Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015)

Automatic Generation of Raven's Progressive Matrices

Ke Wang Zhendong Su
Department of Computer Science
University of California, Davis
{kbwang,su}@ucdavis.edu

Abstract

Raven's Progressive Matrices (RPMs) are a popular family of general intelligence tests, and provide a *non-verbal* measure of a test subject's reasoning abilities. Traditionally RPMs have been manually designed. To make them readily available for both practice and examination, we tackle the problem of *automatically synthesizing RPMs*. Our goal is to *efficiently* generate a *large number* of RPMs that are *authentic* (*i.e.* similar to manually written problems), *interesting* (*i.e.* diverse in terms of difficulty), and *well-formed* (*i.e.* unambiguous). The main technical challenges are: How to formalize RPMs to accommodate their seemingly enormous diversity, and how to define and enforce their validity? To this end, we (1) introduce an abstract representation of RPMs using first-order logic, and (2) restrict instantiations to only valid RPMs. We have realized our approach and evaluated its efficiency and effectiveness. We show that our system can generate hundreds of valid problems per second with varying levels of difficulty. More importantly, we show, via a user study with 24 participants, that the generated problems are statistically indistinguishable from actual problems. This work is an exciting instance of how logic and reasoning may aid general learning.

1 Introduction

Raven's Progressive Matrices (RPMs) are a standardized intelligence test. Each RPM on such a test is an analogy problem presented as a matrix (thus its name). Figure 1 shows an example RPM problem. Each cell in the matrix is filled with a geometric figure, except the last cell, which is left blank. The goal is to select the correct answer from a given set of choices — eight in the example RPM — to fill in the missing cell, so that the nine figures follow some rules. The reader is invited to pause and attempt to solve the given RPM.

There are currently three published versions of RPMs: (1) the original Standard Progressive Matrices (SPM), (2) the Advanced Progressive Matrices (APM), which is more difficult than SPM and designed for adults, and (3) the Colored Progressive Matrices (CPM), which is simpler than SPM and designed for low IQ individuals. Our work focuses on APM problem generation. Intelligence research [Snow *et al.*, 1984] shows that the RPM test provides one of the best single psychometric measures of general intelligence. Since the RPM test is also non-verbal, it is widely used across many cultures and disciplines (*e.g.* clinical, educational, and occupational).

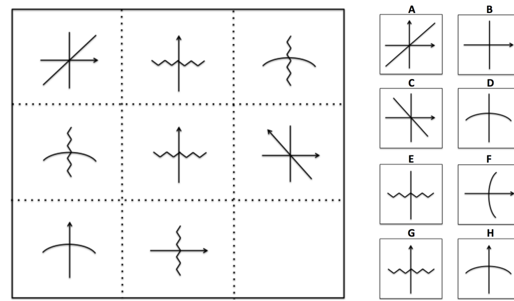
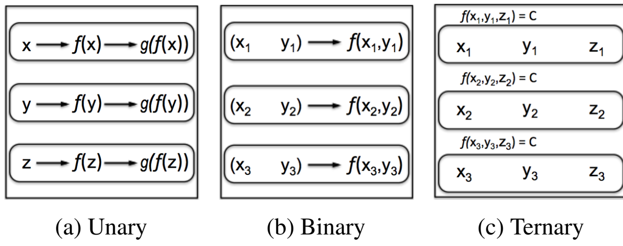


Figure 1: Example RPM problem.

Given their popularity, our goal is to *automatically construct RPMs that resemble the actual problems*. We believe that this automation is very beneficial as RPM problems have been manually designed and are quite limited in number. Thus, our work can provide abundant *high-quality*, *fresh* problems both for practice and examination.

We face two main *technical challenges*: (1) How to model the seemingly irregular and diverse variations in RPM problems? and (2) How to define and only generate valid RPMs? To overcome the first challenge, we formulate RPMs with first-order logic formulae. In particular, we propose a generalization of the concrete rules distilled by Carpenter *et al.* [Carpenter *et al.*, 1990] for describing RPMs. Our generalization is able to model any problem on the APM tests. As for the second challenge, we split a RPM into two parts: the question set (*i.e.* the matrix) and the answer set (*i.e.* the choices), and define respectively the validity of each set. Next, we illustrate how to enforce the generation of valid question and answer sets to form valid RPMs.

We have implemented our approach and extensively evaluated it. Our evaluation focuses on two aspects: (1) performance of the generation procedure, and (2) *authenticity* of the generated RPMs (measured in terms of their conformity to manually written RPMs). Results show that our system is efficient — taking one second to generate hundreds of RPMs,



(a) Unary (b) Binary (c) Ternary

Figure 2: The three categories of relations.

and more importantly the generated problems are similar to manually written problems via user studies.

We make the following contributions:

- We present an abstract representation of RPMs using first-order logic;
- We formulate the notion of well-formedness of RPMs and ensure the generation of only valid RPMs; and
- We realize our approach and demonstrate its clear effectiveness via measurements and user studies.

The rest of the paper is structured as follows. Section 2 presents the overview of our RPM synthesis methodology. Next, we detail our RPM synthesis algorithm (Section 3). Section 4 presents details of our evaluation. Finally, we survey related work (Section 5) and conclude (Section 6).

2 Problem Formulation

We tackle the problem of generating RPMs from two aspects: how to represent RPMs and how to define valid RPMs.

2.1 RPM Representation

RPMs are diverse, and to effectively represent them, it is important to understand their essence. The basic building blocks of a RPM are the figural elements, which possess certain properties, such as *position* and *orientation*. The key characteristic of any RPM is the variation patterns expressed by its figural elements. Carpenter *et al.* [Carpenter *et al.*, 1990] are the first to propose five types of informal rules that govern such variations via a manual inspection of APM problems.

Formalizing and generalizing Carpenter *et al.*'s rules, we introduce an abstract representation of RPM variation patterns using first-order logic. In particular, we model each figural element as an object (denoted by o) and use a finite set of attributes to represent its properties. Each attribute α of o assumes a value, denoted by $\tau(o, \alpha)$, from a fixed domain. The predicate $\rho(o, i, j)$ expresses that object o lies on the ith row and jth column in the matrix. A generic variation pattern may be expressed by the following formula:

$$\exists\alpha\ \forall i \in \{1,2,3\}\ \exists o_1, o_2, o_3\ (\rho(o_1, i, 1) \wedge \rho(o_2, i, 2) \wedge \rho(o_3, i, 3) \wedge P(\tau(o_1, \alpha), \tau(o_2, \alpha), \tau(o_3, \alpha))) \tag{1}$$

where P denotes a predicate.

Intuitively, Formula 1 stipulates that a common attribute of the objects within the cells on each row of the matrix should satisfy the predicate P. Similarly, we can specify such a relationship for objects grouped along each column in the matrix. Depending on how objects are allowed to interact within the same row, we model the predicate P using the following formula:

$$\begin{aligned} \forall\alpha, o_1, o_2, o_3\ (P(\tau(o_1,\alpha), \tau(o_2,\alpha), \tau(o_3,\alpha)) \Leftrightarrow \\ \textsf{UnaryRelation}(\tau(o_1,\alpha), \tau(o_2,\alpha), \tau(o_3,\alpha)) \vee \\ \textsf{BinaryRelation}(\tau(o_1,\alpha), \tau(o_2,\alpha), \tau(o_3,\alpha)) \vee \\ \textsf{TernaryRelation}(\tau(o_1,\alpha), \tau(o_2,\alpha), \tau(o_3,\alpha))) \end{aligned} \tag{2}$$

- ***UnaryRelation***: As Figure 2a depicts, along each row, the element in the first cell determines the second (via unary function f), which in turn determines the third (via unary function g).
- ***BinaryRelation***: As Figure 2b shows, in the binary model, elements in two cells of each row determine the element in the remaining cell (via binary function f). For illustration, we choose the first two cells in a row to determine the last cell.
- ***TernaryRelation***: In the ternary model, elements on the same row satisfy a relation $f(x, y, z) = C$, where C denotes some constant value.

For now, we leave f, g, and C unspecified, but will discuss their choices in Section 3.1.

Besides characterizing RPM variation patterns, Formulas 1 and 2 also establish the basis for creating RPM matrices. Because a matrix can be viewed as an aggregation of variation patterns, generating a matrix reduces to *finding a model* of Formulas 1 and 2. Specifically, we need to (1) assign values to α, $\tau(o_1, \alpha)$, $\tau(o_2, \alpha)$, and $\tau(o_3, \alpha)$ in Formula 1, and (2) implement UnaryRelation, BinaryRelation, or TernaryRelation in Formula 2.

As for generating the answer set, since the correct choice is already determined by matrix generation, we only need to specify the remaining seven incorrect choices. To this end, we use the following formula, where the predicate $\upsilon(o, i)$ specifies that o appears in the ith choice in the answer set:

$$\begin{aligned} \forall i\ \nexists o\ (\upsilon(o, i) \wedge \exists o_1, o_2\ \rho(o_1, 3, 1) \wedge \rho(o_2, 3, 2) \wedge \\ \exists\alpha\ P(\tau(o_1,\alpha), \tau(o_2,\alpha), \tau(o,\alpha))) \end{aligned} \tag{3}$$

Formula 3 ensures that no object exists within any of the incorrect choices that satisfies the predicate P in Formula 1 together with the two given objects on the third row of the matrix. Generating incorrect choices reduces to manipulating elements in the answer cell to create candidates satisfying Formula 3.

2.2 RPM Validity

Now, we tackle the challenge of defining the validity of RPMs. Intuitively, a valid RPM should be unambiguous, *i.e.* there should be a unique answer. We model this intuition with three properties: one concerns the matrix, and other two concern the answer set.

Considering only its matrix (without its answer set), generating a RPM can be viewed as a constraint satisfaction problem: completing the missing cell to satisfy the common constraints from the two complete rows. We first define a *RPM constraint* (which is essentially an instance of the predicate P in Section 2.1).

Definition 1 (RPM Constraint). Let R_1 and R_2 denote respectively the sets of constraints satisfied by the first and

second rows. We define a RPM constraint $\phi \in R_1 \cap R_2$ as the following formula:

$$\exists o_1, o_2\ \rho(o_1, 3, 1) \wedge \rho(o_2, 3, 2) \wedge \exists \alpha, o_3, i\ \phi(\tau(o_1, \alpha), \tau(o_2, \alpha), \upsilon(o_3, i)).$$

C_R denotes a problem R's RPM constraints.

With the notion of a RPM constraint, we can specify the first condition *prerequisite* for a RPM to be well-formed. The condition simply ensures that a RPM must have at least one satisfiable RPM constraint.

Definition 2 (Prerequisite). C_R must be *nonempty*.

Considering the answer set of a RPM, we specify two additional conditions: *correctness* and *necessity*.

Definition 3 (Correctness). C_R is satisfied by a *unique choice* in the answer set.

Definition 4 (Necessity). Any proper subset C of C_R (*i.e.* $C \subsetneq C_R$) is satisfied by *multiple choices* in the answer set.

The correctness condition enforces that if a user identifies all the RPM constraints of a problem, the answer set should have a unique choice. On the other hand, the necessity condition ensures that if the user fails to identify any RPM constraint, the user will face multiple possible choices. Thus, every RPM constraint is necessary.

Definition 5 (Well-Formedness). A RPM is *well-formed* iff it meets the prerequisite, correctness and necessity conditions.

2.3 Example

We illustrate our formulation using the example RPM in Figure 1. In terms of problem representation, the example RPM displays two ternary relations, the union on the *shape* of the vertical (respectively horizontal) elements across each row yields the constant set $\{arrow, wave, curve\}$ (respectively $\{line, arrow, wave\}$).

The example RPM is well-formed because (1) it has two RPM constraints (prerequisite), (2) it has a unique answer (answer "E") considering both RPM constraints, and (3) missing one of the RPM constraints will face multiple choices in the answer set.

3 The RPM Synthesis Procedure

This section presents our RPM synthesis algorithm, which we split into two steps: (1) the generation of the matrix, and (2) the generation of the answer set. We also demonstrate validity of the synthesized RPMs.

3.1 Matrix Generation

Our generation of matrices consists of three steps:

Step 1: Instantiate the abstract representation As explained in Section 2, the generation of a RPM's matrix can be achieved by instantiating Formulas 1 and 2. In particular, we need to assign values to the variables in Formula 1 and implement the relations in Formula 2.

We first introduce how we realize UnaryRelation, BinaryRelation and TernaryRelation, which guide our entire matrix generation procedure:

- ***UnaryRelation***: Let the values of the cells on a row be x, $f(x)$ and $g(f(x))$. The formed relation can be considered an *arithmetic progression* (treating f and g as the same transformation). For example, with the *size* attribute for a row, the relation yields a scaling variation pattern. Note that this pattern may be inapplicable to some attributes, such as the type of the *shape*.
- ***BinaryRelation***: Let the values of the cells on a row be x, y and z. The formed relation is $f(x, y) = z$, where the operator f is chosen to be *conjunction* ($\wedge$), *disjunction* ($\vee$), or *exclusive disjunction* ($\oplus$). For example, with the *texture* attribute and disjunction as the operator f, we obtain the relation that the *texture* of the element in the last cell equals the union of those for the first two cells.
- ***TernaryRelation***: In this case, the values of the cells on a row (namely x, y and z) form a consistent set across each row of the matrix. For instance, the example RPM shows two relations with the *shape* attribute, one for the horizontal elements and one for the vertical elements (see Figure 1 and Section 2.3).

For each instantiation of Formulas 1 and 2, we first select one implementation for the relation among the cells within each row of the matrix. Next, we assign α in Formula 1 a value a from all the attributes of an object. Then, we associate it with three value sequences, denoted by V, computed to satisfy the desired relation. We next illustrate how to fill in the value sequences for each possible implementation of a relation:

- ***Arithmetic progression***: Given the selected attribute a, construct a random sequence of numbers from a's domain to form an arithmetic progression.
- ***Logical operator***: Given the selected attribute a, construct two random values from a's domain, and pick one of the three logical operators to obtain the third value.
- ***Consistent union***: Given the selected attribute a, construct a random collection of values from a's domain, and order them arbitrarily in the array.

If the matrix generation involves multiple instantiations of Formulas 1 and 2, we tag each instantiation to differentiate it from the others. The tag is represented as a name-value pair, *i.e.* by assigning a separate attribute from a.

Step 2: Create the corresponding elements Given each instantiation, we create the objects in two ways:

(1) **Create objects from scratch**: Each instantiation results in the creation of three groups of objects placed in all rows (or columns) of the matrix. Each created object has two attributes (if necessary). The attribute a is assigned a value in V *w.r.t.* the current object's position in the matrix.

For example, if an object resides in the first cell on the second row, the object will be assigned the item V[1][0]. When there are other instantiations, the objects also model the tag from this instantiation.

(2) **Compose attributes on existing objects**: If each cell in the matrix is already filled with objects created from other instantiations, the current instantiation can be modeled as another attribute on the existing objects. In other words, they simply discard their tag and assign the only name-value pair from a and V.

Algorithm 1: Matrix generation

```
1  procedure matrixGeneration()
2    begin
       /* Randomly pick a number greater than 0 */
3      numberOfInstantiaitons ← getRandomNumber()
4      for 1..numberOfInstantiaitons do
5        alpha ← selectAnAttribute()
6        vals ← computeArraysOfValues()
7        if numberOfInstantiaitons >1 then
8          tag ← generateTheTag(alpha)
9        matrix ← createObjects(alpha, vals, tag)
10     generateDistraction(matrix)
```

The mechanism for distributing V onto the existing objects is the same as described in the first case. In particular, we add a and the corresponding value in V as another name-value pair for the chosen existing objects.

Composing the attributes on existing objects is clearly not viable if a selected for the current instantiation has been occupied in the chosen existing objects. In this case, we will need to create new stand-alone objects.

Step 3: Add distractions After having created objects for each instantiation, to increase the difficulty of a RPM, *distractions* may also be added. We can add distractions to a cell along two dimensions: (1) additional attributes for the existing elements, or (2) additional elements. Although those attributes or objects are randomly added, they must not introduce additional legal variation patterns (neither with the existing elements, nor from the objects created for distraction).

These three steps combined lead to our matrix generation procedure, shown in Algorithm 1. Next, we show how to generate the answer set.

3.2 Answer Set Generation

Note first that the correct choice is already determined by the matrix generation procedure. What remains is to generate the other seven, incorrect choices.

Recall Definition 4, which requires that if a user fails to identify any of the RPM constraints C_R, the user will face at least two choices in the answer set. Thus, an obvious approach is to traverse all proper subsets $C'_R \subsetneq C_R$ and use each to synthesize at least two choices. This clearly meets the necessity condition of Definition 4.

In addition, because these synthesized answers only satisfy some, but not all, of the RPM constraints, they are all incorrect. Thus, the correctness condition defined in Definition 3 is also met. Therefore, this brute-force approach does indeed synthesize a valid answer set of a RPM.

However, this process is inefficient; we show how to optimize it to substantially improve the efficiency of answer set generation. First, we describe a property of RPMs that provides the foundation for our optimized procedure.

Proposition 1 (Monotonicity of RPM Constraints). Given two constraint sets C_1 and C_2 for a RPM problem, if $C_1 \subsetneq C_2$, any answer that satisfies C_2 must also satisfy C_1.

Optimization Proposition 1 suggests that it suffices to only consider the C_R's $|C_R|$ *maximal proper subsets* (MPS) with size $|C_R| - 1$ for any RPM. This holds because if all the MPS have more than one answer, any smaller subsets will also have more than one answer. Indeed, each of the MPS can be used once to synthesize an answer. Since a typical RPM has seven incorrect answers, the largest number of RPM constraints is seven (to meet the necessity condition).

Algorithm 2: Answer set generation

```
1  procedure answerSetGeneration(List arrayOfAlphas, Matrix matrix)
2    begin
3      totalFalseChoices ← 7
4      answerObjects ← retrieveAnswerObjects(matrix)
5      iterations ← Ceil(totalFalseChoices/arrayOfAlphas.size())
6      foreach alpha in arrayOfAlphas do
7        value ← getAnswerValue(answerObjects, alpha)
8        domain ← getDomianOfAlpha(alpha)
9        groupOfChoices ← ∅;
10       previousValue ← null
11       for 1..iterations do
12         currentValue ←
             generateNewValue(domain, value, previousValue)
13         choice ← generateAChoice(currentValue)
14         groupOfChoices.add(choice)
15         previousValue ← currentValue
16       answerSet.add(groupOfChoices)
17     answerSet.resort()
18     answerSet.add(answerObjects())
```

Answer Set Generation Algorithm In light of the above optimization, our answer set generation procedure can be simplified to create choices that satisfy each of the MPS of the given RPM constraint set for a problem. Combining with the matrix generation procedure, we realize a RPM constraint as an instantiation to Formulas 1 and 2.

Note that there is a unique value for the answer cell that can satisfy a RPM constraint given two values from the existing cells on the same row (or column). Thus, all the relations (*i.e.* their underlying implementations) are deterministic. Therefore, to create an incorrect choice to satisfy a MPS is essentially to properly alter the answer elements. In particular, we alter the elements determined by the missing RPM constraint.

Putting the above together, we obtain Algorithm 2, our choices generation procedure. Note that Algorithm 2 creates multiple groups to hold the choices generated for each of the MPS (lines 9 and 14). Because the total number of incorrect choices may not be divisible by the total number of MPS, each group may accommodate more choices than necessary (line 5). Later, if needed, the algorithm will randomly remove some extra choices to have exactly seven incorrect choices. It still guarantees that each group has at least one choice that satisfies each of the MPS (line 17).

3.3 Discussions

This section discusses how a generated problem meets the well-formedness property defined in Section 2.

- ***Prerequisite***: As mentioned in Section 3.2, a RPM constraint is essentially accomplished by the realization of an instantiation of Formulas 1 and 2. Because the number of the instantiations realized in the matrix generation procedure is always greater than one, the prerequisite condition is met.
- ***Necessity***: As explained in Section 3.2, for any given

RPM, the generated answer set always contains more than one satisfying choice *w.r.t.* any of the MPS of its RPM constraint set. As mentioned earlier, RPM is monotonic, meaning that any smaller subset will also have more than one satisfying answer. Thus, whichever RPM constraint is overlooked by a user, the user will face at least two choices in the answer set. Thus, the necessity condition also holds.

- ***Correctness***: The only choice in the answer set that meets all the RPM constraints is the one generated by the matrix generation procedure. Any other choice is synthesized to meet only an MPS of the constraint set, *i.e.* it is an incorrect choice. If a user identifies all the RPM constraints, there will only be a single choice in the answer set.

4 Evaluation

This section presents two experiments to evaluate the performance of our synthesis algorithm and the similarity of the synthesized RPMs versus actual APM problems.

We implement our RPM synthesis algorithm in Java SE 1.6 and interface with SVG technology [Ferraiolo, 2000] to render the figural elements in the browser. We design the figures to have fifteen attributes, such as shape, position, orientation and scale. Each of the attributes has a discrete value domain (*e.g.* shape can be assigned *triangle*, *circle*, or *square*), except for position and orientation, which are treated as continuous variables.

4.1 Performance

First, we focus on evaluating the performance of our RPM problem generation algorithm.

We classify the synthesized RPMs into 7 categories according to the number of RPM constraints each problem has. For each category, we synthesize 100 problems and measure the time taken to synthesize each. We conducted our experiments on a desktop with a 4th generation Intel Core i7-4770 processor and 16GB RAM, running Ubuntu 12.04 LTS.

Figure 3a shows the measurement results as a boxplot. We adopt the conventional style of plot where the bottom and top of the box are the first and third quartiles, and the marker inside the box denotes the mean. The two ends of the whiskers represent the minimum and maximum. As shown in Figure 3a, the time taken to synthesize problems increases slightly as the number of the RPM constraints increases. Nevertheless, our synthesis algorithm takes on average under four milliseconds to synthesize a RPM.

4.2 Problem Authenticity

This is the more important aspect of our evaluation. We have conducted a pilot study to carefully assess the authenticity of our synthesized RPMs from the actual APM problems. By authenticity, we mean that RPMs share the same underlying structure (*i.e.* the number and type of the RPM constraints) should be similar in terms of their difficulty level from the user's perspective.

4.2.1 Study Design

Participants We invited twenty-four volunteers in total from our institution to participate in the study. The occupations of the participants vary and range from undergraduate student, graduate student, postdoctoral researcher, and faculty.

Materials The APM test consists of two sets: Set I and Set II. Set I has twelve problems, however, the first four problems are excluded since they are not applicable for reasoning-based solving procedures. The rest of the problem in Set I and the whole Set II are included in the study (44 problems in total). We have synthesized 200 RPMs (without adding distractions) that use similar number and type of RPM constraints as the APMs.

Procedure Each participant was given two test sets, each containing 30 problems randomly sampled from each of the two problem sets.[1] Problems in the APM test are not uniformly distributed in terms of difficulty (there are 21 problems with one RPM constraint, 19 problems with two RPM constraints, and 5 with three constraints). So, we control the sampling process of the synthesized problems to maintain the correlation between the two sets. Specifically, we allow no more than 10 problems with three RPM constraints in the synthesized problem set. Problems in both sets are presented in the order from the simplest to the most difficult. However, in order to imitate the real RPM test environment, participants are free to navigate through the tests. The total time is 30 minutes for each of the test set. Half of the participants completed the APM test set before beginning the other test set, and the other half completed the test set with synthesized problems first.

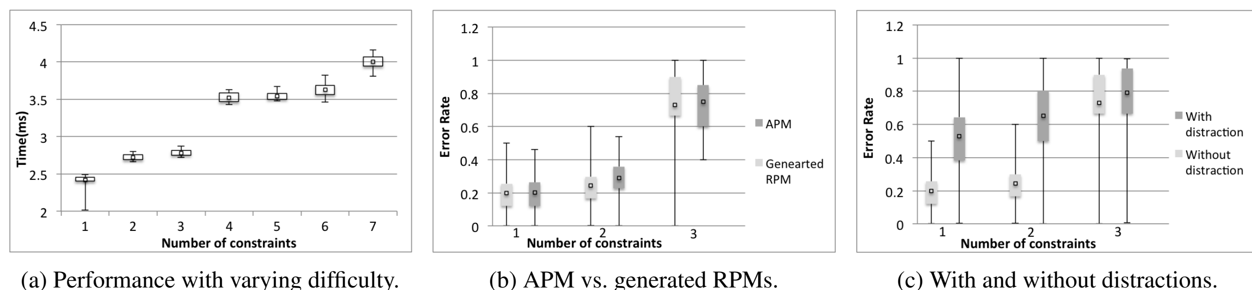
4.2.2 Study Results

In the following discussion, we split the two problem sets into three categories and report the results for each category separately. To measure the similarity of the two problem sets within each of the categories, we run a two one-sided test for equivalence [Schuirmann, 1987] on the participants' error rate. An important decision is how to define the zone of "clinical indifference", *i.e.* a range of effects that can be considered clinically trivial.

For this purpose, we randomly partitioned the APM problems within each category that a participant has been tested on into two halves and summarized the participant's error rates for each half. Next, we apply the two sided test to compute the threshold error margin — anything less than this value would make the two APM partitions dissimilar. But the two randomly partitioned problem sets from APM must be similar, so we can use the threshold error margin to compute the similarity between the synthesized problems and the APM.

Comparison Figure 3b shows the error rates for the two problem sets within each category. The error margins are computed to be 0.089, 0.139 and 0.242 from the two random partitions on the APM problem set within the category of one-constraint, two-constraint and three-constraint respectively. Then, a two-sided test was used to compare each participant's error rates across the two sets of matrices within each of the categories. These tests show that

[1] Please refer to http://www.cs.ucdavis.edu/~su/rpm.html for a sample of 30 synthesized problems as given to the participants. Due to copyright constraints, we do not include the APM problems that we purchased from Pearson at http://us.talentlens.com/pricing#ravens.



(a) Performance with varying difficulty. (b) APM vs. generated RPMs. (c) With and without distractions.

Figure 3: Evaluation results.

the error rate conforms significantly across the two sets of problems [$t(46) = -2.490, p = 0.008$] and [$t(46) = 2.596, p = 0.006$], [$t(46) = -2.572, p = 0.007$] and [$t(46) = 5.016, p < 0.001$], [$t(46) = -3.595, p = 0.001$] and [$t(46) = 4.212, p < 0.001$], indicating that the overall difficulty of the synthesized matrices is similar to that of the original APMs.

Controlled Measure of Problem Difficulty Apart from the error rates reported from Figures 3b, a paired *t* test was used to compare each participant's error rate across the one and two RPM constraints, and two and three RPM constraints matrices in the synthesized problem set only. As expected, the error rate for the two RPM constraints problems was higher than the one RPM constraint problems [$t(23) = 1.98, p = 0.06$] despite the unsubstantial significance, indicating that the two RPM constraints problems were more difficult. Similarly, the error rate for the three RPM constraints problems was significantly higher than the two RPM constraints problems [$t(23) = 9.46, p < 0.001$], indicating that the three RPM constraints problems were more difficult.

We have conducted another user study where the same participants were asked to complete another separate test set consisting of 30 synthesized problems. Those thirty problems are randomly sampled from a corpus, where each problem is also synthesized with distraction. The purpose of this study is to assess the impact of distractions may have on the generated RPMs. Figures 3c shows that problems created with distractions are indeed more difficult than those created without distractions, using the same number of RPM constraints.

A paired *t* test was used to compare each participant's error rate across the two problem sets with or without distraction for the same number of RPM constraints. For problems with one RPM constraint and two RPM constraints, the error rate for problems mixed with distraction was significantly higher than those without [$t(23) = 7.81, p < 0.001$] and [$t(23) = 9.21, p < 0.001$], indicating that problems with distractions were more difficult than those without distractions. As for RPMs with three constraints, the difference on error rate was insignificant [$t(23) = 1.51, p = 0.14$] because the participants' error rates on the three RPM constraints matrices were already very high, leaving little room to be able to perceive the increased difficulty.

Completeness It is pertinent to ask whether our system can generate all the matrices in the APM test set. Thus, we have also manually inspected each of the matrices in the APM set and found several problems that our system cannot synthesize. The main reason is that the recently published APM test set contains some underlying transformation patterns not covered in our current implementation. Nevertheless, they could still be instantiated from the three categories of relations we have proposed in this paper. Thus, conceptually, if we include those specific transformation operators, our tool would be able to synthesize all the problems in the APM set.

5 Related Work

Automated RPM problems reasoning has been an active research topic in cognitive systems. Based on the two qualitatively different RPM strategies, "Analytic" and "Gestalt" proposed by Hunt [Hunt, 1974], we split existing computational models for solving RPMs into two groups, and briefly discuss each. The RPM models developed to resemble Hunt's Analytic algorithm rely on propositional representations of the problem inputs, and do not carry any structural correspondence to what they represent. For example, the computational models proposed by Carpenter *et al.* [Carpenter *et al.*, 1990], Lovett *et al.* [Lovett *et al.*, 2010], Cirillo *et al.* [Cirillo and Ström, 2010] and Rasmussen *et al.* [Rasmussen and Eliasmith, 2011] all reason over a problem's propositional description, which usually is manually converted from the given problem.

Another complementary computational model following the Gestalt algorithm uses iconic representations of an input problem. Those representations, as the name suggests, carries the structural correspondence between format and content. The most notable are the affine model [Kunda *et al.*, 2012; 2013] and the fractal model [Kunda *et al.*, 2012]. A significant advantage of those models is the elimination of the manual conversion from problem inputs to propositional representations. In addition, many existing image transformation techniques can be leveraged to handle those iconic representations.

Unlike the aforementioned work on solving RPM problems, our work tackles an orthogonal dimension, namely the automated generation of RPM problems. The main novelty of our work lies in the logic model of RPM problems and the formalization covering special properties of RPMs.

Matzen *et al.*'s Sandia matrix [Matzen *et al.*, 2010] is the only effort that we are aware of that considers the automated generation of RPM problems. Their approach is to manually extract all types of relations that appear in Raven's Standard Progressive Matrices (SPM) through a manual study, and combine those extracted relations to generate new problems. For comparison, our work proposes the abstract representation of

variation models, from which the concrete relations among the elements within the cells can be automatically synthesized. Matzen *et al.* also represent a strong effort to investigate the difficulty level for each type of the relations and combination of the relations through a user study. In contrast, we tackle the problem at a higher level by identifying the factors that can influence the difficulty of RPM problems. Our user study has clearly confirmed our approach's effectiveness.

6 Conclusion

This paper has introduced an approach to effectively synthesize RPMs. Our evaluation results demonstrate the performance of our synthesis algorithm and strong resemblance of the synthesized RPMs to actual published problems. We expect that our work will facilitate general education and training. Our immediate future work is to reach out to potential user groups that can benefit from our system.

References

[Carpenter *et al.*, 1990] Patricia A Carpenter, Marcel A Just, and Peter Shell. What one intelligence test measures: A theoretical account of the processing in the Raven Progressive Matrices Test. *Psychological Review*, 97(3):404–431, 1990.

[Cirillo and Ström, 2010] Simone Cirillo and Victor Ström. An anthropomorphic solver for Raven's Progressive Matrices. Technical Report 2010:096, Chalmers University of Technology, Goteborg, Sweden, 2010.

[Ferraiolo, 2000] Jon Ferraiolo. *Scalable Vector Graphics (SVG) 1.0 Specification*. Iuniverse Inc., 2000.

[Hunt, 1974] Earl Hunt. Quote the Raven? Nevermore! In L. Gregg, editor, *Knowledge and Cognition*. L. Erlbaum Associates, Hillsdale, NJ, 1974.

[Kunda *et al.*, 2012] Maithilee Kunda, Keith McGreggor, and AK Goel. Reasoning on the Raven's advanced progressive matrices test with iconic visual representations. In *34th Annual Conference of the Cognitive Science Society*, pages 1828–1833, 2012.

[Kunda *et al.*, 2013] Maithilee Kunda, Keith McGreggor, and Ashok K Goel. A computational model for solving problems from the Raven's progressive matrices intelligence test using iconic visual representations. *Cognitive Systems Research*, 22:47–66, 2013.

[Lovett *et al.*, 2010] Andrew Lovett, Kenneth Forbus, and Jeffrey Usher. A structure-mapping model of Ravens Progressive Matrices. In *32nd Annual Conference of the Cognitive Science Society*, 2010.

[Matzen *et al.*, 2010] Laura E Matzen, Zachary O Benz, Kevin R Dixon, Jamie Posey, James K Kroger, and Ann E Speed. Recreating Raven's: Software for systematically generating large numbers of Raven-like matrix problems with normed properties. *Behavior Research Methods*, 42(2):525–541, 2010.

[Rasmussen and Eliasmith, 2011] Daniel Rasmussen and Chris Eliasmith. A neural model of rule generation in inductive reasoning. *Topics in Cognitive Science*, 3(1):140–153, 2011.

[Schuirmann, 1987] Donald J Schuirmann. A comparison of the two one-sided tests procedure and the power approach for assessing the equivalence of average bioavailability. *Journal of Pharmacokinetics and Biopharmaceutics*, 15(6):657–680, 1987.

[Snow *et al.*, 1984] Richard E Snow, Patrick C Kyllonen, and Brachia Marshalek. The topography of ability and learning correlations. *Advances in the Psychology of Human Intelligence*, 2:47–103, 1984.