Faculty of Information Systems and Applied Computer Sciences

University of Bamberg

Applying Structural Analogy to Solve Abstract Reasoning Problems in a More Human-like Way

Master Thesis

Jan Martin (Matr. No. 1796943)

2021/11/12

Supervisor: Prof. Dr. Ute Schmid

Abstract

Creating actual Intelligence, an entity capable of creativity and reason, is an implicit goal of Artificial Intelligence by its very name. The ability to learn, abstracting knowledge from one domain and applying it to a different domain, is a core fundamental of human reasoning that allows learning and understanding a new concept with just a few examples, by virtue of having already learned from examples of other, similar domains. It stands to reason that teaching a machine those abilities will play a pivotal role in creating Artificial General Intelligence, and the measurement of those abilities will be a requirement to evaluating this. This Thesis aims to investigate the viability of applying the Copycat architecture, built to solve 1-dimensional analogies, to 2-dimensional tasks taken from a subset of the Abstract Reasoning Corpus, as used in the related Abstract Reasoning Challenge. Examples will be picked based on complexity, as well as manually chosen, fed into Metacat, a more advanced development of Copycat, and evaluated in regards to both their current output as well as their theoretical potential.
Keywords: Artificial General Intelligence, Learning by Analogy, Analogical Inference

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The idea of artificial intelligence can be traced back to antiquity(Buchanan, 2005). Theorizing about what an artificial being might come to be inspires us to think about ourselves, to ask questions as to the nature of intelligence and self-consciousness, even without the immediate ability to implement our ideas.Since the advent of AI research, an express goal has been to develop a system or machine that could be called intelligent in the same way a human could be. So far, while progress has been made, the goal seems to be about as distant as it was in the 1950s. Mankind has long sought to define, describe, and measure intelligence, usually by evaluating the skill of humans or other test participants at predefined tasks, and turning known cognitive abilities of humans into tasks against which to test AI programs(Turing, 2009). However, intelligence is marked not by skill, but by the ability to attain a skill. Humans are capable of transferring knowledge from situations they know to new situations that they perceive as somehow similar, and with that transferred knowledge, solve problems that they have never seen before. The very concept of a 'Mnemonic device' is a testament to this, allowing us to memorize facts by linking them to some mental picture of something else, more well known to us, that we consider somehow related.
AI research has made huge strides in some areas, rendering possible systems that outperform humans in specific tasks, learning without direct human supervision, or processing and analysing quantities of information that would be impossible for the average human(Russakovsky et al., 2015),yet frequently, they fail because of minute changes to the task, where a human might deem the two tasks very similar, or even indistinguishable. Clearly, those 'narrow AI' programs do not qualify as intelligent. By measuring intelligence purely by its output, we become unable to differentiate actual intelligence and prefabricated skill. The abstraction and generalization abilities of a human can be expressed in generating a solution from just a few examples. Few children, having come into contact with a hot stovetop, will endeavour to repeat the process a few times with different ovens, and analogical transfer allows us to know it is not a good idea to jump down a cliff without having ever witnessed anyone doing so. Efforts to encourage software solutions that bring us closer to Artificial General Intelligence have postulated the need for tasks to be solved to be varied, have few learning examples, and, while the measurement criteria need to be known, part of the evaluation may not be. A dateset specifically designed

to test these on, the Abstract Reasoning Corpus, provides hundreds of varied, human-readable tasks, and has been the subject of multiple recent attempts at bringing us closer to our goal, yet none of them have tried to utilize Copycat or one of it's successors. This raises the question as to why. Shouldn't a system focussed on Mental Fluidity be a good fit for the task? This Thesis aims to answer that question.

Chapter 2

Abstract Relational Reasoning and Learning

Learning takes place when analogy is used to generate a constraint description in one domain, given a constraint description in another, ...
Reasoning takes place when analogy is used to answer questions about one situation, given another situation that is supposed to be a precedent.

Winston (1979)

In the pursuit of AI development, the lofty goal of emulating at least a part of human cognition has been approached over the years from different perspectives. Those Attempts differed in what part of human capability they aimed to emulate, what they considered to be a key feature of this, and how they attempted to reach it. Today, we can differentiate 'Narrow' AI, which focusses on narrow fields of application like processing online searches, predicting the weather, and other usually commercially applicable tasks, and 'General' AI, which is focussed on the pursuit of actual intelligence.

2.1 Artificial General Intelligence

The term 'Artificial General Intelligence', short AGI, denotes a field of AI research concerned with AI that does not aim to solve a specific, pre-defined problem, but instead aims to further the goal to create an intelligent machine. Arguably, all early AI research was concerned with what we would now call AGI, as the proposal for the Dartmouth conference(reprinted in (McCarthy et al., 2006)) mentions goals like "Self-Improvement" and "Creativity". Yet, comparing the status quo to those lofty goals set all those years ago, one would be forgiven to express disappointment. Goertzel and Pennachin (2007) gives an overview over related work.

2.1.1 Measurement

To judge if we are making progress towards our goal, we need to somehow measure it. That requires our goals to be both clearly defined, and accessible to measurement. This, however, incentivises programs that succeed at the task, yet fail at our actual goal, intelligence. Inevitably, by setting the goal to measure intelligence and assign it a distinct value, using a pre-defined evaluation method, we open the door for someone solving the problem without intelligence.

In our many attempts(Neisser et al. (1996),Legg et al. (2007)) to define and measure 'Intelligence', a variety of tests have been created, hoping to find and measure the cognitive abilities of an individual and form it into a value. But frequently, those tests had to be adapted due to bias inherited from their authors(Wicherts and Dolan, 2010)(Pezzuti et al., 2020).

Measuring machine learning exasperates these challenges: When measuring the skill of a human in a specific task, then, if they succeed at the task, we will assume that they have learned that skill somewhere, and are thus attributed the intelligence to do so. We might even go so far as to classify that skill as a subset of a family of skills, and by extension assume the tested individual is able in other abilities, as well. When measuring an algorithm, on the other hand, while we might like to make the same assumption, it is possible that this skill is just an inherent feature of the judged entity, built in by it's creators, or learned through impractical amounts of repetition, exhibiting a high skill value through the weight of data alone, without the desired learning ability, while a human would perform comparably with just a handful of examples(Kühl et al., 2020). Admittedly, it might be a bit unfair to criticise a machine for using pre-existing knowledge, as humans also use previously gained knowledge when solving new and unknown tasks (Lin et al., 2014), but, as mentioned already, a human will have exercised the very skills we want to copy in machines to attain this knowledge, while a program might not have. Yet, the tremendous progress of narrow AI at specific tasks can not be denied. François Chollet argues that this dichotomy is because the fields where AI has progressed were the only ones where "we [have] been able to define our goal sufficiently precisely, and to measure progress in an actionable way"(Chollet (2019b),p.3). Known progress within a limited domain beats the alternative, merely assuming progress in a not clearly defined larger domain. A way to measure the progress we make might just be the requirement to make progress at all.

Analogical transfer is seen as one of the foundations of learning and, indeed, intelligence(Gentner and Markman (1994),Gick and Holyoak (1983)), and so it makes sense to measure the analogy-making ability of a program when evaluating its contribution, if any, to developing AGI.

2.2 The Abstract Reasoning Corpus

The Arbstract Reasoning Corpus (ARC) is a dataset that was proposed by François Chollet in Chollet (2019b) with the goal to "serve as a benchmark for ... general intelligence". ARC consists of two sets of 400 training tasks and 600 evaluation tasks, respectively, each of which may contain up to 5 individual training examples with which a program to be evaluated can learn the task, as well as a single test example to check if the program has drawn the correct conclusions from the earlier examples. The amount of examples is deliberately limited to encourage more human-like approaches, in contrast to Neural Networks which may require tens of thousands. Examples take the

0	0	0	0	0	0
0	0	3	0	0	0
0	3	0	3	0	0
0	0	3	0	3	0
0	0	0	3	0	0
0	0	0	0	0	0

Table 2.1: Example Matrix

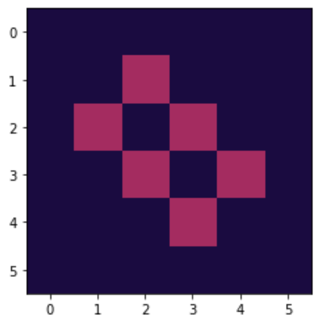


Figure 2.1: Example Plot

form of an input-output pair of two matrices filled with 1-digit numbers2.2, and can be converted to a colour map for human readability2.1. In the latter form, it is not unlike challenges in a human IQ test.

Works using the Corpus as a benchmark so far fall into the following broad categories:

- **Symbolic AI**: The main focus of research in the early decades, symbolic AI focused on human-readable concept representation and logic. Examples include (Newell and Simon, 1961), (Lenat et al., 1985) and (Anderson et al., 1997). Analogous to AI history, most early attempts utilising the ARC also fall within this paradigm.
- **Bayesian Networks**: Probabilisitc Networks used to learn from data (Pearl (1985).
- **Artificial Neural Networks**(ANRs): An attempt to emulate what we know of the structure of a human brain. Especially 'Deep Learning', utilizing convoluted(LeCun et al., 1989) networks has been popular in the recent decade through (Krizhevsky et al., 2012) and others. Schmidhuber (2014) provides an overview of the field. The approach itself has also received criticism(Marcus, 2018), and it has been argued that their contribution to actual Artificial Intelligence has been limited so far, their lack of generalization made apparent by their vulnerability to the tiniest of changes(Mopuri et al. (2017),Moosavi-Dezfooli et al. (2016),Cobbe et al. (2019)).

2.2.1 The Problem Domain

The ARC has served as a basis of measurement for a variety of different approaches, but this has not yet been attempted by a Copycat derivative like Metacat. It should be possible to apply a program meant to emulate mental fluidity to a data corpus that "... is targeted at both humans and artificially intelligent systems that aim at emulating a human-like form of general fluid intelligence."(Chollet (2019b),p.46)

- The Abstract Reasoning Corpus has been used to evaluate a multitude of AGI attempts since its inception.

- Recognized Solutions testing against the full breadth of evaluation tasks have shown success rates under 30%
- No approach has thus far shown to clearly be 'the most promising'
- The ARC hasn't been attempted by a Copycat derivative yet
- The premise of Copycat, "Do the same as in the Example" with a different base, coincides with the tests in the ARC

If Metacat, a successor to Copycat, turns out to show promising results for at least a small subset of the presented problems, it might open up a new approach to tackle this accepted evaluation environment.

Problem Statement:

Design and develop a program to convert some examples from the ARC to input that would be valid for Metacat, and test the performance of Metacat against a selection of those problems to judge the viability of this approach.

2.2.2 The Abstract Reasoning Challenge

The Abstract Reasoning Challenge may be the most prominent employment of the datayet yet. It was a competition on the website kaggle.com (6), hosted by Francois Chollet in the spring of 2020, in which programmers were challenged to write an algorithm that, on predefined hardware and within a set time, could solve as many previously unknown reasoning tasks from the ARC as possible. Training was done on a subset of 100 tasks from the Abstract Reasoning Corpus. The entries were evaluated on a different, hidden subset of the Corpus, to ensure that participants couldn't simply tailor their program to previously known problems. Scoring was based on "top-3 error rate", meaning the program could make three guesses for each task, and if one one them turned out to be correct, it would receive a point. The Challenge garnered some attention, but progress made in the field of AGI was arguably limited: After 3 Months and over 900 submissions, the best solution, a symbolic problem-solver utilizing C++ code optimized for the given domain space, managed to solve just around 21% of the given tasks within the time limit(17). The author of the winning solution themself said they "don't feel like their solution itself brings us closer to AGI". [1] As a user put it on the kaggle forums, "What did we learn from this competition…? Apparently nothing".[2] There has been a late submission claiming a score of 29%, utilizing parts from two of the accepted competition entries. [3]
The author claims that the competition rewards diversity, as a single correct answer out of three would yield the full score for a task, and thus, that by making fewer, more confident guesses with more, different approaches, a higher score could be reached.

[1] User "icecuber", author of the winning submission, in a forum post at https://www.kaggle.com/c/abstraction-and-reasoning-challenge/discussion/154597. Accessed 7/11/21

[2] Discussion at https://www.kaggle.com/c/abstraction-and-reasoning-challenge/discussion/154982. Accessed 7/11/21

[3] Posted on kaggle at https://www.kaggle.com/c/abstraction-and-reasoning-challenge/discussion/234352.Accessed 7/11/21

2.2.3 Dreaming with ARC

An interesting approach detailed in Banburski et al. (2020), that aims to use 'Dreamcoder'(Ellis et al., 2020), a system based on Bayesian Networks, to tackle the ARC. At the time of this writing, no results have been published.

2.2.4 Neural Abstract Reasoner

By utilizing spectral regularisation (see Engl et al. (1996)) with a neural network, Kolev et al. (2020) claim to have achieved a performance rate of 79% on the ARC dataset. The necessary data quantity for an ANR is achieved "by permuting colors and by exploiting that the tasks are invariant to rotation and symmetry"(Kolev et al. (2020),p.7). While the result sounds impressive, it is not entirely comparable to the Abstract Reasoning Challenge entries, as the authors limited their approach to matrices of 10x10 size or lower, which would preclude tasks as shown in 4.3.4.

2.3 Copycat

The Copycat architechture, as detailed in Mitchell (1990) and Hofstadter and Mitchell (1994), is an algorithm built to learn and apply structural analogy in a highly abstracted domain wherein it works with one-dimensional strings of letters. Its aim is to emulate 'Mental Fluidity', using known concepts to generate representations of a given problem space dynamically. It accepts tasks of the format 'A $\rightarrow$ B, C $\rightarrow$?'. The Copycat architecture consists of three main components: Workspace, Coderack, and Slipnet.

The **Slipnet** is a repository of pre-known, linked "platonic concepts". It allows for "conceptual slippage", for example letting the operation "replace letter" spill over to a group of letters, or relating the position of a letter in a given string with the position of the same letter in the alphabet. The distance between concepts will change during a Copycat run depending on the given tasks, making it more or less likely that a connection is made between two concepts in a follow-up task.

The **Workspace** is the main working memory of the program, storing instances of concepts from the Slipnet adapted to the current task, forming temporary structures related to the current progress in solving a given task.

The **Coderack** consists of individual 'codelets', instances of which can be activated non-deterministically by other codelets or the slipnet and perform specific functions. Codelets are also responsible for constructing the structures in the Workspace.

A more detailed explanation can be found in Mitchell (1993).

Copycat regards individual letters based on its understanding of their relation to each other. For Example, a is a predecessor of b, as it comes earlier in the alphabet, and in the string 'ba', the 'b' the positional predecessor of the 'a', as it comes earlier in the string. The Slipnet allows these concepts to be connected, enabling it to transfer known relations from one domain to another. Parts of the string may also be combined into generated groups, for Example two groups of two letters in 'aabb'. In theory, this ordering, while simple, allows a surprising amount of relations to be generated, and represent a lot of the mental operations used to gain insight of a topic. Characters often have a

defined order in alphabets, numbers obviously do, and then groups within a string can also have positional relations with each other. The latter two have been explored in more depth in SeqSee(Mahabal, 2010), and open the door for relations based on more complex pre-defined rules like "is double of".
Copycat's concepts are "designed with an eye to great generality"(Hofstadter and Mitchell (1994),p.34), meaning they are supposed to relate to a breadth of other domains, even if the program itself isn't able to process them. It doesn't require a lot of imagination to see that those concepts should also apply to multi-dimensional problem spaces: up/down and left/right are just different instances of predecessor and successor on different axes.
Modern reimplementations have been done in Java(3) and Python(4), as well as proposed in DrRacket by Huang (2018), who also comments on the potential of the architecture for modern machine learning tasks. Copycat has inspired a couple of follow-up works aiming to build upon the idea by either transferring it to a different domain(SeqSee)(Nichols (2012)), or advance it through additional capability, as seen in Metacat. Tabletop(French and Hofstadter, 1992) explores a similar idea with a focus on object relations on a table instead of letter strings.
Metacat(Marshall (1999) and Marshall (2006)) is a further development of the idea pioneered with Copycat. Developed from scratch in chez scheme, Metacat expands on the original idea of Copycat by introducing the capability of self-watching, meaning the ability of the program to use the same toolbox applied on the given tasks upon its own learning progress. Metacat can thus detect similarities between previously learned analogies, potentially generating meta-analogies.

Chapter 3

The Details

The intent is to find out if an existing analogy engine, developed to tackle a task in a specific domain, can be used to solve similar analogous tasks in a different domain. To enable this, the aim is to pick a small, manually chosen subset of the Abstract Reasoning Corpus, and to attempt an object detection of sufficient quality to enable solving the remainder of the task with the algorithm of the copycat architecture. The goal of this process is strictly to explore the viability of applying the Copycat architecture to this different problem domain, not to solve the problems as fast as possible, or to solve as many problems as possible.
On a sufficiently high level of abstraction, detected objects could be classified and expressed as strings of letters for the purposes of positional analogies, although this is arguably a bit more difficult, as geometric shapes like circles and rectangles have no clearly defined relations the way letters or numbers do.

3.1 Challenges

This section lists perceived challenges that the program should ideally overcome, or otherwise circumvent.

Two-Dimensions. Adding an entire dimension to a problem adds a lot of information to it that a program not designed for this may not be able to process, even if the information itself may be of manageable complexity. As the Copycat architecture and its successors were built to solve analogy problems expressed as small text strings, it is assumed that solving a 2-dimensional problem requires slicing it into 1-dimensional strings. To facilitate this, the matrices taken from the ARC need to be analysed to select matrices that we believe can actually be solved by the chosen algorithm. This poses an additional challenge, namely a lack of context for the program to work with. Many visual puzzles humans could solve might not be so solvable anymore if we cut them into little stripes.

Lack of Change. One fundamental challenge in applying metacat to the problems presented in the ARC is that, being a program designed to process analogies, it expects there to actually be an analogy to be found, that is, a difference between input and output. It is unrealistic to assume that the algorithm would be able to solve every problem put before it, so just taking 'no result' to mean 'no change' will not be a viable approach. Thus, a preprocessing should attempt to filter out inputs where nothing actually changes. The difficulty there is that sometimes, learning that nothing changes with a specific input might well be the point.

Large Problem Space. A single row or column on most ARC tasks is significantly longer the relatively short strings used to demonstrate Metacat's abilities. Also, judging whether to focus on the entire matrix, or just some objects, is relatively difficult.

3.2 Assumptions

To realise this work, some assumptions have been made. Some of those will be put to the test in the evaluation section.

- Copycat, according to commentary in Marshall (2006), will struggle with multiple changes in one line; Thus, Metacat will be chosen for the actual evaluation.
- Not all tasks in the ARC will be solvable by Metacat, even after significant preprocessing
- Some tasks, while theoretically amenable to Metacat, will require an unrealistic amount of previous effort to convert the data sufficiently.
- Further, some tasks will exhibit characteristics common to tasks that metacat can solve, yet be unsolvable without modifying the actual analogy engine. When in doubt, manual pre-tests will be performed.

3.3 Desired Capabilities

The Program for preprocessing data for feeding to Metacat should be able to do the following:

- Detect objects. This can be very simple if the object happens to be a mono-coloured block within a larger area of a different colour, or very difficult if perception has to happen within a noisy environment and context knowledge is required. Tasks should be filtered for this, as object recognition is not actually the focus of evaluation.
- Differentiate — usually irrelevant — background cells from cells inside an object, which depending on the task can be interpreted as part of the object and have a higher likelihood of being changed between input and output. This can also lead to classifications like 'hollow' Objects
- Convert Data. Metacat takes strings of letters. ARC problems are matrices of numbers. Thus, the input needs to be translated somehow.

3.4 Background Detection

An important capability for detecting objects is to detect what is *not* an object. Objects are contrasted against the background, and hence, we need to know what the background is. For this purpose, the values of all cells in a matrix, corresponding to colours of pixels in the human-readable plots, will be counted. This does, however, leave the possibility of a large object taking up the majority of a matrix, yet still being recognizable to a human observer as an object, while the minority cells around it constitute what we would call the background. Because of this, we will value the cells bordering the edge of the matrix higher, not directly, but as a sort of tie-breaker. Thus, the background is determined as follows:

$$BK = \begin{cases} x, & \text{if } |x| \geq \frac{2}{3}T \\ & |x| \geq \frac{2}{5}T \wedge |x| \geq \frac{1}{2}E \\ & |x| \geq |y \qquad\qquad |inT \end{cases} \tag{3.1}$$

Where T is the total amount of cells in the matrix, and E is the set of sells at the edge of the matrix. If neither of those is true, the highest value in the matrix is picked. It is possible that the first condition fails and the second then applies to the second most common value, addressing the above-mentioned case. Ideally, the value totals will be saved for later use. For simplicities sake, only the input matrix will be checked for this.

3.5 Object Generation

The basis building analogies around objects is finding them. As matrices are available in a machine-readable format in the form of matrices, no image recognition is required for this. For the purpose of this test, the object detection will assume a non-noisy environment. The following should be found and saved for analysis:

- **Coordinates:** Where in the matrix the object can be found. This may include redundant data, like spelling out borders in cardinal directions. Additionally, the median, center, as well as the center of mass, should be detected and logged, as this would help in matching objects between input and output. This also enables detection of positional relations between objects, and the detection of neighbours to form 'Macro-Objects'
- **Colours:** Saving a detected Object should include a listing of all colours present in the object, as well as specifically the majority colour, which can then easily be used to help match objects across matrices.
- **Parts:** If a large object consists of smaller objects, those should be listed. This way, Objects could be linked between two matrices even if one part is removed in one of them.
- **Array:** If the found Object is viewed as its own matrix, it may be searched for sub-Objects or patters in the same way that a matrix is searched for

Objects. This adds a layer of recursiveness that is befitting of an attempt to use Metacat, which is able to make analogies of analogies.

- **Negative Space:** Ideally, areas that are not part of objects should also be evaluated as objects. In the case of Object Detection, this will be used inside the Object. The small empty square in a large, hollow square would thus effectively also be seen as an object, and thus accessible to analysis.
- **Object Traits:** Objects can be hollow, have Symmetry, or other qualities. Being able to log those traits, or specific shapes, would enable analogical transfer and enable solving classic IQ test tasks of the form 'Shape 1 $\rightarrow$ Shape 2 is as Shape 3 $\rightarrow$?'. While these are not required for testing Metacat, some of them might have value in matching Objects across matrices.

Actually constructing an output matrix with strings from the generated analogy output is not seen as relevant.

3.5.1 Detection

To begin with the analysis of an example matrix, all non-background cells are listed as sets of coordinates and ordered.

$$while\, L \neq \emptyset : take\, c_0\, in\, L\, as\, C \tag{3.2}$$

$$\begin{aligned} while\, c_m\, is\, neighbour\, to\, c_n\, : \\ take\, c_n\, from\, L \\ add\, c_n\, to\, L \end{aligned}$$

Take the first cell from the cell list L. Then check if any other cells c_n in L are orthogonally adjacent to cells c_m already taken, and add them to C as well. Repeat until no cells are left in L that are adjacent to any cells in C. This C then forms a new Object.

3.5.2 Adjacency

Adjacency between objects is required to generate Macro Objects. Adjacency is calculated like Detection, but instead of orthogonal neighbours, it takes diagonal neighbours, that must not already be part of the same object. Every Object then has a list of cells that are part of it, and a list cells diagonally adjacent to it. The latter is then converted as follows:

$$\begin{aligned} for\, every\, c_a\, in\, C, find\, C_a \supset c_a \\ replace\, c_a\, in\, C\, with\, C_a \end{aligned}$$

Find which generated Object the neighbouring cell belongs to, then replace the cell entry with the Object, and clear up duplicates.

3.5.3 Macro Objects

Macro Objects are the focal point of this analysis. They are created quite similarly to small Objects. While before, individual cells were checked for adjacency, when building a Macro Object, the small objects are checked for adjacency. Once no further objects are found that are adjacent to already checked objects, the macro object will be created. Afterwards, it will be loaded with additional information pulled from the matrix. These include its coordinates, like highest and lowest row, its center, primary colour, and more. Further entries in more detail are:

- top, bottom, left, right: borders of the macro object
- **center**: Where is the middle of the object.
- **com**: Center of mass, weighing all non-background cells equally. It is possible that the ARC contains tasks where a value-specific weighting would be helpful, which may be tested in a future evaluation.
- **median**: The cell that is closest to the center of the object.
- **dominant colour**: Which value occurs most often in this particular macro object.
- **shape**: The shape of the object, with which a similarly sized matrix could be generated, or two matrices compared.
- **parts**: The small objects that make up the macro object. Comparing those might reveal similarity between objects in different matrices that might otherwise not be matched due to different mass and average center.
- **cells and colours**: Despite those being possible to be looked up through parts, all cells and their colours of the macro object are still listed seperately. This detail can also enable further shrinking of tasks forwarded by omitting background cells that are not directly object parts.
- **rows/colums**: Stored more accessibly, as those are forwarded towards Metacat.

After generating a new matrix in exactly the shape of the object, object detection can be run again, this time on the macro object instead of the matrix as a whole, using the object's dominant colour as background. This allows detecting additional features of the object, like colour patterns or holes, meaning spaces of matrix background colour. When comparing two objects across matrices, knowing those sub features and their position allows for easier object matching, as something that would otherwise be a difference counting against similarity is now a changed value on an otherwise similar object.

3.6 Example Filtering

The ARC tasks are saved as a set of matrix pairs, split into input and output, in a json file. Not all of those will be amenable to Metacat. The ability to filter lists of tasks would be helpful in this regard. To raise chances of finding solvable tasks in the portion of ARC used, the first example of each file will be checked, and discarded if the matrix contains too many objects, or more than one macro object. While this could also be used to limit size or other filters, none of those options are used.

3.7 Conversion

The values of the matrices, more specifically of their relevant rows and columns, will be converted to letters which can be understood by metacat with a simple mapping: '0,1,2,3,4,5,6,7,8,9 $\rightarrow$ a,b,c,d,e,f,g,h,i,j'. In case it is desired

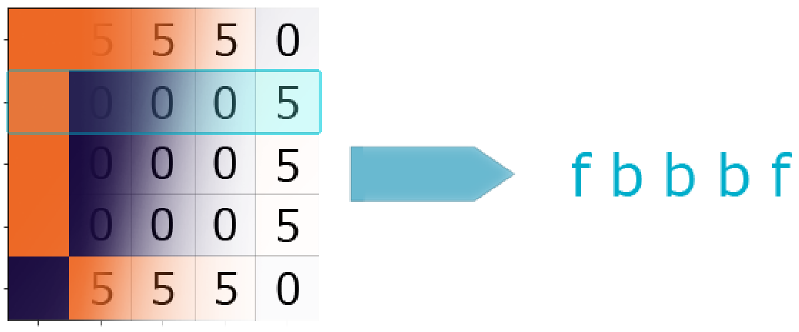


Figure 3.1: From matrix to metacat input.

to ignore a part of an input that would be sent to Metacat, it can either be completely omitted, or changed to a letter later in the alphabet, but preferable not close to the end, so as to not trigger a reverse order analogy in Metacat.

Chapter 4

Realisation and Evaluation

This chapter will give an overview over the program implementation, describe how the tests were run, and describe and the most promising tested tasks in more detail. The plots shown show the input on the left, the output, if any, on the right, and sometimes the difference between the two in the middle. Additional Examples can be found in Appendix C.

4.1 Implementation

The Program is implemented in Python, version 3.8, with the Spyder IDE, version 5.0. The Implementation consists of 7 files, which interact as shown in Figure 4.1.

4.1.1 Conversion

As Metacat is built for solving abstract reasoning problems on the basis of text strings, the values are converted from numbers to letters first, using a straightforward mapping of alphabetical order: '0 $\rightarrow$ a', '1 $\rightarrow$ b', and so forth. Columns and rows that turn out the same are grouped manually, so as to not repeatedly calculate the same operation. The generated converted rows and columns are then manually entered into Metacat's GUI. The Object finder does not directly communicate with Metacat.

4.2 Testing Copycat

Due to the age of the original Copycat, which was written in Lisp, a Python Implementation available from the fargonauts github [1] was used. As expected, the test with Copycat was short and fruitless. Copycat itself, though performing relatively fast, can only process examples with a change at the end of the starting string. This makes it fundamentally unsuited to the task of changing the 'filling' of an object, which implies changes within.

[1]Availble at https://github.com/fargonauts/copycat. The git also has versions of SeqSee and other projects based on this architechture. Accessed 9/11/21

4.3 Testing with Metacat

Initial tests with Metacat started a lot more promising. The program was able to solve tasks with simple hollow objects, which have to be filled with a different colour. The filling colour, in the first example, is constant.

Through testing, Metacat has shown to have significant difficulties with a changing of string length between example input and testing input. As an example, giving the program 'bab → bdb' as an example, it will easily solve letter-string problems of the same type and length, such as 'cbc → ?', but may take significantly longer, or sometimes even produce no result, when the middle letter group is extended to two letters, as in 'daad'. The same problem also arises when the change in length of a letter group is not in the group to be changed. An example of this is 4.2, where Metacat and very quickly presents a result, but completely ignores the extra letter at the end of the testing input.

While 'dad → ded, daad → ?' was solved to 'deed' relatively quickly on a fresh memory, the following 'daad → deed, daaad → ?' couldn't be solved, and giving it 'daad → deed, daaad → deeed' to learn let to a bizarre spectacle wherein Metacat would rapidly declare it had an idea, then either report having made progress or not, before repeating this action. This repeated approximately 30-40 times — it's hard to tell in retrospect as the commentary backlog cut off after 26 instances of this — while showing its current suspicion as to what operation would lead to the given result in the workspace, almost every single one of which contained the correct solution "change letter-category of middle group to 'e'".4.3 It has be assumed that either, the actual solution

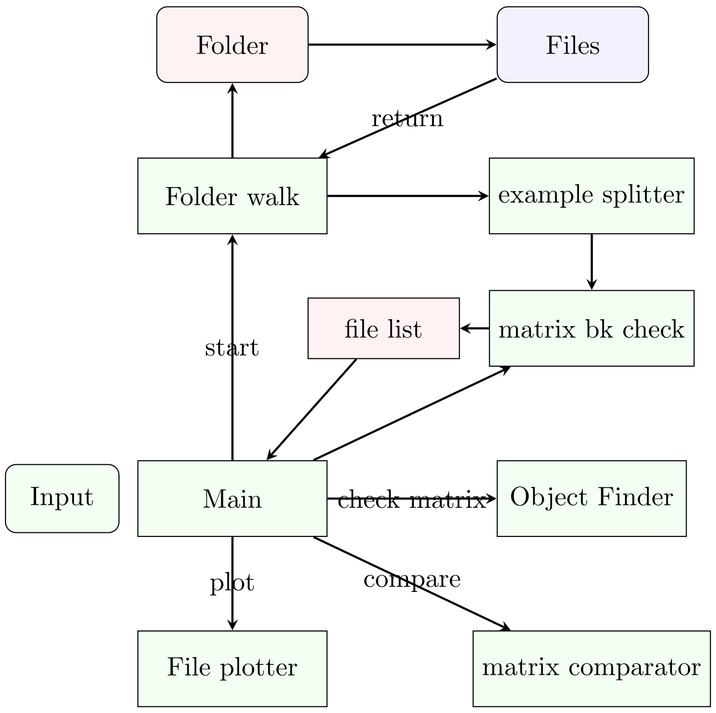
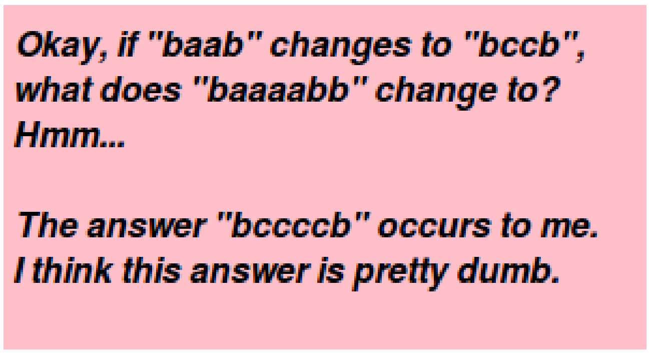


Figure 4.1: Program Layout

was too simple for Metacat, or it somehow couldn't easily make the transfer.

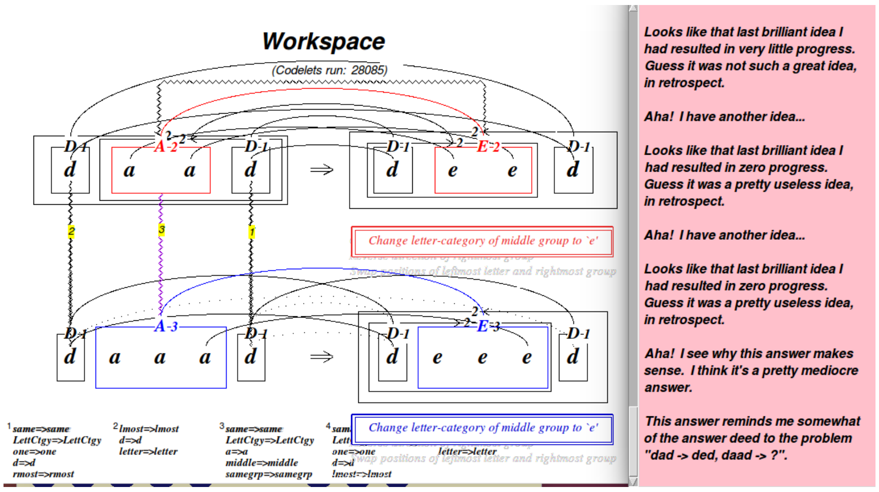
As expected by this point, giving it a challenge will multiple, if similar, groups of the same letter changing in length by an equal amount yielded no useful results, an excerpt of which can be seen in 4.4. The main difference to previous tests is that in this instance, metacat didn't even find any connections to try out and then discard. It just resulted in multiple cycles of "I have no idea", followed by a failure response. Clearly, the internal representation of letter groups still relies at least partially on size, or alternatively, the program is unable to divorce the change to group length with the change in letter type and searches for connections where there are none. Due to those differences, most test runs except for file 0, example 0 were started with a run of 'bab → bcb , baab → bccb' to hopefully give Metacat a slight hint. The test sample size was not big enough, however, to evaluate if this had any effect.



Okay, if "baab" changes to "bccb", what does "baaaabb" change to? Hmm...

The answer "bccccb" occurs to me. I think this answer is pretty dumb.

Figure 4.2: Metacat taking extremely long to understand the similarity of different length strings.


Workspace
(Codelets run: 28085)
D-1
A-2
D-1
d
a
a
d
⇒
D-1
E-2
D-1
d
e
e
d
2
3
1
Change letter-category of middle group to `e'
Swap positions of leftmost letter and rightmost group
D-1
A-3
D-1
d
a
a
a
d
⇒
D-1
E-3
D-1
d
e
e
e
d
Change letter-category of middle group to `e'
Swap positions of leftmost letter and rightmost group
1 same=>same LettCtgy=>LettCtgy one=>one d=>d rmost=>rmost
2 lmost=>lmost d=>d letter=>letter
3 same=>same LettCtgy=>LettCtgy a=>a middle=>middle samegrp=>samegrp
4 same Lett one=>one d=>d lmost=>lmost letter=>letter

Looks like that last brilliant idea I had resulted in very little progress. Guess it was not such a great idea, in retrospect.

Aha! I have another idea...

Looks like that last brilliant idea I had resulted in zero progress. Guess it was a pretty useless idea, in retrospect.

Aha! I have another idea...

Looks like that last brilliant idea I had resulted in zero progress. Guess it was a pretty useless idea, in retrospect.

Aha! I see why this answer makes sense. I think it's a pretty mediocre answer.

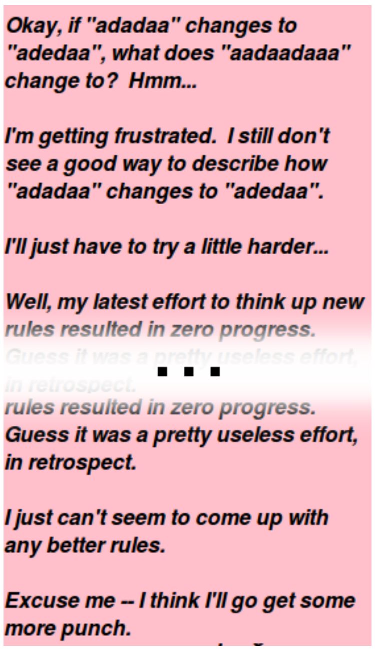
This answer reminds me somewhat of the answer deed to the problem "dad -> ded, daad -> ?".

Figure 4.3: Metacat making a connection to a previous analogy.

4.3.1 Preliminary Tests and Test Planning

Pre-Tests were done with small matrices generated for this task, primarily a hollow object to be filled(4.5. The Object recognition was tailored towards this and performed as expected. Additionally, an 'incomplete' object, the likes of which appear in some ARC tasks, was also used for a cursory function test, with the entire matrix being a 2-colour pattern and the task being to replace 1 of the colours(4.6).

Testing Approach. For testing, the objects detected are sliced sliced into rows and columns, which are converted to letters and then individually given to Metacat. The Matrices themselves will also be subjected to the same treatment. As Metacat either attempts to generate an analogy based on an example change, or reasons about one if given another one as a base, it needs to always receive two string conversions, at least one of which needs to be complete. When a new file is picked, a human-generated string is first attempted to estimate the chance of success, saving time on less promising tasks. For early training, Metacat will receive two full strings from the list, though for testing it is manually assured that they are not identical. It then receives extra strings until it either solves all, or continued testing becomes unsustainable, which did happen several times due to the program failing to respond. A task is considered a success if Metacat manages to solve a combination of rows and columns that would reproduce the entire object, ignoring background cells and strings that don't change. This is admittedly generous, as it is not guaranteed that a program would otherwise be able to shape an object from the pieces. Preliminary testing has demonstrated a high



Okay, if "adadaa" changes to "adedaa", what does "aadaadaaa" change to? Hmm...

I'm getting frustrated. I still don't see a good way to describe how "adadaa" changes to "adedaa".

I'll just have to try a little harder...

Well, my latest effort to think up new rules resulted in zero progress. Guess it was a pretty useless effort, in retrospect.

■ ■ ■

rules resulted in zero progress. Guess it was a pretty useless effort, in retrospect.

I just can't seem to come up with any better rules.

Excuse me -- I think I'll go get some more punch.

Figure 4.4: Changing the length of multiple letter groups proved an impossible task to overcome.

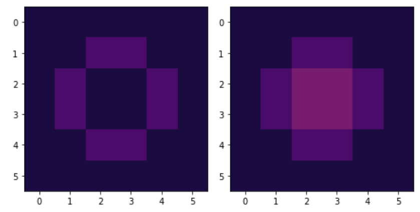


Figure 4.5: **Test Matrix 1**, '6d75e8bb.json', A task that Metacat can succeed at.

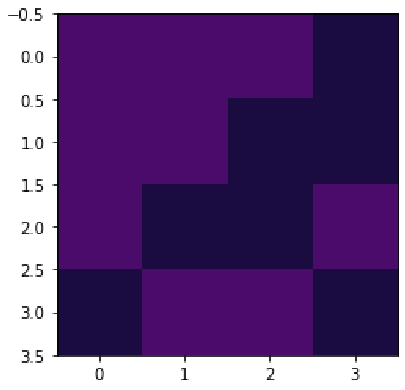


Figure 4.6: **Test Matrix 2**

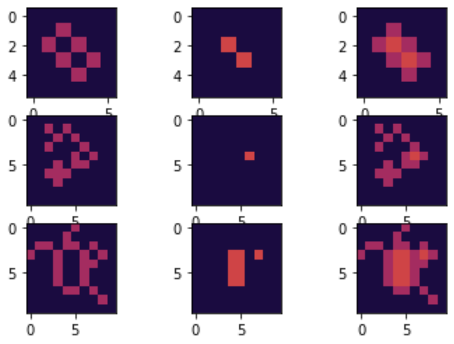


Figure 4.7: **File 1**, '00d62c1b.json'. Partially solvable by metacat.

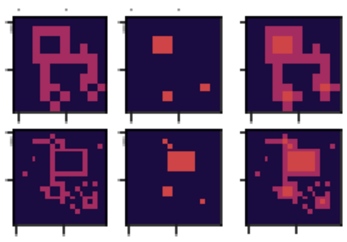


Figure 4.8: Adding visual clutter notably increases the required time.

time requirement for analyzing longer strings, so in this case, too, the field of focus will be manually picked.

4.3.2 Case 1

The testing with simple matrices went relatively well. The examples in File 1(4.7), just require changing the inside of 'hollow' objects, or, in more technical terms, to pick all cells of background colour that are are not connected to the edge of the matrix by other background-coloured cells, and change their colour to a different one, which is always the same in all examples of this file.

The test when ran into the first real issue with Example 3. The challenge contains the same individually simple task, but requires changing two groups in the same line or column in at least one instance. One problematic area is shown zoomed in in 4.9. In this case and similar cases, Metacat, which is only fed individual slices, will in one instance receive 'babaab', and in the other 'baabab', and in the first instance, only the larger group of 'a' would be replaced by c or some other letter, and in the latter case, all 'a' would be replaced instead. If the analogy-making program has information about the surrounding area, this is less of a problem. As the object finder does list all cells of an object that actually belong to the object, this is at least theoretically possible, though it was not implemented for this test. Manually changing the data to change the 'outside' background values to a different colour allowed Metacat to succeed, once. In follow-up tests, when a regular try was clearly not going to work, the "outside" cells were just omitted entirely. Results are listed in 4.1.

Example	Success(base)	Success(mod)
1	3/3	3/3
2	1/3	3/3
3	0/2	1/3
4	0/1	1/1
5	-	0/1

Table 4.1: Results of testing File 1 without and with manual modifications

Overall, the results were not very promising, but hinted at the program's abilities.
Of note in this case test is the inherent learning ability of Metacat, which may end up being counter-productive: After trying a few times without success to solve a manually simplified version of example 7(see 4.3.5), Metacat failed the task 'dad → ded, daad → ?', that is, it couldn't transfer the a→e to a longer letter-group. After clearing the memory, Metacat easily found the solution, running just 1503 codelets.

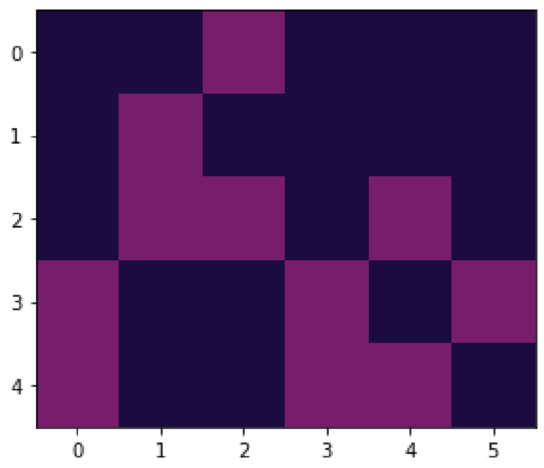


Figure 4.9: **Example 3**, zoomed in.

Case 1A

While filling a hollow object has been shown to be a challenge to Metacat due to not being able to perceive the entire object, completing a square is something it can potentially solve well, and for the same reason. 4.10 consists of tasks that require completing a shape with a different colour. Metacat did not succeed every time, especially early into a run, but running just a few

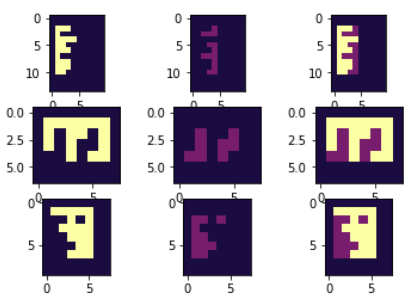


Figure 4.10: **File A1**, '6d75e8bb.json', A task that Metacat can succeed at.

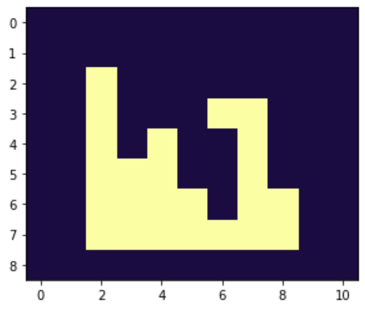


Figure 4.11: Example Plot

reasoning tasks for training would help it achieve an almost 100% success rate, not counting an early failure. While this works well in this case, the fact that the base shape for objects in the object detector just so happens to be a square isn't really a success for the system. Of note, Metacat only managed to reliably solve the analogies on the short axis of each object, and not, for Example, the right-most column of the object in example 1. As expected, Metacat then struggled on the test input (4.11) as it was a length it had not yet seen in this run.

Case 1B

4.12 is yet another example of the "complete object" paradigm, in this case even without any background around it. This makes sure that slicing the whole matrix creates solvable tasks for Metacat, though the used object detection may produce objects that, in a fully automated system, would also be analyzed, potentially confusing the system if the memory was not cleared between the two approaches. This was not tested, however.

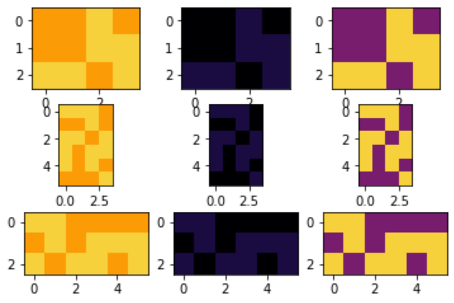


Figure 4.12: **File B1**, 'b1948b0a.json', another example of Object filling.

4.3.3 Case 2

The second file to be deemed viable by the program's filter, Example-Set 2 4.13 is an interesting case of showing potential for a correct output, but then falling short due to no fault of Metacat. While the repetition of an existing pattern is potentially a learnable analogy, the extra dimension of the problem space can not be solved in this case without repetition or extra detection features. The program would be required to run a 1-dimensional analogy detector on the first row and column, then repeat the process for the remaining rows and columns on the already modified matrix, taking into account the contents of the already modified lines. At this point, a detector with the required pattern recognition might as well solve the problem itself. Tests with Metacat did never produce the correct solution in 3 tries, even when just feeding it the first row and column of the first example, possibly due to the length of the resulting string. Maybe exactly one repetition would be more likely to succeed, as a test with a cut-down row did at least recognise a connection. Admittedly, a letter-string analogy of 'abc #*rightarrow* abcabcab' might look a bit odd, though most people would have no problem with transferring this over as "repeat two times, then cut off the last letter". Similarly, an attempt with SeqSee made for comparison, while able to detect the repetition as one possible input, did not provide an easy avenue to apply this to following rows or columns, nor was the output consistent. Also, the nature of the spread out lines in lower examples could turn out to be another roadblock. This might be an application for a pattern-recognition neural network, but not for Metacat.

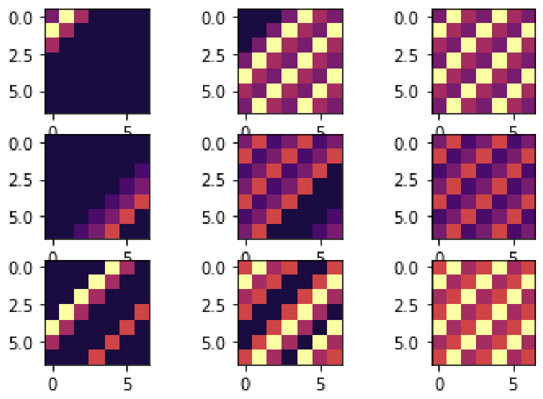


Figure 4.13: **File 2**, '05269061.json'. Not solvable by Metacat.

4.3.4 Case 3

Example-Set 3(4.14) is theoretically solvable by Metacat with significant pre-filtering. The main challenge here is recognizing the grid as background noise. After filtering this out manually and giving metacat converted columns and rows of these examples, it was able to provide a solution more often than not, though this fluctuated heavily with restarts and different generated seeds. On an abstract level, this set of examples shows lines of objects that form larger, 'macro'-objects, not too dissimilar to the objects in Example 1. The main challenge here is recognizing the grid as background noise. A form of pattern recognition would have been required to recognize and filter out the grid, followed by recognizing the larger squares as individual cells, as Metacat still requires a single, one-dimensional string as input.

However, this was not implemented for this test run. Manually filtering out the grid and just feeding Metacat the remaining data enabled it to succeed occasionally at connecting two objects, though this fluctuated heavily with restarts and different generated seeds, and didn't once manage to get an entire matrix solved in one go. Notably, Metacat exhibited its previously shown problem with scaling, meaning that just giving it complete columns, compressed as mentioned above, would nearly always lead to failure. Trying to alleviate this issue by sending Metacat queries consisting just of two squares from the input matrix and the empty squares between them would already require a rough understanding of the task at the detection level, and arguably be more effort than the remaining task to be solved.

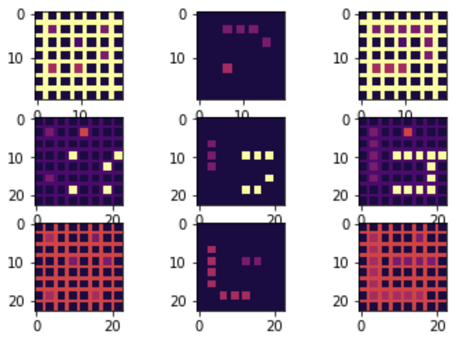


Figure 4.14: **File 3**, '06df4c85.json'. Only theoretically accessible to Metacat.

4.3.5 Case 4

The fact that the challenges in 4.15 have features of width >1 and thus require more than one dimension make them unsuited for an unmodified metacat. Trying the first example in metacat, flatted to one string as 'aaecaa → aaeeee', 'aaceaa → eeeeaa' led to no result after several minutes of trying and 25136 codelets run. However, the workspace looked somewhat promising(4.16). Metacat could clearly make connections between not just the groups of a's that stayed the same, but also from an 'a' group to the larger 'e' group in the outcome. It did, however, also connect each group with its counterpart in the other example, and thus failed to recognize the inverted symmetry. It is possible that a Metacat run starting with tasks involving a lot of invert string operations would solve this.

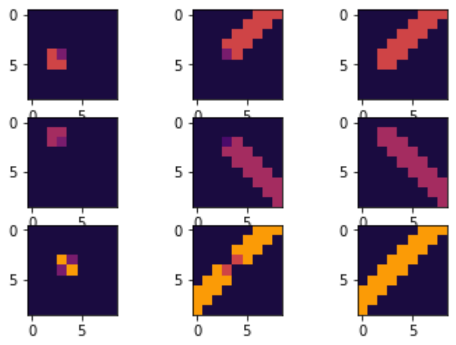


Figure 4.15: **File 7**, '1f0c79e5.json'

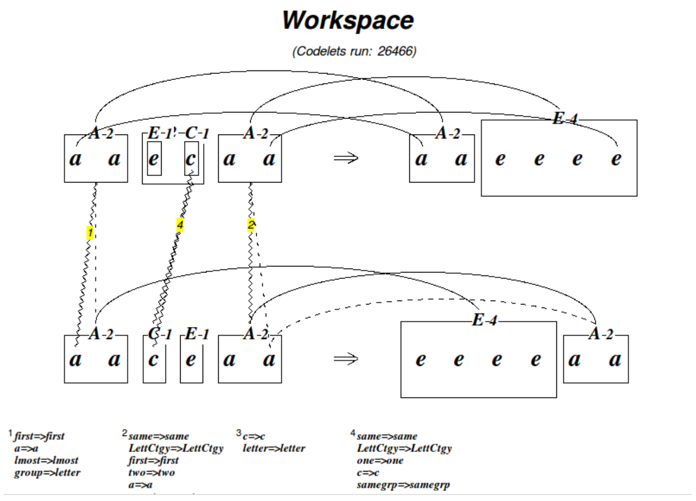


Figure 4.16: Outcome of attempting a manually simplified Example from File 7.

Chapter 5

Conclusions

As somewhat expected by its non-deterministic layout, Metacat has proven difficult to test, as even repeating the exact same inputs in the same order, with the same starting seed, may at some point provide deviating results. The possibility can not be ruled out that, in different circumstances, Metacat would succeed at a task that it has failed at in this evaluation.

The tests have hinted at the potential of a Copycat-like architecture in solving visual analogy problems, but the algorithm has turned out to be poorly scalable. Metacat has proven to be able to solve analogy problems of the type seen in a few of the tasks in the ARC, yet then repeatedly had difficulty solving the actual tasks if encountering minor differences a human testee might gloss over. Not unlike a Neural Network, for which the possibility of these kinds of problems is well known, Metacat would often find a solution for an analogical problem involving changing one letter to a different one, and then struggle or even fail at the same problem, if the input string was just one letter longer. It has to be said that the analogies which could be abstracted or sliced sufficiently to yield a valid input for Metacat always ended up relatively simple, not making use of the capabilities of the program related to conceptual slippage, like connecting position in the string with alphabetical order. In most cases where Metacat could arrive at a solution, it rated the results as 'mediocre' or 'dumb', which is Metacat's way of communicating that those were relatively simple, and watching the workspace showed that in at least two cases, the program found a solution early, but kept searching for something more intricate. Looking at these, Metacat ended up being a strangely good representation of a human test-taker, which will likely also not use every tool in his mental library to solve a simple visual analogy. At other times, the tasks very much looked like solving them would benefit from the application of just those concepts, but there was no easy way, even manually, to translate them from a two-dimensional problem to a one-dimensional string without losing some crucial piece of information.

In the end, the reason this architecture doesn't get as much spotlight in current AGI debate is likely simply that it is quite different from contemporary approaches, and that too much initial effort would be required to adapt it.

Challenges	Result
Hollow Objects	Mostly successful, though not always consistently.
Multiple Hollow Objects	Success depending on Object Detection.
Objects with multiple changes	Failure if in the same line.
Connections between objects	Similar to hollow objects, at significant time cost
Incomplete objects	Almost completely successful, iff a square
Noisy Images	Potential success, but confuses Object Recognition

Table 5.1: Evaluation

5.1 Further Work

The results of some tasks have hinted at the possibility for a copycat-based architecture to solve them, if more information were available to the program. It might be worth exploring just how far this approach can be pushed with a better object detection. In some cases, the lack of an extra dimension could be compensated for with a longer 'alphabet', allowing the encoding of complex 2-dimensional problem spaces into one-dimensional strings, or through a more advanced object classification, which would be able to allow at least some tasks to be solved by abstracting the input until it can be represented by letter strings. A more detailed evaluation would probably make necessary a less rudimentary object generation and handling to automate a larger share of the work. The most likely approach to yield results, however, would be a purpose-built program that is designed from the start to take multi-dimensional input. However, it has to be assumed that the effort likely required for this would be in the realm of years, rather than months.

Bibliography

Anderson, J. R., Matessa, M., and Lebiere, C. (1997). ACT-R: A theory of higher level cognition and its relation to visual attention. *Human-Computer Interaction*, 12(4):439–462. 5

Banburski, A., Ghandi, A., Alford, S., Dandekar, S., Chin, P., and Poggio, T. (2020). Dreaming with arc. Technical report, Center for Brains, Minds and Machines (CBMM). 7

Boland, S. (2014). Javacat. Copycat in Java, https://archive.org/details/JavaCopycat. 8

Brogan, J. A. (2017). Pythoncat. Copycat in Python, https://github.com/fargonauts/copycat. 8

Buchanan, B. G. (2005). A (very) brief history of artificial intelligence. *Ai Magazine*, 26(4):53–53. 1

Chollet, F. (2019a). Abstract reasoning challenge. 6

Chollet, F. (2019b). On the Measure of Intelligence. *arXiv preprint arXiv:1911.01547*. 4, 5

Cobbe, K., Klimov, O., Hesse, C., Kim, T., and Schulman, J. (2019). Quantifying generalization in reinforcement learning. In *International Conference on Machine Learning*, pages 1282–1289. PMLR. 5

Ellis, K., Wong, C., Nye, M., Sable-Meyer, M., Cary, L., Morales, L., Hewitt, L., Solar-Lezama, A., and Tenenbaum, J. B. (2020). Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381*. 7

Engl, H. W., Hanke, M., and Neubauer, A. (1996). *Regularization of inverse problems*, volume 375. Springer Science & Business Media. 7

French, R. and Hofstadter, D. (1992). Tabletop: An emergent, stochastic model of analogy-making. In *Proceedings of the 13th Annual Conference of the Cognitive Science Society*, pages 175–182. Citeseer. 8

Gentner, D. and Markman, A. B. (1994). Structural alignment in comparison: No difference without similarity. *Psychological Science*, 5(3):152–158. 4

Gick, M. L. and Holyoak, K. J. (1983). Schema induction and analogical transfer. *Cognitive psychology*, 15(1):1–38. 4

Goertzel, B. and Pennachin, C. (2007). *Artificial general intelligence*, volume 2. Springer. 3

Hofstadter, D. R. and Mitchell, M. (1994). The Copycat project: A model of mental fluidity and analogy-making. In *Analogical connections*, pages 31–112. Ablex Publishing. 7, 8

Huang, H. (2018). Reimplementation and reinterpretation of the copycat project. *arXiv preprint arXiv:1811.04747*. 8

icecuber (2020). Winning solution of the abstract reasoning challenge. https://github.com/top-quarks/ARC-solution. 6

Kühl, N., Goutier, M., Baier, L., Wolff, C., and Martin, D. (2020). Human vs. supervised machine learning: Who learns patterns faster? *arXiv preprint arXiv:2012.03661*. 4

Kolev, V., Georgiev, B., and Penkov, S. (2020). Neural abstract reasoner. 7

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105. 5

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551. 5

Legg, S., Hutter, M., et al. (2007). A collection of definitions of intelligence. *Frontiers in Artificial Intelligence and applications*, 157:17. 4

Lenat, D. B., Prakash, M., and Shepherd, M. (1985). Cyc: Using common sense knowledge to overcome brittleness and knowledge acquisition bottlenecks. *AI magazine*, 6(4):65–65. 5

Lin, D., Dechter, E., Ellis, K., Tenenbaum, J. B., and Muggleton, S. H. (2014). Bias reformulation for one-shot function induction. 4

Mahabal, A. A. (2010). *Seqsee: A concept-centered architecture for sequence perception*. PhD thesis, Indiana University. 8

Marcus, G. (2018). Deep learning: A critical appraisal. *arXiv preprint arXiv:1801.00631*. 5

Marshall, J. B. (1999). *Metacat: A self-watching cognitive architecture for analogy-making and high-level perception*. Indiana University. 8

Marshall, J. B. (2006). A self-watching model of analogy-making and perception. *Journal of Experimental and Theoretical Artificial Intelligence*, 18(3):267–307. 8, 10

McCarthy, J., Minsky, M. L., Rochester, N., and Shannon, C. E. (2006). A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI magazine*, 27(4):12–12. 3

Mitchell, M. (1990). *Copycat: A computer model of high-level perception and conceptual slippage in analogy making*. PhD thesis, University of Michigan. 7

Mitchell, M. (1993). *Analogy-making as perception: A computer model.* Mit Press. 7

Moosavi-Dezfooli, S.-M., Fawzi, A., Fawzi, O., and Frossard, P. (2016). Universal adversarial perturbations. In *CVPR*. 5

Mopuri, K. R., Garg, U., and Babu, R. V. (2017). Fast feature fool: A data independent approach to universal adversarial perturbations. *arXiv preprint arXiv:1707.05572*. 5

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. 4

Newell, A. and Simon, H. A. (1961). Gps, a program that simulates human thought. Technical report, RAND CORP SANTA MONICA CALIF. 5

Nichols, E. P. (2012). *Musicat: A computer model of musical listening and analogy-making.* PhD thesis, Indiana University. 8

Pearl, J. (1985). Bayesian netwcrks: A model cf self-activated memory for evidential reasoning. In *Proceedings of the 7th conference of the Cognitive Science Society, University of California, Irvine, CA, USA*, pages 15–17. 5

Pezzuti, L., Tommasi, M., Saggino, A., Dawe, J., and Lauriola, M. (2020). Gender differences and measurement bias in the assessment of adult intelligence: Evidence from the italian WAIS-IV and WAIS-r standardizations. *Elsevier*, 79:101436. 4

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252. 1

Schmidhuber, J. (2014). Deep Learning in Neural Networks: An Overview. *Neural Networks*, 61. 5

Turing, A. M. (2009). Computing machinery and intelligence. In *Parsing the turing test*, pages 23–65. Springer. 1

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and Contributors, S. . (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272. 35

Wicherts, J. M. and Dolan, C. V. (2010). Measurement invariance in confirmatory factor analysis: An illustration using iq test performance of minorities. *Educational Measurement: Issues and Practice*, 29(3):39–47. 4

Winston, P. H. (1979). Learning and reasoning by analogy: The details. *AIM*, 520. 3

Appendix A

Relational Concepts

This Appendix Chapter goes into further detail on relational concepts not listed in Chapter 3, most of which were not further explored in the presented algorithm. The listed concepts are split into detection of present objects and existing relational concepts in the shown matrices(Existing concepts), and Operations to be performed whose execution would lead to the difference between the known input and output (Operations:), and listed within those sections in the order in which the author deemed them useful while manually scanning the first 20 unfiltered example files of the ARC in alphabetical order, starting with '00d62c1b.json' and ending with '1f642eb9.json'. Detecting Existing Concepts and Operations may be seen akin to detecting concepts exhibited in the shown matrices and the difference between those matrices, respectively. After the first 10 examples, the first repetitions occurred, that is, concepts or operations deemed helpful or necessary for solving the presented challenge in a previous example we deemed useful again in a new one. Assuming a finite amount of logical concepts, an algorithm capable of learning those concepts would eventually be able to detect and hopefully solve all presented analogy problems. This was not pursued further as it promised no immediate insight about employing Metacat on a subset of the ARC.

A.1 Existing concepts

Describes pre-defined concepts the program is expected to detect in matrices. Some of those may not need to be defined anymore if pattern recognition becomes good enough to generate them during runtime.

- Detect objects. The difficulty of this depends on the noisiness of the image. This has been implemented for simple objects.
- patterns. Especially pattern recognition is a challenge, as the program needs to see not only repetition, but also notice a lack thereof in an otherwise repeating pattern. Further, the repeating entity, whether its a row of colours, a string of objects, or a more complex interaction thereof, needs to be detected before a pattern can be found.
- Differentiate background from the inside of hollow objects. This has been mostly implemented.

- List relations between filled squares or objects. This would both facilitate abstraction, allowing an easier processing for a problem solver, and allow saving in a logic database.
  - Straight/diagonal
  - Colour of neighbours
- Find output in input (colour or greyscale, straight or turned 90/180/-90°). Has been deemed unnecessary for this test as the ideal result would, through turning the image before further analysis, yield no different output to be passed on than a task that didn't require it in the first place.
- Order squares into "areas" and columns.
  - Empty surrounded is an area,
  - colour surrounded by different colour is also an area.
  - Colour surrounded by different colour/edge on each side is also an area A human concept, it so far has not proven necessary.
- Detect Matrix-wide/Object-wide pattern
- Detect Mirror (halves, quarters)
- Detect squares/lines/circles
  - Subconcept: Separator-line, Selector-box
- Group objects of same shape/colour
- List all squares/lines/circles, group by colour.
- Detect "all input"/"all output" having quality x (f.ex. colour)
- Find nearest (Requires a decent object detection in case it needs to detect the 'nearest' of something specific).
- Detect straight/diagonal

Ideally, will list reasoning for each defined concept (as in, why is it necessary, why can't it instead be generated).

A.2 Operations:

() Lists operations that can be observed between input and output matrices.

- Move closest to feature (column, area, etc.)
- Transfer relations from one object to another object
- Subtract objects/areas
- Generate the same pattern repeatedly, with colour-relations
- Move object in relation / to reach relation to other object

    - Gravity
    - Move to / on / to nearest

- “Connect” Objects, shortest path/straight line
- Mirror one half to other, centered or at cluster center, also transposed
- Detect Object based on Interaction between other objects (cross = Intersection of Lines)

Appendix B

Code Details

The presented program for investigating the feasibility of using metacat to solve a subset of ARC problems was implemented in Python 3.8. Python was chosen because it was the programming language used in the Abstract Reasoning Challenge based on the ARC, does not require a lengthy compilation before testing changes, and offers relatively easily human-readable code. It also supports both object-oriented and functional, as well as, through packages, even logic programming. This chapter describes the presented python program in more detail, listing all the modules and what exactly they do. The program was implemented in the Spyder IDE, version 5.0, and makes use of the Spyder GUI for its plotting function.

B.1 Overview

The program reads out json files containing sets of learning examples, consisting of at least two input-output pairs, and one test challenge, also split into an input and an output. All modules are stored in the '/learning' folder, with the example files stored in the subfolder '/learning/training'. Either folder can be renamed, and the program is capable of reading files from a different folder according to user input.

B.1.1 Used Packages

The following external packages were used in the creation of the program:

- **os**: Containing functions concerning the operating system. Specifically, **path** and **listdir** are employed to find and list the provided example files.
- **scipy**: From the SciPy package (Virtanen et al., 2020), **ndimage** is employed for image processing during the filtering of examples to narrow down the problem domain.
- **json**: The package is required to actually read the json files containing the examples
- **numpy**: Used in multiple modules for the analysis of matrices and object detection. Numpy arrays allow a variety of calculations that the list-based standard python matrices don't support.

- **collections** Used for 'Counter' to count the outermost cells in a matrix.
- **matplotlib** This package, through **pyplot** and **cm**, facilitates creating visual representations of the example matrices, significantly increasing human readability.

B.2 Modules

The modules will be listed here with a short explanation and additional commentary where necessary. All modules were written in Spyder 5.0, with UTF-8 encoding. The code is extensively commented, which should explain most of the present functions. Code presented here will have additional line breaks to fit the page.

B.2.1 Main

This is the main method for the Master Thesis, and contains print-outs to ease usage through a non-IDE console, as well as extra command listings not shown here.

```
import time #For console outputs from this file
from Folder_walk import find_examples
from Folder_walk import list_files
from matrix_comparator import matrix_comp
from file_plotter import plot_ex as plex
from Object_finder import obj_gen
# Imported for checking matrices:
from matrix_bk_check import bkcheck as bkcheck
from matrix_bk_check import mat_count as mc

work_file_list = [] #

def start(folder = "training/"):
    """
    Start function of the method, and the entire program.
    Parameters
    ----------
    folder : String, optional
        The folder that the example files are to be found in.
        The default is "training/".
    Returns
    -------
    As this is intended to be used from a console, it just
        produces print-outs.

    """
    global work_file_list #Allows the file list to be used by
        other functions
                          # without a return on the start
                            function.
    work_file_list = find_examples(folder)
    if work_file_list != []:
        print("Files have been found.",
              "Available commands: filelist(), plot(),
                 check_matrix(), compare().")
        print("For more details and advanced commands, type '
            commands()'.")



def filelist(fileA = 0,fileB = 10):
    """
    Allows listing files.

    Parameters
    ----------
    fileA : Int, optional
        Index of the file the list output should start with.
            The default is 0.
    fileB : Int, optional
        Index of the file the list output should end with. The
             default is 10.

    Returns
    -------
    list_files: List
        A List of tuples of file index and file name.

    """
    return list_files(work_file_list ,fileA ,fileB)



def plot(File = 0, test = False, cap = 5, difmap = True ):
    """
    Generates plots, if run in an environment with provision
        for that, like
    an IDE with visual output.

    Parameters
    ----------
    File : Int, optional
        Which file's example matrices to plot. The default is
            0.
    test : Bool, optional
        If, instead of training examples, the test matrix
            should be plotted.
        The default is False.
    cap : Int, optional
        How many matrices should be plotted. Always starts at
            the first, so it is
        not possible to just plot the last 2 examples in a
            matrix on their own.
        The default is 5.
    difmap : Bool, optional
```

```
            If the difference between input and output should also
                be plotted.
            The default is True.

        Returns
        -------
        Image.png
            Returns an image of file type .png.

        """
      #Target function def: (train_in,train_out = [], is_test =
          False, cap = 5, detail = 3)
        show_diff = (len(work_file_list[File][1][0])*len(
            work_file_list[File][1][0][0]) ==
                      len(work_file_list[File][1][1])*len(
                          work_file_list[File][1][1][0]))
                    #Is showing a difference even possible? (same
                        size arrays)
        if show_diff and difmap and not test:
            return plex(work_file_list[File][1][0],work_file_list[
                File][1][1])
        elif test:
            return plex(work_file_list[File][1][2],[],test, cap)
        else:
            return plex(work_file_list[File][1][0],work_file_list[
                File][1][1],test, cap, 2)


def compare(File = 0,Example = 0,Print = True):
        """
        Compares two matrices, producing print-outs if desired,
        and returning a list of linked objects

        Parameters
        ----------
        File : Int, optional
            File Index from work_file_list. The default is 0.
        Example : Int, optional
            Which example to pick from the file. The default is 0.
        Print : Bool, optional
            If additional Print-Outs are desired. The default is
                True.

        Returns
        -------
        None.

        """
        matrix_comp(work_file_list[File][1][0][Example],
                    work_file_list[File][1][1][Example],
                    obj_gen(work_file_list[File][1][0][Example
                        ],[],0,True,[]),
                    obj_gen(work_file_list[File][1][1][Example
                        ],[],0,True,[]),
                mc(work_file_list[File][1][0][Example]),
                mc(work_file_list[File][1][1][Example]),
                Print) #The Boolean turns on print outputs for
                    console operation.

    pass

def check_matrix(File = 0,Example = 0, inout = 0, cbk = 0):
    """
    Compares two matrices, returning a difference map matrix
        as well as a
    list of all rows and columns where changes happen. Tries
        to match macro
    objects.

    Parameters
    ----------
    File : Int, optional
        Index of parsed json or txt file in the work_files
            list.
        The default is 0.
    Example : Int, optional
        Index of Example in the file. Values usually from 0 to
            4.
        The default is 0.
    inout : Int, optional
        0. The default is 0.
    cbk : Matrix background colour, optional
        Uses this colour as the background colour for
            calculations.
        If set negative, will try to figure out the background
            colour
        itself by calling bkcheck.
        The default is 0.

    Returns
    -------
    List
        Returns an obj_gen output from Object_finder.

    """

    sOb,mOb,obmat = obj_gen((work_file_list[File][1][inout][
        Example]),[])
    print(sOb," small Objects were found, forming ", len(mOb)
        ," macro objects.")
    if cbk == 0:
        return obj_gen((work_file_list[File][1][inout][Example
            ]),[])
    elif cbk > 0:
        return obj_gen((work_file_list[File][1][inout][Example
            ]),[],cbk)
    else: # if cbk < 0, and a background check is desired.
        bc = mc(work_file_list[File][1][inout])
        mTot, mSiz, mBd = bc[0],bc[4],bc[3]
        cbk =  bkcheck(mSiz, mTot, mBd[0])
        return obj_gen((work_file_list[File][1][inout][Example
            ]),[],0)

    sOb,mOb,obmat = obj_gen((work_file_list[File][1][inout][
        Example]),[])
    print(sOb," small Objects were found, forming ", len(mOb)
        ," macro objects.")
    return obj_gen((work_file_list[File][1][inout][Example])
        ,[])
```

B.2.2 Folder walk

This method contains two functions, `find_examples and list_files`, responsible for reading and listing all files with a specific ending in a folder, and outputting part of that list on request, respectively. Is called by main.

```
from os import listdir
from os.path import isfile, join
import scipy.ndimage as ndi

import example_splitter as xsplit
from matrix_bk_check import bkcheck as bkcheck
from matrix_bk_check import mat_count as mc
from matrix_comparator import matrix_dif as mdiff

def find_examples(folder_path = "training/", ex_cap = 5,
                  objectlimit = 9, macrolimit = 1):
    """
    Scans the given folder, finding all files, calls other
        modules to read
    and filter them.
    Afterwards, the content of the files is put into a list.

    Parameters
    ----------
    folder_path : String, optional
        The path where the example files are to be found.
        The default is "training/".
    ex_cap : Int, optional
        How many examples should be pulled. It's usually less
            than 5.
        The default is 5.
    objectlimit : Int, optional
        Ignores Matrices with more than that many detected
            objects.
        Meant to limit the testing domain to smaller problems
            that can be
        checked in reasonable time.
        The default is 9.
    macrolimit : Int, optional
        How many Macro Objects are acceptable in the input
            matrix.
        Just like objectlimit , this is meant to limit the
            problem size for
        a first evaluation; If there are problems with few
            objects ,
        more will not solve them.
        The default is 1.

    Returns
    -------
    work_files : List
        A list of all the work files.

    """
    # folder_path = "training/"
    ex_files = [join(folder_path, a) for a in listdir(
        folder_path) if
                isfile(join(folder_path, a))
                and (a.endswith(".json") or a.endswith(".txt")
                    )]
    work_files = []
    for f in ex_files:

        traindata,testdata = xsplit.read_file(open(f),
            folder_path, ex_cap)
        if mdiff(traindata[0][0],traindata[1][0])[0]:
            m_totals, rows, cols, bd, size = mc(traindata
                [0][0])

            if bkcheck(size,m_totals,bd[0])[0]:
                if (ndi.label(traindata[0][0])[1] <=
                    objectlimit and
                    ndi.label(traindata
                        [0][0],[[1,1,1],[1,1,1],[1,1,1]])[1]
                    <= macrolimit and
                    ndi.label(traindata
                        [0][0],[[1,1,1],[1,1,1],[1,1,1]])[1]
                    <= macrolimit+1 ):
                    #Generating work entity:
                    work_entity = [traindata[0],traindata[1],
                        testdata]
                    work_files.append([f,work_entity])


    print(len(work_files)," files are fulfilling the set
        criteria")

    return work_files

def list_files(file_list,fa,fb):
    """
    Allows returning part of the work files list in a print-
        friendly package.
    Just enter the name of the list, and from where to where
        you want it read.
    Parameters
    ----------
    file_list : List
        A list of example files, with their file name saved at
             [i][1].
    fa : Int
        From where to start returning the file names.
    fb : Int
        Until where to read out the list, including the value
            given.

    Returns
    -------
    list_output : List
        Returns a List of Tuples, with the index number and
            the file name.

    """
    list_output = []
    if fa > fb: # Just a failsave
        fa,fb = fb,fa
    for i in range(fa,min(fb+1,len(file_list))):
        #returns files from index A to B, capped at list
            length
        list_output.append((i,file_list[i][0]))
    return list_output
```

B.2.3 Object Finder

Searches an array for non-background values, then generates objects out of neighbouring cells, before generating macro-objects from those.

```
import numpy as np
from itertools import chain as itch
import scipy.ndimage as ndi
cmass = ndi.measurements.center_of_mass

def get_tuples(matrix, bk = 0, cellcolour = 0):
    """
    Gets all those juicy tuples, and puts them in a list

    Parameters
    ----------
    matrix : np.array
        Input one of the matrices, containing only numbers,
        with already determined background.
    bk : Integer, optional
        What is the perceived background colour of the matrix?
        The default is 0.
    colour : Boolean, optional
        Does colour matter? If not, everything not part of the
             background
         will be selected.
        The default is False.

    Returns
    -------
    LIST
        Lists all non-bk tuples found.

    """
    # if cellcolour == bk:
    #     tup1,tup2 = np.where(matrix != bk)[0],np.where(
        matrix != bk)[1]
    # else:
    #     tup1,tup2 = np.where(matrix == cellcolour)[0],np.
        where(matrix == cellcolour)[1]

    if cellcolour == bk:
        return list(zip(np.where(matrix != bk)[0],np.where(
            matrix != bk)[1]))
    else:
        return list(zip(np.where(matrix == cellcolour)[0],np.
            where(matrix == cellcolour)[1]))

def is_adjac(tup1,tup2, diag = False):
    """
    Checks if two tuples are adjacent, orthogonally or
        diagonally, and outputs a boolean.
    Helper function.

    Parameters
    ----------
    tup1 : tuple (a,b)
        A tuple from the list, as found in the original matrix
            .
    tup2 : tuple (a,b)
        Another tuple from the list, as found in the original
            matrix..
    diag : Boolean, optional
        If false, check tuples for direct adjacency. The
            default is False.
        If true. check tuples for diagonal adjacency.

    Returns
    -------
    bool
        True if adjacent (will be considered part of the same
            object), false if not.

    """

    if not diag and(tup1[0] == tup2[0]) and (tup1[1]-1 == tup2
        [1] or tup1[1]+1 == tup2[1]) or (
            tup1[1] == tup2[1]) and (tup1[0]-1 == tup2[0] or
                tup1[0]+1 == tup2[0]):
        return True
    if diag and (tup1[1]-1 == tup2[1] or tup2[1] == tup1[1]+1)
         and (
             tup1[0]-1 == tup2[0] or tup2[0] == tup1[0]+1):
        return True
    return False

#The Main function of this module:
def obj_gen(matrix,objectlist=[], bk = 0, monochrome = True,
    objectlistmatrix=[], check_macro = True):
    """
    Receives a matrix and a set of variables and returns a
        list of (small) objects,
    a list of macro objects consisting of those smaller
        objects, and a matrix showing
    all found objects ennumerated.

    Parameters
    ----------
    matrix : np.array,
        As provided by the tester or example. Contains ints
            from 0 to 9.
    objectlist : List, optional. The default is [].
        A usually empty list, to be filled with objects found
            in the array.
        Objects, in this case, are sets of adjacent cells (or
            just one cell by itself)
        that have a non-background number.
    bk : Integer, optional
        The determined background number, as identified by the
             matrix bk check.
        The default is 0.
    monochrome : Boolean, optional. The default is True.
        If True, checks for everything that is not background.
        Else, it will check colours individually (small
            objects only).
    objectlistmatrix : np.array, optional. The default is [].
        A matrix generated that shows all identified objects
            with incrementing numbers.
        This is empty when running the fuction, and merely
            exists to make it
        easier to output it into a calling function, if
            desired.
    check_macro: Boolean, optional. The Default is True.
        If the function should try to find macro objects.

    Returns
    -------
     objectlist : List (of Ints, Lists, and Dictionaries)
        Lists all objects in the following format:
            [Object ID (Integer),
             [Individual Cell coordinates (Tuples of two Ints)
                 ],
             []]
    macrolist : List
```

```
        A list of macro objects in the form of dictionaries.
            Described in more
         detail in the "mog" function that generates this list,
             below.
    objectlistmatrix : array
        A dummy matrix for object listing, to make it easier
            to work with a relational object.

    """

    if monochrome:
        mDim = matrix.shape
        tulist= get_tuples(matrix, bk, bk)
        checkedlist = [] # List of tuples already checked in
            the followin while loop
        while tulist != []:
            new_temp_obj = [
                #part of object:
                [],
                #adjacent(diagonal) objects:
                [] # <-- these will be the neighbours, used to
                     find macro-objects
                ]
            new_temp_obj[0].append(tulist.pop(0))

            for j in new_temp_obj[0]:
                #Checks adjacency to generate small objects
                for i in tulist:
                    if is_adjac(j,i):
                        #if j and i are adjacent, they are
                            part of the same object
                        new_temp_obj[0].append(i)

                        if i in new_temp_obj[1]:
                            #If, for some odd reason, a cell
                                is already in the "neighbour"
                            # category when being detected as
                                part of the same object, it
                            # will be removed from the
                                neighbour list.
                            new_temp_obj[1].remove(i)
                        checkedlist.append(i)
                for i in checkedlist:
                    tulist.remove(i) # remove i from the list
                        to not check it again.
                checkedlist.clear()
                for i in tulist:
                    #checks diagonal adjacency
                    if i not in new_temp_obj[0] and is_adjac(j
                        ,i,True):
                        new_temp_obj[1].append(i)

            if not check_macro and ( # Checking for internal
                objects in macro objects
```

```
                    any(elem[0] == 0 or elem[1] == 0 or
                        elem[0]+1 == mDim[0] or elem[1]+1 ==
                            mDim[1] for elem in new_temp_obj
                            [0])
                    ):
                    new_temp_obj = [[],[]]
                    continue # Border cells can't be "internal
                        ".

            objectlist.append(new_temp_obj)

    else:
        for k in range(10):
            if k == bk:
                continue
            tulist= get_tuples(matrix, bk, k)
            while tulist != []:
                new_temp_obj = [
                    #part of object:
                    [],
                    #adjacent(diagonal) objects:
                    [] # <-- these will be the neighbours,
                        used to find macro-objects
                    ]
                new_temp_obj[0].append(tulist.pop(0))

                for j in new_temp_obj[0]:
                    #Checks adjacency to generate small
                        objects
                    for i in tulist:
                        if is_adjac(j,i):
                            new_temp_obj[0].append(i)
                            tulist.remove(i)
                    for i in tulist:
                        #checks diagonal adjacency
                        if i not in new_temp_obj[0] and
                            is_adjac(j,i,True):
                            new_temp_obj[1].append(i)
                        #Currently, does not check for direct
                            adjacency of different colours

                objectlist.append(new_temp_obj)

    #Create a dummy matrix for object listing (helper function
         to make it easier
    # to work with a relational object)
    objectlistmatrix = np.zeros(np.shape(matrix),np.int32)

    #Adds a numeric ID to each object, to make the
        objectlistmatrix at least a
    # little bit human-readable. The IDs start at 1 because
        the background is already 0.

    for k in range(len(objectlist)):
        objectlist[k].insert(0,k+1)
        if not monochrome or bk != 0:
            #Fills the objectlistmatrix manually if colour
                matters
            for m in objectlist[k][1]:
                objectlistmatrix[m] = k+1 #Adding actual
                    objects.

    if monochrome and bk == 0:
        #Using scipy's label function in the standard case of
            background 0,
        # no colour seperation.
        objectlistmatrix = ndi.label(matrix)[0]

    #Uses the objectlistmatrix to convert neighbouring cells
        into object IDs
    for k in range(len(objectlist)):
        tempvar = objectlist[k][2]
        tempvar[:] = [objectlistmatrix[i[0],i[1]] for i in
            tempvar]

    #Adds recursive Neighbour links:
    #As the tuple is removed from the list once it is added to
         an object,
    # later objects will not find the previous objects when
        checking
    # for neighbours. Thus, they are added here.
    for k in range(len(objectlist)):
        if objectlist[k][2] is not []:
            for y in objectlist[k][2]:
                if y>k and not (k+1) in objectlist[y-1][2]:
                    objectlist[y-1][2].append(k+1)


    # Remove useless neighbours (Example: Object 6 has
        neighbour Object 6)
    for x in objectlist:
        x[2] = list(filter(lambda y: y != x[0], x[2]))
    #Call function to generate Macro-Objects out of adjacent
        small objects:

    macrolist=[]
    if check_macro: # Does not check for macro objects if set
        to False.
        mog(objectlist, matrix, macrolist, bk)
        print(len(objectlist)," entities were found, forming a
            total of ",
              len(macrolist)," macro-objects.")
    else:
        for i in range(len(objectlist)):
            #We don't care for diagonal neighbours when
                checking
            # surrounded objects
              objectlist[i] = objectlist[i][:-1]
    return objectlist, macrolist, objectlistmatrix



def mog(objectlist,inputmatrix,macro_list=[],bk = 0):
    """


    Parameters
    ----------
    objectlist : List
        A list of small objects, generated by checking direct
            adjacency of
        non-background (or same colour) cells. Inherited from
            the calling function,
        usually obj_gen, and passed along to a sub-function.
    inputmatrix : array
        The matrix in which the objects are found.
    macro_list : TYPE, optional
        DESCRIPTION. The default is [].
    bk : TYPE, optional
        DESCRIPTION. The default is 0.

    Returns
    -------
    macro_list : List
        A list of marco-objects in the form of a dictionary,
            containing an assigned ID,
        a list of parts (smaller objects), a sub-dictionary ("
            coords", see directly below)
        listing the shape and position within the matrix, and
             a listing of all cells that
        are part of the object and not background, as well as
            the colours present amonst
        those cells.
        Some of those datapoints will be redundant, that is,
            can be generated from each other.

    """

    #Recursive Neighbour function to gather all adjacent
        objects:
    checked_objects = []
    for i in range(len(objectlist)):
        parts = []

        coords = {
                "top"    :  None, #Highest row that's part of
                    the object
                "bottom" :  None, #Lowest row that's part of
                    the object
                "left"   :  None, #Left-Most column that's
                    part of the object
            "right"   :  None, #Right—Most column that's
                part of the object
            "com"     :  None, #Center of Mass, weighting
                all non—bk cells
            "median"  :  None, #Cell that is closest to the
                 center
            "smsize"  :  None, #Sub—matrix—size: The shape
                () when creating a matrix from the object
            "center"  :  None, #Center of the matrix shape,
                 top left to bottom right
            "top_left": None, #The cell closest to the top
                 left of the matrix
            "dom_col":  None #The dominant colour(s) in
                the object.
            #Colours are still individually listed
            }

    if not (i+1) in checked_objects:
        # ^ This makes sure that objects which are already
             part of a
        # macroobject are not checked again.
        #Logs important information about the shape, size,
             and location of an object

        recne(objectlist, i, parts, coords)
        #recne is a helper function to simply get the four
             sides of an object
        checked_objects.extend(parts)

        temp_cells = list(itch.from_iterable([objectlist[x
            −1][1] for x in parts]))
        # using https://stackoverflow.com/a/953097 as a
            base
        c_col = cellColCheck(temp_cells,inputmatrix)
        #Checks and lists all the colours that are part of
             the object. Usually just 1 or 2

        coords["com"] = (sum([l[0] for l in temp_cells])/
            len(temp_cells),
                         sum([l[1] for l in temp_cells])/
                             len(temp_cells))
        coords["center"] = ((coords["top"]+coords["bottom"
            ])/2,(coords["left"]+coords["right"])/2)
        coords["median"] = (round((coords["top"]+coords["
            bottom"])/2),
                             round((coords["left"]+coords["
                                 right"])/2))
        coords["dom_col"] = [i for i, j in enumerate(c_col
            ) if j == max(c_col)]
        #inspired by https://stackoverflow.com/a/3989032
        coords["top_left"] = min(temp_cells) #getTopLeft(
            temp_cells,coords["top"],coords["left"])
        #Saved as a list because it is possible that
            multiple colours make up the
```

```
# macro object in equal parts
obSize = ((coords["bottom"]-coords["top"]+1),(
    coords["right"]-coords["left"]+1))
#Made a temp object for ease of reusal and
    readability.

coords["smsize"] = obSize

obArray = object_window(inputmatrix,obSize,(coords
    ["top"],coords["left"]))
#Searches for contained objects by recalling the
    function
# on the object just generated.
contains = obj_gen(obArray,[],coords["dom_col"],
    True,[],False)[0]
lines,columns = [],[]
for i in range(coords["smsize"][0]):
    # Lists all rows of the object
    lines.append((obArray[i,:]).tolist())
for i in range(coords["smsize"][1]):
    # Lists all rows of the object
    columns.append((obArray[:,i]).tolist())

traits = { #Lists detected features of the object
    "hollow": contains == [],     #Contains
        encirled cells of non-dominat colour.
    "axis_sym": np.all(np.transpose(obArray) ==
        obArray),
        #Object can be mirrored on axis from top-
            left to bottom-right
    "rot90_sym":  np.all(np.rot90(obArray, 1) ==
        obArray),
        #Is object rotationally symmetric?
    "rot180_sym": np.all(np.rot90(obArray, 2) ==
        obArray),
        #Can Object be turned by 180 degree?
    "vertical_sym":   np.all(np.flip(obArray, 0)
        == obArray),
         #Is object mirrored across a vertical
             axis?
    "horizontal_sym": np.all(np.flip(obArray, 1)
        == obArray),
         #Is object mirrored across a horizontal
             axis?
    }

#Creates a new macro object, and lists which small
     objects it consists of
new_macro_obj = {
#Contains objects:
"Array": obArray, #Array containing just the
    object, for running functions on
"parts": parts, #What minor objects this macro-
    object consists of
```

```
                "dimensions": coords,
                "cells": temp_cells,     #Cells that are part of
                    this object
                "colours": [i for i, j in enumerate(c_col) if j !=
                     0],
                "rows": lines,           #Rows in the object
                "columns": columns,      #Columns of the object
                "contains": contains,    #If the object contains
                    minor objects of different colour
                "traits" : traits        #Dict listing traits of
                    the object

                # Also "minority colour".

                }

                macro_list.append([new_macro_obj])

    for k in range(len(macro_list)):
    #Adds a numeric ID to each macro object.
    #The IDs start at i+100 to differentiate them from the
        small objects.
    #It is assumed that no matrix with more than 100
        individual objects will
    # be evaluated over the course of this thesis.
        macro_list[k].insert(0,k+100)

    return macro_list

def recne(objectlist, i =0, parts = [], coords={}):
    """
    A recursive helper function to get the four sides of a
        macro-object.
    Starts with an object, then re-calls itself for its
        neighbours until
    there are none left.

    Parameters
    ----------
    objectlist : List
        A list of small objects, generated by checking direct
            adjacency of
        non-background (or same colour) cells.
    i : Int, optional
        The position of the object in the object list. The
            default is 0.
    parts : List, optional
        A list of objects already checked. Prevents a circular
            macro-object
        from being infinitely cycled. The default is [].
    coords : Dictionary, optional
        The coordinates dictionary to later be a part of the
            macro object.
        The first four entries (top, bottom, left, right) are
            filled here.
        Usually starts out empty. The default is {}.

    Returns
    -------
    parts : List
        Parts already checked (see above). Used to prevent
            infinite
        recursion in 'recne' and 'mog'
    coords : Dictionary
        On return to mog, this will contain the four object
            borders.

    """

    parts.append(objectlist[i][0])
    if objectlist[i][2] is not []: # if it's empty, the
        function is done
        for j in objectlist[i][2]:
            #If 'j' wasn't checked so far, runs the function
                on that first.
            if not j in parts:
                recne(objectlist, j-1, parts, coords)
                #The '-1' is due to the count starting at 1,
                    but the
                # list starting at 0.

    for c in objectlist[i][1]:
        coords["top"] = getCoord(coords.get("top"),c[0],True)
        coords["bottom"] = getCoord(coords.get("bottom"),c[0],
            False)
        coords["left"] = getCoord(coords.get("left"),c[1],True
            )
        coords["right"] = getCoord(coords.get("right"),c[1],
            False)

    return parts, coords

def getCoord(key, tup, comp = True):
    """
    A simple min/max function called by recne.
    Checks an existing number against a new number from a
        tuple (cell coordinate)
    and keeps the lower/higher value, depending on input.
    Called by a recursive function(recne).

    Parameters
    ----------
    key : Int
        The current highest/lowest value of the coordinate to
            be checked.
    tup : Int
```

```
        One value from a tuple. Whether its the first or
           second value depends on
        what is being checked, and is decided on input by the
           calling function.
    comp : Boolean, optional
        Tells getCoord whether it should check for min or max.
        'True' means minimum, used for "top" and "left" (
           matrices counting from the top left).
        'False' means maximum.
        The default is True.

    Returns
    -------
    key : Int
        Returns the higher/lower of two input values,
           depending on the comp variable.

    """
    if key is None:
        key = tup
    elif comp:
        key = min(key,tup)
    else:
        key = max(key,tup)
    return key


def cellColCheck(cList,inputmatrix):
    """
    A simple look-up, taking a list of coordinates and
       counting the results.

    Parameters
    ----------
    cList : List of tuples
        A list of cell coordinates making up a macro object.
    colourmatrix : np.array
        The original matrix of cells containing numbers.

    Returns
    -------
    found_colours : List
        A list of colours. Each position in the list is a
           colour corresponding to that position.
        The value of that position is how many times that
           colour exists in the object.

    """
    found_colours = [0,0,0,0,0,0,0,0,0,0]
    for i in cList:
        found_colours[inputmatrix[i]] += 1
    return found_colours

def object_window(matrix,oSize,oPos):
    """
    A simple Array Slicing operation, outsourced to its own
        function to
    facilitate a future refactor

    Parameters
    ----------
    matrix : np.array
        The original Input Matrix.
    oSize : Tuple
        Shape of the object in question.
    oPos : Tuple
        Top Left position of the object.

    Returns
    -------
    np.array window
        View of the matrix at the specific position.

    """
    return matrix[oPos[0]:(oPos[0]+oSize[0]),oPos[1]:(oPos[1]+
        oSize[1])]
```

B.2.4 matrix bk check

Has two functions:

- `mat_count` counts all values in a matrix and returns the count for the entire matrix, all rows, all columns, and all border cells (bordering the edge of the matrix) separately.
- bkcheck takes those values and decides which colour in the matrix will be assumed to be the "background"-colour against which objects are differentiated.

```
import numpy as np
from collections import Counter

def mat_count(matrix,do_tot=True,do_rc=True,do_bd=True): #
    None or True
    """
    Counts values in each row, column, and matrix total,
        followed by border cell counts if desired.
    Some of those steps are later repeated in other methods,
        but flexibility and
    readability are valued higher than performance in this
        case

    Parameters
    ----------
    matrix : np.array
        Input Matrix as an array
    do_tot : BOOL, optional
    Return Totals count of all values or not? The default
        is True.
do_rc : BOOL, optional
    Return aggregated count for each row and column? The
        default is True.
do_bd : BOOL, optional
    Return count of all border values(nodes at the edge of
         the matrix)?
    This is mostly of theoretical value, in case a future
        relational interpretation
    of data can't be ruled out.
    The default is True.

Returns
-------
totals: dict
    A sorted dictionary listing how often each value
        exists in a matrix.
rows:
    Like totals, but lists the values rows in a matrix
cols:
    Like totals Lists the individual columns in a matrix
bd:
    Lists the values of all border cells, that is, cells
        at the edge of the matrix,
    like (0,0).
np.size(matrix):
    A numpy function returning the size of the array as a
        single Integer.

"""
rows,cols = np.array([]),np.array([])
# counts values in the entire matrix
totals = {}
if do_tot:
    totals =  {0 : np.count_nonzero(matrix == 0),
              1 : np.count_nonzero(matrix == 1),
              2 : np.count_nonzero(matrix == 2),
```

The same count will be done for every other possible value.

```
bd = [{}] # initialize border count (border and 1 step
    away from the border)
for i in range(len(matrix)):  # counts values per row #
    range(max(matrix.shape[:])
    if do_rc:
        row = dict()
        np.append(row,{})
        row = {**({0: np.count_nonzero(matrix[i] == 0)} if
                  np.count_nonzero(matrix[i] == 0) else
                      {}),
```

Again, the same here, and then the same transposed.

```
if do_bd:
        border = dict(Counter(list(matrix[0, :-1]) + list(
            matrix[:-1, -1])
                + list(matrix[-1, :0:-1]) + list(matrix
                    [::-1, 0])))
        bd[0] = border
    else: bd[0] = {}

    return totals, rows, cols, bd, np.size(matrix)

def bkcheck(size,totals,bd):
    """
    Decides which colour is seen as "background".
    The values are admittedly a bit arbitrary, but deciding
        which colour
    is the background in a flat image without context is a
        matter of human opinion.

    Parameters
    ----------
    size : Int
        The total amount of cells in a matrix, generated from
            np.size.
    totals : Dict
        Count of all values in a matrix.
        Example: [1,2,3,3,5] would have {3:2, 1:1, 2:1, 5:1}.
    bd : dict
        A dictionary of the border cells. It is assumed that a
             value occuring
        more often at the border of a matrix is more likely to
             be the "background",
        assuming an otherwise equal count of two values.

    Returns
    -------
    Tuple(Bool, # * Tuple(Value, Count)).
           Returns a Tuple of a Boolean and one or multiple
               Tuples.
           If True, the function is sure of its choice of
               background,
           and returns a single tuple of the chosen value and
               how often it occurs.
           If False, there is no clear background, and it will
               return two
           or three tuples of possible values and how often
               they occur,
           in the case of the same count the values are ordered
                small to big.
           The last output is a failsave and just returns False
                with one tuple.

    """

    bk_val = [sorted(totals.items(), key=lambda item: item[1],
        reverse = True)[0]]
    if not    sorted(totals.items(), key=lambda item: item[1],
        reverse = True)[0][1] >= size *3/4:
        bk_val.append(  sorted(totals.items(), key=lambda item
            : item[1],reverse = True)[1]   )
        #If the most numerous value is not at least 75% of the
            matrix,
        # add the second most numerous one to the list
    if not (sorted(totals.items(), key=lambda item: item[1],
        reverse = True)[0][1] +
            sorted(totals.items(), key=lambda item: item[1],
                reverse = True)[1][1]) >= size *3/4:
        bk_val.append(  sorted(totals.items(), key=lambda item
            : item[1],reverse = True)[2]   )
        #If that's still not enough, add a third. This is as
            far as it goes.
    if (bk_val[0][1] >= 0.67 * size or
        (bk_val[0][1] >= 0.4 * size and bk_val[0][1] >= (
         sorted(bd.items(), key=lambda item: item[1],reverse =
             True)[0][1])/2)):
        #One colour makes up at least 67% of the matrix,
        # or >40% and the majority of the outer edges of the
            matrix
        return (True, bk_val[0])
    elif ((bk_val[1][1] >= 0.4 * size and bk_val[1][1] >= (
         sorted(bd.items(), key=lambda item: item[1],reverse =
             True)[0][1])/2)):
        return (True, bk_val[1])
        #It may pick the second most colour if that has the
            border majority.
    elif len(bk_val) > 1 and bk_val[0][1] >= 2* bk_val[1][1]:
        return (False, bk_val[0], bk_val[1])
    elif len(bk_val) > 2:
        return (False, bk_val[0],bk_val[1],bk_val[2])
    else:
        return(False, bk_val[0])
```

B.2.5 example splitter

Splits files into input/output, and individual examples

```
from __future__ import print_function
import numpy as np
import json

#------------------------------------------------------------

def read_file(fl, fpath="training/", cap = 3):
    """
    Reads a file and splits it into train data and test data,
        then further into input- and output-matrices.

    Parameters
    ----------
    fl : json file
```

```
        Contains sets of matrices.
    fpath : String
        The folder where the json files are to be found.
    cap : Int
        How many examples are to be read from each file.
            Default is 3.

    Returns
    -------
    List
        A list of two sets, one with the training examples,
            one with the test example.

    """
    flr = fl.read()
    train_data = np.array(json.loads(flr)['train'])
    test_data = np.array(json.loads(flr)['test'])
    ## Generates Empty Lists to be filled with individual
        examples:
    inputs,outputs = [],[]
    trainr, testr = [inputs,outputs],[]

    for j in range(len(train_data)):
            if j <cap:
                inputs.append(np.array(train_data[j]["input"])
                    )
                outputs.append(np.array(train_data[j]["output"
                    ]))
    for i in range(len(test_data)):
        #The test data usually consists of only one pair of
            matrices, so it does
        # not need to be split into input and output like the
            training data
        testr.append(np.array(test_data[i]["input"]))
        testr.append(np.array(test_data[i]["output"]))
    return [trainr,testr]
```

B.2.6 matrix comparator

Checks two arrays for similarities between containing objects. As this is meant for testing with a console output, it contains a lot of print-outs. Relatively rudimentary as the pass-over to Metacat is done manually anyways.

```
import numpy as np

def matrix_dif(inp,outp):
    """
    Tries to find the difference between input and output, IF
        the same size.
    Will check differences in monochrome and stratified
        colours if necessary

    Parameters
    ----------
    inp : Array
        The first matrix.
    outp : Array
        The second matrix.

    Returns
    -------
    Tuple
        A Tuple of a Boolean and an Array.

    """
    if np.size(inp) == np.size(outp):
        return (True,(outp - inp))
    else:
        return (False,())

#Enter two analyzed matrices here
def matrix_comp(mat1, mat2, m1ob, m2ob, m1c = [], m2c = [],
    pout = False):
    #mXc contain: [0: totals,1: rows,2: cols,3: bd,4: size]

    diff_map = matrix_dif(mat1, mat2)
    diff_col = [[],[]]
    diff_row = [[],[]]
    forswitch = True
    #Rows and Columns of the entire matrices:
    for j in range(len(m1c[1])): # rows
        for key in m1c[1][j] and m2c[1][j]:
            if key in m1c[1][j] and m2c[1][j] and (
                    m1c[1][j][key] == m2c[1][j][key]):
                continue
            elif forswitch:
                diff_row[0].append((j,mat1[j]))
                diff_row[1].append((j,mat2[j]))
                forswitch = False
                continue
            else:
                forswitch = True

    forswitch = True
    for j in range(len(m1c[2])): # columns
        # print(m1c[2][j])
        # print(m2c[2][j])
        for key in m1c[2][j] and m2c[2][j]:
            if key in m1c[2][j] and m2c[2][j] and (
                    m1c[2][j][key] == m2c[2][j][key]):
                continue
            elif forswitch:
                diff_col[0].append((j,mat1[:][j]))
                diff_col[1].append((j,mat2[:][j]))
                forswitch = False
                continue
            else:
                forswitch = True
```

```
#Macro Object Comparison
macrotraits = [[],[]]
object_relation = []
for i in range(len(m1ob[1])): #Matrix 1
    mObTraits = [
        m1ob[1][i][1]["dimensions"]["center"],
        m1ob[1][i][1]["dimensions"]["smsize"],
        m1ob[1][i][1]["dimensions"]["median"],
        (m1ob[1][i][1]["cells"] +
        m1ob[1][i][1]["contains"])#.
        #sort(key=lambda tup: tup[0]),
        ]
    macrotraits[0].append(mObTraits)
for i in range(len(m2ob[1])): #Matrix 2
    nbcell = [k[1] for k in m2ob[1][i][1]["contains"]]
    mObTraits = [
        m2ob[1][i][1]["dimensions"]["center"],
        m2ob[1][i][1]["dimensions"]["smsize"],
        m2ob[1][i][1]["dimensions"]["median"],
        (m2ob[1][i][1]["cells"] +
        nbcell)#.
        #sort(key=lambda tup: tup[0]),
        ]
    macrotraits[1].append(mObTraits)

#Check center, size, median, cells. 2+ equal --> related
for icount, ival in enumerate(macrotraits[0]):
    for jcount, jval in enumerate(macrotraits[1]):
        if majority(trait_compare(ival,jval),2):
            print(m1ob[1][icount][0])
            object_relation.append(
                [
                (m1ob[1][icount][0],m1ob[1][icount][1]["
                    Array"]),
                (m2ob[1][jcount][0],m2ob[1][jcount][1]["
                    Array"])
                ]
                    )


#Console use printout:
if pout:

    if diff_map[0] == True:
        print("Difference map of the two matrices: ")
        print(diff_map)
    print("Different rows: ", diff_row)
    #As no new matrix will be shaped off them, there is
        currently no
    # need to enumerate them.
    print("different columns: ", diff_col)
    #print("Object Relations: ",object_relation)

    for i in object_relation[0][0][1]:
            print(letter_convert(i))
            # Using only the first object pair in the list,
            # As this is deemed sufficient for a first
                evaluation
        for i in object_relation[0][1][1]:
            print(letter_convert(i))


    return object_relation

# taken from https://stackoverflow.com/a/42514511
def majority(itvar, num):
    """
    Checks if a certain amount of values are True.
    Parameters
    ----------
    itvar : Iterator
        The values to be iterated through.
    num : Int
        DESCRIPTION.

    Returns
    -------
    Bool
        True or False.

    """
    print("checking majority")
    itvar = iter(itvar)
    return all(any(itvar) for _ in range(num))

def trait_compare(tr1,tr2):
    """
    Checks if values from two lists are equal, and returns a
        list of booleans

    Parameters
    ----------
    tr1 : List
        A list of hopefully comparable values.
    tr2 : List
        A list of hopefully comparable values.

    Returns
    -------
    tlist : List
        A list of Booleans.

    """
    tlist = []
    for i in range(len(tr1)):
        print(tr1[i])
        tlist.append(tr1[i]==tr2[i])
    return tlist
```

```
def letter_convert(Li):
    """
    Takes a List of Numbers, and returns a List of Letters

    Parameters
    ----------
    Li : List
        A List of Numbers.

    Returns
    -------
    rLi : List
        A List of Letters.

    """
    rLi = [chr(ord('`')+(x+1)) for x in Li]
    return rLi
```

B.2.7 file plotter

Creates a visual output, assuming an environment that supports it.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as mcm

NUM_COLORS = 6
# from matplotlib.colors import ListedColormap,
    LinearSegmentedColormap
pk = mcm.get_cmap('inferno', 10)

#cm1 = pylab.get_cmap('gist_rainbow')
for i in range(NUM_COLORS):
    color = pk(1.*i/NUM_COLORS)  # color will now be an RGBA
        tuple

#Creates a visual representation of matrices with matplotlib.
def plot_ex(train_in, train_out = [], is_test = False, cap = 5,
     detail = 3):
    """
    Here it plots the matrices to allow visualization for a
        human tester/supervisor.

    Parameters
    ----------
    train_in : List of arrays
        The Input matrices of the training data.
    train_out : List of arrays, optional
        The Output matrices of the training data. If no output
            is given, it just plots the input.
    is_test: Bool
        If this is a testing matrix, the matrices are one
            layer higher in the list cascade.
            The default is False. If set to true, the following
                input variables will be treated
            as if set to 1.
    cap: Int
          Shows how many rows should be plotted at max. The
              default is 5, which
          corresponds to the default of five input-output pairs
              being pulled from
          an example json file in Folder_walk.py.
    detail : Int, optional
            If the difference (output - input) should also be
                plotted.
            3: Yes, input, output, difference
            2: No, just Input and output.
            1: No, only plots a single matix.
            The default is 3. Values over 3 make no sense.

     Returns
     -------
     None. As the output is shown in the IDE's plot tab, there
         is no internal output.

     """
     fig = plt.figure()
     ex_count = min(len(train_in),cap)
     if is_test or train_out == []:
         #Test matrices will always be plotted alone, as if
             detail was == 1
         fig.add_subplot(1,1,1).imshow(train_in[0],
             interpolation = 'nearest',
                                         cmap=pk, vmin=-1.0, vmax
                                             =8.0)
         if is_test and not train_out == []:
             plt.show()
             fig = plt.figure()  #Special case: Clearing
                 variable for a double output
             fig.add_subplot(1,1,1).imshow(train_in[1],
                 interpolation = 'nearest',
                                             cmap=pk, vmin=-1.0,
                                                 vmax=8.0)

     elif detail >=3:
         for j in range(ex_count):
             fig.add_subplot(ex_count,3,(1+j)*3-2).imshow(
                 train_in[j], interpolation = 'nearest', cmap=
                     pk, vmin=-1.0, vmax=8.0)
             fig.add_subplot(ex_count,3,(1+j)*3-1).imshow(
                 np.array(train_out[j])-np.array(train_in[j]),
                     interpolation = 'nearest',
                 cmap=pk, vmin=-1.0, vmax=8.0)
             fig.add_subplot(ex_count,3,(1+j)*3).imshow(
                 train_out[j], interpolation = 'nearest', cmap=
                     pk, vmin=-1.0, vmax=8.0)
     elif detail == 2:
```

```
    for j in range(ex_count):
        fig.add_subplot(len(train_in),2,(1+j)*2-1).imshow(
            train_in[j], interpolation = 'nearest', cmap=
                pk, vmin=-1.0, vmax=8.0)
        fig.add_subplot(len(train_out),2,(1+j)*2).imshow(
            train_out[j], interpolation = 'nearest', cmap=
                pk, vmin=-1.0, vmax=8.0)
        fig.add_subplot()
else:
    for j in range(ex_count):
        plt.figure().add_subplot(len(train_in),1,(1+j)
            *2-1).imshow(
            train_in[j], interpolation = 'nearest', cmap=
                pk, vmin=-1.0, vmax=8.0)

plt.show()
```

Appendix C

Checked Example Files

In this Appendix, plots of a subset of the used ARC examples are listed with file name and a short commentary as to metacat's viability for solving them and why they were or were not used.

File 4. C.1 is a task that could potentially be solved by a future development of a Copycat-inspired architecture, but in testing had to fail without even trying, as the made implementation had no provision for diagonals.

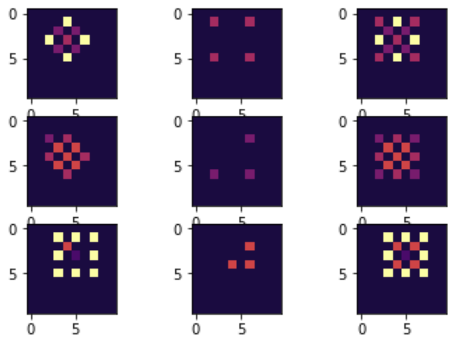


Figure C.1: **File 4**, '11852cab.json', not possible with the existing implementation.

File 8. C.2 showcases in the same example some strength and weakness of Metacat in this application. The 'fill space between' task is doable by metacat after just one training input, independent of the shape of the object, but the different length of individual lines makes it struggle even within the same object of the same example. The attempt was abandoned after two inputs due to low success expectancy and excessive calculation time.

File 14. (C.3) A symmetry task very similar those in File 7 — from a human viewpoint, anyways. It is potentially simpler, as instead of continuing a structure towards a direction it just mirrors a structure present on the input, and it does not contain any diagonals, though it does add an extra feature in the form of recolouring the background.

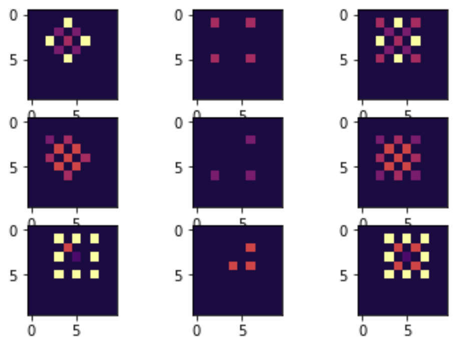


Figure C.2: **File 8**, '22168020.json', Almost possible.

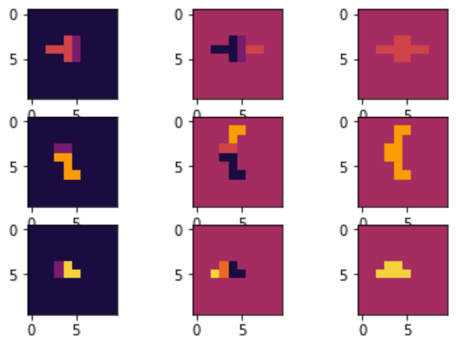


Figure C.3: **File 14**, '2bcee788.json', Another Symmetry Task.

Declaration of Authorship

Ich erkläre hiermit gemäß § 9 Abs. 12 APO, dass ich die vorstehende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Des Weiteren erkläre ich, dass die digitale Fassung der gedruckten Ausfertigung der Masterarbeit ausnahmslos in Inhalt und Wortlaut entspricht und zur Kenntnis genommen wurde, dass diese digitale Fassung einer durch Software unterstützten, anonymisierten Prüfung auf Plagiate unterzogen werden kann.

Bamberg, den ______________ ______________________________

Jan Martin