operator has been used to replace a symbol, this operator itself will count as one symbol for the purposes of calculating new symbols from distances. If a distance was calculated from a symbol that has since been replaced by an operator, the entire operator will be carried over as the new element, and each individual symbol in this operator is increased by the distance. Consider the analogy abc:abd::$ijjkkk$:?:

- structure of A+B: $<(ab)>/<(c)((\$+1))>$
- structure of C: $i\ 2*(j)\ 3*(k)$
- substutition: $(a{\to}i), (b{\to}2*(j)), (c{\to}3*(k))$
- structure of C+D: $<(i2*(j))>/<(3*(k))((\$+1))>$
- distances removed: $<(i2*(j))>/<(3*(k))(3*(l))>$
- decompression: $ijjkkkijjlll$
- result D: $ijjlll$

Representation by Chunking Alternatively, A and C can be represented in terms of *chunkings*, meaning divisions of the concatenated input symbol strings into chunks. These chunks can then be replaced as if they concerned one element. For example, $abcd$ could be chunked into $[ab, cd]$, $[a, bc, d]$, $[a, b, c, d]$, and so on. For A, the chunkings are derived from the compression of A+B, e.g.:

- $[a, b, c]$ is a chunking of abc in the compression $S[(a)(b),(c)]c$.
- $[ab, c]$ is a chunking of abc in the compression $ab2*(c)(\$+1)(\$+1)$.
- $[ab, c]$ is a chunking of abc in the compression $<(ab)>/<(c)((\$+1))>$.
- $[abc]$ is a chunking of abc in the compression $2*(abc)$.

For C, there is no pre-existing structure present that determines how it should be chunked. However, we considered sound to chunk C in such a way that it best corresponds to the chunking of A.

We call the process of creating a chunking of C as similar as possible to a chunking of A *chunking-element matching*, and works as follows: a list is created for the number of symbols in each element in the chunking of A. (e.g. $[a, bc, def]$ results in $[1, 2, 3]$). Next, if the sum of the numbers does not equal the number of symbols in C, as a simple heuristic, the largest number in this list is increased/decreased. Now, this list of numbers can be used to split C into chunks, which can be used to substitute the original elements of the chunking of A. The following example shows how this chunking matching is used to solve an analogy abc:abd::$ijklm$:?

- structure of A+B: $<(ab)>/<(c)((\$+1))>$
- chunking of A in A+B: $[ab, c]$
- lengths of chunking elements of A: $[2, 1]$
- matching chunking lengths of C: $[4, 1]$
- chunking of C: $[ijkl, m]$

- substitution: $(ab{\rightarrow}ijkl), (c{\rightarrow}m)$
- structure of C+D: $<(ijkl)>/<(m)((\$+1))>$
- distances removed: $<(ijkl)>/<(m)(n)>$
- decompression: $ijklmijkln$
- result D: $ijkln$

Representation by Consecutive Chunking A special type of chunking is a *consecutive chunking.* A chunking is consecutive if all elements of the chunking contain exactly one symbol; the elements of the chunking together form all arguments of a single operator. For instance, $[a, b, c]$ is a consecutive chunking of *abc* in the code $<(a)(b)(c)>/<(d)>$, and $[a, b, c]$ is a consecutive chunking of *abc* in the code $S[(a)(b),(c)]$. Consecutive chunkings can be substituted in a special way: instead of replacing individual symbols or elements, the entire chunking can be replaced by a new chunking. The new chunking has one element for every symbol in C. When this consecutive chunking is applied, the entire argument string forming A is replaced with this new chunking (with the corresponding number of parentheses per element). For instance in *abc*:*cba*::*ijklm*:?:

- structure of A+B: $S[(a)(b)(c)]$
- consecutive chunking of A in A+B: $[a, b, c]$
- consecutive chunking of C: $[i, j, k, l, m]$
- substitution: $([(a),(b),(c)]{\rightarrow}[(i),(j),(k),(l),(m)])$
- structure of C+D: $S[(i)(j)(k)(l)(m)]$
- decompression: $ijklmmlkji$
- result D: $mlkji$

2.4 Structure in Parameters

Consider the analogy *aaabb*:*aabbb*::*eeeeef*:?. One way to look at it is that the analogy simply swaps the number of times the first symbol occurs with the number of times the second symbol occurs. However, PISA assigns to A+B the structure $3*(a)S[2*((b))(a)]b$, which does not seem to capture this intuition, as there is no symbol substitution that results in the expected answer *efffff*. The core of this analogy problem does not lie in the structure of the symbols, but in the structure of the structure. By looking at the structure as a series of iterations, this becomes clear; in $3*(a)2*(b)2*(a)3*(b)$, the parameters of the iterations form a symmetry, namely $S[(3)(2)]$. It is this symmetry that forms a plausible basis for solving this analogy. The function written to apply this strategy is separated from the rest of the algorithm. It encodes A+B as a sequence of iterations. Next, the parameters of these iterations are compressed, distances are added and symbol substitution is performed in essentially the same way as described before. This results in new parameters which, combined with symbol substitution on the actual symbols, can produce an answer to the analogy.

However, problems of this type can easily become more complex. When the parameters of a structure can have a structure, the parameters of *that* structure could also have a structure, which could again have parameters with some structure. With larger codes, this 'parameter depth' could become very high.

Algorithm 1: PISA-based compressor

```
Function compress_pisa(graph)
    new_hyperstrings = []
    for hyperstring h in graph do
        Q = QUIS($h$)
        Create and encode S-graphs of $h$ using Q
        for $w$ in h.nodes[1 ... N] do
            Create and encode A-graphs of $h$ using Q, up to node $w$
            for $v$ in h.nodes[w ... 0] do
                find best possible code for $v \rightarrow w$
                add best code to h as edge $v \rightarrow w$
                for $u$ in h.nodes[0 ... v] do
                    if $c(u, w) > c(u,v) + c(v,w)$ then
                        new_code = $u \rightarrow v + v \rightarrow w$
                        add new_code to h as edge $u \rightarrow w$
        add $h$ to new_hyperstrings;
    return combine(new_hyperstrings);
```

Furthermore, relationships between parameters at different ‘depths’ are also possible. Take for instance the analogy *abc*:*aaabbbccc*::*abcd*:. The structure of the parameters could be written as $3*(1)3*(3)$ (neglecting the internal relationships between symbols). To get to a plausible answer *aaaabbbbccccdddd*, there would need to be a relationship between the two 3s out of brackets, and the 3 inside the brackets, which are at different parameter depths. In short, structure in parameters can be very complex. In this work, it has only been explored at a surface level. Other configurations are left to future work.

2.5 Inversion trick

When A:B::C:? does not have a structure that is easily worked with, it is also possible to rearrange the analogy to hopefully obtain better answers. This rearrangement is analogous to this numeric equivalence: $\frac{a}{b}=\frac{c}{d} \Leftrightarrow \frac{a}{c}=\frac{b}{d}$. In our case, while the two forms of analogies might often result in the same answers, one of the two forms might be more solvable using the approach proposed here. An example of this is the analogy *abac*:*adae*::*baca*:?. A+B has a structure of form $<(a)>/<(.)>$, while the left side has a structure of form $<(.)>/<(a)>$. This change in structure is a problem for our algorithm, since it tries to apply the same structure to C+D. Changing the analogy to *abac*:*baca*::*adae*:? results in the structure $S[(a)(bac)]$, which is a structure the solver can deal with more easily.

3 Python implementation of PISA

A string of length N can be represented by up to a superexponential $O(2^{Nlog(N)})$ number of codes [7]. To find the one with the lowest complexity, one could generate each possible code and compare all of these. For long strings this can be very time consuming. The PISA (Parameter load plus ISA-rules) algorithm was designed to efficiently find the minimal coding of a string in the SIT coding language [7]. PISA is significantly faster than a method which generates all possible encodings, being only weakly exponential. A dissection of the PISA algorithm can be found in chapter 5 of *Simplicity in Vision* [6]. Here, we will briefly discuss our re-implementation in Python.

The PISA-based compressor created for this research is written in an object-oriented fashion (the original version[3], being in C, does not support classes), with a central *hyperstring* class. The general outline of the algorithm written for this research can be seen in Algorithm 1. The algorithm processes each hyperstring in the input graph separately (line 3). In line 4, the QUIS algorithm [6] is called for each hyperstring to create an intermediate structure to more efficient representations. The output matrix is indeed used in line 5 and 7 to create S- and A-graphs, representing symmetries and alternations present in the hyperstring. These graphs are themselves also encoded using this function. Lines 6 and 8 loop over every combination of two nodes (v, w) in the hyperstring. For each pair, lines 9 looks for the best possible code for the substring between these two nodes. Using the complexity metric, this best code is chosen from: the current code; the best possible iteration, if any; the best possible symmetry, if any, calculated using the S-graph that has a pivot halfway in between the two nodes; the best possible alternation, if any, calculated for every A-graph. Once the best code has been selected, line 10 adds an edge representing the code to the hyperstring. Next, line 11 iterates over every node that comes before node v. Line 12 looks at the complexities of the codes between u, v and w. If the complexity of the edge $u{\to}w$ can be reduced by creating a combination of the codes in edges $u{\to}v$ and $v{\to}w$, this is done. At the end of the algorithm, all encoded hyperstrings are recombined into a new graph. This graph is then returned.

Given a hyperstring that represents a mere string, the same hyperstring is returned with added edges that represent the best code for each substring, and the code of the edge connecting the first and last node of the hyperstring will be the best encoding of the entire string.

3.1 Configurable Complexity Metric

Besides reproducibility, a major reason why we reimplemented PISA was to gain control over its components, in particular the way in which complexity is measured. The original PISA relies only on one metric, the I_{new} load [7]. We considered instead as basis the more general principles of Kolmogorov complexity [19]. At a more fundamental level, SIT has been conceived for structural information, but analogies require also to look at some metrical information. The complexity metric considered here calculates complexity by taking the number of symbols used for the code and adding, for each operator in the code, a certain value. This value might differ across operators, and can be adjusted later with

[3] https://ppw.kuleuven.be/apps/petervanderhelm/doc/pisa.html

empirical data to find the values for optimal performance of the analogy solving algorithm. Indeed, the I_{new} load does not allow for adjusting of parameters for aligning to human answers and seems somewhat unintuitive, assigning for example the same complexity to the codes a and $9 * (a)$.

3.2 Other differences with PISA

This implementation relies heavily on the theoretical concepts of hyperstrings, S-graphs and A-graphs as PISA does. However, in some points we found the exact working of PISA to be unclear, and that meant that we had to fill in the blanks. The following list outlines the major differences between the two algorithms.

- The explanation of PISA in [6] mentions 'updating its database of S- and A-graphs' at the end of the first for loop. It is however not clear how this update is done. In the proposed compressor the graphs are not updated, but recreated each time.
- PISA updates the A-graphs at the *end* of the first for loop, while this compressor recreates the A-graphs at the *start* of the first for loop. A small exception to this is present in the code; at the end of each v loop, the algorithm does update the repeats of right a-graphs with the encodings of the $v \rightarrow w$ edge. This is not necessary for left a-graphs due to the algorithm encoding the string left to right.
- PISA updates the S-graphs at the *end* of the first for loop, while this compressor creates the S-graphs before the first for loop.
- PISA always returns the one code with the lowest complexity, while this compressor returns a Graph object. In this object, the edge connecting the first and last nodes also represents the edge with the lowest complexity, but other paths represent other codes, which consist of optimally encoded substrings which together form the whole string. This enables us to consider sub-optimal codes as well.

4 Results

To evaluate the proposed analogy solving algorithm, the answers generated by it will be compared against two sets of human answers obtained in distinct experiments. Furthermore, the answers generated by the proposed solver will also be compared to the answers generated by *Metacat* [13].

4.1 Murena's dataset

Table 1 reports data published by Murena et. al. [17] on human answers for analogy tests. In their experiment, 68 participants were asked to solve analogies following the template ABC:ABD::X:?. The left-hand side of the analogy remained the same during the experiment, but the X changed in every test. For each X, the data shows the two most common answers given by participants,

Given X	Solutions	Selected by	P_s	P_M	Given X	Solutions	Selected by	P_s	P_M
IJK	IJL	**93%**	**1**	**1**	BCD	BCE	**81%**	2	**1**
	IJD	2.9%	-	-		BDE	5.9%	**1**	-
BCA	BCB	**49%**	3	2	IJJKKK	IJJLLL	**40%**	**1**	2
	BDA	43%	**1**	**1**		IJJKKL	25%	2	**1**
AABABC	AABABD	**74%**	**1**	**1**	XYZ	XYA	**85%**	**1**	-
	AACABD	12%	-	-		IJD	4.4%	-	-
IJKLM	IJKLN	**62%**	**1**	**1**	RSSTTT	RSSUUU	**41%**	**1**	**1**
	IJLLM	15%	-	-		RSSTTU	31%	2	-
KJI	KJJ	**37%**	**1**	**1**	MRRJJJ	MRRJJK	**28%**	2	**1**
	LJI	32%	-	2		MRRKKK	19%	**1**	2
ACE	ACF	**63%**	**1**	**1**					
	ACG	8.9%	-	-					

Table 1. Human answers to analogies of form *ABC*:*ABD*::X:? from the Murena dataset 1, along with at which position the same answers were given by the solving algorithm proposed in this project (P_s) and *Metacat* (P_M).

as well as the percentage of participants that chose that answer.[4] The last two columns in the Table 1 show the performances of the analogy solving algorithm proposed in this project (P_s) and *Metacat* (P_M) ins terms of the position in which that answer was generated (e.g. a 1 means it was the best answer, 2 means it was the second best, etc.). A dash indicates that the answer was not generated at all. As for speed, the lack of a built-in way to measure the time *Metacat* uses for compression made it difficult to perform an empirical speed comparison. However, when working with *Metacat*, it was clear that this method is much slower than the solver implemented here, sometimes taking more than 10 minutes to generate a single answer. For this reason, only two answers per question were generated using *Metacat*.

The table shows that, for this dataset, the most common answer to the problem is always generated by our solver. Furthermore, the top answer generated by the solver is always one of the two most common answers by the participants. Both of these observations are also true for *Metacat*, with the exception of the problem XYZ, for which it produced none of the answers given by participants. However, there are also answers given by human participants that the solvers did not generate. Overall, the most common human answer matched the top answer 8/11 times (72.7%), both for our solver and *Metacat*. The most common participant answer was in the top 2 generated answers 10/11 times (90.9%) for both algorithms. Answers given by participants were generated 16/22 times (72.7%) for our solver and 14/22 times (63.7%) for *Metacat*.

4.2 Our Dataset

The Murena testset is quite small and the analogies it presents follow all the same template. For this reason, a second testset was constructed on purpose for this work, consisting of 20 more complex analogies. 35 participants (18 male,

[4] In the original experiments some questions were repeated to see the influence of having previously faced similar problems. Since the solving algorithm in this project runs independently of previous answers, repeated questions were omitted here.

Given problem	Solutions	Selected by	P_s	P_M
ABA:ACA::ADA:?	AEA	**97.1%**	**1**	**1**
	AFA	2.9%	-	-
ABAC:ADAE::BACA:?	DAEA	**60%**	2	-
	BCCC	28.6%	21	-
AE:BD::CC:?	DB	**68.5**%	3	**1**
	CC	17.1%	-	2
ABBB:AAAB::IIIJJ:?	IIJJJ	**57.1%**	**1**	-
	JJIII	14,3%	-	-
ABC:CBA::MLKJI:?	IJKLM	**88.6%**	**1**	**1**
	-	-	-	-
ABCB:ABCB::Q:?	Q	**100.0%**	**1**	-
	-	-	-	-
ABC:BAC::IJKL:?	JIKL	**54.3%**	-	-
	KIJL	14.3%	2	-
ABACA:BC::BACAD:?	AA	**57.1%**	**1**	-
	BCD	31.4%	-	-
AB:ABC::IJKL:?	IJKLM	**85.7%**	**1**	**1**
	IJKLMN	11.4%	-	-
ABC:ABBACCC::FED:?	FEEFDDD	**91.4%**	2	1
	-	-	-	-
ABC:BBC::IKM:?	JKM	**57.1%**	7	-
	KKM	37.1%	2	-
ABAC:ACAB::DEFG:?	DGFE	**68.6%**	2	-
	FGDE	14.3%	**1**	-
ABC:ABD::CBA:?	DBA	**51.4%**	**1**	2
	CBB	45.7%	2	**1**
ABAC:ADAE::FBFC:?	FDFE	**94.3%**	**1**	-
	FDFA	2.9%	-	-
ABCD:CDAB::IJKLMN:?	LMNIJK	**80.0%**	-	-
	-	-	-	-
ABC:AAABBBCCC::ABCD:?	AAABBBCCCDDD	**74.3%**	**1**	**1**
	AAAABBBBCCCCDDDD	17.1%	-	-
ABC:ABBCCC::ABCD:?	ABBCCCDDDD	**85.7%**	-	-
	ABBCCCDDD	8.6%	**1**	-
ABBCCC:DDDEEF::AAABBC:?	DEEFFF	**77.1%**	**1**	-
	DCCDDF	8.6%	-	-
A:AA::AAA:?	AAAAAA	**62.8%**	**1**	-
	AAAA	25.7%	2	**1**
ABBA:BAAB::IJKL:?	JILK	**71.4**%	-	-
	JIJM	11.4%	5	-

Table 2. Human answers to analogies collected in this project experiment, along with at which position the same answers were given by the solving algorithm proposed in this project and *Metacat*.

17 female, average age 26.8) were asked to solve the analogies in the testset, the results of which can be seen in Table 2. As before, the table shows the top two answers given by participants, as well as the percentage of participants that gave each answer. In some cases, only the top answer is given. This is done when either all participants gave the same answer, or when there were multiple answers tied for second which all had only one participant.

In this testset, the most common answer given by participants was generated by the solver 16/20 times (80%), whereas it was generated by *Metacat* only 7/20 times (35%). The top answer given by the solver was in the top two participant answers 13/20 times (65%), whereas the top answer generated by *Metacat* was in the top two participant answers 8/20 times (40%). The most common participant answer matched the top generated 10/20 times (50%) for the solver, and 6/20 times (30%) for *Metacat*.

4.3 Complexity values

The complexity values or weights of iteration, symmetry and alternation operators were chosen to optimize the results of the solver on the two testsets. These variables were tested with values ranging from 0.8 to 1.2, with steps of 0.1. Each possible combination of those values was tested on how highly they ranked the participant-given answers amongst all answers. Overall, it was found that small differences in the values often did not change much about the rankings, suggesting that there was not really a risk of overfitting. On a larger scale, the following requirements seem to yield the best results: the weight of iterations should be less than 1; of symmetries less than 1; of alternation more than 1. The final weights chosen were 0.85 for iterations, 0.9 for symmetries and 1.1 for alternation.

5 Discussion

The goal of this project was to use SIT compression as the basis for an analogy solving algorithm. The analogy solving algorithm has shown promising results on the test set created by Murena et. al [17]. However, this test set is very limited, having only 11 questions, each following the same template. The lack of variety and complexity motivated the creation of a second testset.

When compared to *Metacat*, our solving algorithm has shown to achieve similar results on the first testset. This is likely because the questions in this testset share a similar structure which both solvers seem to be able to deal with. On the second testset, our solving algorithm achieves drastically better results. It should, however, be noted that *Metacat* uses randomness in its procedure to generate answers. Therefore, different runs of the algorithm on the problem could result in different, and possibly better, answers. Furthermore, for this comparison, only the first two answers generated by *Metacat* were used, but *Metacat* can often generate more answers than that. The choice to only consider the top two answers was made due to time constraints: generating a single answer using *Metacat* can, in some cases, take up to 10 minutes, against a few seconds for our solver.

It is important also to note that problems in the second testset were created after the implementation of the solving algorithm, and with the capabilities of this solver in mind. Because of this, for many of the questions it was predictable beforehand whether the solver would produce intuitive answers. Therefore, the percentages of correct answers should not weigh heavily in the evaluation of the algorithm. Instead, the set should be used as a showcase of what types of analogies the algorithm can and cannot deal with.

Answers of the solver are ranked by the complexities of the codes they originate from, e.g. answers such as $BCCC$ (28.6%) to $ABAC$:$ADAE$::$BACA$:

?, DB (68.5%) to AE:BD::CC:?, JKM(57.1%) to ABC:BBC::IKM:? and $JIJM$(11.4%) to $ABBA$:$BAAB$::$IJKL$:?. Despite these answers being chosen by (fairly) significant percentages of the participants, they do not rank highly amongst the answers generated by the solver. This results hints that there exist better strategies not adequately taken into account. The reasoning behind these answers (most likely) relies on applying positional distances in the left-hand side of the analogy to the right-hand side. The most significant case of this is the answer $BCCC$ to $ABAC$:$ADAE$::$BACA$:?, which the solver ranks lower than 20 other solutions, despite being picked by over a quarter of the participants. Future work on this project could look at alternate ways which could, either in combination with complexity or on their own, rank answers generated by the solver in a way that corresponds better to human answers.

The answer $LMNIJK$ (80.0%) to $ABCD$:$CDAB$::$IJKLMN$:? might tell us something the cognitive equivalent of what in this project is called chunking (section 2.3). It seems that the structure that best corresponds to participants' interpretation of this problem is $S[(ab)(cd)]$, which essentially represents a swapping of ab and cd in part A to get part B. The same structure in the right-hand side of the analogy that corresponds to the top answer is $S[(ijk)(lmn)]$. This way of symbol substitution corresponds to *chunking element matching*, described in section 2.3; whereas the method used in this project tries to keep as many elements of the chunking as possible at the same length (which results in structures like $S[(ij)(klmn)]$ or $S[(ijkl)(mn)]$), the 3-3 division suggests a preference to maintain the same ratio between the chunking elements.

Finally, other answers that cannot be solved by the algorithm are the ones discussed in section 2.4, where there are relationships between iteration parameters at different levels. In the test, such relationships are (most likely) used for answer $AAAABBBBCCCCDDDD$ (17.1%) to problem ABC:$AAABBBCCC$:: $ABCD$:?, and answer $ABBCCCDDDD$ (85.7%) to problem ABC:$ABBCCC$:: $ABCD$:?. These answers suggest that such relationships are indeed understood and used by participants, although this begs the question of how complex these relationships can be before participants will no longer base their answer on them.

6 Future developments

Structural Information Theory has shown itself to be a useful tool for analogy solving, although it cannot do this on its own. The lack of metrical information, or a way to define relationships between symbols, resulted in the need for a way of defining symbols as distances from other symbols, as well as a way of choosing which symbol to calculate from. Similarly, the necessity to apply structure from one part of an analogy to another entailed the need for a method for symbol substitution. In this work we introduced with some success different intuitive heuristics/strategies for these two aspects, but a general, unifying theory is needed. Additionally, test data confirms that, sometimes, the structure of the symbols has structure itself (section 2.4). A unifying theory based on Kolmogorov complexity might predict that further depth is considered only if yields a reduction of complexity, and this is a required focus for future works.

References

1. Dastani, M., Indurkhya, B., Scha, R.: Analogical projection in pattern perception. Journal of Experimental and Theoretical Artificial Intelligence **15**(4), 489–511 (2003)
2. Dessalles, J.L.: From Conceptual Spaces to Predicates. Applications of conceptual spaces: The case for geometric knowledge representation pp. 17–31 (2015)
3. Evans, T.G.: A program for the solution of a class of geometric-analogy intelligence-test questions. Tech. rep., Air Force Cambridge Research Labs (1964)
4. Gentner, D.: Structure-mapping: A theoretical framework for analogy. Cognitive Science **7**(2), 155–170 (1983)
5. A van der Helm, P., J van Lier, R., LJ Leeuwenberg, E.: Serial pattern complexity: Irregularity and hierarchy. Perception **21**(4), 517–544 (1992)
6. Van der Helm, P.A.: Simplicity in vision: A multidisciplinary account of perceptual organization. Cambridge University Press (2014)
7. van der Helm, P.A.: Transparallel mind: Classical computing with quantum power. Artificial Intelligence Review **44**(3), 341–363 (2015)
8. Hofstadter, D.R.: Analogy as the core of cognition. The analogical mind: Perspectives from cognitive science pp. 499–538 (2001)
9. Hummel, J.E., Holyoak, K.J.: Lisa: A computational model of analogical inference and schema induction. In: Proceedings of the Cognitive Science Society. pp. 352–357. Lawrence Erlbaum Associates Hillsdale, NJ (1996)
10. Itkonen, E.: Analogy as structure and process: Approaches in linguistics, cognitive psychology and philosophy of science, vol. 14. John Benjamins Publishing (2005)
11. Leeuwenberg, E., Van der Helm, P.A.: Structural information theory: The simplicity of visual form. Cambridge University Press (2013)
12. Li, M., Chen, X., Li, X., Ma, B., Vitányi, P.M.: The similarity metric. IEEE transactions on Information Theory **50**(12), 3250–3264 (2004)
13. Marshall, J.B.: Metacat: A self-watching cognitive architecture for analogy-making. In: Proceedings of the Cognitive Science Society. vol. 24 (2002)
14. Mikolov, T., Sutskever, I., Chen, K., Corrado, G., Dean, J.: Distributed Representations of Words and Phrases and their Compositionality. In: Advances in Neural Information Pro-cessing Systems 26 (NIPS 2013). pp. 3111–3119 (2013)
15. Mitchell, M., Hofstadter, D.R.: The emergence of understanding in a computer model of concepts and analogy-making. Physica D: Nonlinear Phenomena **42**(1-3), 322–334 (1990)
16. Morrison, R.G., Doumas, L.A., Richland, L.E.: A computational account of children's analogical reasoning: balancing inhibitory control in working memory and relational representation. Developmental science **14**(3), 516–529 (2011)
17. Murena, P.A., Dessalles, J.L., Cornuéjols, A.: A complexity based approach for solving hofstadter's analogies. In: ICCBR (Workshops). pp. 53–62 (2017)
18. Rogers, A., Drozd, A., Li, B.: The (too many) problems of analogical reasoning with word vectors. *SEM 2017 - 6th Joint Conference on Lexical and Computational Semantics, Proceedings pp. 135–148 (2017)
19. Shen, A., Uspensky, V.A., Vereshchagin, N.: Kolmogorov complexity and algorithmic randomness, vol. 220. American Mathematical Soc. (2017)
20. Sileno, G., Bloch, I., Atif, J., Dessalles, J.L.: Similarity and Contrast on Conceptual Spaces for Pertinent Description Generation, vol. 10505 LNAI (2017)
21. Zachos, K., Maiden, N., Pitts, K., Jones, S., Turner, I., Rose, M., Pudney, K., MacManus, J.: Digital creativity in dementia care support. International Journal of Creative Computing **1**(1), 35–56 (2013)