Cite as: P.C.R. Lane, L. Bartlett, N. Javed, A. Pirrone and F. Gobet, 'Evolving understandable cognitive models', in Proceedings of the 20th International Conference on Cognitive Modelling, 2022.

Evolving Understandable Cognitive Models

Peter C.R. Lane (p.c.lane@herts.ac.uk)
School of Physics, Engineering and Computer Science, University of Hertfordshire, College Lane, Hatfield AL10 9AB, UK

Laura Bartlett (l.bartlett@lse.ac.uk)

Noman Javed (n.javed3@lse.ac.uk)

Angelo Pirrone (a.pirrone@lse.ac.uk)

Fernand Gobet (f.gobet@lse.ac.uk)
Centre for Philosophy of Natural and Social Science, London School of Economics, Houghton Street, London WC2A 2AE, UK

Abstract

Cognitive models for explaining and predicting human performance in experimental settings are often challenging to develop and verify. We describe a process to automatically generate the programs for cognitive models from a user-supplied specification, using genetic programming (GP). We first construct a suitable fitness function, taking into account observed error and reaction times. Then we introduce post-processing techniques to transform the large number of candidate models produced by GP into a smaller set of models, whose diversity can be depicted graphically and can be individually studied through pseudo-code. These techniques are demonstrated on a typical neuro-scientific task, the Delayed Match to Sample Task, with the final set of symbolic models separated into two types, each employing a different attentional strategy.

Keywords: cognitive modelling, genetic programming, model visualisation

Introduction

Developing and verifying the behaviour of cognitive models is a non-trivial task. Ideally, a cognitive model will provide some explanation of how a human performs in a particular experimental setting, and even provide predictions for new settings. In many cases cognitive models are based around computer programs which need to be written. The area of *program synthesis* studies ways to generate executable computer programs from user specifications. In this paper we demonstrate how an evolutionary algorithm can generate programs representing candidate computational models in a typical neuro-scientific experiment. We present techniques to improve the understandability of the resulting programs, which enables their use as the starting point for developing scientific theories.

We use the evolutionary algorithm Genetic Programming (GP) (Koza, 1992) to search a space of programs. The fitness function guiding this search is designed to find programs which effectively simulate the behaviour of human subjects. Unlike many typical GP applications, this fitness function is not based directly on an input-output mapping for the program. In particular, human subjects do not achieve 100% success in our example task, and so the 'best' model is one which replicates this less-than-perfect accuracy. Also, as the responses made by a human take a certain amount of physical time, a simulated time for the program to convert each input into an output must be measured and compared with the observed response time. These two performance measures must be captured in a combined fitness function: how to do this effectively is the first contribution of this paper.

The GP system often generates a large number of candidate models: we want to convert these into a small set of representative, understandable and qualitatively different models. We achieve this with a series of post-processing steps to remove unnecessary operators from the programs and remove duplicates. Finally, the programs can be changed to pseudo-code, for further analysis, and a visualisation made to highlight the relationships between the final solutions. These techniques form our second contribution.

Background

Cognitive modelling is a process in which computational models of a target behaviour are sought in an attempt to understand human behaviour. These computational models are typically developed within a given *framework*, such as a *symbolic* (Simon, 1981) or *connectionist* (Rumelhart & McClelland, 1986) framework. In this paper, we consider a framework of symbolic models, typical of cognitive architectures such as ACT-R (Anderson & Lebière, 1998) or CHREST (Gobet & Simon, 2000). However, even within a single framework, there are still many possible models which could be developed, each with qualitatively different behaviour. For example, the manner in which a visual scene is scanned for information could be systematic and wide-ranging, or task-oriented and narrow, and either way could be sufficient for achieving the target performance: scientifically, it is useful to be aware of both possibilities, but often time constraints or natural bias (oversights) lead to models written by human programmers being constrained to particular groups of solutions.

Using search algorithms to explore a solution space for one or more candidates is a technique with a long history (Langley, Simon, Bradshaw, & Zytkow, 1987; Schmidt & Lipson, 2009). GP approaches to this exploration are also widely known, although there appear to be few studies in the area of cognitive science, exceptions being Frias-Martinez and Gobet (2007); Lane, Sozou, Gobet, and Addis (2016).

Our approach using GP appears unique in developing cognitive models which focus on symbolic, information-processing (Simon, 1981) explanations of human cognition.

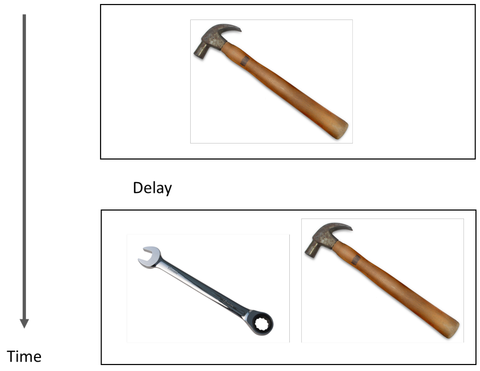

Figure 1: Illustration of DMTS task. (Photos by Danny de Bruyne and Ronaldo Taveira, freeimages.com)

This contrasts with many current approaches in artificial intelligence which rely on connectionist (statistical) explanations based on large datasets: a recent study in this area is that of Peterson, Bourgin, Agrawal, Reichman, and Griffiths (2021).

Model Development System

Our proposed system for automatically developing cognitive models is an example of *program synthesis*. Such systems can be conveniently divided into three parts (Gulwani, 2010): the task definition (*user intent*), to express what makes a good program; a *search space* of candidate programs; and a *search technique*, to explore the given search space for good programs. Here, the developed programs form the control structure for the cognitive models.

Task definition: DMTS

The task studied in this paper is a typical neuro-scientific experiment, popular for studies of short-term memory, which tests the accuracy and reaction time for subjects to recognise images: this is the Delayed Match to Sample (DMTS) task (Chao, Haxby, & Martin, 1999). In this experiment, illustrated in Figure 1, a picture is presented for 1 second in the center of the screen. Then, after a delay of 0.5 seconds, two pictures are presented for 2 seconds, one on the left and the other on the right of the screen. The participant has to select which of those two pictures is the same as the first picture.

The cognitive model must coordinate the perception of time-sensitive information with accurate responses within expected response times. We simplify the task by abstracting away the recognition of images: we have six 'images', represented by the cardinal numbers from 1 to 6.

Although this task is an example of "programming-by-example", where the model must reproduce the example input-output behaviour, the overall quality of the model is not judged on the number of correct input-output pairs. As reported in Chao et al. (1999), across the complete set of presentations, human subjects only score 95.7% accuracy, with an average response time of 767ms: the model's accuracy and simulated response times are judged against these values.

Search Space: Cognitive Models

Each individual cognitive model is defined by a control program to be interpreted within a simple cognitive architecture. This architecture has some task-specific input/output components: a set of *inputs* and a *response*. It also has some task-independent components: a fixed-size short-term memory (STM), and a working memory *current*. Finally, each model has a *clock*, to record its current in-task time.

The model control program is composed from a set of operators, listed in Table 1. These operators define a simple imperative programming language, where operators can be combined in sequence, selected with a conditional statement, and repeated in fixed-cycle loops. The model's current working value, STM and clock values can all be manipulated, inputs read and a response prepared: the current response is "made" when the program ends. Operators are arranged in groups, matching their simulated execution time (based on estimates from the psychological literature): *input operators* (100ms), *output operators* (140ms), *cognitive operators* (70ms), *STM operators* (50ms) and *syntax operators* (0ms).

Search Technique: Genetic Programming

Genetic Programming (GP) (Koza, 1992) is an evolutionary search technique which works by creating a population of candidate solutions and then gradually evolving this population through several iterations until a termination condition, such as the number of iterations, has been met. The evolution step is loosely based on biological evolution, with candidate solutions selected based on their *fitness*. New candidate solutions are created from existing candidates through the processes of *crossover* and *mutation*, which respectively combine or modify existing solutions.

Fitness Function The fitness function is used to rank different candidate solutions when choosing which candidates should be combined or used when the GP process constructs the next population. The fitness function used here is constructed from three components: *accuracy*, *response time* and *program size*. Accuracy is the overall performance of the model, based on the proportion of input-output pairs that it gets correct: accuracy is assessed in the range $[0,1]$. Response time is measured in simulated milliseconds, and program size is the number of operators in the control program.

As described above, accuracy is compared with the performance of human subjects: the closer the value of accuracy is to 0.957, the better it is. Similarly, the closer the value of the response time is to the target average of 767ms, the better. For response time, because the values can become large, we use a half-sigmoid function to rescale the numbers into the

Table 1: Overview of operators used in DMTS models.

Name	Function	Type
input-X	sets model 'current' to value of left/right/target input, if it is visible	input
respond-X	sets model 'response' to "R"/"L", if inputs are visible	output
access-N	sets model 'current' to STM item N ($N \in \{1,2,3\}$)	stm
compare-M-N	compares value of STM items M and N ($M \neq N \in \{1,2,3\}$) and sets 'current' to 1 if equal, or 0 if not	cognitive
nil	sets model 'current' to 0	cognitive
put-stm	pushes value in model 'current' to top of STM	stm
dotimes-N	repeats a given expression ($N \in \{2,3,5\}$)	syntax
if	executes condition, executes one of two expressions based on value in model 'current'	syntax
prog-N	sequence of expressions ($N \in \{2,3,4\}$)	syntax
wait-N	advances model clock ($N \in \{25,50,100,200,1000,1500\}$)	syntax

range $[0,1]$. Program size is treated like response time, with an arbitrary target of 10 operators. All three components are evaluated so that values closer to 0 indicate a 'better' fitness.

Formally, the three components of the fitness function are:

1. $f_a = |\text{accuracy} - 0.957|/0.957$: this is the difference of the model's and target accuracy, scaled to the range $[0,1]$.
2. $f_t = \text{half-sigmoid}(|\text{response-time} - 767|/\text{RT})$: this is the difference of the model's and target response time, with a variable scale factor RT.
3. $f_s = \text{half-sigmoid}(|\text{program-size} - 10|/\text{PS})$: this is the difference of the model's and an arbitrary target program size of 10, with a variable scale factor PS.

where $\text{half-sigmoid}(x) = 2 \times (1/(1+e^{-x}) - 0.5)$ is the usual sigmoid function which we rescale from [0.5,1] to [0,1], because all our values of x are positive. The variable scale factors in f_t and f_s control the steepness of the sigmoid slope.

The *overall fitness* is computed as a combination of these three, with multipliers $a+b+c=1$ ensuring that the overall fitness is in the range $[0,1]$:

$$f = a \times f_a + b \times f_t + c \times f_s$$

Phased Evolution In earlier experiments, GP struggled to find solutions using this overall fitness function. The difficulty appeared to be that the requirement to minimise program size or meet a target response time would override the need to observe and predict a correct response. Hence, the idea of what we call *phased evolution* was created, to break this multi-component problem into stages. The evolutionary process is separated into three phases based on which of the three components are used in the fitness function: phase 1 uses one component (f_a), phase 2 uses two components (f_a and f_t), and phase 3 uses all three. The system starts in phase 1. It moves to the next phase when the best model's fitness is less than a threshold value (0.1 here).

More precisely, in:

Phase 1 fitness $f = f_a$

Phase 2 fitness $f = (a \times f_a + b \times f_t)/(a+b)$

Phase 3 fitness $f = a \times f_a + b \times f_t + c \times f_s$

The intention of this phased introduction of fitness components is that the GP system should first evolve models to perform accurately on the task, when compared to the target behaviour. Once models have been created which meet the required threshold ($f < 0.1$), then they must additionally match the required reaction time. When the final component is added in, the GP system should already have a population of models able to achieve good accuracy and response times, and can now concentrate on reducing the size of the models.

Post-Processing

Genetic programming (GP) is highly effective at locating candidate programs which fit target behaviour in complex applications. However, the range of interesting solutions is obscured by the large number of evolved candidates, formed from a combination of dead code (bloat), functionally similar program segments with varying contents, and genuine differences in possible solutions. In order to make the candidate programs more understandable, we introduce a series of post-processing steps to generate fewer, high-quality solutions.

Dead code removal

A standard problem with GP systems is that of "bloat" (Langdon & Poli, 1998): an example of bloat is where programs contain operators which are not executed when the task is run. This *dead code* can occupy the majority of a program, frequently over 90%. One way to remove dead code is to add the program size as one of the components in the fitness function. However, as we find in our experiments, this is not completely effective.

A more effective way to remove dead code starts by tracing the operation of each evolved program on our task and recording those parts of the program which are not executed: it is important that our task is *deterministic* so this can be done reliably. All non-executed code is then replaced with the special node "UNUSED". Conveniently, all non-executed code must be on one branch of an IF-statement: the code is not run because the condition on the IF-statement always returns a value which uses just one branch of the IF-statement. For example, if some CONDITION always returns a true value, its else branch will never be executed:

```
(IF (CONDITION) (SOME-CODE) (UNUSED))
```

The programs can be simplified by replacing all such code to remove the UNUSED branch:

```
(PROG2 (CONDITION) (SOME-CODE))
```

The condition must still be executed as it could contain side-effects and takes up some execution time, which is critical for the timing performance of the model.

This step helps in two ways:

1. The population of candidate models is reduced dramatically, by removing those which differ only in the contents of the dead code.

2. Each individual model is simplified, with only important parts of its control program remaining.

Time-only code removal

There is a further aspect of the candidate models which can be simplified. As the model is optimised to perform against time, some of the operators within the control programs can be important only for their timing – they do not affect the performance. For example:

```
(PROG2 (INPUT-LEFT)  (INPUT-RIGHT))
```

In this program, the model first looks at the left input, and then looks at the right input. The second operation will *always* override the behaviour of the first operation, and hence the first operation only affects the model's clock and not its accuracy. Other operators could be used in place of INPUT-LEFT to take up a similar amount of time, e.g. INPUT-RIGHT, but these would, superficially, look like different models. However, by replacing each operator with a specific WAIT operator we get the same timing and performance behaviour but with a clearer model. i.e. the previous example is replaced with:

```
(PROG2 (WAIT-INPUT) (INPUT-RIGHT))
```

This step has two advantages:

1. The programs of the candidate models are made clearer, with all time-only operations written as WAIT- operators. This improves the comprehensibility of the final model.

2. Behaviourally similar models have syntactically similar control programs. This means more redundant models can be removed from the candidate models.

Pair-wise similarity

Clustering and visualisation can be helpful to understand the models' programs as a group. We introduce a pair-wise similarity measure between programs to make this possible. Each program is separated into a set of node+child-labels segments. For example, the following program is converted into eight segments of two parts and six individual node names:

```
(if (access-1)
    (prog2 (input-right) (input-left))
    (input-target))

parts: (if access-1 prog2 input-target)
       (prog2 input-right input-left)
names: if access-1 prog2 input-target
       input-right input-left
```

The pair-wise similarity (Jaccard Index) divides the number of common segments in the two programs (the set intersection) by the total number of segments (the set union).

Pseudo-code

As shown in the preceding examples, individual models are represented internally as abstract-syntax trees: we can rewrite each model in a more readable pseudo-code. Although not fully automated, this step also combines consecutive WAIT operators, further simplifying the models. An example is shown in Figure 4.

Simulation Experiments

Table 2 shows a typical set of results, where we have varied the hyperparameters *a*, *b*, *c* and *RT*, with each run using a population of 500 individuals and 2000 generations. Recorded are the generation and performance measures for the best models found in each run. Most of the runs produced "good" models, with excellent fits to both accuracy and response time. However, due to the stochastic nature of the search algorithm, the last two runs failed to converge: over 5 repeats of the 6 shown sets of parameters, 5 runs failed to converge to a model with good accuracy, and a further 9 runs failed to converge to a good model of response time.

Phases in evolution

Table 3 gives summary statistics on which generation each phase was reached. In some cases phases 2 and 3 were reached very quickly, in less than 100 generations.

By analysing results against generation, we can investigate how the phases affect or reflect changes in the fitness function. Figure 2 shows overall fitness, f_a, f_t and f_s against generation number for the best model in each generation, for the first 100 generations.

Phase 1 of evolution lasts only up to generation 10, where the accuracy is optimised (the red line). As the accuracy improves, it improves the overall fitness (the green line) beyond the threshold of 0.1, and phase 2 begins.

Table 2: Table of results from ‘phased’ evolution simulation (PS = program size parameter).

a	b	c	PS	Generation	Fitness (f)	Accuracy	Response Time	Program Size
0.80	0.1	0.10	100	202	0.040	1.00	775.0	17
0.85	0.1	0.05	100	376	0.040	1.00	770.0	26
0.89	0.1	0.01	100	491	0.040	1.00	830.0	18
0.80	0.1	0.10	500	176	0.040	1.00	760.0	17
0.85	0.1	0.05	500	78	0.140	0.92	8695.0	53
0.89	0.1	0.01	500	56	0.140	1.00	8595.0	72

Table 3: Generation when phase reached (out of 30 runs).

Phase	Frequency	Minimum	Maximum	Average
2	25	6	271	79.28
3	16	55	1529	294.56

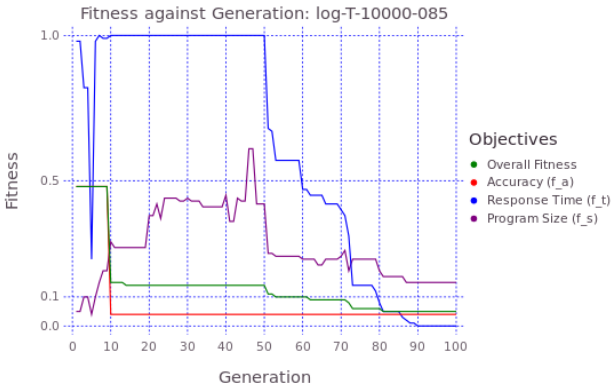

Figure 2: Progress of fitness against generation for the best model. Only the first 100 generations out of 2000 are shown.

Phase 2 lasts from generation 10 to 60, and optimises both accuracy and response time (the blue line). Around generation 50 the response-time accuracy starts to improve, as does the overall fitness. As the threshold of 0.1 is crossed by the best model, phase 3 begins. Notice how the program size appears to grow from generations 20 to 50 before the response time can begin to improve. Due to the phased introduction of the components, this increase in program size does not affect the fitness.

Phase 3 lasts from generation 60 to the end. As is evident in Figure 2, there are still some gains to be made in the response-time, which falls to an almost negligible error by generation 90, and, en passant, halves the overall fitness. The main change from this point is a steady reduction in program size: when phase 3 starts, at generation 60, the best model has 46 nodes, whereas by generation 2000 the best model only has 24 nodes, almost halving its complexity.

Effects of post-processing

Combining the candidate models from each of the six runs means the GP system produces 1164 distinct models with a good fitness value (less than 0.1). This set of models is too large to analyse and understand. In particular, the programs are obscured with bloat (only 40% of the population has less than 10% dead code) and the intention of different parts of the solution (to solve the accuracy or the reaction-time) is hidden.

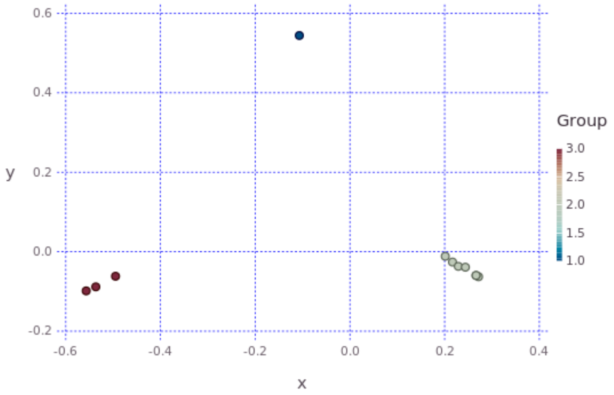

Figure 3: Visualisation of model diversity: Model distance is inverse of similarity.

Our two post-processing techniques reduce this number dramatically: removing the dead-code leaves 248 distinct models, and further removing the time-only operators reduces these to 11 distinct models.

Visualisation of models

Figure 3 depicts model diversity in a graphical form, using multi-dimensional scaling to convert pair-wise similarity into cartesian coordinates. What is most striking about this image is that the models have split into three distinct groups. The top model is an outlier, there are three models in the left-hand group, and the remaining seven models are in the right-hand group. Figure 4 shows an example from the left-hand group.

Analysing the pseudo-code of these models helps to understand the two types of solution. One (shown in Figure 4) uses a fixed delay between reading the target and the input: initially the model reads the target, then places this into STM. The model then uses a loop to wait the required time before it can see the input, followed by some processing to set up the appropriate response. The second type uses a more general perceptual loop, which tries to first read the target and then the input stimulus in turn. Because of how the environment

```
if target is visible:
  set model 'current' to target
wait for 140ms
push model 'current' onto top of STM
loop 3 times:
  loop 5 times:
    if stimuli are visible:
      set model 'current' to left input
if stimuli are visible:
  set model 'response' to "R"
push model 'current' onto top of STM
if first item in STM equals second item:
  set model 'current' to 1
else:
  set model 'current' to 0
if model 'current' is 1:
  if stimuli are visible:
    set model 'response' to "L"
else:
  wait for 70ms
wait for 70ms
```

Figure 4: Example Program: Pseudo-Code

timings work, the input stimuli will only be available in a later loop of the program and so the model will arrange the target and input in its STM as required to complete the task.

The remaining models fit these patterns, mostly with negligible differences in the ordering of operations and whether the model looks at the left or right stimulus. The outlier model is a variation on those of the second kind, but uses one outer loop repeated multiple times, rather than having a long delay within the outer loop, as in the second kind of model.

A concern when confronted with these multiple automatically-generated models is whether they are explainable or qualitatively match human behaviour. This is a topic we intend to address with improved heuristics and constraints in the GP system. However, we do not see the system as standing in isolation, but as a tool to aid the cognitive scientist. The system generates a range of *candidate* models, and the cognitive scientist using the system has the responsibility to select from or modify the generated models to create a final model and/or theory.

Discussion

A weakness of our approach is that the empirical data are the result of averaging across several individuals (e.g. Gobet, 2017; Gobet & Ritter, 2000; Siegler, 1987): one model represents that average individual. One way to simulate group behaviour is to modify GP to manage several programs instead of just one; each program would represent a single person, and the average performance of these programs would be compared to the given average.

However, more recently, psychologists have begun to publish more of their empirical data, including the performance of individual subjects. The analysis process developed in this study can use runs capturing not just one but multiple subjects, and combine the candidate solutions to see how similar or different the behaviour of different individuals is. In particular, as of now, the 'preferred' type of data are choice and reaction times, which are extremely popular outputs in fields such as decision making or psychophysics, and will be areas where the approach in this paper should be beneficial.

Further areas for future work include a co-evolution approach, to optimise the operator time parameters, and domain-specific heuristics for the GP algorithm.

Acknowledgements

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. ERC-ADG-835002).

References

Anderson, J. R., & Lebière, C. (Eds.). (1998). *The atomic components of thought*. Mahwah, NJ: Lawrence Erlbaum.

Chao, L., Haxby, J., & Martin, A. (1999). Attribute-based neural substrates in temporal cortex for perceiving and knowing about objects. *Nature Neuroscience*, *2*, 913–20.

Frias-Martinez, E., & Gobet, F. (2007). Automatic generation of cognitive theories using genetic programming. *Minds and Machines*, *17*, 287–309.

Gobet, F. (2017). Allen Newell's program of research: The video game test. *Topics in Cognitive Science*, *9*, 522–532.

Gobet, F., & Ritter, F. E. (2000). Individual data analysis and Unified Theories of Cognition: A methodological proposal. In N. Taatgen & J. Aasman (Eds.), *Proceedings of the Third International Conference on Cognitive Modelling* (pp. 150–57). Veenendaal, The Netherlands: Universal Press.

Gobet, F., & Simon, H. A. (2000). Five seconds or sixty? Presentation time in expert memory. *Cognitive Science*, *24*, 651–82.

Gulwani, S. (2010). Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on principles and practice of declarative programming* (p. 13-24).

Koza, J. R. (1992). *Genetic programming: On the programming of computers by means of natural selection*. Cambridge, MA: MIT Press.

Lane, P. C. R., Sozou, P. D., Gobet, F., & Addis, M. (2016). Analysing psychological data by evolving computational models. In A. Wilhelm & H. Kestler (Eds.), *Analysis of large and complex data. Studies in classification, data analysis, and knowledge organization* (pp. 587–97). Springer, Cham. doi: 10.1007/978-3-319-25226-1_50

Langdon, W. B., & Poli, R. (1998). Genetic programming bloat with dynamic fitness. In W. Banzhaf, R. Poli, M. Schoenauer, & T. C. Fogarty (Eds.), *European conference on genetic programming: Eurogp* (pp. 97–112).

Langley, P., Simon, H. A., Bradshaw, G., & Zytkow, J. (1987). *Scientific discovery: Computational explorations of the creative processes*. Cambridge, MA: MIT Press.

Peterson, J. C., Bourgin, D. D., Agrawal, M., Reichman, D., & Griffiths, T. L. (2021). Using large-scale experiments and machine learning to discover theories of human decision-making. *Science*, *372*, 1209 - 1214.

Rumelhart, D. E., & McClelland, J. L. (Eds.). (1986). *Parallel distributed processing* (Vol. 1 and 2). Cambridge, MA: MIT Press.

Schmidt, M., & Lipson, H. (2009). Distilling free-form natural laws from experimental data. *Science*, *324*.

Siegler, R. S. (1987). The perils of averaging data over strategies: An example from children's addition. *Journal of Experimental Psychology: General*, 250–264.

Simon, H. A. (1981). Information-processing models of cognition. *Annual Review of Psychology*, *30*, 363-96.