CHALMERS

An Anthropomorphic Solver for Raven's Progressive Matrices

Simone CIRILLO
Victor STRÖM

Supervisor: Claes Strannegård
Examiner: Claes Strannegård

Department of Applied Information Technology
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2010

Report No. 2010:096
ISSN: 1651-4769

An Anthropomorphic Solver for Raven's Progressive Matrices
Simone CIRILLO, Victor STRÖM

Report No. 2010:096
ISSN: 1651-4769
Department of Applied Information Technology
Chalmers University of Technology
SE-41296 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Göteborg, Sweden
June 2010

Exemplaria in oculo spectatoris sunt

An Anthropomorphic Solver for Raven's Progressive Matrices

Simone CIRILLO, Victor STRÖM
Department of Applied Information Technology
Chalmers University of Technology

Abstract

This report describes a computer program for solving Raven's Progressive Matrices (RPM), a multiple choice test of abstract reasoning introduced by Dr. John C. Raven in 1936. Each RPM problem consists of a grid (or matrix) of 2x2 or 3x3 cells with graphical content, where the cell content in the bottom right corner is omitted; the solver's task is to pick the missing content from a set of eight solution candidates.

We argue these problems are not only mathematical, but also psychological in nature. Due to this and other considerations such as algorithmic transparency, the program makes use of a simple cognitive model.

The program solves RPM problems in a fully automatic fashion, without taking the solution candidates into account. The input is an RPM problem represented as a vector graphics file; the output is a complete or partial solution for the missing entry, represented in the same format. Internally we use multi-layered structures which enable the perception of the problems' different organizational levels.

The program was tested on sections C, D and E of the Standard Progressive Matrices (SPM) and produced correct solutions for 28 of the 36 considered problems.

Keywords: Antropomorphic Artificial Intelligence, Cognitive Model, General Artificial Intelligence, Intelligence Tests, Raven's Progressive Matrices

Acknowledgements

We would like to thank all those guiding, helping and supporting us during this project and the writing this report, only a proportion of which we have space to acknowledge here.

First and foremost special thanks to our supervisor and examiner Claes Strannegård for his dedication to the project and active contribution. Without your continued support this work would not be what it is today.

We would also like to thank Carl-Martin Allwood for interesting discussions during the course of the project and Ingvar Lind for sharing with us the thoughts behind constructing IQ tests.

Simone: I would like to thank all the people who enabled me to take part in this project and who supported and endured me throughout its course: my parents, D.ssa Barbara Loddo, my friends, The Pentacle, and the Synth scene.

Victor: A special thanks goes to my friends and family that have supported me during this project, especially my fiancée Kajsa for inspiration and encouragement along the way.

Simone Cirillo and Victor Ström
Göteborg, June 2010

Contents

Chapter 1

Introduction

In this Master's thesis project report we will describe the creation of a computer program aimed at solving as many problems as possible from a well-known set of Intelligent Quotient (IQ) tests, Raven's Progressive Matrices (RPM).

1.1 Motivation

Presume a program achieving perfect scores on an established and standardized IQ test was created; would that program truly be intelligent?

1.2 Background

Giving an answer to the provocative statement above is not trivial: human intelligence and artificial intelligence (AI) have both been given several definitions. Without specifying what is intended by intelligence and whether that is actually measured by IQ tests, it is impossible to answer the question in a proper fashion.

Although this thesis is rooted in computer science and will not discuss the intricacies of cognitive psychology and the human brain, we will start by providing a quick overview of what intelligence and artificial intelligence are and by introducing Raven's Progressive Matrices.

We will examine related previous work, describe the similarities and differences of our approach and discuss some key concepts about the task of solving RPM before going into the details of our implementation.

1.2.1 Intelligence and Artificial Intelligence

As mentioned above, many definitions of intelligence exist. Bringsjord & Schimanski (2003) propose the following, strictly operational one, which brings together human and artificial intelligence: *"Some agent is intelligent if and only if it excels at all established, validated tests of intelligence"*, later amended to include *"tests of artistic and literary creativity, mechanical ability, and so on"*.

According to Legg & Hutter (2007), being able to solve such tests successfully is a necessary, but not sufficient, condition for intelligence. They argue that a collection of algorithms specifically constructed for solving the tests with no real usefulness or application in any other domain does not qualify as an intelligent agent.

However, looking at the discipline of Artificial Intelligence, a standard textbook in the field from Russell & Norvig (2003) discusses four alternative definitions: *"Systems that think like humans"*, *"Systems that act like humans"*, *"Systems that think rationally"* and *"Systems that act rationally"*. Elaborating on the last definition, the one preferred in the book, a system is considered to be rational if it always chooses to perform the actions that maximize the (expected value of) its performance measure.

If we wish to use the same performance measure to evaluate both human and artificial intelligences, nothing prevents us from using the scores obtained on various standardized tests. However, claiming the test-solving algorithms are intelligent is wrong, since that capability is a necessary condition, but not a sufficient one.

McCarthy (2007) defined AI as: *"the science and engineering of making intelligent machines"* . Given that humans are intelligent machines, and that humans can solve intelligence tests, it is still relevant for the discipline to investigate such tests to understand the competences proper artificial intelligent machines should have.

With respect to the Raven's Progressive Matrices, the investigation is still not complete. In this report we describe a methodology and a program for solving RPM problems in a fully automatic fashion.

1.2.2 Raven's Progressive Matrices

An issue with some standardized IQ tests is that they are verbally based, requiring considerable language proficiency as a prerequisite. Raven's Progressive Matrices (RPM), however, are pictorial tests of abstract reasoning where the language factor is entirely removed.

Extensive analysis and studies on these tests have been performed in the past sixty years, evidencing high levels of correlation with other multi-domain intelligence tests and measures of achievement (Raven & Court, 2003) (Snow et al., 1984). As a result RPM are often used as a test of so-called general intelligence (Snow et al., 1984), which was also one of Raven's (1936) goals when first constructing the test.

Each RPM problem is presented as a 2x2 or 3x3 matrix of pictures following a pattern. The bottom right position in each matrix is left blank and the solver's task is to choose the missing picture for that cell from a list of eight provided solution alternatives. Figure 1.1 shows an example of such type of problems; for copyright reasons all the illustrations in this report will not feature actual RPM problems, but constructed equivalents. The first and most common set of RPM are the Standard Progressive Matrices (SPM), developed in 1936 (Raven, 1936) and published in 1938. This set contains five sequential sections: A-E, each more difficult than the previous one. Each set contains 12 matrices to solve, also ordered by increasing difficulty. The two easiest sections, A and B, consist of 2x2 matrices while C, D and E feature 3x3 matrices. All SPM problems are represented in black ink over a white background. Later on several other RPM sets have been created in order to better assess different population segments and to increase the discriminative

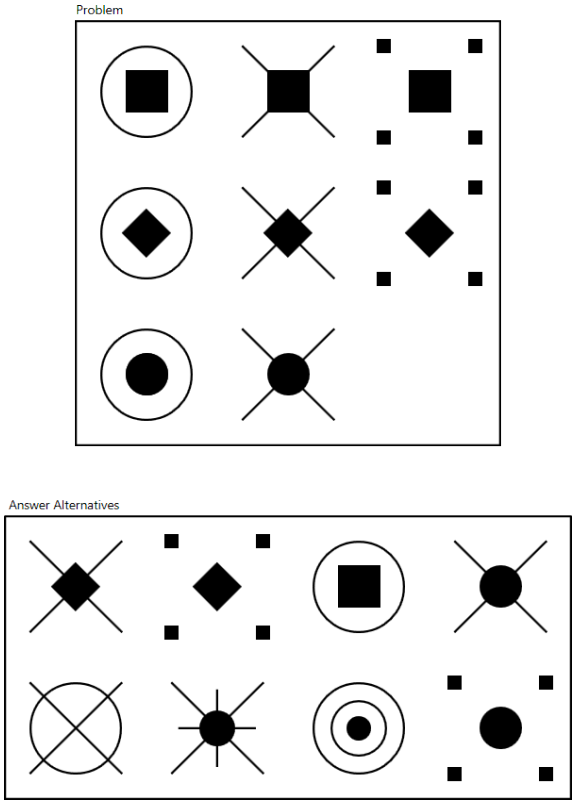

Figure 1.1: Example of a 3x3 RPM-like problem

power of the SPM: the Advanced Progressive Matrices (APM) for adults and adolescents with above average intelligence and the Colored Progressive Matrices (CPM) for children, elderly and people with learning disabilities.

The work presented here focuses on the three hardest sections (C, D and E) of the SPM, all 3x3 matrix problems. The 3 sections consist of 12 items each, for a total of 36 problems to solve (Raven, 1990*c*).

1.2.3 Automatically Solving Raven's Progressive Matrices

As of today the cognitive and computational characteristics of RPM aren't yet well understood (Kunda et al., 2009) and no general algorithm for solving them in their entirety has been developed. However several attempts to build an automated solver have been made throughout the years.

1.3 Previous Work

There have been four documented attempts at solving the RPM with computer programs since they were originally constructed: Carpenter et al. (1990), Bringsjord & Schimanski (2003), Lovett et al. (2007), and Kunda et al. (2009).

Carpenter et al. (1990) conducted a seminal study along with implementations of an automated solver in 1990. They investigated human RPM solution strategies by means of experiments involving think-aloud protocols and tracking gazing patterns and successively implemented two distinct solvers based on production systems and cognitive models.

The two solvers were aimed at mimicking the RPM performance of their average and high scoring subject, respectively. Both programs achieved results that correlated well with the experimental data.

The input to these solvers consisted of manmade individually tailored textual descriptions of the RPM problems. However, as pointed out by Meo et al. (2007), this approach relies heavily on human pre-processing on the levels of perception, encoding and information filtering. Therefore, the two solvers do not solve RPM problems strictly speaking, but substantially simplified non-pictorial versions of them.

In 2003 Bringsjord & Schimanski reported the implementation of theorem prover-based agents *"able to infallibly crack not only geometric analogies, but RPM items they have never seen before"*. Any other information regarding the implementation or the obtained results is however missing.

A more recent attempt was made Lovett et al. (2007), using the sKEA/CogSketch sketch understanding architecture (Forbus et al., 2004, 2008). They managed to get very good results on SPM sections B and C using an analogical reasoning strategy, but did not publish any attempts on any other sections.

In 2009 Kunda et al. proposed fractal coding and affine transformation based approaches. No results were mentioned in their preliminary study and at the time of writing this report they are

in the process of evaluating them.

Summarizing, several different approaches have been proposed and tried to automatically solve RPM, but the challenge of finding a solution methodology that handles the entire set of problems is still open.

1.4 Goals

The goals of this Master's thesis project are:

- Solving as many RPM problems as possible from the three hardest sections of SPM: C, D, and E
- The definition of a knowledge representation structure, a rudimentary cognitive model and a solution process enabling the automated generation, rather than the choosing, of RPM solutions.

1.5 Disposition of This Report

- In Chapter 2, Problem Analysis, several problems tangent to the task of solving Raven Progressive Matrices will be discussed and a general overview of our methodology will be provided.
- Knowledge Representation, Chapter 3, will introduce and explain the format of the inputs and the internal problem representation structures used by the program.
- Chapter 4, Computational Model, describes the process used to compute the RPM solutions.
- In Chapter 5, Implementation, we detail the software application developed to implement the previously discussed algorithms and strategies.
- Chapter 6, Results, presents the obtained results and some immediate considerations.
- In Discussion & Conclusions , Chapter 7, we will articulate several interesting points emerging from the results, delineate strenghts and weaknesses of our implementation, and propose directions of possible future development.

Chapter 2

Problem Analysis

After examining the layout of RPM, the ideas behind them and relevant previous work, we now proceed to analyze the problems themselves in more depth.

2.1 Background

To design a way to solve the problems we drew inspiration from human solving strategies. Since RPM are fully pictorial tests, we looked into psychological and computational theories of the organization of visual stimuli. Secondly, we examined the notion of simplicity as that concept is often brought up as the general criterion to be used when solving intelligence tests.

2.1.1 Visual Perceptual Organization

Gestalt psychologists such as Koffka, Wertheimer and Köhler in the 1930s and 40s sought to define the principles guiding the perception of raw stimuli into organized wholes. In particular they focused on visual perception and tried to explain the perception of groups of objects and parts of objects. Their findings are coded in the *Gestalt Law of Prägnanz*, stating the following criteria: closure, similarity, proximity, symmetry, continuity and common fate (Wertheimer, 1939).

Structural Information Theory (SIT), initiated in the 1960s by Leeuwenberg (1968, 1969, 1971), is argued the best (Palmer, 1999) framework derived from Gestalt concepts because it overcomes the entirely qualitative aspect of the original Gestalt laws by providing a mathematical formalism for generating perceptual interpretations. SIT considers the simplicity princple the criterion for interpreting stimuli: the preferred interpretation of a stimulus is the one yielding the simplest code. Simplest codes imply a hierarchical stimulus organizations in terms of wholes and parts (van der Helm, 2007).

2.1.2 RPM Solution

To solve RPM problems, the test candidates are instructed to choose the solution alternative that completes the pattern from a eight listed options. Completing the pattern traditionally means that the global information content of the problem grid is minimized. In terms of Kolmogorov complexity this corresponds to finding the generating function of minimum length, the shortest description.

2.1.3 Simplicity

Consider the following question about number progressions: Which are the successive three numbers in the sequence:

$$1, 1, 2, 3, 5$$

People trained in mathematics would immediately see that the progression corresponds to the Fibonacci series: $F_0 = 0,\ F_=1,\ F_n = F_{n-1} + F_{n-2}$. Hence for them the full sequence will be:

$$1, 1, 2, 3, 5, 8, 13, 21$$

Another possible approach would be performing a polynomial expansion, which with the fourth degree polynomial $P(n) = 0.0833n^4 - n^3 + 4.4167n^2 - 7.5n + 5$ would yield a very good fit for the given values. The resulting answer would then be:

$$1, 1, 2, 3, 5, 11, 26, 57$$

So, which of the two answers is the correct one?
If the question above was an item within an intelligence test, the answer sheet would almost certainly list the first one. Fibonacci and similar "notable" series are a common feature in such tests.

However, the second option is based on a very straightforward and mechanical approach and, given it fits the given series, it cannot be said to be wrong.

Why is there a bias towards the Fibonacci function then?

Many would confidently affirm that it is the simplest one generating the progression. The notion of simplicity, however, has no mathematical meaning unless it is properly defined. Technically, simplicity corresponds to a drop in Kolmogorov complexity but to apply this definition it is first necessary to specify the description language, to instantiate the machine.

Simplicity theory, a cognitive theory seeking to explain the attractiveness of situations or events to human minds (Chater & Vitányi, 2003), avoids the problem of the a priori incomputability of Kolmogorov complexity by positing that when dealing with cognitive systems complexity is to be defined as relative to the specific observer. The shortest, and thus simpler, description is the shortest one among the ones currently available to the observer and it heavily depends on the cognitive model used (Dessalles, 2010).

In the case of number progressions, the cognitive model is the set of patterns (functions, operations, etc) known to the individual problem solver and which can be computed given the limited

available cognitive resources as well as the heuristics used to guide the search, for example a precedence relation between the functions. Regarding RPM the situation appears to be completely analogous. The problems are similar in the sense that the RPM can be viewed as glyph progressions on a bidimensional grid.

Computer programs designed to solve such problems in human-like fashion need to feature similar cognitive models in order to be able to produce solutions of maximum simplicity within their own scope.

2.2 Our Approach

Our objective is to construct a program that performs as well as possible on RPM problems. Although our approach superficially coincides with that Carpenter et al. (1990), amongst other things the goals differ; Carpenter et al. had the aim to produce cognitive models modeling human solving strategies, we try to extract only the positive parts of the human problem solving and not fail on problems where humans tend to fail - all in all our goal is to contribute to AI rather than cognitive psychology.

Despite that we, when constructing the program, took inspiration from the fact that our target problems are solvable by humans. If our program could imitate their strategies closely enough, then it should be able to solve the problems as well.

In agreement with the complexity definition offered by Simplicity Theory, our program features a very rudimentary cognitive model. It consists of several patterns and a precedence relation between them; the patterns were isolated through introspection of our own solving strategies. Our cognitive model does not include any limitation in its resources (such as working memory), but does subscribe to the approach of anthropomorphic artificial intelligence (Strannegård, 2007), where the idea is to use cognitive models as frameworks for solving AI problems.

Following Lovett et al. (2007), the inputs to our program are files containing vector graphics representations of the RPM problems. Given that such files contain the same information as the original problems, this choice of format enables us to investigate not only the pattern-matching, but also the perceptual strategies to some extent. In terms of level of pre-organization of the input information, vector graphics representations lie between the two extremes of raster images (bitmaps) and textual, propositional encodings.

A characteristic feature of our program is its internal knowledge representation. While not directly implementing their strategies, we have drawn inspiration from Gestalt and SIT principles. From our input, we automatically construct hierarchical structures to capture different levels of possible perceptual organization for the problem, enabling our pattern-matching to work on different levels.

Another important thing to note is that the program does not choose the answer from the provided alternatives but rather computes the solution entirely on its own, something made possible by relying on a cognitive model. Therefore we deal with a task much harder than conventional RPM solving, with the bonus of being able to detect inconsistencies and other issues with the problems' specifications that might not be obvious when choosing from a list of possible answers.

Chapter 3

Knowledge Representation

As we have introduced earlier, we want to be able to process the RPM problems using different levels of abstraction. From the input format we therefore need to perceive and represent the problem in a flexible multi-leveled fashion.

3.1 Input Format

Vector graphic representations of Standard Progressive Matrices sections C-D-E were created; each file encodes one problem and its eight candidate solutions. Specifically the chosen file format is XAML (Extensible Application Markup Language): a user interface markup language also capable of general purpose serialization.

3.2 Overview of the Perception Stage

The conceptual model we use abstracts RPM according to different organizational levels in order to facilitate detection of the underlying logical patterns. The four main levels the problems can be perceived at are:

- **Attributes**
  Isolated properties of the graphical elements or groups: height, width, color, etc...
- **Elements**
  Single graphical elements, such as a filled black circle with a specific size
- **Groups**
  Sets of distinct elements behaving as a single one, such as two lines making up an X or a + figure.
- **Cells**
  Entire cells of the RPM matrix.

The first stage of our software is a procedure that, starting from an input file, produces a problem representation in terms of this conceptual model.

3.3 Representation Structure

The conceptual model introduced above is implemented using a layered graph structure built up of interconnected nodes. Each layer corresponds to an organizational level of the input problem and each node corresponds to a feature on that level. Nodes are allowed to be connected only if they belong to different layers.

The main reason for having such a complex structure is that, since our aim is to both compute and visualize the solutions, we want to be able to solve the problem using the appropriate abstraction level while still retaining every piece of information needed to go back to a graphical representation.

In the complete structure eight node layers are present and from top to bottom they represent:

1. **Attribute Nodes**
   The relevant attributes directly extracted from the vector graphics objects (shape, line thickness, etc) or simple geometrical properties computed from them (bounding box width, center position, etc).

2. **Attribute Value Nodes**
   Values of the extracted attributes (type=rectangle, line thickness=2, etc)

3. **Element Model Nodes**
   Sets of *Attribute Value* representing maximum common denominators with respect to the *Element Instances*

4. **Element Instance Nodes**
   Position-less single visual elements. Are made up of a combination of links to *Attribute Value* nodes and *Element Models*.

5. **Relative Position Nodes**
   Positions *Element Instances* relative to others within a group

6. **Group Nodes**
   Groups *Relative Position* nodes to make groups of *Element Instance* nodes comparable with single *Element Instance* nodes.

7. **Absolute Positions**
   Positions *Element Instance* nodes or *Group* nodes inside the *Cell* nodes.

8. **Cell Nodes**
   Represents the actual cells of the RPM problem. Contains a number of *Cell Objects*, which are links to the *Absolute Position* nodes contained in the cell.

Figure 3.1 shows a progressive matrix and the representation structure the program extrapolates from it; for visual clarity reasons, however, the diagram is simplified: it does not depict all the considered attributes and the *Position* attribute is presented as a single, qualitative one instead of two quantitative coordinates. The *CellObject* nodes of the diagram correspond to *Absolute Position* nodes of the actual structure; *CellObject* nodes are also used in the graphical visualization of

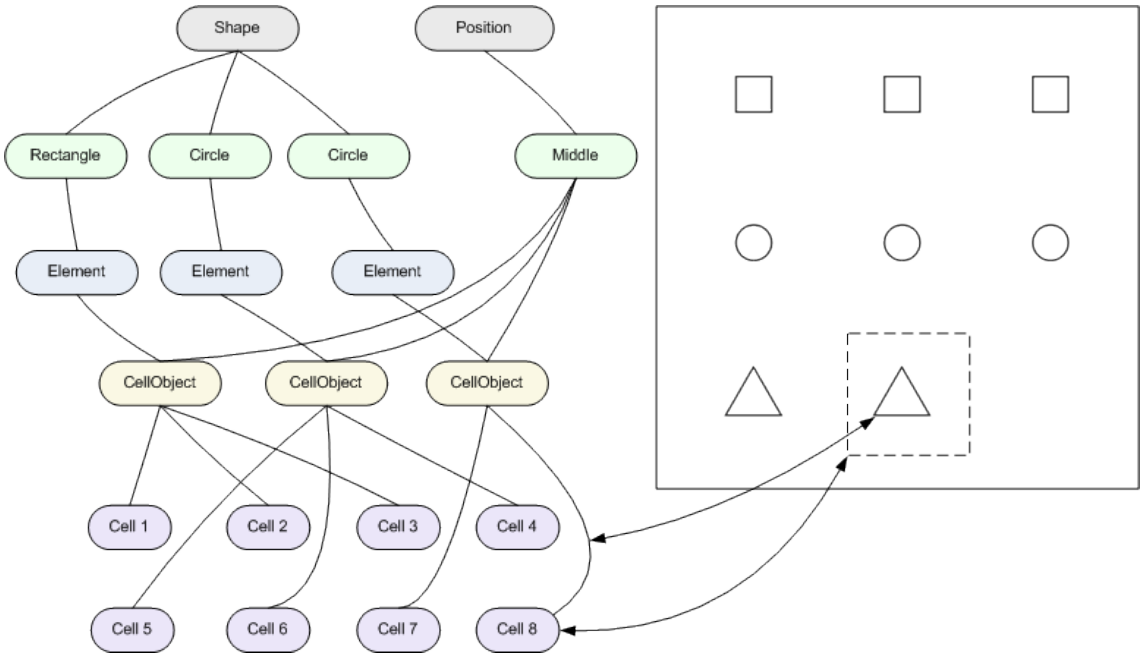

Figure 3.1: Representation structure for a simple progressive matrix

the structure featured in the program, as will be explained in Section 5.3.

In Figure 3.2 instead we have the graph representation for a single cell of a more elaborate progressive matrix. Here a *Group* is present: the four squares at the corners; note how all of them are represented by the same *Element* node and the different positions are modeled at a successive layer. *Group Element* nodes in the diagram correspond to *Relative Position* nodes in the actual structure.

3.3.1 Differences from the conceptual model

Specific layers for the positioning were added because we want to consider the graphical object and its position within the cell as two separate features. Given the overall nature of the RPM sets, multiple identical looking objects in different positions in the same cell can occur, the separation of objects and positions is an efficient way of representing such situations.

Since not every RPM problem contains all features, layers 3, and 5 and 6, are populated only when the presence of *Element Models* and *Groups*, respectively, are detected.

3.3.2 Computation

The graph structure is computed only from the information contained in the input file, and expanded in several stages. Each stage roughly corresponds to a conceptual operation and they are performed in the following order:

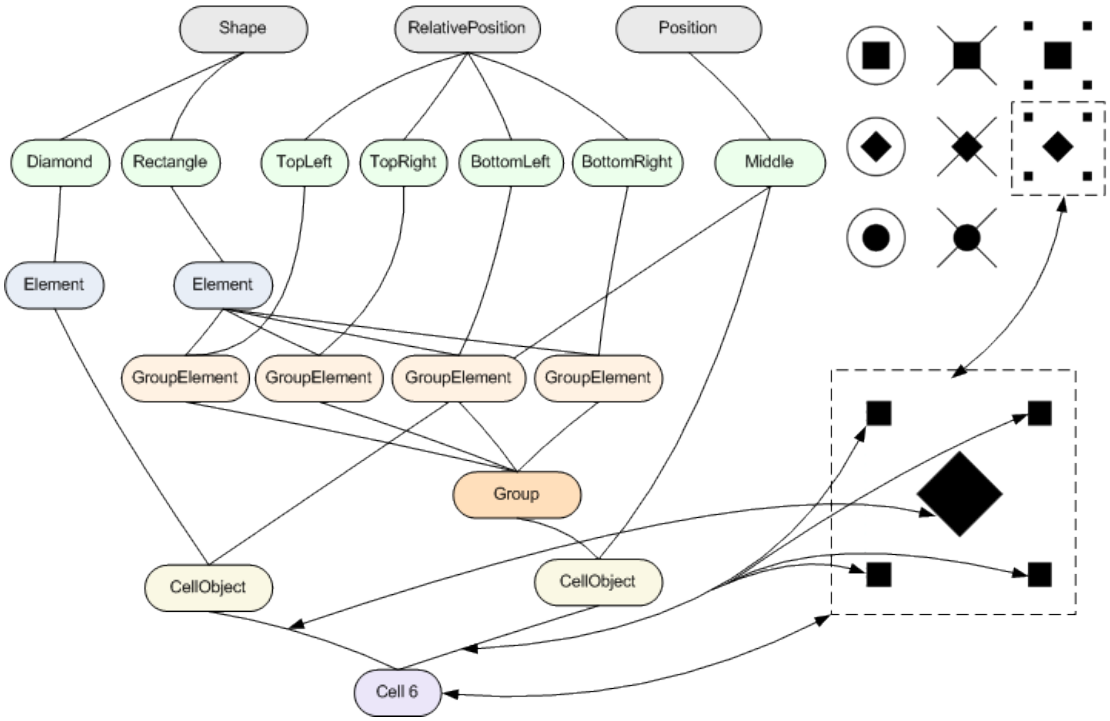

Figure 3.2: Representation structure for a single cell of a moderately elaborate progressive matrix

1. Initialization
2. Common Attributes Extraction
3. Element Instances Cleanup
4. Element Models Creation
5. Common Elements Extraction
6. Element Groups Creation and Group Merging

Initialization

The input file is read and the vector graphics objects of each cell are processed one by one. For each object the names and values of some predefined attributes are extracted and used to populate layers 1 and 2; for a complete list of the attributes taken into consideration refer to Appendix A.

Next, an *Element Instance*, layer 4, is created and connected to the *Attribute Value* nodes.

Finally an *Absolute Position* node, layer 7, is generated and connected to the *Attribute Value* nodes of the object's X and Y position coordinates, the previously created *Element Instance*, and the *Cell* node, layer 8, currently being processed.

Common Attributes Extraction

Attributes appearing in every generated *Element Instance* node with the same value are removed from the structure and not processed any further because they can be considered problem-wide constants, providing no information about the solution pattern.

These common attributes will be factored back in at a later stage; specifically they will be added to the attributes of each generated solution element.

Element Instances Cleanup

Redundant *Element Instance* and *Absolute Position* nodes are merged in this method.

This means that now all identical looking graphical elements in the input are represented by a single *Element Instance* node and that no duplicate *Absolute Position* nodes for the same *Element Instance* are allowed.

Element Models Creation

This procedure creates layer 3, *Element Models*. *Element Models* are sets of attribute values representing maximum common denominators with respect to the *Element Instances*. *Attributes* considered for model creation do not include the ones related to positioning and connected to the nodes in layer 7.

The purpose of *Element Models* is to provide a way to infer additional *Attribute Values* from a single one when inspecting the problem at such level.

Common Elements Extraction

Absolute Position nodes occurring in every cell are at this point removed from the structure and consequently from further processing.

Common Absolute Positions represent constant elements with respect to the cells and are added to the objects contained in the solution cell, along with a reference to the Element Instance they belong to.

Element Groups

Two or more *Absolute Position* nodes are grouped if and only if they appear in the same cells throughout the problem. This concept was introduced is to correctly represent single elements which are impossible to draw with a single glyph in the vector format, for example X shapes or + shapes, but it is also useful to model other situations where multiple elements either appear concurrently or none of them is present.

Absolute Position nodes to be grouped are detected by constructing their cell appearance lists and comparing them: the nodes whose lists are exactly the same will be grouped accordingly. The positions of all group elements are first processed to calculate the group's own center and bounding box; successively they are recomputed as relative, expressed as offsets, with respect to the group's.

In terms of the representation structure the grouped *Absolute Position* nodes are removed from the graph and each of them is replaced by a *Relative Position* node in layer 5; Groups are added as nodes in layer 6; a new *Absolute Position* node for the whole group is added in layer 7.

Group Merging

The main reason we implemented groups was to be able to compare visual objects made up of one *Element Instance* with visual objects made up of multiple *Element Instances*. However, our grouping algorithm works only on the basis of co-occurrence, without considering any other information. This can lead to situations where several compound visual objects are grouped together but elsewhere in the problem a separate group for the single compound visual object has been constructed.

To resolve this issue, from "compound" groups where a subset of the pairs of the *Element Instances* and corresponding *Relative Position* ancestors constitutes the full ancestor set of a different, smaller group, the *Relative Position* ancestors are removed and to represent the corresponding graphical elements an *Absolute Position* node pointing to the minimal group is introduced instead.

Chapter 4

Computational Model

In this chapter we will describe the methods used to solve RPM problems in the form of the structures described in the previous chapter.

The core idea in our solving algorithm is a pattern matching process. We process the problems row by row or column by column, looking for patterns that match on both the first and second row/column and then try to apply them to the last one.

In the example shown in Figure 4.1 the pattern finding algorithm would realize that the *Identity* pattern matches on the first and second rows and would predict the ninth cell, the solution cell, to contain a triangle, since that object shows up in cells seven and eight.

The other important concept is being able to process problems using different abstraction levels. Always in the very simple problems, like the one in Figure 4.1, the whole problem can be solved on the Full Cell level, meaning that it is enough to consider the full content in each cell - no specific processing of the individual objects inside the cells is required.

If instead a slightly more challenging problem is encountered, like the one in Figure 4.2, a cell-wide processing would not be able to predict the content of cell nine. Instead we need to reduce the level of abstraction and process the individual cell objects, resulting in two *Identity* patterns being found, a horizontal one and a vertical one.

In this section we will detail the computational models employed, discuss how we use the concept of abstraction and what are the patterns we currently detect. We will also discuss the possibility of expanding the program to be able to solve more types of problem.

4.1 The Use of Abstraction Layers to Solve RPM

The main reason to use different levels of abstraction in the processing of RPM is to be able to look at each problem in as little detail as possible, while still being able to solve it. If a problem is not solvable at one abstraction level, we move on to the next. The levels of abstraction our representation structure allows the problem to be perceived at, where patterns can occur on, are the following:

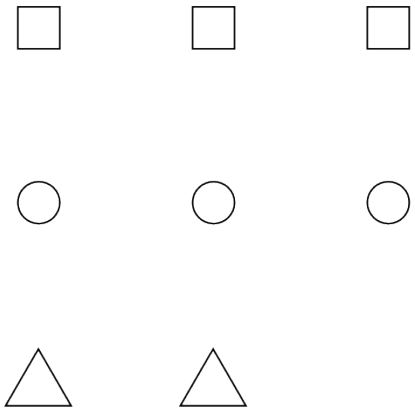

Figure 4.1: A very simple progressive matrix solved by the Identity *pattern on the* Cell *processing level*

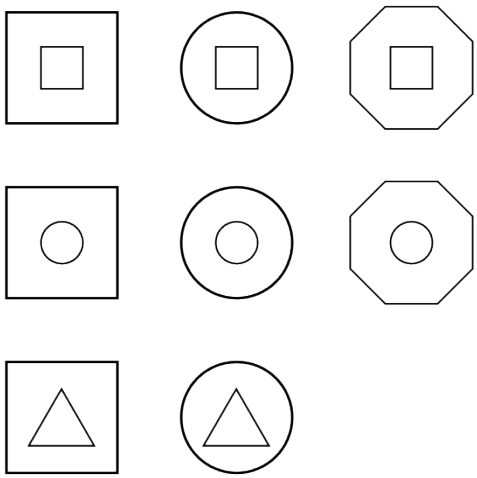

Figure 4.2: Progressive matrix solved by two Identity *patterns on the* Cell Objects *processing level*

- **Cells** - pattern entities are entire cells
- **Object Count** - pattern entities are the numbers of objects within the cells
- **Cell Objects** - pattern entities are positioned objects in the cells
- **Instances** - pattern entities are un-positioned cell objects
- **Element Models** - pattern entities are sets of attributes constituting maximum common denominators between different elements

When a pattern is found to match, it outputs a part of the solution for the missing RPM cell; the level of these partial solutions is the same as where the pattern was found. This means that if a pattern matches on the *Cell* level, the entire content of the solution cell will be predicted and the solution process can stop.

If on the other hand a *Cell Object* is predicted, the processing might not be complete - for this level processing stops only when the number of *Cell Objects* predicted on the *Objects Count* abstraction level has been found.

The process of solving an RPM problem is then defined as iterating through the patterns until the missing cell has been completely determined or until all the patterns on all levels have been tried.

4.2 Pattern Detection

In practice **the patterns are built as separate, easily extendable modules** and are implemented in two parts: one algorithmic function able to operate on the graph structure and one pattern-matching function testing if the pattern holds for a list of entities found by the algorithmic part.

4.2.1 Algorithmic Function Part

For each pattern, the processing of the graph structure can be done in two different ways, supervised or independent. The pattern itself chooses what abstraction levels or types of entities to process in the different modes.

- **Supervised Mode**
  The pattern matching method is iteratively called on all combinations of entities processable at the current abstraction level. The algorithm works in an exhaustive, brute force, way: all combinations will be tested, but the mode is relatively slow. This is the most commonly used mode.
- **Independent Mode**
  Processes the tree in a free, function-specific way. One example would be to simultaneously look at horizontal and vertical patterns and build a compound entity in the solution cell based on both horizontal and vertical pattern findings.

4.2.2 Pattern Matching Part

The pattern matching function of each pattern gets a list of entities and returns a flag to the algorithm that tells if the pattern can be applied or not, predicting a value for the next cell if possible. There are currently seven implemented patterns; as previously mentioned the architecture is very modular and more patterns can easily be added. The final important property is what levels of abstraction are relevant for each pattern. The implemented patterns and the abstraction levels each of them is able to act on are shown in Table 4.1. The details of each function are now going to be explained in more detail.

Pattern	*Abstraction Levels*				
	Cells	**Object Count**	**Cell Objects**	**Instances**	**Element Models**
Identity	X	X	X	X	X
Distribution of Three Entities	X	X	X	X	X
Numeric Progression		X			
Translation				X	
Binary AND	X				
Binary OR	X				
Binary XOR	X				

Table 4.1: Patterns and relevant abstraction levels

Identity

Identity is the first tested function for each level. As the name implies it checks for identity, meaning that it succeeds if and only if all entities sent to the pattern matching function are identical. If they are, the *Identity* function predicts the next value in the series to be the same as the others. The function works for a matrix of arbitrary size.

Distribution of Three Entities

During processing row by row, the *Distribution of Three Entities* function works by identifying three entities that appear in each row, but in different order. For the first row, the function will save the entities passed in and succeed if all entities are different. The second row succeeds if the same entities saved during the processing of the first one are encountered here as well. The final row will succeed if the two entities passed in are found in those in the first row and predict the missing entity.

The pattern works the same way for other processing directions and, despite its name, can process square matrices of any size.

Numeric Progression

The *Numeric Progression* pattern can be thought of as a $+n$ function and only works on the *Objects Count* level, that is on sets of integers $n_1, \ldots, n_m$ where $m \geq 2$, meaning that the matrix' size is 3x3 or larger. For each set of integers passed in the functions checks that the difference between all pairs of integers n_{a+1} and n_a, where $a < m$, are the same. If so it predicts the next value to be the $n_m + n_{a+1} - n_a$. The function checks both the horizontal and the vertical direction simultaneously, making sure that the numeric progression holds for both of them. If the pattern would only check one direction local, false, numeric progression patterns could be detected.

Translation

The *Translation* pattern tracks objects moving in a consistent way between the cells. It first checks that all Absolute Position nodes passed in point to the same instance node, be it an element instance or a group.

When processing the first row it saves the translation (displacement of x and y position) from each cell to the next and for subsequent rows it checks that the translation is the same. When encountering a set of entities containing one element less than those previously processed, it predicts a new element at the last position encountered plus the saved translation between the next to last and last objects in the processed sets.

Binary AND

Binary AND works only on the full *Cell* level and is hence only concerned with lists of *Cell Objects*. It starts with the list of objects in the first cell and performs a binary AND operation of all cells objects in the cells between the first and the last one in the row or column being processed. Finally it checks that the objects left match with the objects in the last cell of the row or column.

Binary OR

The *Binary OR* pattern works the same way as the *Binary AND* one, except that the function performed between the cell objects is binary OR and not binary AND.

Binary XOR

The *Binary XOR* pattern works the same way as the two binary functions introduced above, but employs the XOR function.

4.3 Notes Abobut the Overall Processing

The overall processing is done by testing all patterns for all abstraction levels. Both the patterns and the abstraction level are ordered and after each pattern has been tested the current state of the solution is evaluated to determine if the full solution has been computed or not.

The abstraction levels processed are in order, as previously mentioned:

1. Full Cell
2. Cell Objects Count
3. Cell Objects
4. Instances
5. Element Models

The patterns processed are, in order:

1. Identity
2. Distribution of Three
3. Numeric Progression
4. Translation
5. Binary AND
6. Binary OR
7. Binary XOR

The order of the patterns above is the default order - in chapters 6 and 7 the outcomes of changing this order and omitting some functions are discussed.

Once a pattern function succeeds it returns an entity, whose type depends on the abstraction level currently being processed, that according to the algorithm discussed in 4.2.1 is added to the structure of the solution cell.

Chapter 5

Implementation

The methodology described in this report was implemented as a Windows application written in C#.Net version 3.5. The application's GUI has a tabbed layout with the following panels:

1. Choose Problem
2. Active Problem
3. Visualization
4. Logger
5. MultiLogger

The application is operated by manually selecting the desired input file from the first panel. As soon as the file is loaded the representation structure is built and the solution search begins. Once this process is completed, regardless of whether the problem was solved or not, control is returned to the user and the second panel takes focus, where the problem and the found solution are displayed. At this point it is also possible to access panels 3 and 4 to visualize the constructed graph structure and the event log, respectively.

The five panels are now going to be described in detail.

5.1 Choose Problem

This first panel, shown in Figure 5.1, allows the user to choose and load input files into the program. The file selection dialog features a custom built preview pane where the RPM problem contained in the currently selected file is shown. This preview visualization does not include the solution alternatives.

The Choose Problem panel also includes buttons to automatically pass to the next or previous input file without going back to the file selection dialog. Lastly, after loading a problem file, the panel offers the possibility to run through the whole problem set and output a condensed log to a text file. The condensed log contains information regarding the solution, specifying the matched

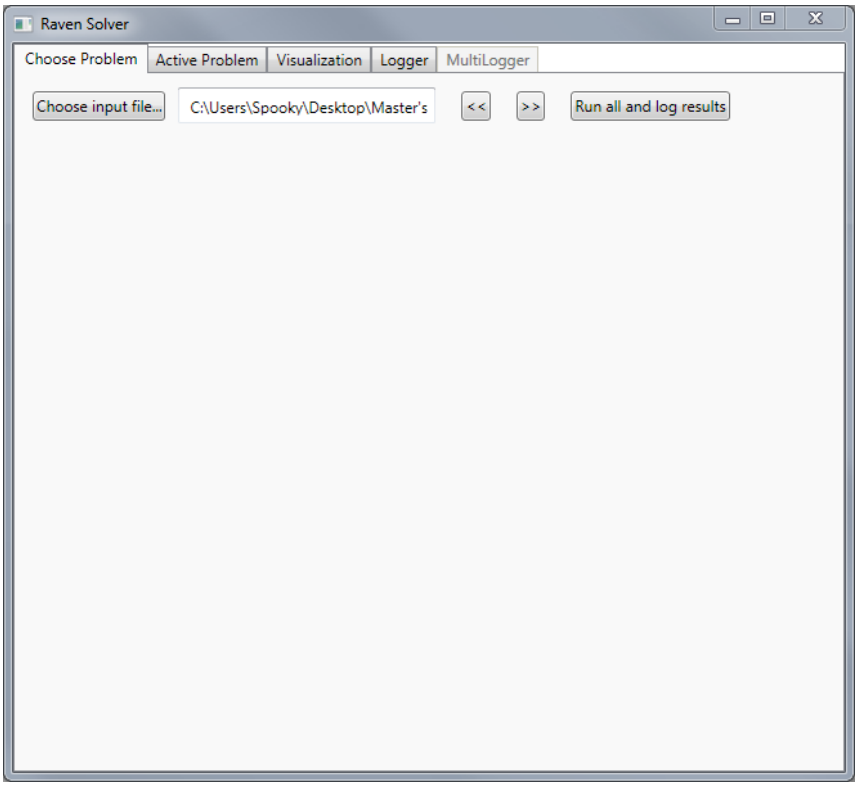

Figure 5.1: Choose Problem interface panel

patterns and the abstraction levels, or lack of thereof for each of the problems, as well as record benchmarking data.

5.2 Active Problem

Here the currently loaded RPM problem is displayed. Additionally, a text label indicating the input file name, zoom controls to scale the drawing and the same previous/next file buttons as in the first panel are present as well. The Active Problem panel also features a checkbox acting as a toggle for the display of the computed partial or total solution to the problem. Finally, the bottom section of the panel includes textual information about the obtained solution indicating which pattern-finding functions have matched, how many matches were found and on which processing levels. The Active Problem panel is depicted in Figure 5.2.

5.2.1 Drawn Solution

When the corresponding checkbox is toggled the found solution is displayed, positioned in its intended location at the bottom-right cell of the matrix. This is accomplished by converting the

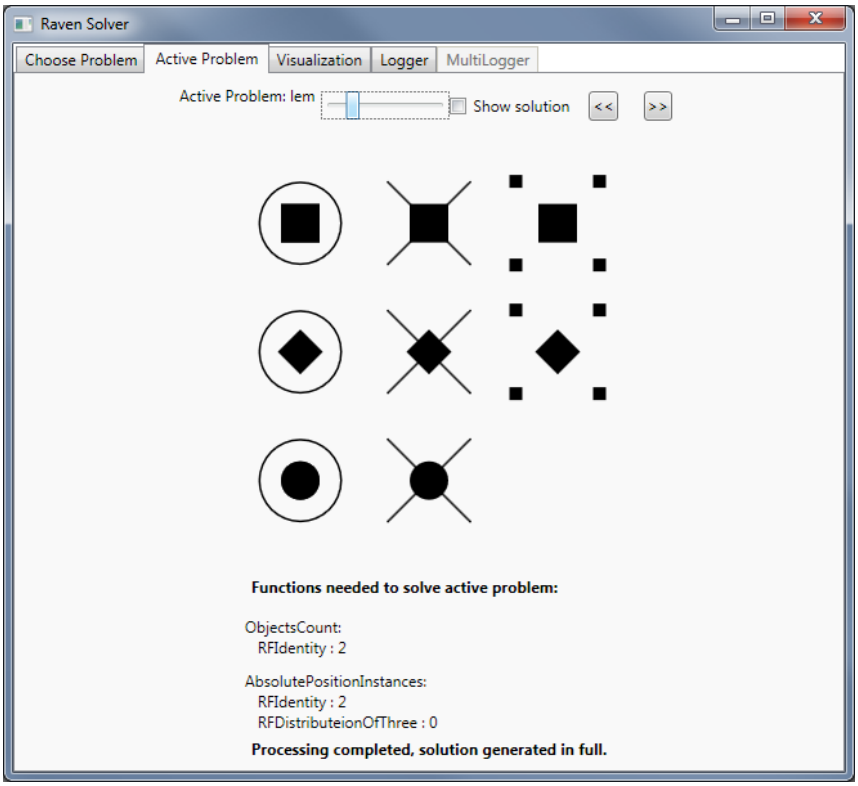

Figure 5.2: Active Problem interface panel

solution from its nodes and arcs form back to vector graphics.

Each drawable (Absolute and Relative Positions, Groups, Element Instances, Cells) node in the structure has a DrawIn method which renders its content relative to a translation passed in as argument. A full cell is drawn by incrementally adding translations and delegating the actual drawing to each individual element: when calling DrawIn on a Cell node the method call is propagated upwards along the tree adding the encountered Absolute and Relative positioning offsets until it reaches the Element Instances, where all the other attributes from itself and its models are retrieved and the graphical object finally drawn.

5.3 Visualization

Here the full internal representation structure is displayed. Each rectangle represents a node and lines are the connecting arcs. The visualization is represented as going from left to right following the layer order explained in section 3.3 but it features an additional Cell Objects layer to more conveniently show the Absolute Positions contained in each cell; Cell Object nodes are just a visual construct and they are not part of the representation structure.

Layer 1 and 2 nodes are labeled with the actual attribute name or value, while labels for other nodes just indicate their type. Layer 2 nodes colored in green represent common attributes, while Layer 7 nodes colored in blue are the common elements.

Figure 5.3 shows the Visualization panel.

Hovering the mouse cursor over a node highlights the arcs connecting it to its parents and children and displays a tooltip providing more information about it. The tooltip contains extended information about the node's makeup in terms of its ancestors all the way up to the individual attributes. When possible the tooltips also feature a drawing of their corresponding node. For nodes not specifying positioning information (Element Instances, Groups) the drawing is rendered as if they were centered within a cell.

Given the large dimensions reached by drawn structures, the Visualization panel includes a zoom control.

5.4 Logger

This panel shows a textual log of messages generated during input file loading and parsing, graph structure construction and problem solution. The messages allow users to keep track of what happened at every processing step and are generated during code execution. The principal log messages for every stage are:

- **Loading/Previewing** Indication of whether files have been correctly loaded and whether problem and solution candidates were found.
- **Structure Construction** Graphical objects found in each Cell and their attributes. Common Attributes detection results. Cells state after Absolute Positions generation. Element Instances and Absolute Positions count after merging the duplicates. Number of Element Models created. Cells state after Element Models creation. Number of created Groups.

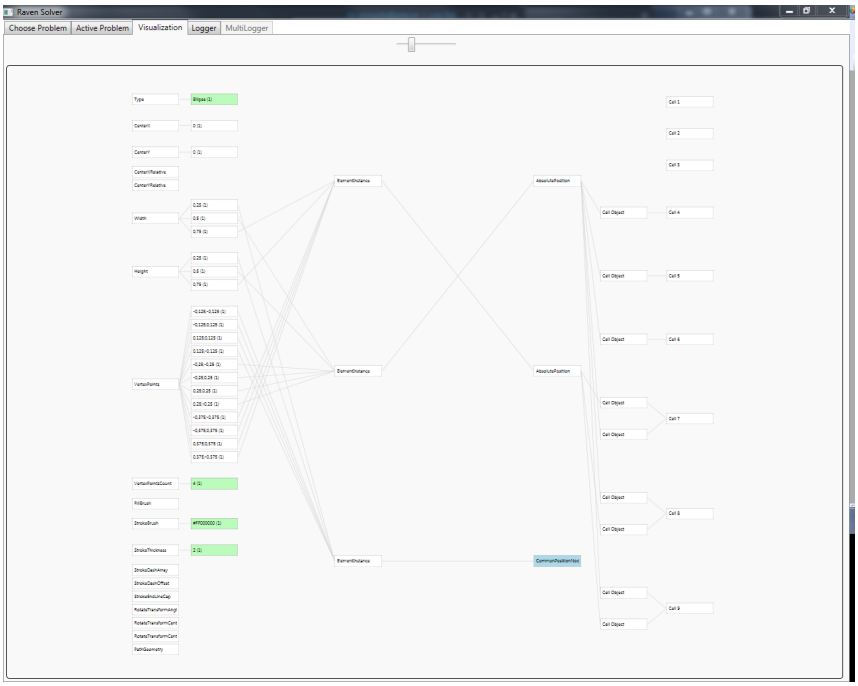

Figure 5.3: Visualization interface panel

- **Solution Process** Notification of which functions are being run, on which level and in which processing direction.
- **Final Report** Same information displayed in bottom part of the Active Problem panel. Reports the functions which found a match, the level where the match were found at, and the number of matches found for each function, level pair.

Figure 5.4 is a screenshot of the Logger panel.

5.5 Multilogger

This last panel, displayed in Figure 5.5, becomes available when the option to run through all the RPM problems at once has been selected in the Choose Problem panel. Here the generated condensed log is displayed.

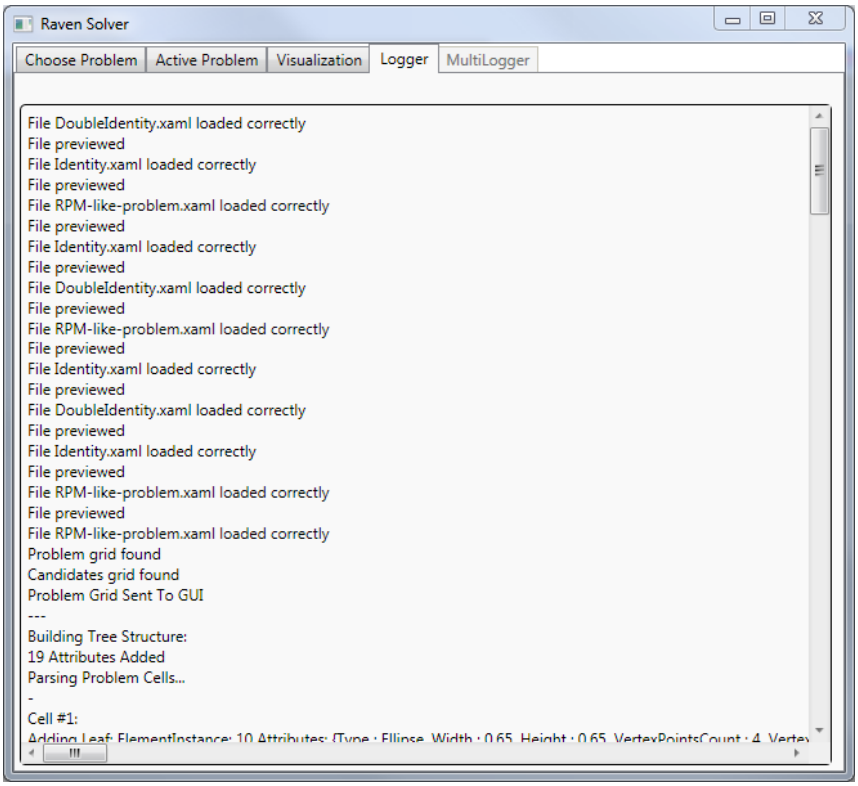

Figure 5.4: Logger interface panel

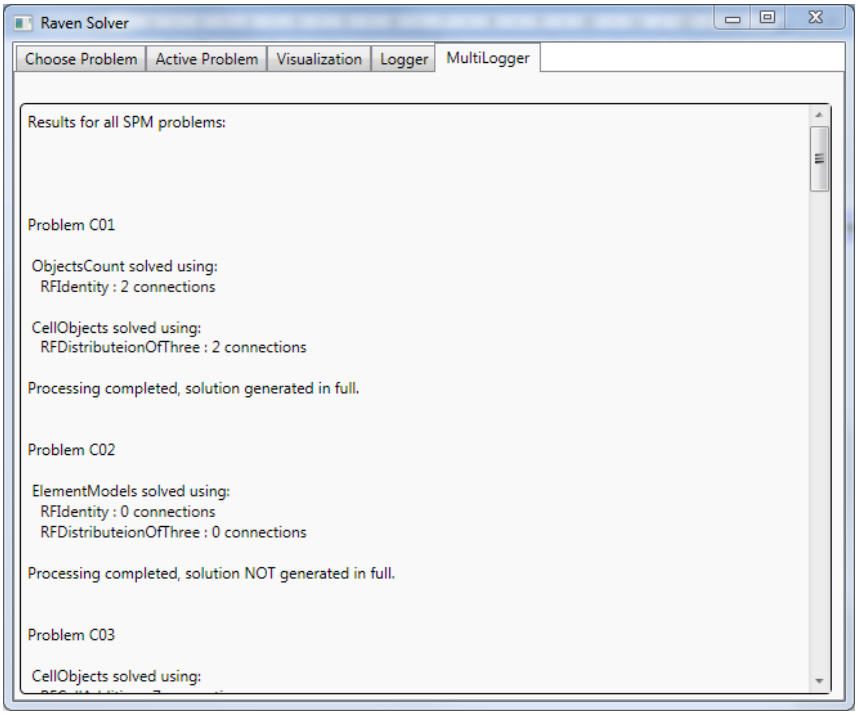

Figure 5.5: Multilogger interface panel

Chapter 6

Results

Our program solves 8 of 12 SPM set C problems, 10 of 12 for set D, and 10 of 12 for set E. In total 28 of 36 problems are solved, 78%.

Removing the problems not solvable without considering the answer alternatives, we solve 85% of the problems.

6.1 Obtained RPM Solutions

Table 6.1 presents a detailed account of the obtained results: for each problem in SPM sections C, D, and E the matched patterns are evidenced along with the abstraction level they were found at, the levels are indicated using the numbered ordered explained in Section 4.3. An indication of whether a complete solution was produced is also present. In the bottom row of the table the sums of the occurrances of each pattern are shown.

6.1.1 Alternative Function Orders

In addition to the processing order of Section 4.3 the program was run using three other function orders, specified as follows:

- **Alternative 1**: In this alternative, the *Binary XOR* pattern is processed before the two other binary patterns, completely eliminating the need for the *Binary OR* function.
- **Alternative 2**: Here we use the original order but run the *Distribution of Three Entities* pattern before the *Identity* pattern, almost removing the need for the *Identity* function
- **Alternative 3**: The last alternative shown combines the two changes mentioned above.

The sums of the each pattern's occurrances for the original and alternative orders are reported in Table 6.2.

When looking at the resulting tables it is evident that a person could get a very high score knowing only *very* few patterns.

Problem	Identity	Distribution of Three	Numeric Progression	Translation	Binary AND	Binary OR	Binary XOR	**Solved**
C01	2	1						✓
C02								
C03						1		✓
C04	3							✓
C05	3		2					✓
C06	2							
C07	2			4				✓
C08			2					
C09	2			4				✓
C10				4				✓
C11			2					
C12		3	2					✓
D01	2	1						✓
D02	2	1						✓
D03		2, 1						✓
D04	2, 3	3						✓
D05	2, 3							✓
D06	2, 3	3						✓
D07	2, 3	3						✓
D08	2	5						✓
D09	2	3						✓
D10	2	3						✓
D11	2							
D12								
E01						1		✓
E02		2				1		✓
E03						1		✓
E04							1	✓
E05							1	✓
E06							1	✓
E07								
E08	2, 3							✓
E09					1			✓
E10		2					1	✓
E11							1	✓
E12								
Occurrances	17	13	4	3	1	4	5	28

Table 6.1: Detailed account of the obtained RPM solutions

	Identity	Distribution of Three	Numeric Progression	Translation	Binary AND	Binary OR	Binary XOR	Solved Problems
Original	17	13	4	3	1	4	5	28
Alternative 1	17	13	4	3	1	0	9	28
Alternative 2	3	20	4	3	1	4	5	28
Alternative 3	3	20	4	3	1	0	9	28

Table 6.2: Pattern occurrances sums for alternative function orders

Taking the results to the extreme by only using *Distribution of Three Entities* and *Binary XOR*, the program correctly solves as many as 21 problems of the total 36, over 58%! Not taking into account the three ones which is not possible to solve with the current approach, those two functions alone solve over 63% of the attempted problems!

Including the *Identity* function as well, which most candidates probably, know 3 more problems are solved, giving a total of 24 solved problems (66% or 72% solved, depending on method used). Adding the *Translation* pattern to the repertoire solves another 3, for a total of 27 solved problems (75% or 81% solved depending on method). To beat the last problem solved by the program in the current implementation the *Binary AND* function is required.

6.1.2 Patterns

As shown, most of the problems can be solved using a very small subset of the implemented patterns. The distribution of the patterns across the sets is worth noting as well: while the makeup of section C is pretty varied, in section D 9 out of the 10 solved problems *need* the *Distribution of Three Entities* pattern to be solvable and in 11 out of 12 problems the *Objects Count* is predicted using *Identity*. For the E set, 8 out of the 10 solved problems only need *Binary XOR* to be solved; some can however be solved with the *Binary OR* function instead.

6.1.3 Abstraction Levels

In 22 out of 36 problems the number of the graphical objects of the solution cell is predictable separately from any other kind of information about them, as evidenced from the results achieved by the Object Count processing level. Since this is the most abstract of the considered levels, it is the only determined one in the four cases where we have obtained partial solutions: C06, C08, C11, and D11.

14 of the 28 solved problems are determined on the Full Cell abstraction level; 13 are solved on Cell Objects level, of which 3 feature two distinct patterns; only one problem, D08, is solved at the Element Models level.

6.2 Computation Times

The execution time for the complete solution process is around 5 seconds, and includes structure computation, solution search, generation of the structure's visual representation and writing of log data to screen and disk.

Looking only at the solution search, not including the rest of the processing described above, the situation is very different. Considering over 10 runs the average problem is solved in under 10ms, the hardest one in around 55ms and the easiest one in less than 4ms. The mean value of the standard deviation over the test problems was around 10ms. This means that the full set of SPM problems C-E is processed in less than half a second.

6.3 Points of Failure

The problems we do not solve fall in two categories:

- Problems for which the solution is incomputable unless the alternatives are considered:
  - C2: the figure gets qualitatively larger, but its components do not scale uniformly or consistently, making it impossible to predict the exact size and position of each.
  - D12: impossibility to determine the curvature and the position of the solution elements
- Problem requiring not implemented patterns or capabilities:
  - C6: point-level geometrical operations (shape morphing)
  - C8: point-level geometrical operations (shape morphing)
  - C11: numeric and positional progression
  - D11: advanced morphing, possibly qualitative
  - E7: categorization
  - E12: abstraction of graphical features to arithmetical entities

Chapter 7

Discussion & Conclusions

In this Chapter we are going to discuss the outcomes and consequences of the work performed and draw some conclusions based on that. First we are going to analyze the results obtained by the implemented program and evaluate its performance against the goals set in Section 1.4.

Next we will compare our approach with the previous attempts documented in 1.3 and make some reflections about Raven's Progressive Matrices as a problem.

Finally, we are going to propose several directions for future improvements of both the program and the representation framework.

7.1 Remarks on the solution process and on the obtained results

Currently, seven patterns are implemented: *Identity*, *Distribution of Three Entities*, *Numeric Progression*, *Translation*, *Binary AND*, *Binary OR*, *Binary XOR*.

However, the patterns are unhomogeneously represented in the full problem set: depending on the application order, *Distribution of Three* can solve as many as 21 problems, while *Binary AND* is used only once.

From the cumulative results of the alternative function orders described in subsection 6.1.1 we can deduce that the classes of problems solvable by the different functions do sometime overlap: *Distribution of Three Entities* "masks" *Identity* in most cases, when applied first; while *Binary XOR* completely "masks" *Binary OR*.

The first behavior is caused by the fact that in many problems there is a horizontal *Distribution of Three Entities* pattern together with a vertical *Identity* or vice-versa and that they can be solved by using either one of them.

The second behavior instead is caused by the equivalence of the XOR and OR operators when the operands do not have "1" values in the same positions:

$$1010 \oplus 0001 = 1011$$

$$1010 \vee 0001 = 1011$$

Which is exactly the situation appearing in the involved RPM items.

Another very important deduction stemming from the results is that the cognitive model used, the order the functions are applied in our case, *does* contribute to the way the solution is computed.

Other variables coming into play are the allowed processing directions, and of course the availability or not of a given function.

7.2 Goals And Performance Evaluation

With respect to the goal of solving as many problems as possible from sections C, D, and E of SPM, the program achieved a good level of performance, 78%. Estimating an IQ score for the program based on those results, however, is well outside the scope of this work.

Our second goal, designing a system able to automatically generate RPM solutions, was achieved even better than the first one, with a success rate of 85% of the total problems where this is actually possible.

We also believe we succeeded in constructing a very powerful and easy to understand conceptual framework to represent and solve RPM and RPM-like problems in, especially considering that by slightly expanding the program's pattern repertoire 100% of the SPM problems where the answer alternatives are not strictly needed would be solvable. While this program still has some way to go to become the optimal RPM solver, it comes closer to the goal than any documented previous program we know of. We also believe our program constitutes a general platform that can potentially handle any kind of progressive matrices problems, even far more advanced than RPM.

Next the different sections of our architecture are going to be critically evaluated.

7.2.1 Representation Structure

The designed representation structures enable the perception of the RPM problems in a way very easy for people to understand and follow, capturing all the available levels of organization at the same time not losing or discarding any information present in the original input. The graph structure used is also economical in the sense that on every level of the hierarchy no duplicate nodes exist.

Given that, considering the complete problem set, patterns are detected on every implemented abstraction level, the representation strategy results seem to be well-structured with respect to the pattern detection methods.

Compared to the work of Carpenter et al. (1990), our structures allow the solution of the same kind of abstract, purely logical, patterns they seek but starting from an unfiltered, drawable description of the problem.

Another positive effect produced by introducing an intermediate representation layer between the input and the solver is the capability of dealing with all those RPM-like problems which are

logically equivalent but graphically different from the original solved test items.

7.2.2 Cognitive Model

The cognitive model we implemented is very simple and rudimentary, consisting only of a series of pattern-finding functions and an application order between them, but it allowed us to solve 85% of the attempted, possible, problems.

Its best conceptual characteristic is that it enables finding the same pattern on different levels, therefore subscribing to a very abstract definition of pattern; a strong technical point instead is the modularity of the code implementing the pattern-finding functions, providing convenient extendability.

About our cognitive model one can argue that its functions can be considered too mechanical: when the processing is ran in supervised mode, they are effectively brute-forcing the subproblem defined by the three specified cells and the objects they contain at the considered abstraction level. Given the very small size of such kind of search spaces, this doesn't result in computational black holes or noticeable slowdowns; however it can produce the effect of masking patterns that may be more obvious to the human observer but that are further down the processing list.

A solution to these concerns could for example be limiting the pattern-finding functions in their supervised processing mode and substituting the simple ordered list of patterns and abstraction levels with a proper action planner in order to orchestrate the solution process in more elaborate ways.

7.2.3 Anthropomorphic Aspects

Anthropomorphic means "Having human characteristics".

In this acception the proposed approach to RPM solution has several anthropomorphic aspects:

1. The representation stage follows heuristics inspired by studies of human visual perception
2. It uses a cognitive model, albeit a very rudimentary one
3. It features patterns that human solvers report to use, proved by the studies of Carpenter et al. (1990) and introspection.
4. It autonomously generates the solutions, behavior found to occur in high achieving human solvers (Carpenter et al., 1990).

The obtained results indicate that anthropomorphic artificial intelligence, the use of cognitive models to solve AI problems while at the same time not modeling the performance bottlenecks of human cognition, and simplicity theory, a framework providing a computable definition of complexity for cognitive systems, are tools worth considering when constructing computer programs for solving tests of human intelligence.

7.3 Considerations on the RPM

Although most RPM problems are well defined and follow certain patterns (most of our pattern functions produce matches in more than one occasion), there are some issues with the underlying problem formulations which will be discussed below. The discussion is purely based in computer science and does not attempt to make any claims about the psychological and psychometrical aspects of the RPM.

7.3.1 Underspecified Problems

One of our objectives is to construct a methodology able to autonomously generate pictorial solutions to RPM problems, without considering the answer alternatives. We have already clarified that this is not the actual protocol for RPM tests, since there candidates have access to the list of proposed solution alternatives, and that ours is a harder task.

However it is not always possible to generate the complete solution without considering the answer alternatives.

While most of the SPM problems are fully determined there are some cases, namely three (C02, D12, E12), where the information extrapolated from the problem grid alone is not sufficient to produce an exact graphical rendition of the solution. There, considering the answer alternatives becomes a necessity in order to solve the task. Specifically the missing information regards the positioning, size or orientation of the graphical elements within the cell.

We ignore if this feature is by design, acting as a discriminant in assessing the capabilities of the test subjects; however it does constitute an important aspect in determining the computational characteristics and requirements of RPM as a whole. Apart from these "unsolvable" problems, our approach shows that the the rest of SPM set are a completely mechanical pattern-matching task requiring no additional information or creative step do be solved. The fact that all but three of the inspected 36 problems can be solved in a completely mechanical way is on its own a very interesting result.

7.3.2 Ad-Hoc Patterns

Most of the RPM items feature and are solvable using patterns that keep recurring throughout the section, set, and even beyond (to the APM problems). In stark contrast, some items feature patterns never occurring again, ad-hoc for all intents and purposes. This characteristic makes it difficult to create a general model for them and even to understand if they are mathematically formalizable at all on the graphical level or if they are just intended to be qualitative, highly abstract relations where the solution alternatives are required to produce a solution as explained in Section 7.3.1.

7.3.3 Vector Graphics Versus Raster Image Representation

In its current state our program starts from a vector graphics representation. Since the intended input stimulus for human solvers is the printed RPM booklet, a collection of raster images, one

might argue that by starting from vectors instead some of the initial pre-processing stages are skipped.

However, the images in the booklet are most certainly created using vector graphics software and the raster images seen in the booklets are just there because no commercial vector graphics printing format exists. Additionally, we argue the human solution process does not start from a raster representation: when solving progressive matrices the conscious mind does not perceive the problem as a dot matrix, but as collections of objects instead.

So, which one of the two formats is the correct one to start from? In this work we focused on the conscious solution process, so for our purposes it is enough to start from a vector graphics representation. We also think the bias of considering the raster level as irrelevant is a reasonable one to have; the progressive matrices problems are most certainly not constructed to test raster image relationships. However we have built the program in such a way that it can easily be extended by a raster to vector graphics preprocessor, thus encompassing also the subconscious parts of human pre-processing.

7.3.4 APM Results Speculation

APM are the natural follow-up to the SPM test we worked on. Following a qualitative analysis we speculate that in its current state our program would be able to correctly solve 7 out of 12 problems in APM Set I. Regarding APM Set II we predict circa 15 out of 36 problems solved.

7.4 Future Work

Below we will suggest future work extending or building on this one. Based on the modular structure of the architecture employed, it should even be possible to directly expand the program described in this report.

7.4.1 More Patterns

Pattern-finding functions for the “ad-hoc” or seldom occurring patterns could be added to the system using the already existing modular function structure. Examples of such patterns are simple arithmetical operations, shape folding, shape stretching, etc.

7.4.2 Advanced Progressive Matrices

The speculations and predictions made about the results of our software on the Advanced Progressive matrices could be verified by producing XAML vector graphics representations of those problems and monitoring the results.

7.4.3 Raw Image Processing

For the reasons discussed in 7.3.3 above, we decided to start our processing from the vector graphics level. However, we welcome future work that builds on our results and expand them with a raster

image preprocessor.

7.4.4 Answer Alternatives

Since for a few RPM items it is not possible to compute the complete solution, the capability of processing the answer alternatives will make it possible to optimally deal with such situations as well.

An implementation for this could consist of creating separate representation structures for the eight solution candidates, using the very same algorithms used to create the one for the problem, compare them one by one with the computed (complete or partial) solution across all levels, and pick the most similar as the answer to the problem.

7.4.5 Architectural Changes

Architectural revisions could be performed in order to more closely mimic human cognitive processing while solving RPM. For instance a limited working memory could be introduced, or an explicit action planner to direct the solution process. This, however, was not one of the goals of the work presented here.

7.4.6 Psychological Dimension

Lastly, it would be interesting to get a psychological view on the herein proved fact that solving RPM only requires a very mechanical approach and that good results can be achieved knowing as few as two patterns and what implications that has for the RPM test as whole.

List of Figures

List of Tables

Bibliography

Bethge, H., Carlson, J. & Wiedl, K. (1982), 'The effects of dynamic assessment procedures on raven matrices performance, visual search behavior, test anxiety and test orientation', *Intelligence* **6**(1), 89–97.

Bringsjord, S. & Schimanski, B. (2003), What is artificial intelligence? psychometric ai as an answer, *in* 'IJCAI'03: Proceedings of the 18th international joint conference on Artificial intelligence', Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 887–893.

Carpenter, P., Just, M. & Shell, P. (1990), 'What one intelligence test measures: A theoretical account of the processing in the raven progressive matrices test', *Psychological review* **97**(3), 404–431.

Chater, N. (1999), 'The search for simplicity: A fundamental cognitive principle?', *The Quarterly Journal of Experimental Psychology A* **52**(2), 273–302.

Chater, N. & Vitányi, P. (2003), 'Simplicity: A unifying principle in cognitive science?', *Trends in cognitive sciences* **7**(1), 19–22.

Dessalles, J.-L. (2010), 'Simplicity theory - unexpectedness'.
URL: *http://www.simplicitytheory.org/*

Forbus, K., Gentner, D., Markman, A. & Ferguson, R. (1998), 'Analogy just looks like high level perception: Why a domain-general approach to analogical mapping is right', *Journal of Experimental & Theoretical Artificial Intelligence* **10**(2), 231–257.

Forbus, K., Lockwood, K., Klenk, M., Tomai, E. & Usher, J. (2004), Open-domain sketch understanding: The nusketch approach, *in* 'AAAI Fall Symposium on Making Pen-based Interaction Intelligent and Natural', AAAI Press, pp. 58–63.

Forbus, K., Usher, J., Lovett, A., Lockwood, K. & Wetzel, J. (2008), Cogsketch: Open-domain sketch understanding for cognitive science research and for education, *in* 'Proceedings of the Fifth Eurographics Workshop on Sketch-Based Interfaces and Modeling'.

Freed, K. & Ram, H. (2005), Finding patterns in series - measuring complexity of human recollection, Master's thesis, Chalmers University of Technology.

Galos, P., Nordin, P., Olsén, J. & Ringnér, K. S. (2003), A General Approach to Automatic Programming Using Occam's Razor, Compression, and Self-Inspection, *in* 'Genetic and Evolutionary Computation GECCO 2003', Vol. 2724 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg.

Kunda, M., McGreggor, K. & Goel, A. (2009), Addressing the ravens progressive matrices test of general intelligence, *in* 'Fall AAAI Symposium on Multimodal Representations'.

Leeuwenberg, E. (1968), *Structural information of visual patterns: an efficient coding system in perception.*, Mouton, The Hague.

Leeuwenberg, E. (1969), 'Quantitative specification of information in sequential patterns.', *Psychological Review* **76**(2), 216–220.

Leeuwenberg, E. (1971), 'A perceptual coding language for visual and auditory patterns', *The American journal of psychology* **84**(3), 307–349.

Legg, S. & Hutter, M. (2007), Tests of machine intelligence, *in* '50 years of artificial intelligence: essays dedicated to the 50th anniversary of artificial intelligence', Springer-Verlag, Berlin, Heidelberg, chapter Tests of Machine Intelligence, pp. 232–242.

Lovett, A., Forbus, K. & Usher, J. (2007), Analogy with qualitative spatial representations can simulate solving ravens progressive matrices, *in* 'Proceedings from the 29th Annual Conference of the Cognitive Science Society', pp. 449–454.

McCarthy, J. (2007), 'What is artificial intelligence?'.
URL: *http://www-formal.stanford.edu/jmc/whatisai/whatisai.html*

Meo, M., Roberts, M. J. & Marucci, F. S. (2007), 'Element salience as a predictor of item difficulty for raven's progressive matrices', *Intelligence* **35**(4), 359 – 368.
URL: *http://www.sciencedirect.com/science/article/B6W4M-4MH8BFY-1/2/7d47b60fcbc995267633f85fdfcb2245*

Palmer, S. (1999), *Vision science: Photons to phenomenology*, MIT Press, Cambridge, MA.

Raven, J. C. (1936), Mental tests used in genetic studies: The performances of related individuals in tests mainly educative and mainly reproductive., Master's thesis, University of London.

Raven, J. C. (1990*a*), *Advanced Progressive Matrices Set I*, Oxford Psychologists Press Ltd.

Raven, J. C. (1990*b*), *Advanced Progressive Matrices Set II*, Oxford Psychologists Press Ltd.

Raven, J. C. (1990*c*), *Standard Progressive Matrices Sets A, B, C, D & E*, Oxford Psychologists Press Ltd.

Raven, J. C. & Court, J. H. (2003), *Manual for Raven's Progressive Matrices. Research Supplement No. 2 and Part 3, Section 7.*, Harcourt Assessment, San Antonio, TX.

Russell, S. & Norvig, P. (2003), *Artificial Intelligence: A Modern Approach*, Prentice Hall.

Snow, R., Kyllonen, P. & Marshalek, B. (1984), The topography of ability and learning correlation, *in* R. J. Steinberg, ed., 'Advances in the Psychology of Human Intelligence', Vol. 2, Erlbaum, Hillsdale, NJ, pp. 47–103.

Strannegård, C. (2007), Antropomorphic artificial intelligence, *in* 'Kapten Mnemos Kolumbarium', Vol. 33 of *Filosofiska Meddelanden - Webbserien*, Göteborgs Universitetet.
URL: *http://www.phil.gu.se/posters/festskrift2/mnemo_strannegard.pdf*

van der Helm, P. (2007), 'Structural information theory'.
URL: *http://www.nici.kun.nl/ peterh/doc/sit.html*

Wertheimer, M. (1939), Laws of organization in perceptual forms, *in* W. D. Ellis, ed., 'A source book of Gestalt Psychology', Routledge, London, pp. 71–88.

Appendix A

List of Processed Atttributes

The attributes processed by the program are either originally present in the input files, or simple quantities derived from them. These attributes are:

CenterX, CenterY
Bounding box coordinates, expressed in a reference frame centered at the cell's center and where the cell height and width are unitary.

CenterXRelative, CenterYRelative
Bounding box coordinates, expressed relative to the group's bounding box.

FillBrush
Brush used to paint the shape's fill. Solid color or textured pattern.

Height, Width
Element height and width, expressed in cell length units.

PathGeometry
Provides a uniform way to operate at the same time on open (Line, Path) and closed (Ellipse, Rectangle, Polygon) shapes. Contains the list of the shape's vertex points sorted according to drawing order, the type of the connecting segments (straight line, arc, etc) and their relevant settings.

RotateTransformAngle, RotationTransformCenterX, RotationTransformCenterY
Parameters of the shape's rotation transform: clockwise rotation angle and rotation center coordinates, respectively.

StrokeBrush
Brush used to paint the shape's outline. always a solid color in this project. Stroke patterns is specified with the next set of attributes.

StrokeDashArray, StrokeDashOffset
Define an outline dashing pattern.

StrokeEndLineCap
Shape at the end of shape outline segments:

- Flat
- Square
- Round
- Triangle

StrokeThickness
Width of the shape's outline.

Type
Element class type:

- Line - Straight line between two points
- Ellipse - Ellipses and circles
- Path - Series of connected lines and curves, arcs, open shapes
- Polygon - Connected series of lines
- Rectangle - Squares and rectangles