

Learning Programs for Modeling Strategy Differences in Visuospatial Reasoning

By

James Ainooson

Dissertation

Submitted to the Faculty of the
Graduate School of Vanderbilt University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

December 16, 2023

Nashville, Tennessee

Approved:

Prof. Maithilee Kunda, Ph.D., Chair

Prof. Armando Solar-Lezama, Ph.D.

Prof. Bradley Malin, Ph.D.

Prof. Gautam Biswas, Ph.D.

Prof. Tyler Derr, Ph.D.

Copyright © 2023 James Ainooson
All Rights Reserved

To Adwoa Bruwaa, Akyedze Abaka, and Adzepa Amoaba. Thanks for the constant love, patience, and support.

ACKNOWLEDGMENTS

First, thanks to my advisor, Dr. Maithilee Kunda, for providing me with this wonderful opportunity. I am very grateful for her support, advocacy, inspiration, mentorship, encouragement, and guidance. She always helped me see the big picture when things were bleak, and she provided an intellectually welcoming and stimulating environment that encouraged me to push myself.

My gratitude also goes to the members of my committee: Dr. Armando Solar-Lezama, Dr. Bradley Malin, Dr. Gautam Biswas, and Dr. Tyler Derr. I am very grateful for contributions to this work. I appreciate the wonderful feedback and comments they provided, the thought-provoking questions they posed, and the enjoyable defence experience they provided.

I would also like to thank members of the Laboratory for Artificial Intelligence and Visual Analogical Systems (AIVAS). To Joel Michelson, Deepayan Sanyal, Yuan Yang, and Tengyu Ma, for their contributions and help with my work. To Dr. Effat Farhana and Dr. Caoimhe Harrington Stack for their fantastic work on the WOMBAT system and for coordinating the online pilot study. To Dr. Fernanda Elliot and Dr. Xiaohan Wang for their help in getting me settled in the lab, and to all members, past and present, who played roles in the work I did. Especially, the hard-working team of undergraduate researchers who contributed immensely to data annotations.

I am also grateful to members of the Neurodiversity Inspired Science and Engineering (NISE) fellowship. To Dr. Julie Vernon, Dr. Keivan Stassun, and all the other NISE fellows.

I also want to show gratitude to my family. To my wife Janice Brako-Boateng, whom I can't thank enough for the sacrifices she made and continues to make. She put her dreams and accomplishments aside, and left her comfort behind to join me on this uncertain journey. To my kids Ekow and Efe Ainooson, who fill me with so much pride, and show me so much love and patience. To my Parents Kobina and Margaret Ainooson, for the encouragement and prayers, and most importantly, for laying a foundation that has brought me this far. To my father inlaw Oheneba Boateng, for all the encouragement and guidance. To my aunties Ruth Ainooson and Effie Bondzie, and uncles Eben Ainooson and Nana Appah Dankyi thanks for the immense support and all the help you provided in helping me get settled in the United States. To my lovely sisters Rhoda, Magdalene, Ewurafua, and Delali, your constant check-ins and encouragement kept me going.

Finally, this research was funded in part by NSF awards 1029679 and 1730044.

TABLE OF CONTENTS

	Page
LIST OF TABLES	x
LIST OF FIGURES	xi
List of Algorithms	xvi
1 Introduction	1
1.1 Defining Visuospatial Reasoning	3
1.2 Reasoning with Computer Models	4
1.2.1 Representations and Computational Models	5
1.2.2 Representations for Visual Reasoning	5
1.3 Computational Cognitive Models and Strategies	6
1.4 Motivation for this Dissertation	7
1.5 Overview of this Dissertation	8
1.5.1 Problem Statement	9
1.5.2 Research Questions	9
1.5.3 Summary of Work Done	10
1.5.4 Organization of this Document	12
2 Background	13
2.1 Computation, Cognition and Visuospatial Reasoning	14
2.1.1 Theories of Representation	14
2.2 Visuospatial Reasoning in Humans	16
2.3 Visuospatial Reasoning in AI	18
2.3.1 A Brief Survey of Visual Imagery Based AI Systems	19
2.4 A Brief Survey of Program Synthesis	22
2.4.1 Grammar Enumeration	23
2.4.2 Deductive Search	24
2.4.3 Constraint Solving	25
2.5 An Overview of Tasks Studied in this Dissertation	25
2.5.1 The VZ-2 Punched-hole Paper Folding Task	26
2.5.1.1 Computational Modelling of Paper Folding Task	28
2.5.2 The Block Design Test	29
2.5.2.1 Computational Modelling of the Block Design Task	30
2.5.3 Leiter International Performance Scale-Revised	30
2.5.3.1 Structure of the Leiter-R	32
2.5.4 Abstract Reasoning Corpus	32
2.5.4.1 Structure of the Abstract Reasoning Corpus	33
2.5.4.2 Prior Knowledge for ARC Tasks	35

2.5.4.3	Related Work	35
3	Hand Coding Strategies for Visual Reasoning Tasks	37
3.1	A Computational Model for the VZ-2 Paper Folding Task	38
3.1.1	Inputs and Pre-processing	38
3.1.2	Core Operations	40
3.1.2.1	The Initialize Operation	40
3.1.2.2	The Fold Operation	40
3.1.2.3	The Punch Operation	41
3.1.2.4	The Unfold Operation	41
3.1.3	Walk-through of a Sample Problem	42
3.1.4	Results and Discussion	43
3.2	Building Models for Leiter-R Tasks	46
3.2.1	Implementation Details	46
3.2.1.1	Similarity	47
3.2.1.2	Containment	47
3.2.1.3	Rotation and Scaling	48
3.2.2	Inputs and Pre-processing	48
3.2.3	Results of Sufficiency Experiments	48
3.2.4	Discussion of Sufficiency Experiments	49
3.3	Visuospatial Reasoning Environment for Experimentation	51
3.3.1	The Environment	51
3.3.2	Objects	52
3.3.3	Affordances	53
3.3.4	Agents	54
3.3.4.1	Operations and Affordances	54
3.3.4.2	Memory	54
3.3.4.3	Reasoning Rules	55
3.3.4.4	Vision Attention and Gaze	55
3.4	Exploring Strategy Differences on the Leiter-R with VREE	56
3.4.1	Selected Sub-tasks and their Specific Strategies	56
3.4.1.1	The Associated Pairs Subtest	56
3.4.1.2	The Figure Rotation Subtest	57
3.4.1.3	The Matching Subtest	58
3.4.1.4	The Repeated Pattern Subtest	59
3.4.1.5	The Sequential Order Subtest	59
3.4.1.6	The Visual Coding Subtest	60
3.4.2	Setting up the Environment	61
3.4.3	Walk-through of Agent’s Reasoning Strategies	63
3.4.4	Results and Discussion	64
3.5	Exploring Strategy Differences on the Block Design Task with VREE	64
3.5.1	Reasoning Model for Solving the BDT	66
3.5.2	Walk-through of the Block Design Task Strategy	68
3.5.3	Results	69
3.5.4	Discussion	70
4	Synthesizing Strategies for the Abstract Reasoning Corpus	73

4.1	The Abstract Reasoning Task	73
4.2	Visual Imagery Reasoning Language	74
4.2.1	Generating ARC Strategies with VIMRL	74
4.2.2	Execution of VIMRL Programs	76
4.2.3	Details on VIMRL Instruction Types	77
4.2.4	A Walk-through of High Level Function Execution	79
4.3	Operations for Reasoning about the ARC	81
4.4	Overview of Selected Operations	82
4.4.1	The <code>self_scale</code> Low Level Operation	82
4.4.2	The <code>find_enclosed_patches</code> Low Level Operation	83
4.4.3	The <code>solve</code> High Level Operation	83
4.4.4	Title <code>connect_pixels</code> High Level Operation	85
4.5	Synthesizing VIMRL Strategies for ARC Tasks	86
4.5.1	Tree Traversal with Classic Search Algorithms	87
4.5.2	General Structure of ARC Search	88
4.5.3	Estimating Probabilities for Stochastic Successor Generation	89
4.5.3.1	Details on Sampling Instructions	90
4.5.3.2	Details on Sampling Arguments for Instructions	90
4.5.4	Ground Truth Programs for Stochastic Search Methods	92
4.5.5	Applying Monte-Carlo Search Techniques	92
4.5.6	Search Space Pruning and Optimization	95
4.5.6.1	Depth Limiting	95
4.5.6.2	Reference Gaps	96
4.5.6.3	Logically Equivalent Programs	96
4.5.6.4	Early Program Evaluation	96
4.6	Implementation Details of VIMRL ARC Experiments	97
4.7	Experiments	99
4.7.1	Experiment I: Tree Traversal	99
4.7.1.1	Results	100
4.7.1.2	Discussion	100
4.7.2	Experiment II: Pruning and Selection	102
4.7.3	Results	103
4.7.4	Discussion	103
4.8	2022 ARCATHON Run	105
4.9	Conclusion	105
5	Exploring Strategy Differences on the Block Design Task	107
5.1	Extending VIMRL with Control Structures	107
5.2	An Overview of State Machines	107
5.2.1	Formal Representation of State Machines	109
5.2.2	Prior-work in Synthesizing State Machines	109
5.3	Modelling Reasoning with State Machines	110
5.4	Integrating VIMRL and State Machines	111
5.5	Synthesizing State Machines	113
5.5.1	Exploring the Space of State Machines	113
5.5.2	Unrestricted exploration	114
5.5.3	Heuristics for Improving Search	115

5.5.3.1	Single Blank State	116
5.5.3.2	Reversed Machine	116
5.5.3.3	Unreachable States	116
5.5.3.4	One-way True Conditions	117
5.5.3.5	Executable Machine	117
5.5.3.6	Reference Gaps	118
5.5.3.7	Logically Equivalent Machines	118
5.5.4	Stochastic Successor Generation	118
5.5.4.1	Sampling Successors that Added States	119
5.5.4.2	Sampling Successors that Added Operations	119
5.5.4.3	Sampling Arguments for Operations	120
5.5.4.4	Sampling Transitions between States	120
5.6	Experiments on Synthesizing Strategies for the Block Design Task	121
5.6.1	Simplifying the Block Design Task	121
5.6.2	The Single Block Task	122
5.6.3	The Single Block Single Axis Experiment	123
5.6.4	The Single Block, Multiple Axes Experiment	123
5.7	Results for Strategy Search Experiments	124
5.8	Discussions	124
5.8.1	Exploring Single Block Search Tree	125
5.8.2	Single Block, Single Axis	127
5.8.3	Single Block Multiple Axis	128
5.9	Conclusion	128
6	Analysing Human Strategies on the Block Design Task	129
6.1	The Block Design Task	129
6.2	Automatically Scoring the Block Design Test	130
6.2.1	Overview of the Three Iterations	131
6.2.2	Setup for Testing	132
6.2.3	Measuring Block Placements	133
6.2.3.1	Isolating Individual Blocks	134
6.2.3.2	Detecting the Face of the Blocks	135
6.2.3.3	Filtering Out Hands	136
6.2.4	Measuring Gaze and Attention	137
6.2.4.1	A Brief Background on Gaze Tracking	137
6.2.4.2	Our Approaches to Gaze Tracking	138
6.3	Measuring Gaze Through Corneal Imaging	139
6.3.1	Gaze Estimation with Corneal Images	140
6.3.2	Collecting and Analysing Corneal Imaging Data	141
6.3.3	Experiment 1: Evaluating Image Fidelity by Human Annotation	142
6.3.3.1	Results and Discussion of Experiment 1	143
6.3.4	Experiment 2: Automating Corneal Image Analysis	144
6.3.4.1	Results of Experiment 2	145
6.3.4.2	Discussion of Experiment 2	146
6.4	Analysing Block Placement Sequences	147
6.5	Combining Gaze and Block Placement Data	150
6.6	Introducing Novelty with Blue Blocks	151

6.7	Case Study: The Block Design Task as Featured on CBS' 60 Minutes	153
6.7.1	Results and Analyses: Anderson Vs. Dan	153
6.8	Taking the Block Design Test Online	156
6.8.1	First Participant Study	157
6.8.2	Assessing Strategy Differences on The study	158
6.8.3	Future Work on WOMBAT	161
7	Conclusions and Future Work	162
7.1	On Hand Coded Strategies	162
7.2	On Machine Generated Strategies	163
7.3	On Investigating Human Strategies	164
7.4	Summary of Contributions	165
7.5	Future Work	166
7.5.1	The Abstract Reasoning Corpus and Search	166
7.5.2	Synthesizing and Understanding Strategies	167
7.6	Final Thoughts	167
	Bibliography	168
A	List of VIMR operations for ARC	179
B	State transitions of the two different Blocks for the BDT	182

LIST OF TABLES

Table	Page
2.1 All 20 items in the Leiter-R, listed with brief descriptions of what the test requires.	31
3.1 All 10 affordances used in the block design environment for VREE	66
4.1 Grammar for the Visual Imagery Reasoning Language-I (VIMRL). These also double as production rules for generating code during program synthesis.	75
4.2 VIMRL program listings for possible solutions to the ARC tasks displayed in Figure 4.1	78
4.3 A list of selected ARC operations and their assumed core knowledge priors.	81
4.4 Listings for VIMRL programs that solve tasks in Figure 4.3	83
4.5 Listing for a possible VIMRL solution to the task described in Figure 4.4 .	84
4.6 Listing for a possible VIMRL solution to the task described in Figure 4.4 .	86
4.7 Best results observed for the different forms of search traversal. For stochastic methods, these were results taken over several experimental runs.	100
4.8 The sets of three programs generated by both the Unique picker and the Best 3 picker.	105
4.9 Final leader board for the 2022 International ARCATHON Competition showing the best entries. The entry from this dissertation is shown in boldface.	106
5.1 Grammar for language used in specifying state operations.	112
5.2 A sampling of state machines obtained for all experiments.	125
6.1 Accuracy results for different approaches.	137
6.2 A brief comparison chart of various forms of gaze tracking.	139
6.3 Cohen’s Kappa scores for the inter-rater reliability among the four raters on our research team.	143
6.4 Average accuracy and precision scores for the neural network blink classifier	146
6.5 Average accuracy and precision scores for the neural network gaze target classifier	147
6.6 Average accuracy and precision scores for the kNN gaze target classifier .	147
6.7 Average accuracy and precision scores for the MLP gaze target classifier .	148

LIST OF FIGURES

Figure		Page
1.1	A sample visual reasoning puzzle. This puzzle is variation of the Visual Coding task from the Leiter International Performance Scale-Revised (Leiter-R) (Roid & Miller, 1997). To protect the secrecy of the Leiter-R, this sample is not a real Leiter-R problem.	1
1.2	Rules for the visual reasoning puzzle found in Figure 1.1.	2
1.3	Two ways of representing the concept of an apple: on the left there is a proposition representation spelling out the English word “apple”, on the right an illustration of an apple.	5
2.1	A tally diagram that represents a number. Even without knowing the exact value of the number, it is easy to know what the remainder is when it is divided by 5.	15
2.2	Sample stimuli from Shepard and Metzler (1971)’s mental rotation experiment. For each of these test pairs, the participant was supposed to determine if the image on the right is a three-dimensional rotation of the one on the left.	17
2.3	Figures from WHISPER (Funt, 1980). (a) An input image for whisper in its unstable state. (b) The final state of the blocks after the naive physics simulation.	19
2.4	Results from Gardin and Meltzer (1989) showing how particles were used to simulate flexible strings, bars of different flexibility, and liquids.	20
2.5	A sample item from the VZ-2 paper folding task. The two images to the left of the vertical bar depict the sequence of folds and the eventual punch. The other four images labelled A through E are possible answers from which the test taker is to make a choice.	28
2.6	A closeup shot of a person taking the block design task	29
2.7	Sample tasks from the ARC’s training set. (a) A task that requires solvers to isolate the object depicted in the grid. (b) A task that requires solvers to complete the patten by replacing the pixels in cyan coloured patch. (c) A task that requires solvers to split the input into four same sized quadrants and overlay the output into a single image output.	33
2.8	A plot of task image sizes from the ARC dataset. The circle’s centre represents the width and height of a given image size and the size signifies the number images with that size.	34
3.1	Image inputs at different stages of processing for the paper folding task. (a) The redrawn inputs as they were presented to the model. The top row shows the sequence of input folds, and the bottom row shows the possible answers. (b) The results of a thresholding operation on the input images, a final step before they were fed to the model.	39

3.2	The sequence of operations that lead to a fold detection and the ultimate application of a fold. (a) A bounded version of the input image is inverted and its intersection with each of the images on the stack is computed to generate the corresponding fold flaps. (b) The same input image is further intersected with the images on the stack and the results are morphologically dilated by a single pixel. (c) The dilated image from <i>(b)</i> is intersected with the fold flap to determine the fold line. (d) The image from the fold flap is mirrored along the fold line to simulate a fold.	43
3.3	(a) The sequence of inputs and the effect they have on the image stack three-dimensional representation of the folded paper. (b) From top to bottom, this image shows how the images on the stack are combined to represent unfolding the paper.	44
3.4	The solution to an arbitrary punched-hole paper task, which simulates a paper snowflake. The top row of images show the input to the model, with the blue sections representing solid sections of the paper. The bottom row shows the unfolding sequence as generated by the model.	45
3.5	Results of the Leiter-R sufficiency experiment.	49
3.6	The sequence of steps an image goes through before every comparison is made with either metric.	50
3.7	A diagram of the VREE system showing how the different components existed within the environment, and how they interacted with each other.	51
3.8	A sample item from the Associated Pairs task from the Leiter-R (Roid & Miller, 1997). In this illustration, the section labelled A will be briefly exposed to the subject, then the section labelled B will be exposed while the A section will be hidden. To protect the secrecy of the Leiter-R, this sample is not a real Leiter-R problem.	57
3.9	A sample item from the figure rotation task of the Leiter-R (Roid & Miller, 1997). In this instance, the subject will be required to select one of the four slots in which the card at the bottom best fits. To protect the secrecy of the Leiter-R, this sample is not a real Leiter-R problem.	58
3.10	An instance of the matching task from the Leiter-R (Roid & Miller, 1997), with 4 slots and 4 answer cards. To protect the secrecy of the Leiter-R, this sample is not a real Leiter-R problem.	59
3.11	An instance of the repeated pattern task from the Leiter-R (Roid & Miller, 1997), which requires the subject to fill in the missing parts of the given sequence. To protect the secrecy of the Leiter-R, this sample is not a real Leiter-R problem.	60
3.12	An instance of the sequential order task from the Leiter-R (Roid & Miller, 1997), in which the subject must complete the sequence by placing easel cards. To protect the secrecy of the Leiter-R, this sample is not a real Leiter-R problem.	61
3.13	An instance of the visual coding task from the Leiter-R (Roid & Miller, 1997), in which subjects are required to provide the cards that best fill the slots labelled 1 and 2. To protect the secrecy of the Leiter-R, this sample is not a real Leiter-R problem.	62
3.14	Results from running the two different strategies on the six selected Leiter-R subtests, while alternating forgetfulness.	63

3.15	The internal representation used to store block states and transitions for the block design task.	65
3.16	A flow diagram constructed around the state machine representation of the template strategy. Note the slots into which the two sub-strategies for block search and face search are executed.	67
3.17	A view of the VREE environment for a few initial steps while solving a block design item. The design to be replicated is on the left, the block bank is on the right, and the final construction is in the centre. The blue square indicates the current gaze position of the VREE agent.	69
3.18	The results of evaluating all six main strategies on eight BDT puzzles. . .	70
3.19	The results of evaluating all six main strategies on eight BDT puzzles with forgetfulness. Notice the variability in response times for the best and worst performing strategies.	70
4.1	Tasks for the programs displayed in Table 4.2.	78
4.2	A walkthrough of the execution of Program (d) from Table 4.2 on the task from Figure 4.1 (d). On the left side of the image, each box shows a time step in the execution of the program. The instruction executed is on the top of each box, and the bottom of each box shows the state of all variables available in the runtime environment. The left section shows the modified task passed to the <code>recolor_objects</code> operation. For each task item, the top section shows the original task, and the bottom section shows the list of objects passed on to the <code>recolor_objects</code> operation.	80
4.3	Two tasks from the ARC that make use of the <code>self_scale</code> function. (a): <code>007bbfb7</code> ; (b): <code>cce03e0d</code>	82
4.4	Task <code>e999362f0</code> from the Abstract Reasoning Corpus.	84
4.5	A visual demonstration of how data items from tasks are converted into the dataset for the neural network.	85
4.6	A sample task for demonstrating the <code>connect_pixels</code> operation	85
4.7	A flow diagram of how instructions and their arguments were sampled in a multi-tier sequence.	92
4.8	Characteristics of the ground truth dataset. The chart on the left shows the distribution of operation occurrences across all programs in the ground truth dataset. The upper-middle chart shows the distribution of the number of ground truth programs per task. The upper-left chart shows the distribution of program sizes across the entire ground-truth dataset. And the lower left chart shows the distribution of task sizes in the full dataset versus those that have been solved in the ground truth dataset.	93
4.9	A task and its MCTS generated solution. Instructions shown in bold were actually responsible for solving the problem. The others greyed out instructions had no effect on the final solution.	101
4.10	Function distribution	102
4.11	103
4.12	A sample task from the training section of the ARC	104
5.1	A state machine model for a simplified trunk door with two states and two inputs. The circles represent the states and the arrows represent the transitions that take place on specific actions.	108

5.2	An extended version of the state machine described in Figure 5.1. This extension adds an extra state to represent a locked door and an extra input for locking and unlocking the door.	108
5.3	A depiction of state machines that has a single state (left) and a state machine that has multiple empty states (right).	116
5.4	A reversed machine in which a state transitions into the initial state. . . .	116
5.5	A state machine with an unreachable cluster (existing in the dotted line). . . .	117
5.6	A partial rendering of the expanded search tree for the single block task. Each node in the search tree represents a machine that was generated. The colour of the edges, as well as the letter on the edge in the search tree, signify the type of action that was taken to generate the new machine. Orange or the character 'a' signifies the addition of an assignment, red or the character 'c' signifies the addition of a call to an operation, blue or the character 't' signifies the addition of a transition, and green or the character 's' signifies the addition of a state.	126
6.1	A setup of the block design task, showing the differences between sensor setup in the different iterations, as well as, how attention zones were defined. (a) Shows an overhead view of the setup in our first iteration, as captured by the overhead camera, and (b) Shows the setup for our second iteration.	133
6.2	Feature detector for quadrant based block face detection. Samples taken from the quadrants labelled Q1 through Q2 are each guaranteed to contain a uniform colour. By combining these in multiple ways, the design on the face of the blocks can be found.	136
6.3	A visualization of the two main classes of gaze tracking technology widely available today (wearable on the left and remote in the middle), and the corneal imaging technique (right).	138
6.4	A reflection of the world as seen on the cornea of a person solving a block design task.	139
6.5	Fitting an eye model to the limbus for obtaining the exact gaze point with respect to the corneal image.	141
6.6	A visual comparison of annotations produced by three different raters on the same block design puzzle.	144
6.7	A visual comparison between annotations generated by human raters and the neural network algorithm for the same puzzles.	146
6.8	(a) Regions of the block construction area, labeled as vertices, sides, and the inner area. (b-e) Examples of spatial block placement strategies, with arrows representing the order of blocks placed: (b) row-by-row, (c) subsection, (d) perimeter-complete, and (e) vertices-first.	148
6.9	A visualization of sequences in which subjects from our study responded to puzzles from the block design task, and the predictions made by our system on strategy. For the green patches, earlier blocks are represented by lighter greens and later blocks are represented by darker greens. The following abbreviations are used on the charts: (r-r) for row by row, (c-c) for column by column, (s-s) for subsections, (p-c) for perimeter complete, and (v-f) for vertices first.	149
6.10	A plot of reaction time against the number of errors made.	151

6.11	A synchronized plot showing gaze data, along with block movement and placement, on a single plot.	152
6.12	Net diagrams of the blocks showing the red blocks (left) and the blue blocks (right).	153
6.13	A chart of Anderson's and Dan's the response times on all the blue and red block tasks.	154
6.14	A visualization of the gaze and attention patterns exhibited by Dan and Anderson as they reasoned through a single instance of the block design task.	155
6.15	Block placement charts that show how Anderson (a) and Dan (b) placed their blocks respectively, while solving the 7th red puzzle from Figure 6.13	156
6.16	A screenshot of the WOMBAT system showing a block design task session in progress. In this view, the user is manipulating a block in the construction area with the pattern and block bank obscured.	157
6.17	A box plot showing the response times of the autistic and non-autistic participants.	159
6.18	Attention charts for all participants on the 6th task from the BDT. The graph on the left shows the attention of all Non-ASD participants, and the graph on the right shows the attention of all ASD participants.	160
B.1	All the states of the RED block design Chart	182
B.2	All the states of the BLUE block design Chart	183

List of Algorithms

- 3.1 *TimeKeeper* routine from the Environment that coordinates the activities of agents, affordances and objects 52
- 3.2 Imagery search algorithm for finding the matching face to the cell in memory. 68

CHAPTER 1

Introduction

As we reason about the world around us, we have the ability to capture what we see, recall what we have seen, synthesize new things to “see”, and most importantly, we are able to mentally manipulate these various forms of “seen” images to solve some of the problems we face (Tversky, 2005). This, in simple terms, is what visuospatial reasoning through mental imagery allows us to do. Reasoning with visual imagery, seemingly makes it possible for an engineer to correctly visualize the intricate functioning of a gearbox mechanism before even building one (Hegarty, 2004), or a person to picture a *purple elephant flying with white fluffy wings across an orange sky*—a sight almost impossible to observe in real life.

Figure 1.1 presents a visuospatial reasoning task. In this task, your goal is to find one of the images from those labelled A through F that fit in the slots labelled 1 and 2. Without any description of what the actual task requirements are, what choices will you make? And after you make your choices, can you consider what informed the strategy in making your decision?

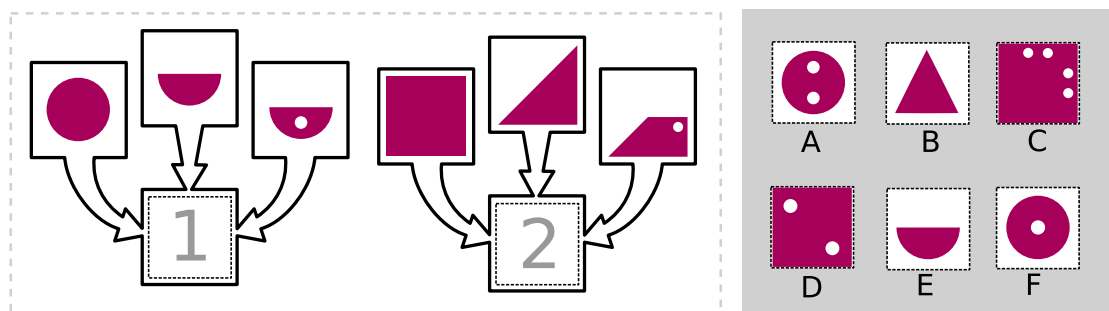


Figure 1.1: A sample visual reasoning puzzle. This puzzle is variation of the Visual Coding task from the Leiter International Performance Scale-Revised (Leiter-R) (Roid & Miller, 1997). To protect the secrecy of the Leiter-R, this sample is not a real Leiter-R problem.

Now, provided you managed to find a solution for the problem, do you think your choices will be changed after you receive the information that the images in Figure 1.2 are a set of rules for solving the problem in Figure 1.1? If both of your responses are the same,

and you factored the rules in Figure 1.2 on your second attempt, then you most likely relied on a different strategy in each of your attempts.

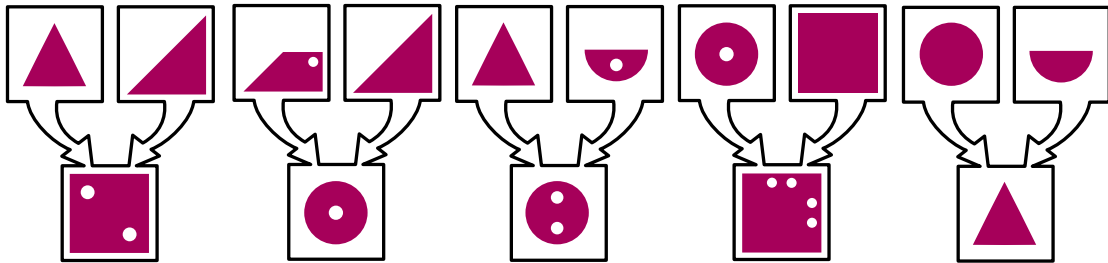


Figure 1.2: Rules for the visual reasoning puzzle found in Figure 1.1.

In case you could not find a convincing solution for the task, here are the two possible strategies you could have used:

1. For the problem as presented in Figure 1.1, you can consider the three images that point to an empty slot to be depicting the sequence of folding and ultimately punching a hole in a shaped piece of paper. The best choice from the possible answers that fits a slot will be the one that best shows the pattern made when this punched paper is unfolded.
2. You can consider the series of images from Figure 1.2 as rules by which pairs of images can be combined into new ones. By recursively reducing pairs of images in each problem (from left to right) until a single image remains, a valid solution for a problem can be obtained.

The example above was specifically designed to elicit multiple strategies. However, although this multi-strategy problem was intentional by design, it reflects the reality of situations where different people may exhibit completely different strategies on the same problem based on how they perceive it. Factors that influence such strategy differences are interesting to study, and it is even more interesting when you consider how such an ability to dynamically form strategies can be implemented in artificial agents.

For my dissertation work, I focused on how cognitive differences in intelligent agents affect strategy choices when agents are faced with visuospatial reasoning tasks. I investigated

this in two ways:

1. Through computational models of reasoning that were designed to form strategies dynamically using program synthesis techniques.
2. By working on tools for measuring and analysing factors that may influence strategy choices humans make.

1.1 Defining Visuospatial Reasoning

Visuospatial reasoning in several ways involves reasoning about the world around us. We are surrounded by different visuospatial properties that help us function in our daily lives. From the shape and texture of objects, to colours, distances and reference points; a significant part of living involves reasoning about visuospatial entities. The visuospatial entities we reason about could be real or imagined, and they could also be static or dynamic.

Visuospatial reasoning can be considered as critical for several human endeavours. From computer programming (Petre & Blackwell, 1999), mathematics (Giaquinto, 2007), physics (West, 2009) and medicine (Birchall, 2015; Lufler et al., 2012; Wanzel et al., 2002), to visual arts (Goldsmith et al., 2016) and language comprehension (Marschark et al., 2015), all sorts of activities people perform are known to directly benefit from visuospatial reasoning.

It is also especially known that success in Science, Technology, Engineering and Mathematics (STEM) fields (both in practice and education) is also tied to visuospatial reasoning abilities (National Research Council, 2009; Wai et al., 2009). Several prominent scientists like Albert Einstein, Nicola Tesla, and Temple Grandin have been reported to rely on exceptional visuospatial abilities in developing some of their inventions and theories (Grandin, 1995; West, 2009).

Like other skills humans possess, visuospatial reasoning in people can be improved through training (Uttal et al., 2013). With better understanding of how the brain works during visuospatial reasoning, we can produce effective training and evaluation tools that take advantage of the true underlying mechanisms of the brain. But the brain is very complicated to study, and as it currently stands, we do not have a good “under-the-hood” understanding of how reasoning in the brain works. To fill this gap, some researchers have

adopted computational models as a means of studying human reasoning.

1.2 Reasoning with Computer Models

Computers have the ability to exhibit behaviours that can be considered as intelligent, and such intelligent actions in computers are performed in ways that can mostly be inspected (Newell & Simon, 1976). This openness to scrutiny makes computer modelling of intelligence a viable tool for studying human reasoning. A researcher can propose a theory on how a particular reasoning process is carried out, build a computational model that simulates this theory, and evaluate the model (Thagard, 2005). Evaluation can be carried out with experiments on specific tasks that can be performed by both human subjects and instances of the computer models. Because the computational models and human subjects can be evaluated on the same task, there is a possibility of validating model behaviours against that of humans.

This technique of using computational models to study human reasoning has been shown to be successful in some cases. There are accounts of studies in which computational models have been used in modelling situations, like the process by which children learn the rules of subtraction in mathematics (Young & O’Shea, 1981), how people learn to read (Just & Carpenter, 1987), and even the decision-making processes of combat flight pilots (R. M. Jones et al., 1999), all with some similarity to how humans perform such tasks.

Regardless of these accomplishments, computational models still have some serious limitations. Most of the time, to reduce complexity, models are built to evaluate only specific, narrow, mostly stripped down versions of tasks. Additionally, in most cases, designers have to provide a significant amount of the knowledge needed by the model to function. Without a clear boundary on what a model is intended to evaluate, it may be difficult to delineate where a designer’s knowledge ends and where the model’s starts (Chollet, 2019). Another potential setback designers may introduce to their models could be found in cases where models are designed around a set of “friendly” inputs that yield desired results, or cases where models are evaluated through experiments that cannot be repeated due to some special circumstances the designers found themselves in.

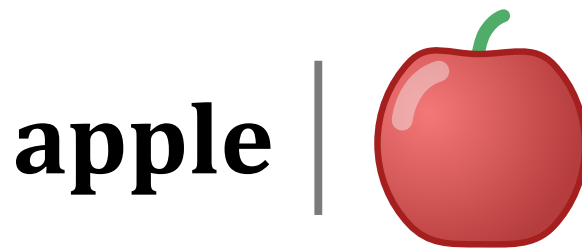


Figure 1.3: Two ways of representing the concept of an apple: on the left there is a proposition representation spelling out the English word “apple”, on the right an illustration of an apple.

1.2.1 Representations and Computational Models

In designing and implementing computational models for reasoning, one important factor to consider is the underlying knowledge representation used by the system. The choice of representation determines the kinds of knowledge a system can express, and this in turn determines the kinds of operations a system can ultimately reason with (Thagard, 2005).

Larkin and Simon (1987) illustrate this point with a real-world example that considers an architect who has to communicate the detailed design of a floor plan in a natural language. For all the floor plan’s details to be correctly covered, a large amount of text will be needed. If a drawing is used instead, a single page of lines will always suffice. Not only will a drawing be good medium for communicating said design, it would also be easier to modify, and could also be used as an actual basis for construction.

1.2.2 Representations for Visual Reasoning

Models of visuospatial reasoning, can be built on representations that fall into two main categories: propositional and iconic (Kunda, 2018). Propositional representations are symbolic, language-like, and descriptive in nature. You can consider them to be similar to reasoning with words and symbols. Conversely, iconic representations are image based and structured to look like whatever they are representing. They can be considered analogous to mental imagery in humans.

1.3 Computational Cognitive Models and Strategies

Computer systems that take inputs and merely produce meaningful outputs for tasks cannot necessarily be considered as computational models of cognition. To properly fit the description of a computational cognitive model, the system must exhibit similar problem-solving patterns to those used by humans, including exhibiting the effects that cognitive deficiencies in humans bring, such as limited memory and attention (Chown, 2014). To put it simply, a good computational cognitive model will strive to employ strategies similar to what a human would.

When humans reason about problems, a first step they are most likely to take before making an attempt at a solution will be to form a strategy. For problems involving complex sub-tasks, this strategy forming phase may be obvious, and it may even involve detailed planning and rehearsal of some sub-tasks (Land & Tatler, 2009). But for simpler, more repetitive tasks, strategies can be selected sub-consciously without much effort from a person.

Strategy choices in humans are influenced by several factors that include a person's skill level, age, cognitive abilities (like working memory capacity), and the like (Schunn & Reder, 1998). Generally, humans have the ability to remember strategy choices made in previous task instances, and as people get more experience with a task they are also able to modify and evolve their strategies. In some cases, people may even adapt earlier strategies to form newer ones when they encounter tasks that seem related (Ackerman, 1988). When considered on a broad spectrum, people have the ability to easily form strategies that end up being successful when faced with completely novel tasks (Mumford et al., 1993).

Cognitive computational models, on the other hand, may either have a fixed strategy (usually supplied by the system's designer), or they may approximate the human approach of selecting a strategy based on the task at hand. Production rule based cognitive architectures like SOAR (Rosenbloom et al., 1993) and ACT-R (Anderson, 1993), which evaluate rules through inference engines as a primary means of reasoning, tend to exhibit the later more flexible approach. In these systems, core knowledge is typically represented through a complex network of *if-then* rules, and reasoning involves reducing these rules and inputs to the model by executing them through an algorithm, such as forward chaining (Doumas

& Hummel, 2012; Young, 2001).

Another approach for generating reasoning strategies, which is worth considering, could be program synthesis. A system that generates strategies through program synthesis will always generate a new program—essentially generating a new strategy—for any particular instance of a task it faces. Program synthesis itself is a widely used technique in computer science that involves systems that automatically generate programs to meet some user defined criteria (Gulwani et al., 2017). Although it is primarily found in the programming languages community, applications of program synthesis can be found in machine learning and artificial intelligence (Gulwani et al., 2015).

Unlike production systems, synthesized programs can be used in systems built on top of a wide variety of representations. For example, instead of writing elaborate if-then rules, systems can be made aware of their task environments through input-output examples (as done with the Abstract Reasoning Corpus in Chapter 4), task demonstrations (Lázaro-Gredilla et al., 2018), sample programs, or in some extreme cases brute-force search of some representation of the problem’s task space.

1.4 Motivation for this Dissertation

Human reasoning is both complex and fluid. As exhibited in the example from Figures 1.1 and 1.2, we can form strategies for certain novel tasks, often without much effort, depending on the information available to us. And even when the strategies we form are unsuccessful, we are able to make reasonable attempts at correcting our mistakes and working towards valid solutions. Current artificial intelligent systems cannot exhibit this level of fluidity. This fact is still true when current advances in machine learning techniques (for example, the use of Large Language Models) are considered. Currently, the best performing machine learning algorithms are extremely resource intensive; they require a tremendous amount of training data, they have long training times, and they consume immense computational power, not to mention the energy required to operate these systems (Kasneci et al., 2023).

A primary motivator for me in pursuing this line of work was an interest to investigate how intelligent systems could be given the potential to exhibit this human-like level of fluidity in forming strategies. This motivation unfolds in two ways:

1. In pursuing this line of work, I would have to explore possible cognitive mechanisms people may be employing to reason when faced with certain tasks. This means I would be working on studies and experiments that involve understanding how aspects of human cognition may be working.
2. Through the application of some of the knowledge obtained from understanding strategy forming in humans, newer artificial intelligence techniques could potentially be unearthed.

Having a good understanding of human reasoning and its associated cognitive processes, including a good grasp of how people form strategies, provides the opportunity for us to improve human life. With good knowledge about how human cognition works we can have the right resources to build better educational tools that are more effective at promoting learning, we can have accurate ways of evaluating and classifying the cognitive capabilities of individuals, and we can have better ways of providing care to people who are experiencing mental health issues.

Even more importantly, in a world where we are increasingly becoming aware of our neurological diversity, such knowledge can help in building an environment in which people of different neurodiverse leanings can comfortably live meaningful lives and work together to achieve both our personal and common goals.

1.5 Overview of this Dissertation

My primary focus through this dissertation was to investigate strategy differences between intelligent systems on specific tasks. I paid particular attention to exploring three different ways in which strategies were implemented in intelligent systems. Specifically, I looked at how strategies could be specified through hand-coding by an AI system's designer, how machines could generate strategies through algorithms (like program synthesis), and to complete the loop, I studied ways in which we can analyse human strategies on certain tasks.

1.5.1 Problem Statement

Work on this dissertation was meant to address the following problem statement:

Humans have the ability to form strategies when faced with novel tasks, and different people often form different strategies for the same task. Most AI systems, on the other hand, do not exhibit this level of fluidity. Currently, there is limited research on how to formally represent a space of strategies such that individual strategies can be systematically synthesized, scrutinized, and transferred. This limitation acts as a barrier to building AI systems that reason fluidly.

1.5.2 Research Questions

In addressing this problem statement, I pursued the following research questions:

1. To what extent can we define information processing strategies using imagery based representations that are sufficient for solving tasks from the paper folding, block design, and Leiter-R intelligence tests?
2. To what extent can we establish a search space defined by an imagery-based domain specific language along with search algorithms based on heuristic and probabilistic graph search in order to synthesize strategies that successfully solve problems from the abstract reasoning corpus and the block design test?
3. To what extent can we identify and categorize patterns in human strategy choices from rich multi-modal behavioural trace data collected during performance of the block design task?

In answering these questions, I focused on 4 different visual reasoning tasks that can be considered as intelligence tests. These were the VZ-2 Punched-hole paper folding task (Ekstrom & Harman, 1976), the Block Design Task (BDT) (Kohs, 1920), the Leiter International Intelligence Scale-Revised (Leiter-R) (Roid & Miller, 1997), and the Abstract Reasoning Corpus (Chollet, 2019). VZ-2 Punched-hole Paper folding, Leiter-R, and BDT

were used for the work in answering the first research question, BDT and ARC were used in answering the second research question, and the final question was focused only on the BDT. All these tasks are components of intelligence tests, with ARC specifically designed with machines in consideration.

The VZ-2 punched-hole paper folding task comes as part of a battery of intelligence tests, and it requires subjects to predict the pattern on a piece of paper after it has been folded, punched, and unfolded. Similarly, the Leiter-R is a complete battery of twenty non-verbal intelligence tests that are split into two categories for evaluating attention and memory, and visuospatial reasoning respectively. The BDT, is a non-verbal intelligence test that requires participants to reproduce patterns with specially designed blocks. Although the block design task was originally produced to be administered by itself, it is currently typically issued as part of standardized intelligence tests. Finally, ARC is also an intelligence test for humans and AI systems, designed specifically to be challenging for AI systems to solve.

1.5.3 Summary of Work Done

Work towards answering my research questions was focused on validating ideas about imagery representations through experiments on hand-coded, designer supplied strategies on the VZ-2 punched-hole paper folding task, the BDT, and the Leiter-R (see Chapter 3); building a program synthesis toolkit for generating reasoning strategies in intelligent agents through imagery based representations for the ARC and the BDT (see Chapters 4 and 5); and contributing significant work towards a system for recording and analysing the performance of humans on the block design task (see Chapter 6).

Here is a summary of all the work I did in answering the first research question:

- I built a computational model for reasoning about the punched hole paper folding task. With this model, I was able to show how a representation entirely based on imagery knowledge representations is sufficient for solving the paper folding task through a recursive reasoning strategy. The strategy used was not human inspired, used a lot more memory than what a typical human is likely to use, and was successful at completely solving all items on the VZ-2 paper folding task.

- In a bid to switch towards more human inspired architectures, I worked on a Visual Reasoning Experimentation Environment(VREE), through which I was able to deploy virtual agents to interact with virtual objects while reasoning about them visually.
- I used VREE to perform experiments on the Leiter-R and the BDT. In the experiment on Leiter-R, I investigated how two different strategies, which were influenced by the structure of most multi-choice Leiter-R tasks, perform. By taking advantage of the simulated environment of VREE, I was also able to evaluate the effects of forgetfulness on performance of the task.
- In addition to the work on Leiter-R, I performed experiments on the BDT. For these experiments, I investigated how a general template strategy into which other sub strategies can be plugged performed on the BDT. By evaluating different combinations of these sub-strategies, I was able to find which strategy works best, and I was able to lay foundations for future work on synthesizing strategies for the BDT.

On the second research question, I worked on the following:

- I developed a domain specific language for representing visual reasoning strategies. This language, named the Visual Imagery Reasoning Language (VIMRL), formed the basis of on which I ran all my experiments in exploring machine generated strategies. Programs in VIMRL have instructions which could either be represented as straight line instructions, or they could be represented as state machines when branching logic is needed. One interesting feature of VIMRL is the ability of operations to determine their own arguments by performing local searches on the problems they were meant to solve.
- With the VIMRL I built a solver for reasoning about tasks from the ARC. This solver was capable of solving a wide variety of tasks, and it tied for fourth place on a recently held competition for ARC solvers, the 2022 International ARCATHON challenge.
- Additionally, I experimented on building solvers for the BDT using agents whose reasoning is synthesized in state machines whose logic was driven by VIMRL programs.

Finally, on the third research question, I did significant work on building and evaluating systems for measuring human performance on the block design task. I worked on the following:

- I developed a scheme for annotating data collected from cameras that recorded a participant's actions when they took the block design task.
- I additionally developed tools for visualising the data collected and for extracting information about possible strategy patterns people may be exhibiting.
- Due to an interruption in in-person activities, much of the data collection work was moved to an online task evaluation platform that was developed in-house, named WOMBAT. For this effort, I contributed some guidance on development and I ported over all the tools used in analysing data from the in-person data collection systems.

1.5.4 Organization of this Document

The rest of this dissertation is organized as follows: In the second chapter, I review literature that provides background for this work. Then in the third chapter, I go ahead to provide details on work done in investigating hand coded strategies to answer the first research question. Work on the second research question is captured in the third and fourth chapters. The fourth chapter specifically focuses on work around the development of VIMRL and the experiments on the ARC, while the fifth chapter focuses on the work done in synthesizing strategies for the Block Design Task. The sixth chapter details work I contributed to an effort in building systems for automatically quantifying human performance on the BDT. And I conclude the work with thoughts on possible future directions in the seventh chapter.

CHAPTER 2

Background

Pioneers of artificial intelligence were not only inspired by the prospects of reproducing human-like intelligence, they also identified the potential for artificial intelligence as a tool for understanding the mind and human reasoning (McCarthy et al., 1955). This is probably the reason why thinkers like Turing (1950) had already made predictions about the viability of digital computers for reproducing human-like reasoning, even before the first digital computers were built.

Although this dream of reproducing true human-like reasoning—as optimistically envisioned by the pioneers—has not yet been achieved, inspiration drawn from human-reasoning has led to the development of several AI systems that perform well, even doing better than humans in some cases, in their specific task domains. Currently, there are AI systems that dominate humans in tasks like playing the game of chess (Campbell et al., 2002) or Go (Silver et al., 2016); and there are other systems that exhibit exceptional performance in tasks like recognizing objects in pictures (Zhai et al., 2021), translating text between natural languages (Zhang & Zong, 2020), generating text with human-like prose (Brown et al., 2020), and producing visual art in convincing human-like style (Ramesh et al., 2021). Interestingly, regardless of how these systems improve in their performance on various task domains, human performance still remains the benchmark against which they are evaluated (Winnerman, 2018).

While artificial intelligence systems are getting better at performing human-like tasks, the other goal of understanding the human mind through artificial intelligence has also seen a lot of progress. Researchers in this area, typically work in multidisciplinary teams to build models of human reasoning with the goal of better understanding the ways in which people may be thinking (Young, 2001). Generally, such work may rely on models that are built around elaborate architectures that simulate human cognitive processes like memory, learning, and planning (Kotseruba & Tsotsos, 2018). These architectures allow models to be built for tackling a wide array of tasks, with some systems even having the capability of

introspection through meta-cognition (Sun et al., 2006).

For the rest of this chapter, I will be providing some background and reviewing literature on how computational models have been used for studying visuospatial reasoning. Then, I will briefly discuss program synthesis and how it has been used in the field of artificial intelligence, while providing a brief literature review. Finally, I will introduce the tasks that were studied in this dissertation, and I will discuss some of the prior work that has been done regarding computational models for reasoning about the tasks.

2.1 Computation, Cognition and Visuospatial Reasoning

All computational cognitive systems for studying human reasoning, regardless of how complex they may be, are underpinned by a simple fact: human reasoning can, in some ways, be considered as computational in nature (Newell & Simon, 1976; Thagard, 2005). Thus, unlike the brain whose underlying cognitive processes may be difficult to study—even when modern neuroimaging is considered—computer programs that implement reasoning models can be inspected in much more thorough detail. The underlying instructions and associated data structures used in a program can be studied, and the factors that influence the choice of a particular strategy over another (provided these strategies are interchangeable in the program) can also be identified.

At a fundamental level, cognitive scientists (an interdisciplinary group of researchers consisting of computer scientists, psychologists, neuroscientists, philosophers, anthropologists, linguists and education experts) study reasoning, the mind and intelligence through the theoretical framework of mental representations (Thagard, 2005). Marr (1982) defines representations as a formal system through which knowledge or information is expressed. In even simpler terms, Winston (1992) defines representations as how things are described.

2.1.1 Theories of Representation

The choice of representation in a system determines what the system can express. This choice also has a direct effect on the kinds of operations that can be performed in the system. For example, when we consider the quantity represented as a tally in Figure 2.1, it is possible to tell what the remainder of this quantity is when divided by 5, even

without knowing the actual value represented. However, when asked about the remainder when this same quantity is divided by 7, a complete count and conversion to a different representation (probably Arabic numerals) may be necessary for a correct response. It is easier to tell the remainder of a division by 5 with just a look at Figure 2.1, because the tally representation of grouping items into fives naturally lends itself to such problems.

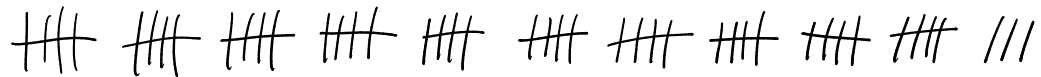


Figure 2.1: A tally diagram that represents a number. Even without knowing the exact value of the number, it is easy to know what the remainder is when it is divided by 5.

Different forms of representations have been proposed for studying visuospatial reasoning. But before exploring the different ways in which knowledge is represented for visuospatial reasoning, it is worth understanding the possible ways in which all forms of knowledge representations, in general, can be evaluated. Given that the potential operations of a system are constrained by its internal representations, being able to compare different forms of representations is worthwhile.

Thagard (2005) proposes a framework with five evaluation criteria under which different forms of representations can be evaluated and compared. According to this framework, representations can be evaluated under the following properties:

- Representational Power
- Computational Power
- Psychological Plausibility
- Neurological Plausibility
- Practical Applicability.

Representational power describes how efficiently a representation expresses knowledge of a particular kind. Through the concept of representational power we are able to know

things such as the kinds of operations the representation lends itself to, how efficiently these operations (or computations) can be performed, and in some ways, how explicable problem-solving with the representation is. Since representations are directly tied to operations, the *computational power* of a representation additionally describes a representation's problem-solving capabilities.

Further, if we intend to study human reasoning, then the choices of representations must produce outcomes that are close, if not exactly equal, to what humans will generate. This measure of similarity to human ability is the *psychological plausibility* of a representation. Representations with good psychological plausibility give us the opportunity to indirectly study how the mind operates through high level cognitive processes. If our interest, however, is to understand the mind in terms of its actual biological composition, then a measure of a representation's *neurological plausibility* becomes important.

Ultimately, any knowledge gained from studying reasoning through computational models may be useful if (such knowledge is) transferable to practical areas of application like education, healthcare, intelligent systems, etc.

2.2 Visuospatial Reasoning in Humans

The exact form in which humans performed visuospatial reasoning with imagery based representations had been under debate for several decades (Pearson & Kosslyn, 2015). It was not disputed that humans could experience imagery; what was in contention was how depictive imagery was represented in the mind. The theory in one camp was that all reasoning occurred through propositional representations, and imagery was only "experienced" as a side effect when people processed these representations (Marr, 1982; Pylyshyn, 1973). The other side argued for mixed representations, which may be depictive and structurally represented when reasoning with imagery, and symbolic when reasoning verbally (Kosslyn, 1980; Paivio, 1990). This debate has since been resolved through modern neuroimaging, after it was demonstrated that mental imagery and visual perception appear to rely on the same mechanisms in the brain to operate (Albers et al., 2013; Kosslyn et al., 1995; Slotnick et al., 2005; Stokes et al., 2009).

However, before direct evidence for imagery in humans was provided through neu-

roimaging, research into understand mental representations in humans had already yielded compelling results about the possible use of imagery. Early experiments by Shepard and Metzler (1971) showed that when participants were presented with two-dimensional images of two three-dimensional objects the response time in determining whether the two items were the same, was linearly proportional to the angle of rotation between the two objects. This indicated that participants were probably mentally rotating these objects.

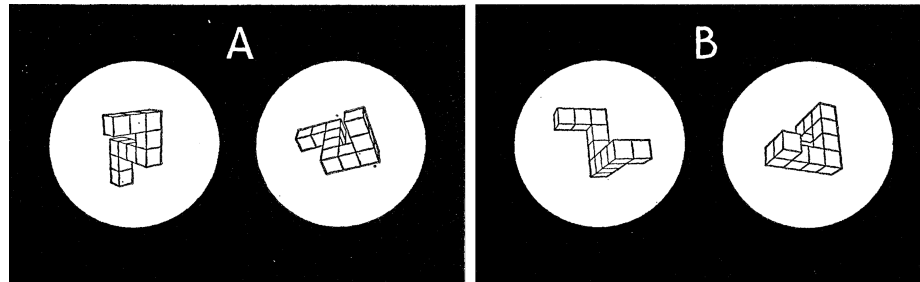


Figure 2.2: Sample stimuli from Shepard and Metzler (1971)'s mental rotation experiment. For each of these test pairs, the participant was supposed to determine if the image on the right is a three-dimensional rotation of the one on the left.

Proponents of propositional representations argued that the linear correlation Shepard and Metzler (1971) showed could be the result of a piece-wise comparison between stimulus pairs, and not necessarily the effect of a complete mental rotation of the three-dimensional object. Indeed, through eye tracking, Just and Carpenter (1976) were able to show that the number of fixations between the images of the objects also increased proportionally to the angular rotation.

These arguments led to other mental rotation experiments, such as L. A. Cooper and Shepard (1973) where subjects were presented a single familiar stimulus, like the upper case letter "R", instead of two three-dimensional objects. They were still required to tell if the letters were mirrored or not. The results still showed response times proportional to the angle by which the stimulus had been rotated, and extra time was even required in cases where the letter was actually mirrored, since people may have to additionally perform the flipping operation.

Mental rotations have not been the only means of studying imagery representations in humans. Kosslyn (1973) demonstrated that when people are shown an image and later

made to recall features from the image, the amount of time it takes to shift their mental focus from one part of the image to another was directly proportional to the actual distance between the items. Hegarty (2004) also presents a review of the potential ways in which people may be using a mix of imagery and propositional rules to reason through mechanical problems by simulation.

2.3 Visuospatial Reasoning in AI

When it comes to mental representations, propositional representations are symbolic, more language-like and descriptive. They are meant to represent content much in the same way as language does. Imagery representations, on the other hand, are more depictive of the content they describe, such that representations exhibit matching spatial properties. For artificial intelligence research into visuospatial reasoning, both representation types are actively used.

Classes of visual reasoning systems that do not use imagery operations are those that take images as inputs, but process them through other internal representations. An example of this could be an image classifier built with support vector machines. Although such a classifier can easily reason about an image and make some inferences about its content, such as correctly identifying what is depicted in the image, much of the internal knowledge of this classifier is represented as a complex system of linear equations.

Visual reasoning systems that are imagery based must meet the following criteria (Kunda, 2018): First, such systems must use internal representations that are visual, image-like, and iconic. This means the representations must have some visual or, at least, structural likeness to the content they are representing. Second, the content of the representation must not be exactly the same as whatever inputs the system receives through its perception. For systems to meet this second criteria, they must have ways of internally modifying their inputs into other visual forms, or even generating newer visual images for reasoning internally. Finally, the image representations must play an active role in the system's reasoning processes; they must not just be passively stored to be simply reproduced later.

2.3.1 A Brief Survey of Visual Imagery Based AI Systems

One of the earliest demonstrations of imagery as a representation for reasoning was presented by Kosslyn and Shwartz (1977). Kosslyn and Shwartz’s work was meant as an exploration of how humans may be using imagery as a medium for solving problems. Specifically, their interest was in how imagery, as experienced by humans, was possibly rooted in propositional descriptions that are themselves built on prior perceptual inputs. Although the system did not reason about any particular tasks, it exhibited image generation abilities that could yield images to be manipulated through the system’s “mind’s eye”. It could perform operations like mental scanning and rotation on these generated images—operations that Kosslyn (1973) also investigated in human subjects.

An early use of imagery in problem-solving can be seen in WHISPER (Funt, 1980), a system for physical reasoning about block worlds using naive physics. WHISPER had the ability to take an input image of a blocks world structure, and when the structure was depicted in an unstable state, WHISPER could predict how the structure was going to collapse. By taking inspiration from how humans were possibly paying selective attention, WHISPER included a simulated radial retina that could be fixated on different parts of the input. This retina was also foveated, giving it a resolution that decreased as you moved away from its centre, much like it is in a human retina. Additionally, the radial nature of the retina allowed for its “receptors” to be connected in a concentric grid, yielding the additional benefit of directly performing rotation and scaling operations right on the retina in a bottom up manner.

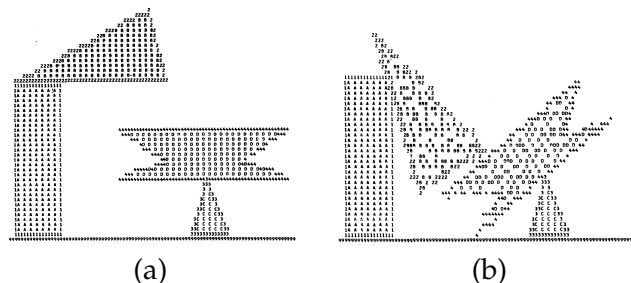


Figure 2.3: Figures from WHISPER (Funt, 1980). (a) An input image for whisper in its unstable state. (b) The final state of the blocks after the naive physics simulation.

Gardin and Meltzer (1989) demonstrated a system for physical common sense reasoning, that dealt with the properties of objects like strings and liquids through analogical models. Much like WHISPER, this system reasoned about physical objects through simulations, and much like WHISPER's retina, Gardin and Meltzer's system was based on the interaction of connected particles that communicated with each other. Essentially, all objects in this system were designed to be made of particles that interacted with each other according to a few simple rules. For strings, particles in the system could rotate around each other, have external forces acting on them, and could collide with other objects in the environment. For liquids, the particles were acted upon by gravity, could collide with other particles and objects in the environment, and always moved to fill spaces. With just these particle systems and a few rules, Gardin and Meltzer's system was able to naively reason about strings, rigid rods, and fluids flowing into spaces, without solving any actual physical modelling equations.

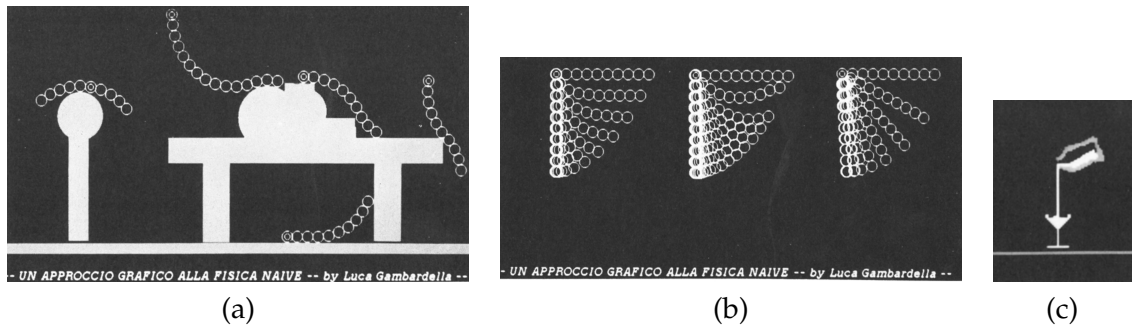


Figure 2.4: Results from Gardin and Meltzer (1989) showing how particles were used to simulate flexible strings, bars of different flexibility, and liquids.

In the same vein as Kosslyn and Shwartz (1977) on representations, Glasgow and Papadias (1992) also presented a model on how imagery can be used in computational systems. The representation, aptly named "Computational Imagery", was not necessarily meant to be a cognitive model of human reasoning, nor was it meant to solve any specific tasks. It was meant to be a demonstration of how imagery can be used in computational systems in ways that are human inspired. Based on this inspiration, the system had a long term memory which stored symbolic information about the relationships between objects, and this information from long term memory could further be used to invoke image based

visual representations or array based spatial representations. As far as Glasgow and Papadias's "Computational Imagery" representation was concerned, visual thinking—and in effect imagery—dealt with what an object looked like, and spatial reasoning dealt with where an object was located in space with respect to others.

There are some classes of systems that are built to reason using mixed representations. As examples consider the following: Tabachneck-Schijf et al. (1997) present a computational model of an expert that explains economic concepts with the help of graphs, Roy et al. (2004) demonstrate a conversational robot that exhibits spatial awareness in language through images of what it sees in its world, (Bertel et al., 2006) a system for reasoning about things through sketches, and (Lathrop et al., 2011) present a model of a fighter pilot's reasoning in combat situations.

Another class of tasks that have been of interest in AI research are psychometric intelligence tests (Hernández-Orallo et al., 2016). Because most of these tests are meant to be tools for measuring human intelligence, there is a lot of value to be gained when AI Systems are built to solve these tests. One of the earliest AI systems for solving a psychometric test was ANALOGY (Evans, 1964). ANALOGY was built to solve geometric analogies, a class of problems that are inherently visuospatial in nature. At the time of its introduction, ANALOGY was believed to be the largest LISP program ever written. Although geometric reasoning tasks are spatial in nature, ANALOGY was based on symbolic representations, which were direct descriptions of the visual problem items. Inputs to the system were manually converted from their original depictive geometric forms into descriptive LISP lists. An assumption the authors made was that future improvements will include programs that will automatically convert the geometric images into the symbolic format. At the core of ANALOGY's reasoning were pattern recognition routines and geometric operations like rotations, translation and scaling. ANALOGY's success led to several similar systems for solving matrix based analogy problems through propositional representations.

In exploring other representations, Kunda et al. (2013) built the Affine-and-Set Transformation Induction (ASTI) AI system that reasoned through matrix geometric analogy problems from the Ravens Progressive Matrices (Raven & Raven, 2003), using only image representations and operations. ASTI worked by trying out different operations across

the matrix until it figured out a series of operations that created a change in a row or a column. Once these operations were found, an image was synthesized with this operation and compared with the possible answers to find a match. When tested on the standard Ravens Matrices, ASTI was able to score 50 out of the 60 problems without extracting any form of verbal or propositional information. An extension of ASTI (Yang et al., 2020), which considered multiple other ways of slicing and combining the matrices, while still relying on imagery representations with improved similarity matching, was able to extend the score to 57 out of 60 problems.

It is important to note that an intelligent system that reasons through visual imagery representations, must have its image inputs play a significant role in the overall reasoning process (Bertel et al., 2006; Kunda, 2018). Imagery should not only be captured and stored, but must be an active component in of the system’s decision-making process. Thus, when used for reasoning, images must be transformed in some way and inferences about the task at hand must be drawn from these transformations.

2.4 A Brief Survey of Program Synthesis

Program synthesis is the automatic generation of computer programs to meet certain user defined requirements. These requirements for synthesized programs can be presented in various forms, such as formal logical constraints, input-output examples, task demonstrations, and natural language descriptions (Gulwani et al., 2017). Program synthesizers typically work by searching a hypothesis space of possible programs for one that meets the specified requirements. A bulk of the work on any program synthesis system comes from defining the underlying language and its associated search algorithm.

Techniques from program synthesis have been applied across different research communities in computer science such as programming languages, machine learning and artificial intelligence. Typical applications include code optimization, suggestion, and repair; data processing and modelling; robotics etc.

Program synthesis is fundamentally a search problem, and the combinatorial nature of this problem, coupled with the complexity of the languages in which programs are expressed, creates significantly large hypothesis search spaces—even when the simplest of

problems are considered. This makes program synthesis a difficult problem.

At a high level, program synthesizers search for programs by systematically generating possible programs according to the grammar of the language, and verifying these programs with the supplied specifications. What varies among synthesizers is how this space is explored. Some synthesizers enumerate possible programs from the language's grammar by growing programs through the addition of operations. There are other synthesizers that take a deductive approach of generalizing the solution by breaking problems down into simpler sub-problems whose solutions can later be combined. Another group of synthesizers work by solving constraints that exist between the program's specifications and the language. And, as a final example, there are synthesizers that rely on large language models built into neural networks to generate code, or others that explore their program spaces with genetic algorithms.

2.4.1 Grammar Enumeration

The simplicity of this search approach makes it one of the most used program synthesis techniques. But, like many other combinatorial search processes, this technique is prone to explosions in the search space. For enumeration to be successful, optimizations such as the use of domain specific languages to ensure efficient search, type checking to exclude erroneous programs, stochastic exploration of selected relevant nodes, and the application of logical rules to prune branches that are expected to yield invalid programs must be employed.

Enumeration has been in used in synthesizers like DeepCoder (Balog et al., 2017), `MAGICHASKELLER` (Katayama, 2010), and in work from Lázaro-Gredilla et al. (2018). Exploration in these systems are based on top-down tree search algorithms, that generate partial programs which are evaluated and extended as search progresses. Additionally, each of these systems incorporate unique ways of limiting explosions in the search space.

DeepCoder solves coding competition-like problems, and relies on an optimized depth first search traversal to synthesize programs from input-output examples. There are a couple of optimizations in DeepCoder's search, but one technique that stands out is the use of a neural network to predict the possible functions a program could have when given

an input-output example. The neural network was trained with input-output pairs and functions from a corpus of ground-truth programs. By limiting the search to just these predicted functions, DeepCoder searches a significantly reduced space around a functional programming DSL.

Lázaro-Gredilla et al. (2018) also demonstrates a system that learns to synthesize programs from input-output image examples. The ultimate goal of their work was to use the system to train pick-and-place robots on concepts of how objects can be arranged. Using a DSL and a system called “Visual Cognitive Computer”, inputs from images can be parsed to identify objects (with the help of a Convolutional Neural Network), and output programs can manipulate a robotic arm to arrange items on a stage according to concepts elicited from the input-output image pairs. Just like DeepCoder, synthesis was driven by a depth first search, but here the search was guided by a probabilistic Markov chain model that predicted how instructions in the DSL are sequenced.

2.4.2 Deductive Search

Systems that take a deductive approach tend to rely on the peculiar nature of their task domains and implementation languages to carry out their unique search techniques. THESYS (Summers, 1977) was an early program synthesizer that generated programs from input-output examples through a deductive process. In line with work from Shaw et al. (1975) and Biermann (1978), program requirements were presented as part of a LISP program that took advantage of the structural nature of *s-expressions* in the LISP language. Search in THESYS started off with a program fragment that was consistent with one of its input-output examples, and this program fragment was recursively extended to make it consistent with other examples as needed.

A similar approach to THESYS can be seen in IGOR2 (Schmid & Kitzelmann, 2011). However, instead of plainly synthesizing programs from input-output examples, IGOR2 solves the kinds of number series problems found in intelligence tests. IGOR2 also uses data from its input problems to synthesize programs in functional languages—MAUDE and later HASKELL. Much like THESYS, IGOR2 takes advantage of the nature of the task and language to implement a search strategy that is more deductive.

Perhaps the most widely deployed program synthesizer, FlashFill (Gulwani, 2016) is a system for learning complex string transformations from input-output examples that has been shipping with Microsoft Excel since version 2013. FlashFill learns string transformations from a few input output examples. Because of its unique use-case, FlashFill’s DSL consists of regular expression based string manipulation operations that facilitate breaking down the main program into smaller sub goals for a search that efficiently works backward from output to inputs in a bottom up manner.

2.4.3 Constraint Solving

Systems that rely on constraint solving typically express the relationship between a DSL and input-output examples as constraints that must be solved, in most cases, through Satisfiability Modulo Theories (SMT). Unlike enumeration which involves the churn of trying out as many values as possible for variables while searching for programs, SMT solvers work like SAT solvers with the added benefit of having built-in specializations, through theories, for solving special kinds of constraints.

SKETCH (Solar-Lezama, 2008) is a synthesizer that takes an input template program with holes and replaces these holes with synthesized expressions that meet the specifications. Search in SKETCH proceeds through a counter-example guided search, and an SMT solver is used to solve SAT expressions that are derived from the template program and its associated specification. SKETCH uses a C-like language, which makes it easy to incorporate its results into programs from most languages with that style. Another synthesizer that takes a similar approach is ROSSETTE (Torlak & Bodík, 2013), which is built for the Racket programming language.

2.5 An Overview of Tasks Studied in this Dissertation

A wide range of visuospatial reasoning tasks exist, but for the purposes of my dissertation I worked with specific standardized visual reasoning tests. This choice was motivated by three main factors. First, using standardized tests provides a set of well-defined tasks whose goals and scoring methods have been standardized and properly documented. Second, because some of these tasks have been in existence for a while, there is a wealth of data

available on human performance, which provides the opportunity to compare the performance of any models with that of humans. Third—and probably most importantly—using standardized tests, which can be tested on humans, provides a great medium to transfer knowledge about any strategies or possible underlying mechanisms that are obtained from experiments on models to humans.

Work on this dissertation covered four different visuospatial reasoning tasks. These were the Block Design Test (BDT) (Kohs, 1920), the Leiter International Performance Scale-Revised (Leiter-R) (Roid & Miller, 1997), the Abstract Reasoning Corpus (ARC) (Chollet, 2019), and the VZ-2 Punched-hole Paper Folding Task from the Kit of Factor-Referenced Cognitive Tests (Ekstrom & Harman, 1976). The block design is a test of visuospatial reasoning that requires test takers to manipulate coloured blocks to replicate a given design. The Leiter-R is a battery of twenty different visuospatial reasoning tests that test visualization, attention, and memory. The ARC, although not necessarily standardized (as a means of measuring human intelligence) is an abstract reasoning benchmark specifically designed to evaluate how well AI systems can learn to generalize concepts from very few examples.

2.5.1 The VZ-2 Punched-hole Paper Folding Task

Paper folding tasks are a family of tasks that require people to reason about fold patterns in physical objects and their effects on the shapes of such objects. Generally, in these tasks a person must either predict the outcome of a given fold pattern, or they must forecast the sequence of folds that could result in some given final shape. Other variants of the paper folding task—like the punched-hole version discussed throughout this section—require participants to determine the patterns that will be made when a folded piece of paper is punctured and unfolded. Because these tasks must be performed mentally, people may have to rely on some imagined model of whatever they may be folding in order to be successful on such tasks.

Although the exact cognitive mechanisms used for paper folding remains unknown, it is believed that mental rotations (Shepard & Metzler, 1971) and mental folding (Shepard & Feng, 1972) play a major role in how people reason through the paper folding tasks (Wright

et al., 2008). Additionally, for the punched hole variants, people may need the working memory capacity to deal with the additional complexity introduced by keeping track of folds and punch patterns.

Paper folding tasks are usually components of intelligence test batteries, like the Leiter-R (Roid & Miller, 1997) and the Kit of Factor-Referenced Cognitive Tests (Ekstrom & Harman, 1976), and they are also widely used as stand-alone tests for measuring spatial reasoning skills. The following is a small sample of studies in which the paper folding task was used as a means of assessing the spatial reasoning skills of subjects.

- In a study of the relationship between a soldier's cognitive abilities and their marksmanship, researchers from the Warfighter Health Division of the United States Army Aeromedical Research Laboratory used paper folding as one of the tools to measure a subject's spatial ability (Kelley et al., 2011).
- Mayer and Massa (2003) and Kozhevnikov et al. (2002), in separate studies on the verbalizer-visualizer hypothesis of learning styles in individuals (Jonassen & Grabowski, 1993), used the paper folding test as a measure of spatial ability in their subjects.
- Keehner et al. (2004) were able to show that for inexperienced surgeons, skill level in laparoscopic surgery was higher in subjects who scored better on a battery of tests that included paper folding tests. For experienced surgeons, however, there was no significant difference in performance.
- Silvia (2008) used paper folding as a tool for measuring fluid intelligence, while investigating the relationship between creativity and intelligence.

With so many variants of the paper folding test in existence, I chose the form that appears as the second visualization task (VZ-2) in the "Kit of Factor-Referenced Cognitive Tests" battery (Ekstrom & Harman, 1976) for my work. The VZ-2's paper folding task was ideal because its multi-choice, pen and paper administration made it a good candidate for modelling in a computer system. Each item in the VZ-2 punched hole test presents a series of images that depict the steps of folding a piece of paper that is later punched. The subject

must pick from a list of given options, the pattern generated when this folded paper is unfolded. A sample item from the VZ-2 paper folding test is shown in Figure 2.5.

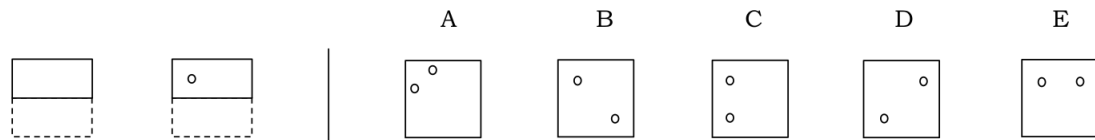


Figure 2.5: A sample item from the VZ-2 paper folding task. The two images to the left of the vertical bar depict the sequence of folds and the eventual punch. The other four images labelled A through E are possible answers from which the test taker is to make a choice.

The VZ-2 contains 20 test items that are split into two parts of 10. Subjects are required to complete the first part within the first three minutes, take a break, then proceed to take the second part, which must also be completed within three minutes. In terms of difficulty and structure, there are no significant differences between the test items in both parts. As far as difficulty is concerned, harder items seem to be randomly mixed up with easier ones across the test. In fact, the test's instructions encourages subjects not to waste time, but rather skip items they may find difficult.

2.5.1.1 Computational Modelling of Paper Folding Task

Although Paper Folding tests have been extensively used in research, mainly for measuring people's visuospatial reasoning skills (for examples see Keehner et al., 2004; Kelley et al., 2011; Kozhevnikov et al., 2002; Mayer and Massa, 2003; Silvia, 2008), not much work has been done in computationally modelling how people reason about it. Lovett and Forbus (2013) while studying the information processing load required for mental rotation and paper folding, implemented a model that reasons through paper folding problems using CogSketch, a visual sketch understanding system for cognitive science research. In their work, they showed that one possible strategy for reasoning through folding and rotation problems involves simplifying stimuli by reducing complexity, a capability they proposed may be a skill people with high spatial ability may possess.

2.5.2 The Block Design Test

The Block Design Test (BDT) was introduced by Kohs (1920) as a non-verbal test for measuring human intelligence. It is normally administered as part of other intelligence test batteries like the Wechslers Adult Intelligence Scale (WAIS) (Wechsler, 2008). The goal of Block Design is to have subjects reconstruct designs using specially coloured blocks. Each block is a two colour cube, configured such that four faces have a single colour, and two faces have both colours painted on the opposite sides of a diagonal. The final distribution of colours makes each block look as if two differently coloured triangular prisms were fused together. Originally, the test had two different colouring schemes: some blocks were coloured red and white, while others were blue and yellow. Currently, in most use cases, such as in the Wechslers Intelligence Scale for Children-revised (WISC-R) (Wechsler, 1974), only the red and white blocks are used.

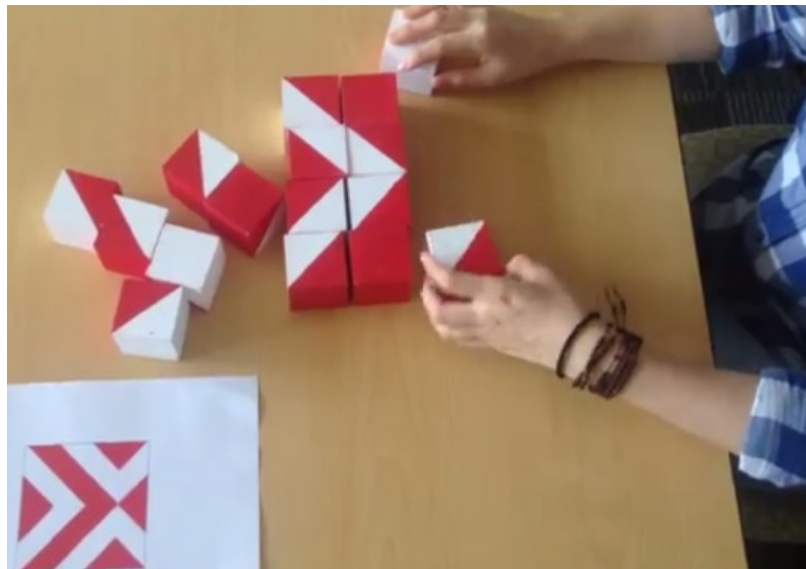


Figure 2.6: A closeup shot of a person taking the block design task

The non-verbal nature of the BDT makes it a good test of visual reasoning. According to Kohs (1920), instructions for the test can be given with simple demonstrations or pantomimes, if necessary. In a typical test, easier designs will be presented first, with the difficulty increasing as the test proceeds. Elements such as the use of diagonals, symmetry, boundaries and even the number of blocks are introduced to alter the difficulty of items.

Given the active use of the BDT for intelligence testing, there is a wide range of human performance data available. For example, it has been observed that performance on the BDT increases gradually with age, for children and then plateaus for adults, with a later decline in the senior years (Rozencwajg & Corroyer, 2002). It has also been shown that autistic children perform better on the block design than their typically developing counterparts (Shah & Frith, 1993).

2.5.2.1 Computational Modelling of the Block Design Task

Block Design Tests have been used in lots of studies aimed at understanding human cognition, but most of these have been human studies. Very little work has been done on the task as far as studies through computational models are concerned. In an early demonstration of robotic capabilities, Bringsjord and Schimanski (2003) solved the block design task using actual physical blocks and a robot named PERI. PERI was able to receive visual information about the state of the blocks, and with image matching operations, it was able to solve the task by moving the actual blocks into place. Kunda et al. (2016) used a similar approach for the block design solver, except everything was solved in a simulated environment, with an agent that could simulate visual attention and had a volatile short-term memory.

2.5.3 Leiter International Performance Scale-Revised

The Leiter International Performance Scale Revised (Leiter-R) (Roid & Miller, 1997), is a cognitive test for evaluating visuospatial reasoning in children and young-adults between the ages of 2 and 21. One significant feature of the Leiter-R is how the test is administered in a pure non-verbal format, whereby instructions for tests are given to test-takers through gestures and pantomimes. This property makes the test widely accessible to people of diverse cognitive and verbal abilities.

The entire Leiter-R test is made up of twenty subtests, split in two categories. Ten of these subtests are grouped into a Visualisation and Reasoning (VR) battery, which contains tests intended for measuring reasoning, problem-solving, and visualization, while the other ten are grouped into an Attention and Memory (AM) battery, which contains tests for measuring attention and memory. Table 2.1 provides a summary of all items on the

Leiter-R test.

Subtests on the Leiter-R are designed to test human abilities such as visual scanning (through tests like the Figure Ground), rule forming from visual inputs (through tests like the Sequential order and Repeated patterns test), mental rotation (through tests like the Figure Rotation) and other related visuospatial reasoning skills.

Table 2.1: All 20 items in the Leiter-R, listed with brief descriptions of what the test requires.

Subtest	Task for the test-taker	Battery
Associated Pairs	Match an item with its corresponding item with a rule that is learned after a training period.	AM
Attention Divided	Identify items from one set of images while sorting another set of images.	AM
Attention Sustained	Find and mark as many instances of a given image as possible from a collage of similar images.	AM
Classification	Match images with others that are semantically related. Example, socks and shoes.	VR
Delayed Pairs	Remember images from the Associated Pairs task without looking at the original images.	AM
Delayed Recognition	Recognize items from the Immediate Recognition task without looking at the original items	AM
Design Analogies	A collection of geometric matrix analogy problems.	VR
Form Completion	Reconstruct an image from broken parts.	VR
Figure Ground	Search an image for specific, exactly matching fragments.	VR
Forward Memory	Remember the sequence in which a series of images are presented.	AM
Figure Rotation	Match an image with its rotated counterpart.	VR
Immediate Recognition	Recognize items that have been removed from a collage that was displayed briefly.	AM
Matching	Match items to their exact copies.	VR
Picture Context	Select items that are semantically related to others in a given image.	VR
Paper Folding	Select images that are either folded or unfolded representations of materials.	VR
Reverse Memory	Recollect items that were seen in the earlier Forward Memory task.	AM
Repeated Patterns	Determine a pattern in a sequence of images and complete it.	VR
Spatial Memory	Remember where items were placed in an image.	AM
Sequential Order	Determine the sequence in a series of images and complete the sequence.	VR
Visual Coding	Perform simple logic operations with rules that are defined through images.	AM

2.5.3.1 Structure of the Leiter-R

Most subtests on the Leiter-R are presented as multiple-choice style problems. Typically, problems are displayed on the pages of flip-book easels, and subjects are given cards with the possible answers from which to make a choice. Answer choices are made by placing cards into special slots provided under the easel. This particular tactile design—of placing cards in slots—gives subtests on the Leiter-R an intuitive interface that eases the test-administrator’s responsibility of describing task requirements to subjects without words. In all, there can be as many as seven answer slots on an easel page. For some classes of problems, all required easel slots must be filled, and for others they must be partially filled. This response mechanism of having to supply a varied number of choices makes it hard for someone to guess their way through the test.

Not all problems on the Leiter-R are in a multiple-choice format. Some subtests require subjects to provide pencil and paper style responses by either drawing or writing on paper, while other subtests may require subjects to point out responses on the easel with their fingers, or sometimes with a special task-specific card.

The difficulty of subtests on the Leiter-R tend to increase as the test progresses. In most cases, a subtest will begin with simple training items whose correct answers are supplied. Through mimes and other non-verbal gestures, test-administrators use these training items to explain a subtest’s requirements to subjects. After the training phase, tests progress with items increasing in difficulty, in a manner that makes initial test items very easy and later ones significantly difficult. Due to how difficult some items later in a test can be, some difficult items on the Leiter-R are age limited. Test administrators are advised to stop giving out a particular test after a subject consistently makes errors in a row, or appears to find the items too difficult.

2.5.4 Abstract Reasoning Corpus

The Abstract Reasoning Corpus (ARC) was introduced by Chollet (2019) as a general intelligence test for both humans and artificial intelligence systems. As a test, ARC is particularly difficult for most traditional machine learning algorithms (especially when trained from scratch) because it requires the generalization of concepts over limited training

sets that have an average of about 3 training items.

Figure 2.7 shows sample tasks from the Abstract Reasoning Corpus. These tasks demonstrate the major characteristic of tasks from the ARC: tasks are presented on grids, where some unknown rule transforms one input grid into an output grid. An ARC solver’s goal is to find this rule and supply the output grid for a given challenge input grid.

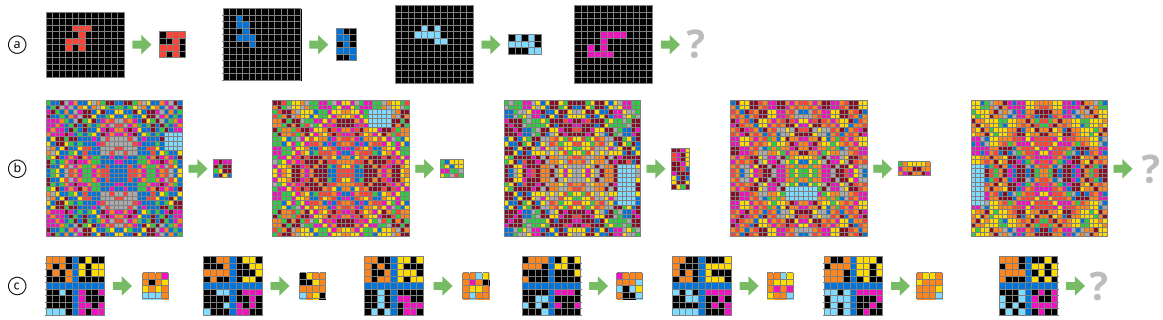


Figure 2.7: Sample tasks from the ARC’s training set. (a) A task that requires solvers to isolate the object depicted in the grid. (b) A task that requires solvers to complete the pattern by replacing the pixels in cyan coloured patch. (c) A task that requires solvers to split the input into four same sized quadrants and overlay the output into a single image output.

Tasks in Figure 2.7 also show how the underlying concepts of task items on the ARC tend to vary. For example, the first task on the top requires solvers to isolate objects, while the second task in the middle requires solvers to complete a pattern. For the third task on the bottom, solvers are expected to split the grid into four quadrants along the blue cells and output a grid in which the isolated quadrants are combined into one grid by another unknown rule that the solver must additionally determine.

According to Chollet (2019), the ARC tasks were designed with the consideration of program synthesis as a more probable solution strategy. This is even more evident in its structure of having only a few training and test items for each task.

2.5.4.1 Structure of the Abstract Reasoning Corpus

All tasks on the ARC are presented as input-output grids (as shown in Figure 2.7) with grid sizes ranging anywhere from 1×1 through 30×30 that are not necessarily squared. A distribution of sizes of all images in the dataset is shown in Figure 2.8 Each cell in the grid holds one of 10 symbols, 0 through 9, which are typically represented by unique colours

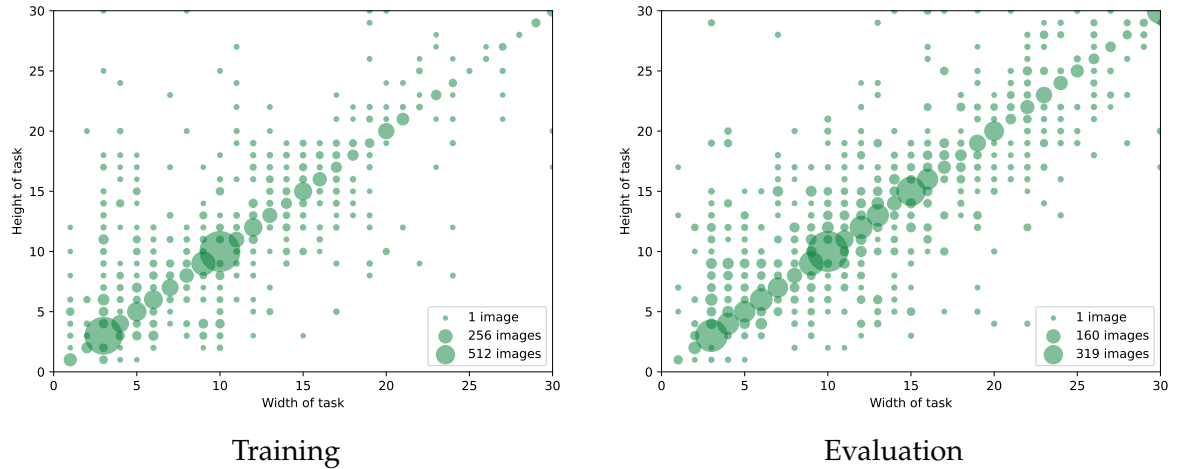


Figure 2.8: A plot of task image sizes from the ARC dataset. The circle’s centre represents the width and height of a given image size and the size signifies the number images with that size.

when tasks are visualized. It is also worth noting that tasks on the ARC are uniquely tagged with eight character hexadecimal codes. So, for example, the tasks displayed in Figure 2.7 have the codes: `1cf80156`, `0934a4d8`, and `a68b268e` respectively.

As originally released by Chollet (2019), ARC contains a total of 1,000 different tasks. Of these, 800 are publicly available to researchers, while the other 200 are kept private for evaluation. Developers building solvers have access to the complete solutions (containing both the input and output grid pairs) for all the test and train sections in the 800 publicly available task items. Access to the private tasks are, however, limited. Researches hoping to test their solvers on this private test items can either run their solvers through online an evaluator on Kaggle¹, or through the occasional ARCATHON² competition.

When a solver is evaluated by any of the private dataset providers—either Kaggle or ARCATHON—the only feedback provided is the total number of tasks correctly solved. For a solution to be considered correct, all cells in the output must be predicted exactly, and the output’s grid size must also be correctly determined. This requirement—for an *all-or-nothing* prediction both in the grid’s size and cell contents—certainly increases the difficulty of solving ARC tasks. To compensate for some of this inherent complexity, solvers

¹<https://www.kaggle.com/competitions/abstraction-and-reasoning-challenge>

²<https://lab42.global/arcathon>

are given the opportunity to make three predictions on tasks during private challenges. This multi-response provision allows solvers to account for any ambiguities that may be encountered in learning a task's underlying rules. Certainly, having multiple outputs also means solvers can attempt diverse strategies even if there are no specific ambiguities observed.

2.5.4.2 Prior Knowledge for ARC Tasks

Chollet (2019) provides brief descriptions of some *core knowledge priors* that solvers must possess to be successful on ARC tasks. Note that while the ARC is labelled as an “abstract” reasoning test, it presents problems in a visual format, and thus requires some degree of visual processing. In addition, core knowledge priors are about object properties expressed visually and spatially. Thus, visual reasoning abilities representing functions of both perception and inference are key for solving ARC items.

The **objectness** prior requires solvers to deal with the segmentation, permanence and interaction of objects. **Goal-directedness** requires solvers to deal with processes. Tasks that require *goal-directedness* may exhibit input-output grids that can be considered as the start and end states of some abstract process, such as an object moving from one point in the grid to another.

Numbers and counting priors are required in situations where quantities, like frequencies of object occurrences and sizes of objects, are considered as numbers for operations like comparison and sorting. The **geometry and topology** prior requires an agent to have knowledge about shapes, lines, symmetry, relative positioning of objects, and the ability to replicate objects in different ways.

2.5.4.3 Related Work

The Abstract Reasoning Corpus is a relatively new test, and with its development still in progress, not much work has gone into its verification. Currently, the only known human tests on the ARC are trials performed by the ARC's authors on human subjects during development (Chollet, 2019), and work by (Johnson et al., 2021) to measure how well humans were able to infer the underlying concepts of 40 randomly selected items on the

task.

Although the scope of tasks for the study by (Johnson et al., 2021) were quite limited, it showed that humans had the ability to quickly generalize the concepts behind tasks on the ARC to effectively solve them. To show how well this generalization occurred, each participant was made to provide a natural language description of their strategy, which was later compared to the sequence of actions they performed while actually solving the task.

Of more interest to our work are AI solvers that attempt the ARC. Through a recent Kaggle competition, several submissions for ARC solvers were made, and the best performing solver was able to successfully solve 27 items on the restricted ARC set. This solver was a heavily optimized, handcrafted system that efficiently searched a space of graphical operations to find a sequence in which operations can be applied to yield solutions on the ARC.

Kolev et al. (2020) also present Neural Abstract Reasoner (NAR), a solver that relies on neural networks, specifically Differential Neural Computers to reason through items on the ARC. Although according to the published results, the NAR scores an accuracy of 78.8% on items of size 10×10 or lower, it is not clear which section of the ARC was evaluated.

There is yet to be a program synthesis system that performs reasonably well on the ARC. The potential for these however can be seen in works like the puzzle solver from Butler et al. (2017), which synthesizes programs to reason through puzzles like sudoku and nanograms, and the teachable robot from Lázaro-Gredilla et al. (2018), which learns to generalize concepts about the placement of objects from input-output image pairs.

CHAPTER 3

Hand Coding Strategies for Visual Reasoning Tasks

A researcher interested in whether a given knowledge representation is suitable for a task, or whether a proposed theory for how some reasoning process works through a particular task, may perform a series of sufficiency experiments to test out their ideas. The goal of such experiments may vary among researchers. Some may just be interested in the ability of their system to make reasonable choices, while others may be interested in accuracy, etc. Regardless of the researcher's goal, models of reasoning built for such experiments are not expected to exhibit exact similarities to human cognitive mechanisms; they are meant to be a means for answering whether a given task can be reasoned about in some particular manner, and if it can, what operations and processes are necessary.

These types of experiments are interesting to me, because they represent a good example of hand-coded, designer supplied strategies. AI systems are typically built to solve specific tasks, and the designers who put these systems together already have a good idea of some of the steps the system may take in solving problems. Thus, these steps are built into the system.

The first research question I addressed for this dissertation dealt with investigating the extent to which information processing based strategies can be used to reason through given visual reasoning tasks. The experiments I performed in answering these questions were, thus, sufficiency experiments. For the rest of this chapter, I will be describing experiments that formed the basis of my larger strategy learning work by allowing me to test my ideas and investigate which sets of operations were sufficient for reasoning in the tasks I studied.

My experiments were performed in the task domains of the Punched Hole Paper folding task (Ekstrom & Harman, 1976) (see Section 2.5.1), the Block Design Test (Kohs, 1920) (see Section 2.5.2), and the Leiter Intelligence Scale-Revised (Leiter-R) (Roid & Miller, 1997) (see Section 2.5.3). Although my goal for these experiments were mainly to explore how hand coded generated strategies can be implemented and analyzed, I was also able to demonstrate the sufficiency of imagery as a representation for these tasks.

In summary, the following outcomes were obtained from the work described in this chapter:

- Work on the punched hole paper folding task showed how certain tasks are more amenable to being reasoned about with imagery representations (Ainooson & Kunda, 2017) when compared to purely symbolic approaches. The model for this task was built around a reasoning strategy that involved recursively folding a three-dimensional model of a paper.
- Work on the Leiter-R showed that a small set of imagery operations were sufficient for reasoning through problems from a battery of related visual tasks (Ainooson et al., 2020).
- On the Block Design Task, I developed an imagery based reasoning model that worked by combining different sub strategies for various parts of the task (Ainooson et al., 2020).

3.1 A Computational Model for the VZ-2 Paper Folding Task

One possible strategy for reasoning through the VZ-2 Paper Folding Task may be to have a mental model of the paper that can be manipulated according to the problem being solved. I implemented a computational model which relied on a similar strategy by manipulating a three-dimensional representation of a piece of paper, as required by a given paper folding problem, to make predictions. This three-dimensional paper representation was implemented as a stack of same-sized two-dimensional bitmap images. Four different operations were then used to manipulate this stack to generate solutions for items in the VZ-2 paper folding task. The model for reasoning about the task was implemented in the Python 3 programming language (Van Rossum & Drake, 2009), with image operations provided by the OpenCV library (Bradski, 2000).

3.1.1 Inputs and Pre-processing

Inputs to the model were taken from carefully re-drawn versions of the VZ-2's original test images. Instead of scanning the originals, I chose to redraw all inputs because this model's

goal was not for studying perception—a problem that is significantly complex in its own right—but to try and understand how images can play an active role in the reasoning process.

Sticking to the format of the original VZ-2 items, the input images used by the model were also line drawings. However, unlike the original items, in the redrawn versions, sections of the image meant to represent solid pieces of paper were filled out in a different colour. Also, all image inputs to the system were converted into a binary format through a thresholding operation. In the thresholded binary image, pixels meant to represent space occupied by paper were assigned a value of 1 (*True*), and those for empty spaces were assigned a value of 0 (*False*). The use of binary images made it simpler to use boolean operations for low level image manipulation and also to perform some other computations. Figure 3.1(a) shows the redrawn version of the inputs from Figure 2.5, and Figure 3.1(b) shows the final binary images.

Whenever the system was solving a test item, the pre-processed inputs were presented to the model in three stages. The first stage consisted of images for the fold sequence, the second stage contained the punched image, and the final stage was a series of images representing possible answer choices.

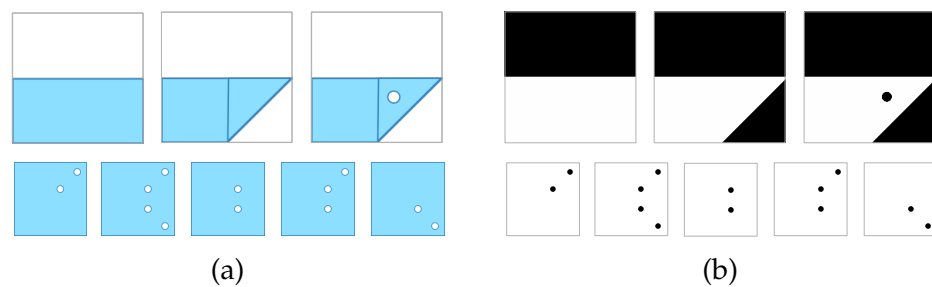


Figure 3.1: Image inputs at different stages of processing for the paper folding task. (a) The redrawn inputs as they were presented to the model. The top row shows the sequence of input folds, and the bottom row shows the possible answers. (b) The results of a thresholding operation on the input images, a final step before they were fed to the model.

3.1.2 Core Operations

A paper folding problem, x , can be considered as a tuple of fold inputs $f_1 \dots f_n$, a punch input p , and five possible answer options $a_1 \dots a_5$, all of which are images. Formally, a full problem can be represented as $x = \langle f_1, \dots, f_n, p, a_1, \dots, a_5 \rangle$, where n is the number of fold steps in the problem.

In the reasoning model, a stack of images, S_p , was used to represent the paper, and to keep track of all operations performed, a second stack, S_l , was kept. The reasoning strategy for the model worked to solve items with the help of four main operations, named *Initialize*, *Fold*, *Punch*, and *Unfold*. These operations are described as follows:

3.1.2.1 The Initialize Operation

For every test item, the *Initialize* operation was first performed to create both the image and operation stacks, S_p and S_l . When executed, it placed a square image filled with each pixel set to a value of 1, l_0 , which represents a blank sheet of paper, on the image stack such that: $S_p \leftarrow l_0$. Additionally, it reset the operations stack by emptying it out, $S_o \leftarrow \emptyset$.

3.1.2.2 The Fold Operation

This operation was executed once for every fold input, f_i , to simulate a fold in the paper. Because the size of the stack, S_p , represented the third dimension of the paper, every image on the stack was a layer of folded paper. For each layer, l , of folded paper, the fold operation proceeded in the following steps:

1. The folded flap was detected.
2. The layer's image was replaced with one that depicted the fold input's effects.
3. A fold line around which folds would occur was estimated.
4. The folded flap was mirrored across the estimated fold line.
5. The mirrored fold flap was added to the image stack as a new fold layer.

In detecting the folded flap, an image, b_i , which contained a filled rectangle obtained from the smallest bounding box of the contents in the fold input, f_i , was first generated.

Then, for each image, l_i , on the stack, S_p , the corresponding folded flap was found by computing the fold input's inverse, f_i , masked by b_i , such that a new list of fold flaps, A , was generated by $A \leftarrow \{l^{fold} : \forall l \in S_p \ l^{fold} \leftarrow \neg l \cap b_i\}$.

The effects of the fold input on each layer on the stack, S_p , could then be computed as the intersection between the fold input, f_i , and the corresponding layer on the stack, l , such that $S_i \leftarrow \{l' : \forall l \in S_i \ l' \leftarrow f_i \cap l\}$.

With the list of fold flaps and the fold inputs' effects computed, the actual fold operation could be finally performed by mirroring each fold flap image from A across a fold line. Because a single fold line was going to be sufficient for all layers, one image from A and another from the stack S_p could be combined to find the fold line.

To actually find the fold line, the selected layer from the stack was morphologically dilated with, a 3×3 structural element, d . The dilated layer was then intersected with the selected fold flap image to yield the image of a line. The two extreme pixels of the resulting line image could then be used as the coordinates of the fold line. These coordinates were placed on the second operations stack, S_l , to be used later during unfolding. Formally, this procedure for obtaining the fold line's image can be expressed as: $\exists l \in S_i \exists k \in A [f \leftarrow (l \oplus d) \cap k]$.

For the final step of the fold operation, each flap image in A was mirrored across the fold line, and the resulting image was put back on the image stack, S_p , to represent the next set of layers that could be possibly manipulated by subsequent fold inputs.

3.1.2.3 The Punch Operation

This operation took the punch input, p , and intersected it with each of the images on stack, S_p , to simulate the punch going through all layers of the paper.

3.1.2.4 The Unfold Operation

Unfolding worked by recursively reducing images from the image stack, S_p , while simultaneously simulating unfolds by reversing the mirror operations using coordinate information from the operations stack, S_l . During calls of the unfold operation, a temporary image stack, S'_p , was created to hold the layers as they were unfolded. First, coordinates of the

most recent fold line was popped off the operations stack to help guide reversals. Then the two items at the top and bottom of the image stack, S_p , were extracted, after which the bottom image was mirrored across the fold line to simulate unfolding. The reversed flap could then be combined with the top image to create an unfolded layer, which was then added to the temporary image stack, S'_p . This process of popping the first and last images, and unfolding through mirroring continued until the image stack, S_p , was empty. Once S_p was empty, the unfold operation terminated and either returned a completely unfolded paper if there was just a single item on the stack, or recursively called itself (the *Unfold* operation) with the temporary stack, S'_p , as the new image stack, S_p .

3.1.3 Walk-through of a Sample Problem

For a walk-through of the model's strategy, consider the problem presented in Figure 2.5. To solve this problem, the first step would be to convert the images, as displayed in 2.5, into binary images through the process shown in Figure 3.1. Afterwards, the problem would be presented to the model, formatted as such:

$$x = \langle f_1 = \begin{array}{|c|} \hline \blacksquare \\ \hline \end{array}, f_2 = \begin{array}{|c|} \hline \blacksquare \\ \hline \end{array}, p = \begin{array}{|c|} \hline \blacksquare \\ \hline \end{array}, a_1 = \begin{array}{|c|} \hline \cdot \\ \hline \end{array}, a_2 = \begin{array}{|c|} \hline \vdots \\ \hline \end{array}, a_3 = \begin{array}{|c|} \hline \cdot \\ \hline \end{array}, a_4 = \begin{array}{|c|} \hline \cdot \\ \hline \end{array}, a_5 = \begin{array}{|c|} \hline \cdot \\ \hline \end{array} \rangle$$

Solving would start with an execution of the *Initialize* operation to set up the stacks, S_p and S_i , after which there would be two calls to the *Fold* operation. As explained in Section 3.1.2, Figure 3.2 shows the various stages of the fold operation for the second fold input, f_2 .

Once all fold inputs were complete, the punch operation would be performed with the punch input, p . Figure 3.3(a) shows how the various operations affect the image stack. It is obvious from the figure how the stack started with a single image (to represent a blank piece of paper), and how its size doubled each time a new fold input was added. When time to unfold and predict an answer came, this stack would be recursively recombined into a predicted unfold pattern, y , by the *Unfold* operation as shown in Figure 3.3(b).

Finally, with a predicted unfold image, an answer choice would be made by picking from the possible options, a_1 through a_5 , the image that was most similar to the one predicted. Similarity, D , in this case will be measured by the total number of pixels that

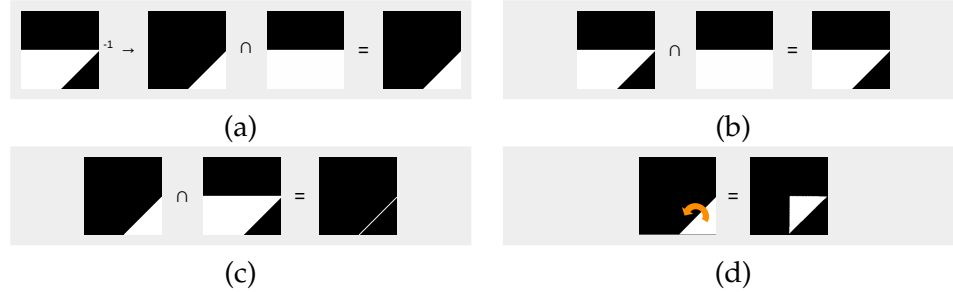


Figure 3.2: The sequence of operations that lead to a fold detection and the ultimate application of a fold. **(a)** A bounded version of the input image is inverted and its intersection with each of the images on the stack is computed to generate the corresponding fold flaps. **(b)** The same input image is further intersected with the images on the stack and the results are morphologically dilated by a single pixel. **(c)** The dilated image from *(b)* is intersected with the fold flap to determine the fold line. **(d)** The image from the fold flap is mirrored along the fold line to simulate a fold.

were exactly the same in both the output, y , and the possible answers, a_1, \dots, a_5 . Formally, the choice, k , would be chosen by:

$$k = \underset{i \in \{1, \dots, 5\}}{\operatorname{argmax}} D(y, a_i) \quad (3.1)$$

3.1.4 Results and Discussion

When evaluated on the VZ-2 (Ekstrom & Harman, 1976), this paper folding model correctly solved all 20 items, and it did so with just the single hand-coded strategy described above. This was not surprising, considering that the model had advantages, like access to pristine redrawn inputs, enough information to help segment test problems, and an almost infinite amount of memory to keep track of all folds.

Redrawing inputs allowed the model to clearly identify which sections of the images represented solid paper and which ones were for empty space. The knowledge required to make this differentiation is one that humans may obtain from experiencing numerous illustrations that are presented as simple line drawings, and physical interactions with paper that occur through other activities.

The complexity of the knowledge required to understand these images is even more obvious when you consider just the positions of the punched holes. Because the model internally simulated a model of a paper being folded, the fold positions were exact, having

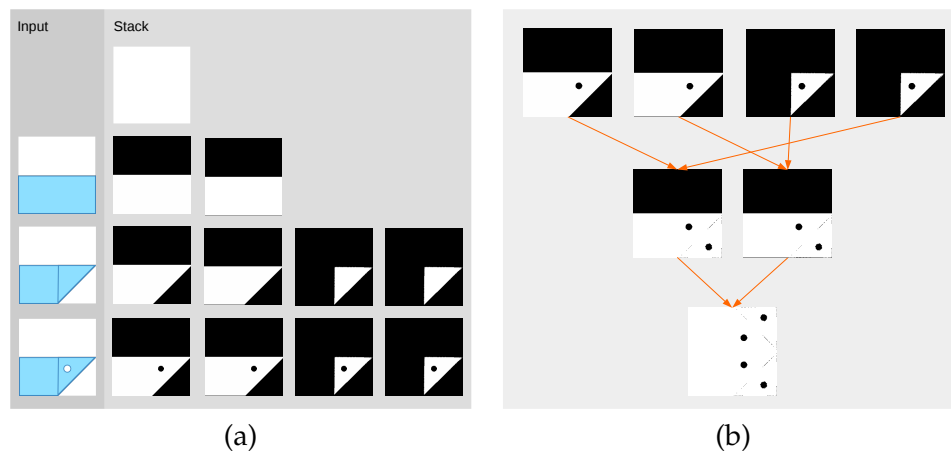


Figure 3.3: **(a)** The sequence of inputs and the effect they have on the image stack three-dimensional representation of the folded paper. **(b)** From top to bottom, this image shows how the images on the stack are combined to represent unfolding the paper.

all the punched holes aligned in the right places. When the original test items from the “Kit of Factor-Referenced Cognitive Tests” are observed, however, these punched holes were approximated and not in the exact locations. In order for the model to perform well on the task with its current strategy, the punched holes had to be realigned when the inputs were redrawn for the system. In fact, still on the issue with punch hole alignment, when the original scanned items were evaluated on the current model, it failed to solve any of the test items.

Due to its use of imagery as a representation for reasoning, this model could work through any arbitrary paper folding task. For example, consider the snowflake simulation performed by the model shown in Figure 3.4. It can be observed that the model generated an output that corresponded exactly with the fold inputs it received.

Although this system was mainly built to evaluate the sufficiency of an imagery based strategy for solving the paper folding task, one important question is still worth asking: could the strategy, as described in the sections above, be used by a human for reasoning through the VZ-2? That definitely cannot be guaranteed. One interesting factor to consider is how the model uses memory when reasoning through items on the task. For every single fold, the number of layers the system has to keep track of doubles. If a human had to use this exact strategy, and the stack is considered to be analogous to a person’s working

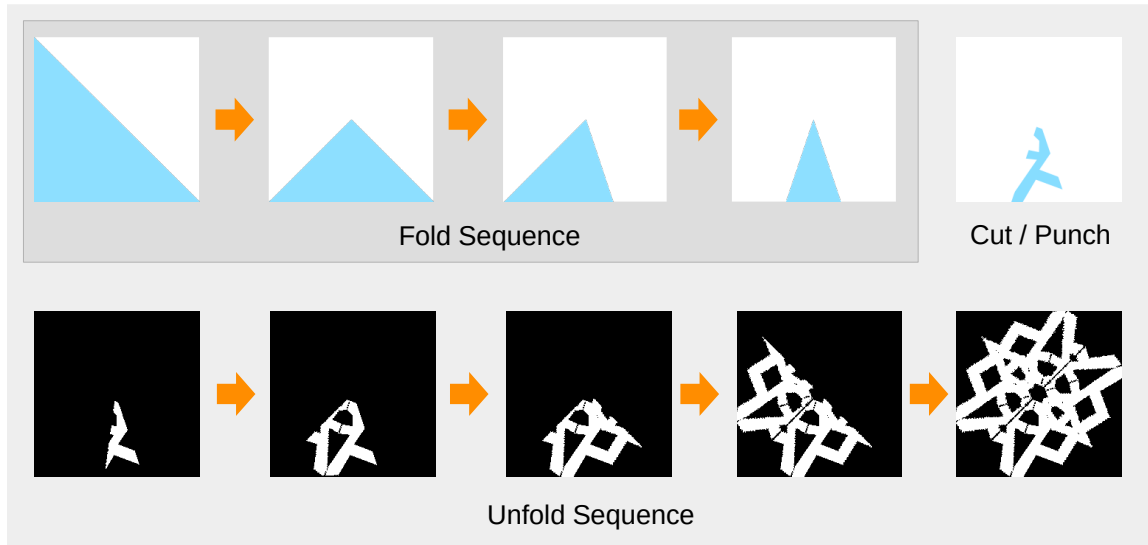


Figure 3.4: The solution to an arbitrary punched-hole paper task, which simulates a paper snowflake. The top row of images show the input to the model, with the blue sections representing solid sections of the paper. The bottom row shows the unfolding sequence as generated by the model.

memory, they may have to be keeping track of an exponential amount of items for every fold.

Short-term memory in humans is believed to be very limited. Miller (1956) places it at seven items plus or minus two, but more recent work from Cowan (2001) shows it could simply be just about 4 items. And these limits come about from considering short-term memory as a single store for all kinds of information regardless of modality. When Baddeley and Hitch (1974)'s model of working memory, which uses different types of stores for different modalities is considered, an analysis by Luck and Vogel (1997) shows that people may be integrating the features of individual objects (like colour, shape, size, etc.) when storing them. Essentially, people can remember a few objects at a time, but can remember a lot about each of those items through these integrations.

Items on the VZ-2 have anywhere from 1 to 3 folds per problem. In sticking with the assumption that the model's stack is stored in a person's working memory, people may have to keep track of anywhere from 2 to 8 items. This figure is still within the reasonable range of working-memory in humans. It could also be considered that people may be integrating folds as features of the various input images, as described by Luck and Vogel

(1997), and as a result may not be using as much space in their working memory.

3.2 Building Models for Leiter-R Tasks

The Leiter International Performance Scale-Revised (Leiter-R) is a battery of non-verbal intelligence tests for children and young adults Roid and Miller (1997). Being a test of visual reasoning in humans, the Leiter-R is well positioned as a task domain for testing computational models that reason visually. Additionally, with most problems in the Leiter-R presented through diagrams and pictures, opportunities to evaluate reasoning strategies based around imagery representations can also be realized. This also means it should be possible, just as it was with the punched-hole paper folding task, to formulate reasoning models based on fixed hand-coded strategies for the Leiter-R that do a bulk of their reasoning through image representations.

As a test of this hypothesis, I conducted a series of experiments to find out the extent to which imagery representations and their associated operations were sufficient for reasoning through items on the Leiter-R (Ainooson et al., 2020). The primary goal of this work was to build models that reasoned through problems using any possible strategy that can be implemented through computer code, without an attempt to mimic any human-like cognitive capabilities. The choice of using a common framework of image processing operations also meant I could later analyse the strategies to determine the common patterns in which operations were used throughout the model.

3.2.1 Implementation Details

The reasoning strategy for an item on the Leiter-R was expressed as a sequence of visual reasoning operations (v_0, v_1, v_2, \dots) interspersed with control operations (c_0, c_1, c_2, \dots) ¹. Visual operations took images as input, and output either a derived image or a number (representing some property of the image). Control operations, on the other hand, determined the sequence in which visual operations were executed—they worked similarly to control statements like loops and conditions typically found in high level programming

¹Although the use of control operations remains in this description, in the actual implementation I found out that it was much easier to use the native control flow structures supplied by the Python programming language.

languages. The operations used for this study are described in the sections below.

3.2.1.1 Similarity

The similarity operator, $similarity(a,b) \rightarrow r[0,1]$, took as input two images, a and b , and outputs a real number between 0 and 1. This output number represented how similar the images were, according to a metric in which higher values indicated a higher similarity. To allow experimentation with different similarity metrics, the operator provided in the framework acted as a front end to different internal metrics. There were two such metrics on the back-end: Jaccard similarity and Euclidean similarity.

Jaccard similarity, $similarity_j(A,B)$, was computed directly with pixels from two images, A and B , as a ratio of their number of intersecting pixels to the number of union pixels.

$$similarity_j(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Euclidean similarity, $similarity_e(A,B)$ was based on the euclidean distance, $E(A,B)$ between both images. It was computed as:

$$similarity_e(A,B) = \frac{1}{1 + E(A,B)}$$

3.2.1.2 Containment

Through the containment operator, a measure of the relative size difference between two objects in separate images could be obtained. Containment took two images, a and b , and a colour, g , as input. It then returned a real number between 0 and 1, which could be considered as an estimate of the size of the object in the image a relative to that in b , when the colour, g , was considered as the background of both images.

Another way of looking at the containment operator, is to consider it as a measure of how much an object displayed in one image could be contained by another displayed in the other image. Because this operator measured sizes of objects, and not sizes of images, each image to be compared had to contain a picture of a single item over a plain background with colour g . Interestingly, in this formulation, computing containment could also be

thought of as finding the ratio of the number of non-background pixels in both images (which was how this operator was implemented).

3.2.1.3 Rotation and Scaling

Unlike the other operations, the rotation and scaling operations transformed the input images instead of computing a metric based on them. The scaling operation, $scale(a, f) \rightarrow b$, took an image, a , and a real number, f , as inputs and returned a scaled version of the input. Similarly, the rotation operation, $rotation(a, \theta) \rightarrow a$, took an image, a , and a real number, θ , and returned an image containing a rotated by θ degrees. Because both operations could alter the image's bounding box size, these operations returned an image that was larger than the input in most cases. For example, rotation will always increase the bounding box size of the image.

3.2.2 Inputs and Pre-processing

When it came to implementing the reasoning models, all image inputs for the experiments were scanned from the original Leiter-R items (easels, answer cards and other booklets). These scanned images were further annotated with rectangles to highlight the areas of interest that any solvers may need to pay attention to. For the multiple-choice easels, annotations were made on the locations of the slots where cards would be placed, and for the answer cards, annotations were placed on the single central image contained on the card. Other test items, which did not follow the easel and card format, were annotated subjectively according to what I felt may need attention.

3.2.3 Results of Sufficiency Experiments

Using a framework of operations and inputs described previously, I was able to obtain programs for solving 17 of the 20 subtests on the Leiter-R². The three subtests for which I failed to obtain any programs were Paper Folding, Picture Context, and Attention Divided. See Figure 3.5 for a breakdown of the results.

²Sixteen of these programs were written over a weekend hackathon session, involving myself and two other PhD students, Deepayan Sanyal and Joel Michaelson, while the program for the Figure Ground task was adapted from an earlier study on the Leiter-R by Palmer and Kunda (2018).

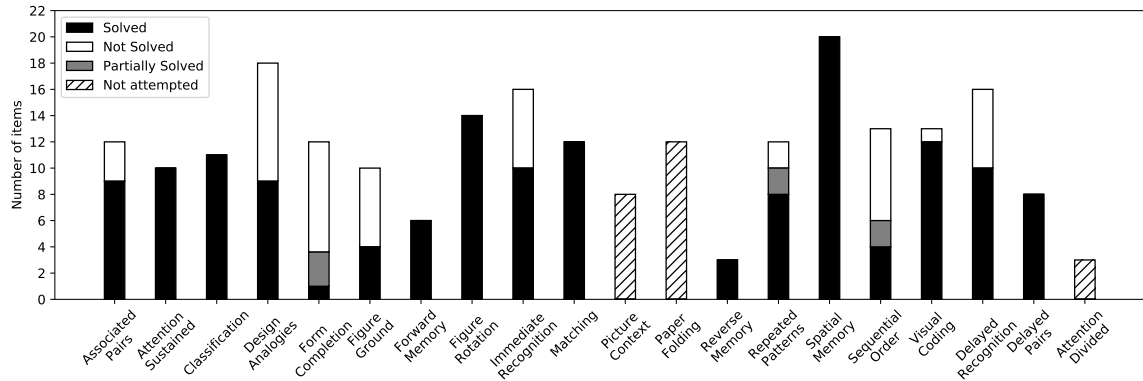


Figure 3.5: Results of the Leiter-R sufficiency experiment.

3.2.4 Discussion of Sufficiency Experiments

From the results, it can be seen that every subtest from the Attention and Memory battery was completely solved, while little progress was made on most items from the Visualization and Reasoning battery. Solving all items on the Attention and Memory battery appears to be a trivial task for computers, especially since these tasks mainly require people to remember facts for later recollection. Unlike the limited, volatile working memory of humans (Cowan, 2001; Luck & Vogel, 1997; Miller, 1956), computers can indefinitely hold on to whatever is stored in their memories. Although it may seem “pointless to apply memory tests to a computer model” as Hernández-Orallo et al. (2016) puts it, and rightfully so, in the context of visuospatial reasoning, valuable observations can be made when these tests are applied in environments where human-like attention and memory are simulated.

It was particularly difficult to develop reasoning strategies for subtests in the Visualization and Reasoning battery due to the particular way in which the Leiter-R deals with task difficulty. Although all items in a subtest fall within a particular task theme, changes in difficulty tend to be introduced through subtle, and sometimes drastic modifications to task requirements. For example, the paper folding task in the VR battery starts with simple two-dimensional items that have single folds, then progresses to multiple-folds in two-dimensions, then folds become three-dimensional cuboid cut-outs, which are then modified to require test takers account for differently coloured faces, before finally ending with irregular geometric shapes in three dimensions.

The most interesting observation from these experiments was how the few operations described in Section 3.2.1 were able to provide strategies for working through some items across most of the Leiter-R's subtests. Of all these operations, *Similarity* was the most important in terms of implementation. Because every test item required some form of visual comparison to be made, variations in results were observed for the different similarity metric implementations. Of the two metrics available for these experiments, Jaccard was the most accurate.

For both underlying similarity metrics to exhibit a decent performance, some pre-processing of the scanned input images had to be performed. Especially for the Jaccard metric, images needed to be properly aligned and had to be of the same size; any inconsistencies in image orientation introduced through the initial scanning process had to be manually corrected before any images were able to produce useful results. Additionally, before every comparison was made with any of the metrics, images had to be cropped and resized to match each other in size, and the colours had to be flattened. See Figure 3.6 for a visual description of the image pipeline.

In applying the Jaccard similarity, cropping was applied by finding the smallest bounding box, and colour flattening involved converting all colours from 8 bits per channel in the input RGB image to a single bit per channel RGB. With a single bit per channel, all possible colours were reduced from a potential 16 million to just 8, effectively reducing the margin by which errors could occur during comparisons.

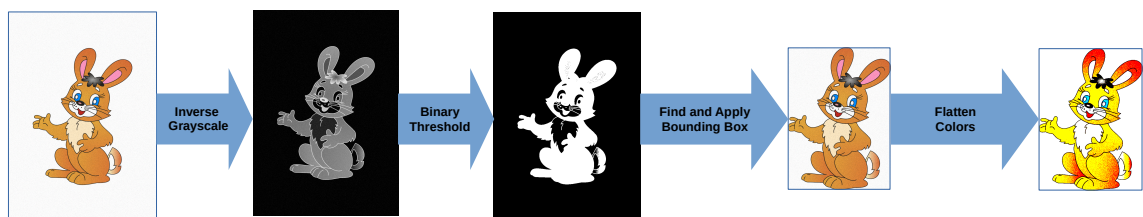


Figure 3.6: The sequence of steps an image goes through before every comparison is made with either metric.

3.3 Visuospatial Reasoning Environment for Experimentation

The experiments I have described so far were performed with systems built on fixed, hard-coded strategies. One of the primary goals of this work was to study strategy differences, and to that effect, I worked on an integrated environment for conducting visuospatial reasoning experiments (Ainooson et al., 2020). This environment, named the Visuospatial Reasoning Environment for Experiments (VREE), was organized as a virtual environment in which agents interacted with objects, and was based around ideas expressed in Kunda (2017).

Structurally, VREE contained an Environment, Agents, Affordances and Objects. Figure 3.7 presents a simplified block diagram of VREE that shows its internal components and how they interacted.

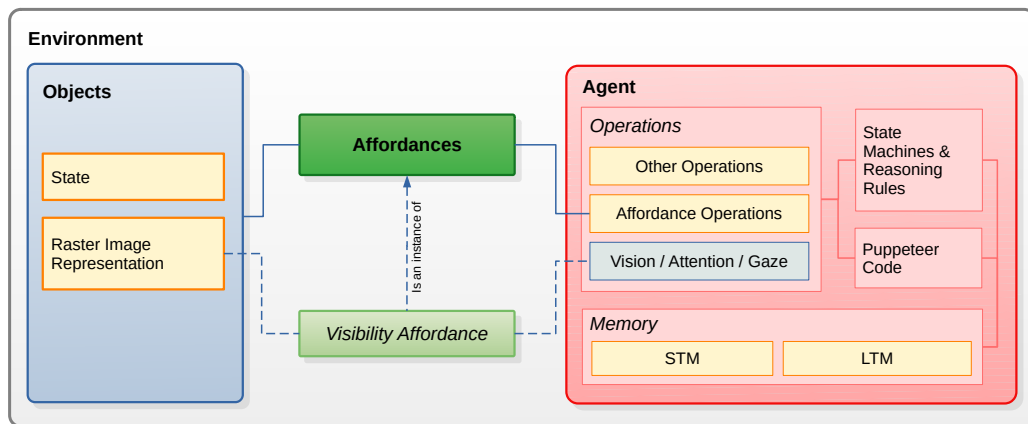


Figure 3.7: A diagram of the VREE system showing how the different components existed within the environment, and how they interacted with each other.

3.3.1 The Environment

VREE's environment was the world in which all experiments took place. The environment contained objects, agents, affordances, and a *TimeKeeper*. Agents performed the intelligent actions in VREE, objects passively existed to be manipulated by agents, affordances constrained agent-object relationships, and the *TimeKeeper* coordinated the entire environment by defining the sequence in which agents executed their reasoning logic. Formally, the environment can be defined as: $Environment \rightarrow \langle \mathbf{Objects}, \mathbf{Agents}, \mathbf{Affordances}, \mathbf{TimeKeeper} \rangle$

Being a framework for visual reasoning experiments, VREE’s environment could be rendered as an image. This image was produced as a two-dimensional top-down view. The choice of a top-down view was made for two primary reasons:

1. A two-dimensional space reduced the complexity of code required to handle perception and other vision related computational workloads.
2. The tasks to be studied in VREE were mostly table-top tasks that people most likely performed while hunched over a table—essentially giving them a top-down view.

The *TimeKeeper*, a simple routine that advanced agents in fixed time-steps, tied the environment and its components together. Within a time-step, all qualifying affordances—those whose conditions for execution were met—were activated, and agents were given the opportunity to make their decisions about the current state of the environment. Algorithm 3.1 shows the pseudo-code for the *TimeKeeper* routine. Note that in the implementation of VREE described in this chapter, there was only a single agent in the environment at any time.

Algorithm 3.1: *TimeKeeper* routine from the Environment that coordinates the activities of agents, affordances and objects

```

do
  foreach object ∈ Environment.Objects do
    foreach affordance ∈ Environment.Affordances do
      if affordance.AgentType =
        affordance.Type(agent) ∧ affordance.ObjectType = Type(object) then
        | add affordance.Actions to agent.Operations
    execute Agent's next step
  forever

```

3.3.2 Objects

When agents were performing any tasks, they had the ability to manipulate objects in the environment. Objects were the primary interface through which agents performed their tasks. Through objects, agents received all information needed for tasks, and agents could equally communicate the results of their reasoning actions by manipulating same objects.

All objects in VREE were assigned specifically defined *object-types*. An object's type characterized the structure of its internal state, which always included a raster image representation for visualizing the object. Additionally, because objects had to be visible in the environment, a two-dimensional world coordinate that specified an object's position was also present in the object's state. The state's raster image was what agents perceived when they attended to the object. An object in VREE can formally be defined as: $Object \rightarrow \langle Type, State, Location, Image \rangle$

3.3.3 Affordances

Generally, affordances can be thought of as the relationships existing between objects and agents that allow agents to "use" objects (Norman, 2013). In VREE, I adapted this definition to make affordances represent the knowledge agents had about objects of a particular *object-type*. Within VREE, for any defined *object-type*, affordances could be used to specify the actions agents could perform when interacting with objects of the type. Additionally, a physical variant of affordances existed in the environment which allowed objects to interact with each other. This type of affordance allowed for the definition of passive interactions between two objects (such as physical collisions or overlaps). Thus, in VREE, affordances could be defined between agents and object-types as *agent-object* affordances, or between any two object-types as *object-object* affordances.

Every affordance contained a condition and a set of actions. The condition governed when the affordance's actions could be performed, and were usually defined over the internal states of the related components (agents or object-types). Whenever an affordance's conditions were met, the related operation executed the affordance's actions, which further altered the internal states of any related components. For object-object affordances, operations were executed immediately conditions were met. On the other hand, operations for agent-object affordances are optionally executed at the agents discretion.

An affordance can formally be defined as: $Affordance \rightarrow \langle Conditions, Actions \rangle$

3.3.4 Agents

Agents performed the intelligent actions in the VREE system, and in some ways, they could be considered as mini cognitive architectures. Each agent contained a collection of operations, different forms of memory for storing information, and various mechanisms for executing reasoning rules. Also, due to the visual nature of the tasks that were intended to be performed in VREE, every agent had a perception system.

3.3.4.1 Operations and Affordances

Operations in VREE's agents were analogous to function calls in computer programs. Whenever an operation was executed, some pre-defined actions of the agent were performed. Multiple operations could be put together to implement reasoning rules. Because every agent had a built-in perception system, agents were additionally equipped with operations for controlling gaze and attention. Whenever an agent was being defined (or implemented), additional operations for other agent specific abilities could be added.

To simplify the construction of agents, affordances were transparently applied as though they were operations. Whenever an affordance's conditions were met, the agent had the opportunity to execute the affordance's actions just as it would any of its operations. This transparency was meant to reduce the complexity required to build agents, since agents only needed to know about operations they could perform in a time step and nothing about affordances.

3.3.4.2 Memory

When reasoning through tasks, agents had access to different forms of memory for storing information: There was a short-term memory for storing temporary facts, a long-term memory containing built-in knowledge, and a spatial memory for keeping track of the locations of interesting items. Although the long-term memory of the agent was typically hand-populated while designing an agent, there was an option for agents to persistently store facts from short term memory into long term memory during execution.

3.3.4.3 Reasoning Rules

An agent's reasoning rules represented its model of reasoning about any tasks in the environment. There were primarily two ways in which reasoning rules could be specified: They could either be through state machines or through scripts written in general purpose computer programming languages (like python). Regardless of how reasoning rules were specified, agents always worked by taking visual input from the environment, making deductions through its internal rules, and communicating their results through affordances that may manipulate objects in the environment.

3.3.4.4 Vision Attention and Gaze

Since the tasks to be reasoned about were all visual in nature, agents had a gaze window through which they could visually attend to objects. The gaze window's size is variable, and it can be moved to any location of the environment. Gaze movements are controlled by the agent, and the decisions about where to pay attention are influenced by a salience map. Much like peripheral vision in humans, the salience map told the agent where interesting things in the environment may be, without providing any fine details.

Two visibility affordances in the environment were responsible for managing the gaze and vision system. There was a salience map affordance that updated the agent's salience maps with the coordinates of objects after every timestep, and a collection of gaze affordances that allowed an agent use to move its gaze to the different locations highlighted on the salience map. The salience map affordance was always active, and the gaze affordances only became active when the object to which a particular gaze affordance was linked to, was not obscured in an agent's view. As far as implementation went, a hash, which mapped an object's unique identifier to the coordinates of its location, was used for the salience map.

Gaze affordances were not always mapped one-to-one between objects and agents. Some objects, such as a simulated piece of paper displaying a puzzle, could have multiple areas of interest. In such cases, an object could supply an additional salience map of its local areas of interest. Any object supplied salience maps were merged with the main map to present a transparent distribution of gaze targets.

3.4 Exploring Strategy Differences on the Leiter-R with VREE

As an initial experiment to test out studying strategy differences through VREE, I investigated how different constraints placed on agents affected their performance on selected Leiter-R subtests. For this experiment, I evaluated two different high level reasoning strategies that were executed as slight modifications to the original strategies defined in Section 3.2.3. These high level strategies altered the original strategies by standardizing how answer choices were provided for some tasks in the Leiter-R.

In the first of these strategies, the agent starts with an option from a card and applies a task specific strategy to find a compatible easel slot. I called this the *card-to-slot* strategy, and it is akin to how people may attempt multiple choice problems by working from their answer choices to the question.

Conversely, for the second strategy, the agent starts from an easel slot and works towards finding a matching card, similar to how people may also attempt multiple choice problems by working from the problem to the answer choices. This second strategy was called the *slot-to-card* strategy.

3.4.1 Selected Sub-tasks and their Specific Strategies

To test out these ideas on strategy differences, I focused on six subtests from the Leiter-R. These were Associated Pairs, Figure Rotation, Matching, Repeated Patterns, Sequential Order and Visual Coding tasks. All the tasks, except for the Associated Pairs, come from the visualization and reasoning battery of the Leiter-R. As already stated, these subtasks each had their own task specific strategy (which were earlier developed through the work described in Section 3.2.3). Each of these subtests and their associated strategies are explained in the following sections.

3.4.1.1 The Associated Pairs Subtest

Associated Pairs is a memory test in which a subject is required to memorize the relationship between arbitrary pairs of images. When this test is administered, the subject is exposed to an easel page containing a number of image pairs for up to 5 or 10 seconds (depending on the difficulty of the task). After this exposure, the subject is presented with one or

two easel pages that contain some (or all) of the image pairs from the previous easel page with their counterparts removed. The subject's task is to supply the other missing halves. There could be anywhere from 2 to 12 pairs displayed, and the subject may be required to recollect up to 8 of these. See Figure 3.8 for a sample of this task.

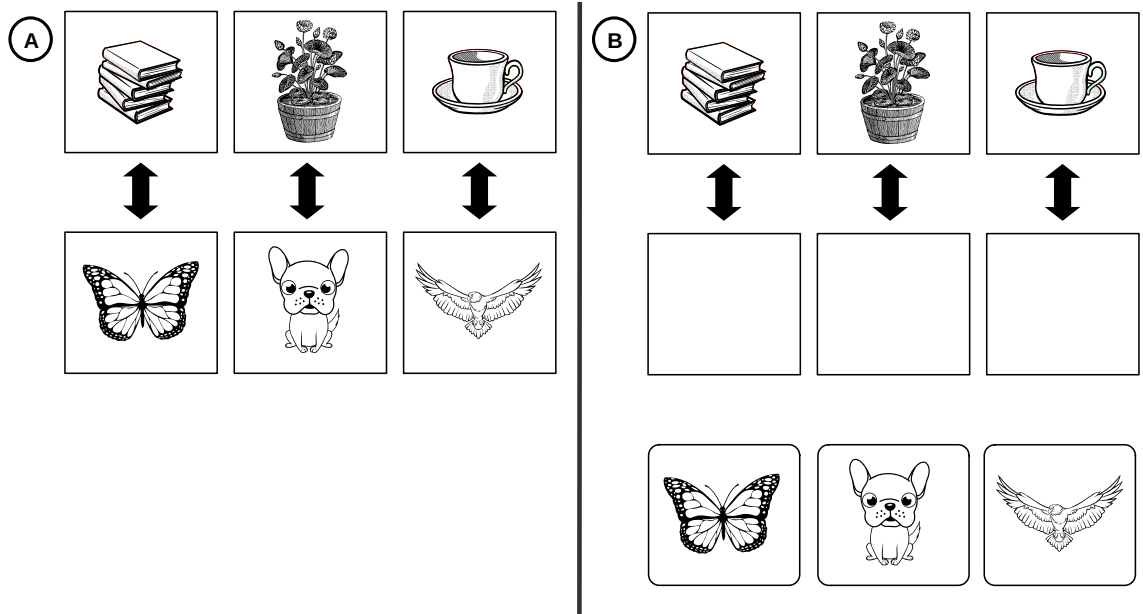


Figure 3.8: A sample item from the Associated Pairs task from the Leiter-R (Roid & Miller, 1997). In this illustration, the section labelled A will be briefly exposed to the subject, then the section labelled B will be exposed while the A section will be hidden. To protect the secrecy of the Leiter-R, this sample is not a real Leiter-R problem.

The reasoning strategy used for the associated pairs task simply recorded all pairs in the agent's short term memory. When the page was turned for testing, the stored pairs were looked up, and matches were made with the similarity operation to find which card best fit each of the slots.

3.4.1.2 The Figure Rotation Subtest

The figure rotation subtest requires subjects to match easel slots with cards, such that images on both card and easel slots are rotated counterparts of each other. An example of the Figure Rotation is displayed in Figure 3.9.

The original figure rotation strategy (as defined in Section 2.5.3) compared pre-rotated versions of a test's image with their unmodified counterparts to find a match. This com-

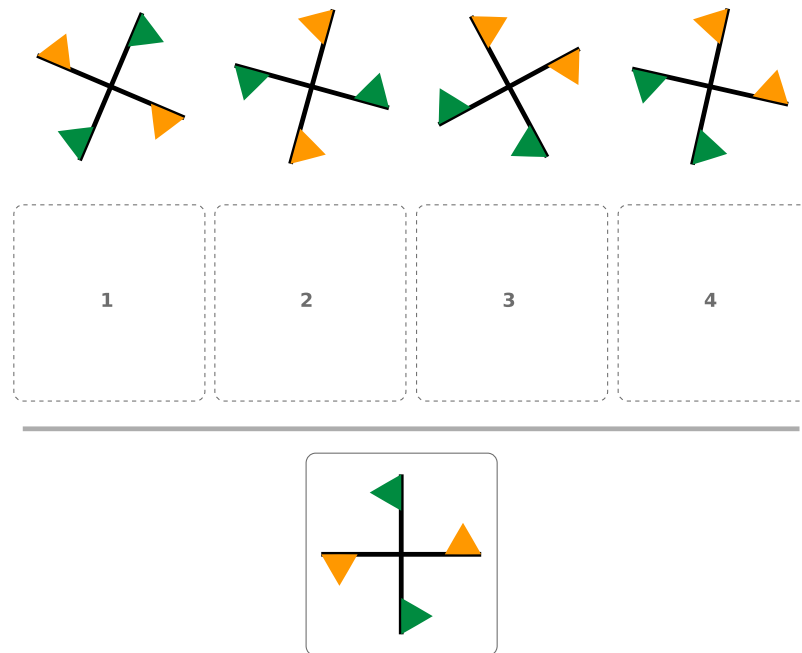


Figure 3.9: A sample item from the figure rotation task of the Leiter-R (Roid & Miller, 1997). In this instance, the subject will be required to select one of the four slots in which the card at the bottom best fits. To protect the secrecy of the Leiter-R, this sample is not a real Leiter-R problem.

parison was executed iteratively, such that if a match was not found after any iteration, the test image was further rotated by a fixed angle before it was tested again. A set of iterations were run until the test image had been rotated over a complete 360 degree cycle. If a match was still not found after the complete cycle, the angle by which test images was rotated after each trial was halved, and the whole process was repeated until a match was found or the angle of increment became less than 1° , by which time a match should have been found.

3.4.1.3 The Matching Subtest

The matching subtest requires subjects to find images from their cards that best match with others on the easel. The original reasoning strategy for this subtest relied on the similarity operator to perform a simple matching with a similarity value that exceeds a threshold. See Figure 3.10 for a sample of this task.

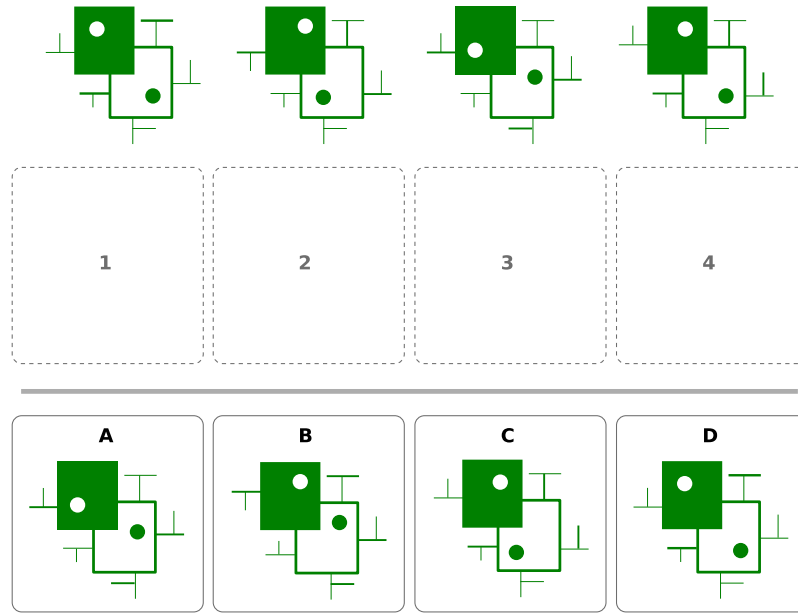


Figure 3.10: An instance of the matching task from the Leiter-R (Roid & Miller, 1997), with 4 slots and 4 answer cards. To protect the secrecy of the Leiter-R, this sample is not a real Leiter-R problem.

3.4.1.4 The Repeated Pattern Subtest

In the Repeated Patterns test, subjects are presented with a sequence of images that are arranged on the easel page in a given pattern. The subject's task is to figure out the pattern and complete the blank sections presented on the easel with responses from their cards. See Figure 3.11 for a sample of this task.

The strategy for the repeated pattern subtest involved generating identifiers for all the distinct images on both the cards and the easel slots. These identifiers were generated through a simple clustering technique, where all images having Jaccard similarities within a threshold were given a common identifier. Once the identifiers were generated, the sequence of identifiers were used to find which items best completed the pattern.

3.4.1.5 The Sequential Order Subtest

Much like the repeated pattern subtest, the sequential order requires subjects to complete a sequence of images presented on the easel. The difference here, however, is that the sequence of images change by some property in every step. Changes could be in the size, or orientation of the image, and in extreme cases the frames of an animation sequence.

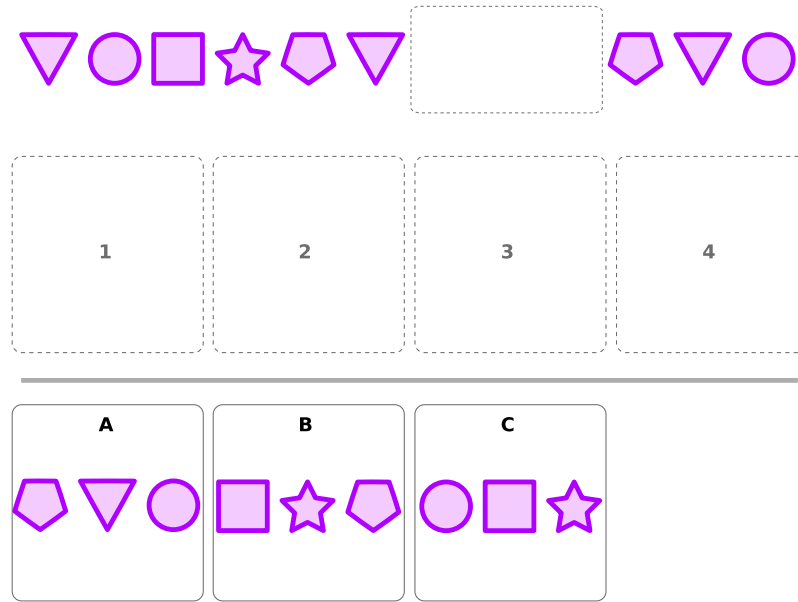


Figure 3.11: An instance of the repeated pattern task from the Leiter-R (Roid & Miller, 1997), which requires the subject to fill in the missing parts of the given sequence. To protect the secrecy of the Leiter-R, this sample is not a real Leiter-R problem.

A metric based on an equally weighted combination of the similarity score (returned by the similarity operation), containment score (which is the value returned by the size comparison operator), and approximate RGB values (converted from 24 bits per channel to 1 bit per channel) between any two images was used as a sorting key to determine the position an item had in the sequence. This strategy only worked for items which had size changes, and not those that featured animations.

3.4.1.6 The Visual Coding Subtest

The visual coding task requires subjects to learn arbitrary rules about how images relate to each other, with which they could solve given problems. In some ways, the Associated Pairs and the Visual Coding subtests are similar; both require associations to be identified and later mapped.

Because of its similarity to the associated pairs task, both Visual Coding and Associated Pairs shared a similar strategy. The only difference between the two strategies was from the storage of image associations. For the associated pairs, because the associations were not going to be available during testing, they were stored in short term memory. For visual

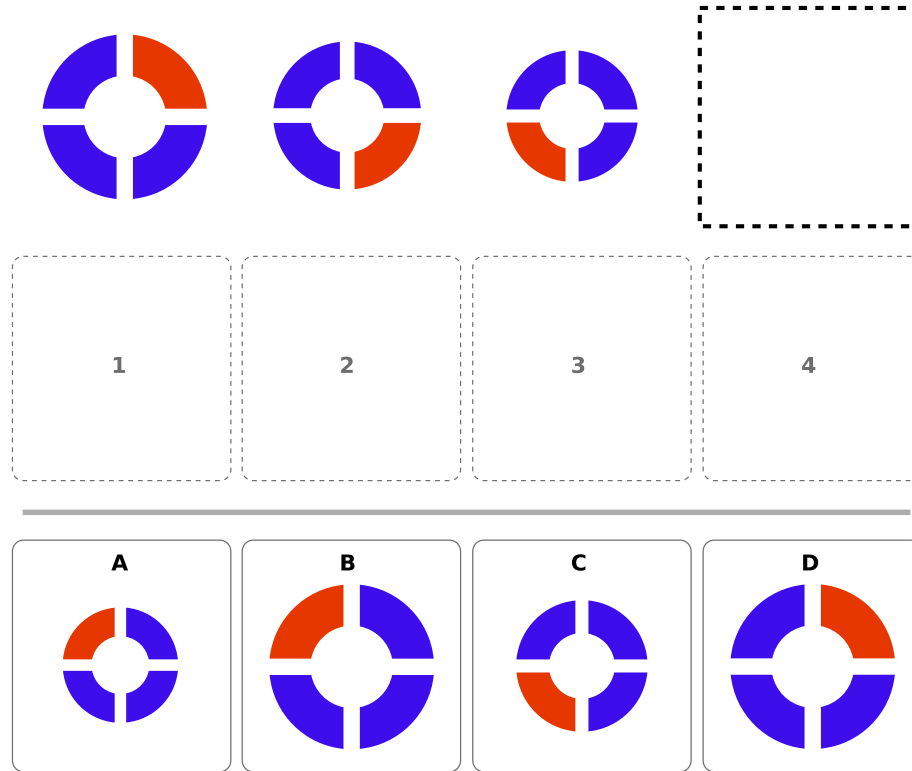


Figure 3.12: An instance of the sequential order task from the Leiter-R (Roid & Miller, 1997), in which the subject must complete the sequence by placing easel cards. To protect the secrecy of the Leiter-R, this sample is not a real Leiter-R problem.

coding, however, the associations were always going to be visible during testing, so the agent could simply attend to it. The associations used in visual coding were never stored by the agent.

3.4.2 Setting up the Environment

Before running experiments in VREE, a few things had to be in place: the environment had to be setup with the task's objects, the necessary affordances required by the agents and objects needed to be established, and the reasoning logic for the agent needed to be configured. For these series of experiments, I created two object-types to represent easel pages and cards, and I established two affordances to allow agents to place cards into easel slots or pick them out.

The reasoning agent was equipped with the operations that were used in the earlier Leiter-R attempt (see Section 3.2.1.) These operations—similarity, containment, rotation

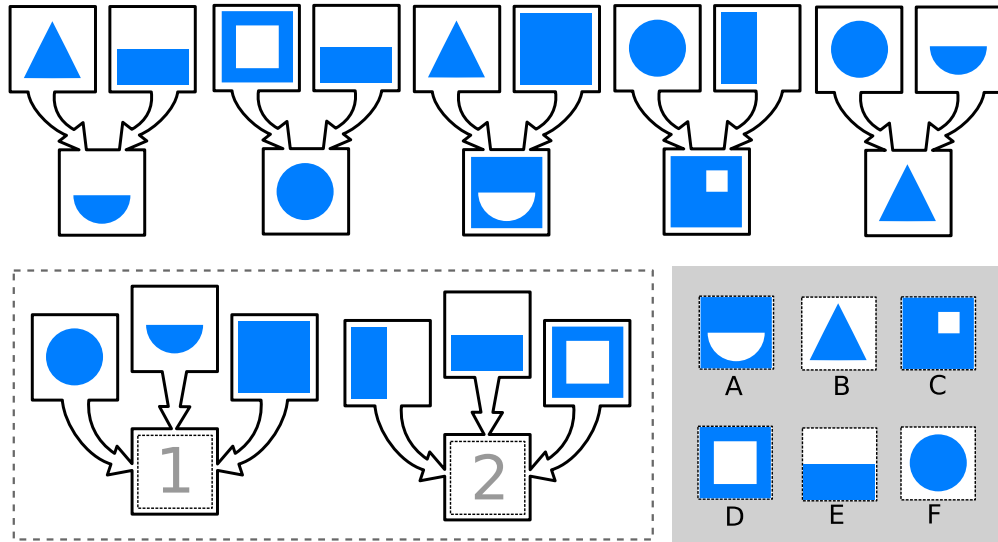


Figure 3.13: An instance of the visual coding task from the Leiter-R (Roid & Miller, 1997), in which subjects are required to provide the cards that best fill the slots labelled 1 and 2. To protect the secrecy of the Leiter-R, this sample is not a real Leiter-R problem.

and scaling—allowed the agent to also use the reasoning strategies developed from the earlier attempt.

During the execution of a task, an instance of the easel page object type was always available to display the problem currently being solved, and as many instances of the card object were also created for all the possible answers of a given problem. Just as it was with the earlier Leiter-R experiments (see Section 3.2.3), pictures for the task items were taken from scans of the original easels and cards. These scans were further annotated to highlight salient regions for driving the agent’s attention.

Finally, to simulate the effects of memory on the different tasks, agents were given the ability to forget items they had in short-term memory. This forgetfulness was implemented such that any image in the short-term memory was gradually corrupted with random noise after every time step, to ensure the entire image was covered with noise after seven time steps.

3.4.3 Walk-through of Agent’s Reasoning Strategies

Let us consider how both strategies (*card-to-slot* and *slot-to-card*) can be applied by working through an example of the Leiter-R’s Matching Task shown in Figure 3.10. As already stated (see Section 3.4.1.3), the strategy for solving the Matching task finds an image on the easel that is most similar to one from a given card through the agent’s similarity operation. When applying these high level strategies, the original strategy—as described in Section 3.4.1.3—is actually the *card-to-slot* strategy. To transform this to the *slot-to-card* strategy, the similarity of images from the slot will be maximized against those from the card instead.

When solving with the *card-to-slot* strategy, the agent picks one card (images labelled A through D) and compares them with those on the easel (images labelled 1 through 4). Assuming the similarity threshold was chosen to be 99, and the agent selected the card labelled A, the largest similarity over the threshold would be at the third easel slot with an exact match. The agent will then move onto the next card labelled B which will fail to meet the threshold for all its similarity values. This process repeats until all cards are exhausted. Similarly, for the *slot-to-card* strategy, the agent could start with slot 1 and test it against cards A to D, with D maximizing similarity over the threshold after all the slots have been exhaustively tested.

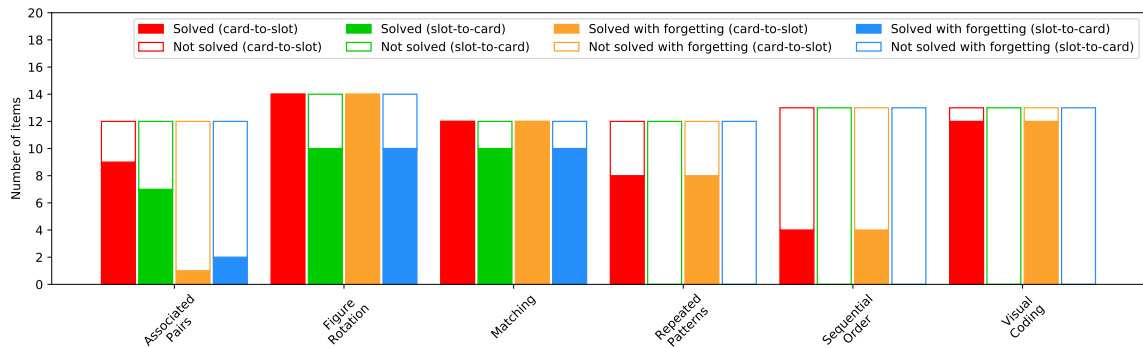


Figure 3.14: Results from running the two different strategies on the six selected Leiter-R subtests, while alternating forgetfulness.

3.4.4 Results and Discussion

Although both strategies—*card-to-slot* and *slot-to-card*—appeared to be similar, and even had the same performance on most of the tasks, there were still differences when performance on the Associated Pairs task was considered. This discrepancy was caused by how the strategy’s choice of goal images placed a bias on search progression. For example, if an image from a card was wrongly matched through the *card-to-slot* strategy, the answer option provided by this card was no longer available to other slots later in the search. In a similar vein, when a wrong problem slot match was made in the *slot-to-card* strategy, the slot was locked up for the remainder of the search. It is worth noting that both strategies could be altered to ensure that choices made could be changed after some reconsideration.

It could also be seen from the results how forgetfulness always reduced performance. Some tasks, like the visual coding and repeated patterns, could not be solved at all when forgetfulness was in place. Although the robustness of similarity metrics could be blamed to an extent, this proof-of-concept implementation showed there was a lot that could be learnt from experiments in VREE when the right questions are asked.

3.5 Exploring Strategy Differences on the Block Design Task with VREE

After the initial work with VREE on the Leiter-R, I performed another set of experiments on the Block Design Task (BDT). The tactile, non-verbal nature of the BDT made it a good candidate for experiments in VREE. In the BDT, there are blocks that can be modelled as objects for agents to manipulate through affordances and there are gaze targets to which agents can attend for the necessary information to perform their tasks.

To accommodate all the block design’s physical properties, I defined two object types in VREE. The first type, *Block*, was used to represent the blocks, and the second type, *Design*, was used to represent a space for displaying the design to be replicated.

The environment for BDT experiments in VREE was set up as follows:

1. Each block had the name of the upward face as its internal state. See Figure 3.15 for a description of this naming convention.
2. The agent’s hand was simulated by an internal state variable, *HandContents*, that



Figure 3.15: The internal representation used to store block states and transitions for the block design task.

could either be empty or contain a reference to the block currently being held by the agent.

3. The design was represented in the environment with a *Design* object whose internal state held the size and an image of the design. To make perception easier, the design image was pre-segmented into blocks (through the salience affordance) and the image on each block's face was also stored in the internal state.
4. The agent's long term memory contained information about the location in the environment from which blocks could be picked, also known as the block bank, and also the location where blocks could be placed for construction. To further help with properly aligning the blocks during assembly, the location of the construction area was broken up into cells.
5. A symbolic model of a block was also stored in long term memory. This model provided a description of each of the block faces and the transitions between them. See Appendix B for a visual representation of this model and associated state transitions.
6. There were ten affordances in the environment to allow the agent to manipulate and move blocks around. See Table 3.1 for a full listing of all these affordances.

Table 3.1: All 10 affordances used in the block design environment for VREE

Affordance	Description	conditions
Pick Block from block bank	Picks a block from the block bank and puts it in the “hand” of the agent. This affordance is only active when the agent’s “hand” is empty. There are sixteen instances of this affordance, with one for each block.	$\neg Agent.BlockInHand$
Drop block to block bank	Drops a block that was earlier picked back to its original position in the block bank. This affordance is only active when the agent has a block in hand, which was possibly picked through the “Pick block from block bank” affordance. There are sixteen instances of this affordance, with one for each block.	$Agent.BlockInHand$
Drop block to construction area	Drops a block into the grid representing the solution for a given design. There are sixteen instances of this affordance, with one for each grid cell.	$\forall_x Block(x) \Rightarrow x.Location \neq ConstructionSlot$
Flip block up	A single instance of this affordance flips the current block in hand up.	$Agent.BlockInHand$
Flip block down	A single instance of this affordance flips the current block in hand up.	$Agent.BlockInHand$
Flip block right	A single instance of this affordance flips the current block in hand up.	$Agent.BlockInHand$
Flip block left	A single instance of this affordance flips the current block in hand up.	$Agent.BlockInHand$
Spin block clockwise	A single instance of this affordance flips the current block in hand up.	$Agent.BlockInHand$
Spin block counter-clockwise	A single instance of this affordance flips the current block in hand up.	$Agent.BlockInHand$

3.5.1 Reasoning Model for Solving the BDT

A general template strategy was used in the block design model for reasoning in VREE. This template strategy worked by laying out a main flow of reasoning, but left out two key steps—one for selecting a block from the bank and another for finding the right face of the block—that were later implemented as sub-strategies. When solving a block design puzzle, reasoning proceeded by considering each cell in the design as an intermediate goal to be solved. Thus, the model had agents working through any designs block by block.

Reasoning through a given puzzle with this model proceeded as follows:

- The design, as presented in decomposed cells, is considered cell-by-cell from the top-left to bottom-right.
- In each iteration, the agent fixes its attention on the current cell under consideration as the intermediate goal and memorizes it in short-term memory.
- With the block in memory, the first sub-strategy is used to find the best block to pick from the block bank, and the second sub-strategy, which runs after the first, is used to find the right face for the block.
- Once the right face is found, the block is placed in the construction area to end the intermediate goal.

This process is repeated for all cells of the design, row-by-row, from the top-left to the right-bottom. A state machine representation of this general reasoning strategy is displayed in Figure 3.16

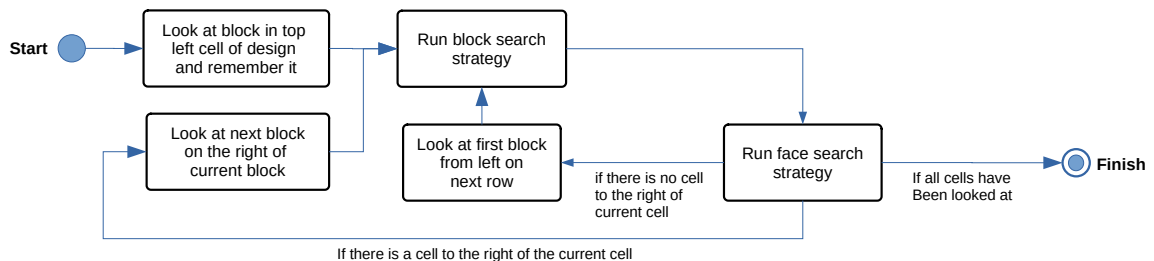


Figure 3.16: A flow diagram constructed around the state machine representation of the template strategy. Note the slots into which the two sub-strategies for block search and face search are executed.

For the block search strategy, there were two different choices: the agent could pick the next block from the bank, regardless of its face, or the agent could scan the block bank for the block whose face best matches with the design’s cell under consideration. The first choice was known as the sequential strategy, since the agent picked blocks according to the sequence in which they were presented, and the second choice was known as the closest looking strategy.

When it came to the strategy for finding the right face for the block, there were three different choices for the sub-strategy. The first, which also happened to be the simplest,

had the agent randomly flipping the block until the right face was found. The second took advantage of imagery to speed up random flips by deciding the direction of the next flip based on the colours of the current face. The final strategy, which did not use any imagery at all, performed a breadth first search on the symbolic representation of the block in the agent’s long term memory to find the optimal sequence of flips that reached right face from the initial starting point.

Algorithm 3.2: Imagery search algorithm for finding the matching face to the cell in memory.

```

ActionQueue  $\leftarrow$  {Nothing, FlipUp, FlipRight}
BlockSpun  $\leftarrow$  False
while ActionQueue is not empty do
    pop nextAction from ActionQueue
    perform nextAction
    if face on block matches with cell in memory then
        | end loop
    if current face has red and white  $\wedge$   $\neg$ BlockSpun then
        | ActionQueue  $\leftarrow$  {SpinClockwise, SpinkClockwise, SpinClockwise,
        | FlipUp, FlipRight, FlipRight, FlipRight}
        | BlockSpin  $\leftarrow$  True

```

3.5.2 Walk-through of the Block Design Task Strategy

Here is a walk through of the steps taken by the agent to solve a given instance of the block design task. For this walk-through, the agent is working with the sequential sub-strategy for picking blocks and the imagery sub-strategy for selecting the right block face. The agent works as follows:

1. The agent first places its gaze window on the top left cell of the design.
2. According to the sequential strategy, the agent picks the next available block from the top left corner of the block bank. In the case of this particular run, the block happens to have a white face.
3. The agent compares this face to what it has in short-term memory. This comparison, however, fails.

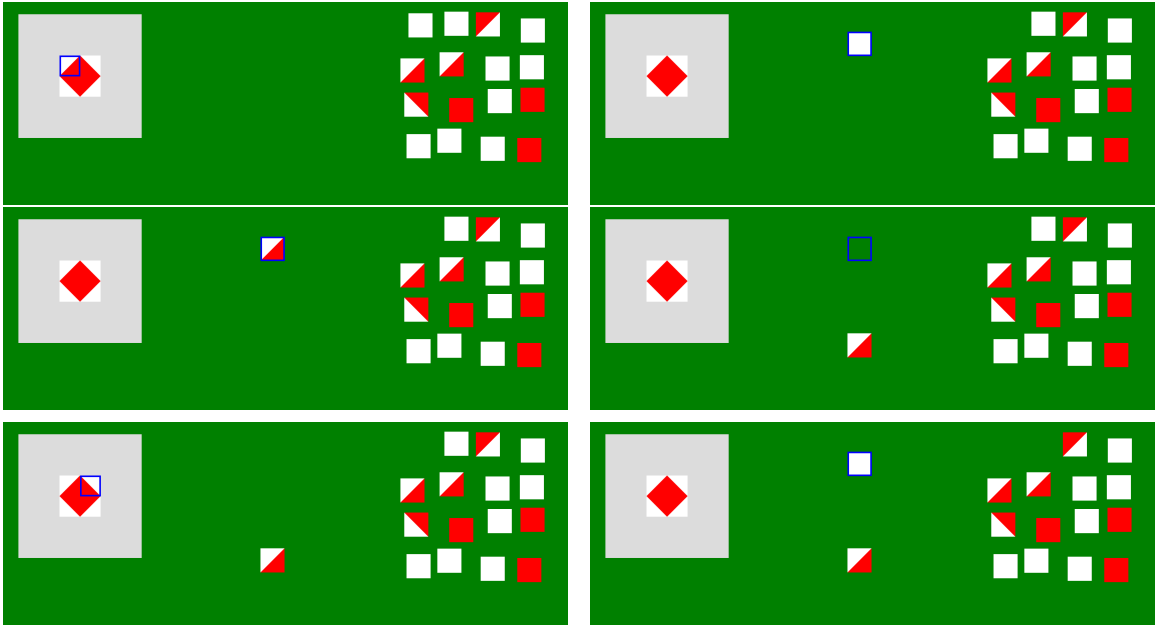


Figure 3.17: A view of the VREE environment for a few initial steps while solving a block design item. The design to be replicated is on the left, the block bank is on the right, and the final construction is in the centre. The blue square indicates the current gaze position of the VREE agent.

4. The agent then flips the block up as required by the imagery strategy to reveal a red and white face with the SE orientation which matches with the cell in short term memory.
5. This process continues until all the cells are solved.

See Figure 3.17 for a visualization of the first few steps of this process.

3.5.3 Results

By combining the different sub-strategy stages, a total of six main strategies for reasoning through the block design task were obtained—two for the first sub-strategy and three for the second. These six strategies were further evaluated with the agent’s short-term memory forgetfulness turned either on or off, leading to a total of 12 different experimental trials. In each trial, eight different designs were evaluated and the results for these are shown in Figures 3.18 and 3.19. Figure 3.19 shows all the trials in which forgetfulness was enabled, placing emphasis on the variability in response times, whiles Figure 3.18 shows the trials

without forgetfulness.

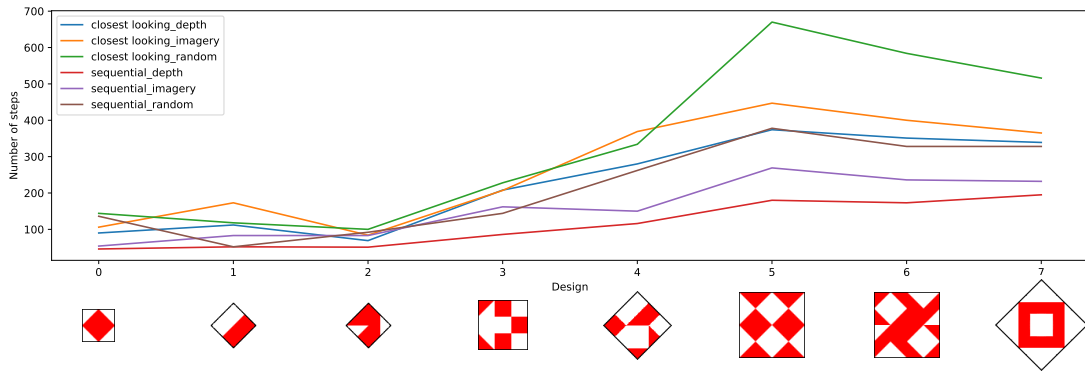


Figure 3.18: The results of evaluating all six main strategies on eight BDT puzzles.

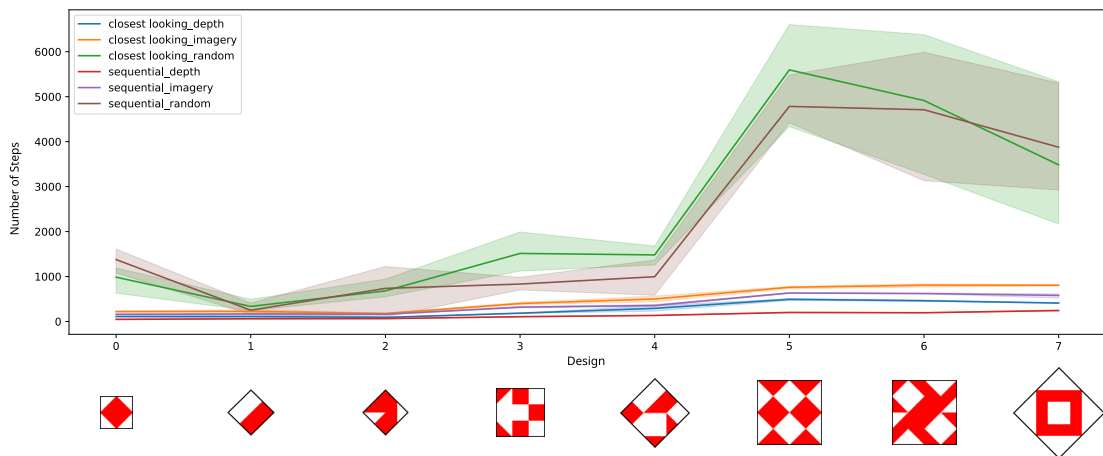


Figure 3.19: The results of evaluating all six main strategies on eight BDT puzzles with forgetfulness. Notice the variability in response times for the best and worst performing strategies.

3.5.4 Discussion

When considered in their composite form, the best performing sub-strategy pair in both experimental setups was the sequential block search and depth first face search combination. On the other hand, the consistent worst performing pair was the closest looking block search and random face search combination. This observation was due to how the different strategies spent their time. The sequential block search always spent a single

time step, whereas the closest looking search strategy required extra steps to search for the right blocks. This search ultimately led to the closest looking strategy being slower than sequential.

On the face search strategies, the best performing was the depth first search. This strategy being purely symbolic was guaranteed to be optimal every time, meaning the fewest block flips will always be taken to reach the correct face. Conversely, the worst performing strategy was the random search which was uninformed and flipped the block until the correct face was found. The inefficiency of the random face search was even more apparent when the experiments involving forgetfulness were analysed. With forgetful random search in place, the variance in the number of steps taken was wide, making the random search strategy's performance significantly worse when compared to the other forgetful random search strategies.

Overall, the imagery based strategies (those consisting closest looking and imagery search) did not perform as well as their symbolic counterparts (those consisting sequential and depth search). But, given that imagery search operations were more informed, they performed better than random search.

Given that VREE was human inspired, it is worth asking if these strategies exhibited could be used by humans. We may not know how humans may be reasoning through the block design task, but from analyses of human performance, it can be seen that people exhibit diverse strategies which could be influenced by several factors. Some people may be influenced by the gestalt figures that the design exposes, others may work through the task by breaking them into segments (like going row-wise or column-wise). Also, some people may take in the design as a holistic piece while working through the test. With VREE on the BDT, the strategies employed always split the design out into its individual block components and solved the problems in a block-by-block fashion (according to the selected sub-strategies).

Some similarities already exist in certain human strategy patterns and those exhibited in VREE's BDT solver. For example, concerning block segmentation, it has already been studied that people generally perform better on the BDT when designs are pre segmented (Stewart et al., 2009). And also, how people distribute their attention and their preference

to spatial placements affects their performance (see Cha et al., 2020; Rudolf Burggraaf and van der Geest, 2016; and Chapter 6 of this Dissertation).

The takeaway here is that, regardless of the differences between how VREE agents solve the BDT (in rigid a fixed manner) and how humans do (in a fluid and adaptive manner), systems like VREE still form a solid base on which experiments that involve human performances on tasks like the BDT, where attention and motor actions play an active role in reasoning, can be studied.

CHAPTER 4

Synthesizing Strategies for the Abstract Reasoning Corpus

Some cognitive scientists have been studying the idea that, as humans, we build a repertoire of strategies from which we select the best candidates for solving problems we face (see Marewski and Link, 2014). This, however, brings up another question: *How exactly are these strategies represented?*. Simon and Newell (1971) discuss extensively how human problem-solving and strategies can be expressed in information processing terms as computer programs.

In this chapter, I discuss work performed towards building visual reasoning systems that form their own strategies when faced with novel tasks. These strategies are represented as programs in a domain specific language built for reasoning about visual tasks. I relied on the Abstract Reasoning Corpus (ARC) as a task domain because of its amenability to program synthesis techniques.

This work was in direct continuation to the experiments discussed in Chapter 3. With the sufficiency work in the previous chapter exploring hand-coded strategies, here I will delve into machine generated strategies for ARC tasks.

4.1 The Abstract Reasoning Task

Formally, an ARC task, T , that has n training items and m tests items can be defined as:

$$T = \left\{ \langle I_1^t, O_1^t \rangle, \dots, \langle I_n^t, O_n^t \rangle; I_1^e, \dots, I_m^e \right\}$$

where I^t and O^t are input and output training grids, and I^e represents a test input grid for which the solver must provide an output. In the case of publicly available ARC tasks, the solutions, O_i^e , for all test inputs, I_i^e , are provided to help developers test solving techniques. Thus, in the case of items in the public set, a task, T , can in turn be defined as:

$$T = \left\{ \langle I_1^t, O_1^t \rangle, \dots, \langle I_n^t, O_n^t \rangle; \langle I_1^e, O_1^e \rangle, \dots, \langle I_m^e, O_m^e \rangle \right\}$$

4.2 Visual Imagery Reasoning Language

My approach to solving items on the ARC was based on the tried and tested approach of performing program synthesis in a domain specific language (DSL) designed to only solve problems in a given task domain. For the work described in this dissertation—modelling studying visual reasoning in selected standardized intelligence tests—I developed a domain specific language, named Visual Imagery Reasoning Language (VIMRL) to form the basis of all my program synthesis experiments.

As an imperative style language, VIMRL was designed around imagery operations, and built specifically for reasoning about visual tasks. Instead of relying on control instructions, VIMRL placed emphasis on the sequence of instructions to control the state of a program during execution. Values in VIMRL programs could be variable or literal, and they always had a fixed data type. VIMRL operations consumed these values as arguments for their internal computation, and in turn returned values that could be assigned to variables. Variables were only created when they are assigned to. The grammar for VIMRL is displayed in Table 4.1.

4.2.1 Generating ARC Strategies with VIMRL

For solving ARC tasks, a VIMRL based solver searched a space of potential programs for ones that were capable of solving items from a task’s training items. A program that performed well on these training items was most likely to solve the test items from the task.

When given an ARC task, $T = \{ \langle I_1^t, O_1^t \rangle, \dots, \langle I_n^t, O_n^t \rangle; I_1^e, \dots, I_m^e \}$, the solver searched a space of VIMRL programs for a candidate program, $y = \varphi(x)$, where x was the input grid and y was the predicted output grid. For a program to be considered as a viable candidate solution, it had to satisfy: $\left(\frac{\sum_{i=1}^n \lambda(\varphi(I_i^t), O_i^t)}{n} > \alpha \right)$, where $\lambda(x, y) = \begin{cases} 1 & x = y \\ 0 & \text{and, } \alpha, \text{ was a} \end{cases}$ threshold value within which the agent had to be accurate. Thus, a candidate program would only be selected if it solved enough items in a task’s training items with an accuracy higher than the value of α .

In solving ARC tasks, values in VIMRL could assume one of 6 types. They could either be typed as image, object, color, number, list or grid values. The actual nature and

Table 4.1: Grammar for the Visual Imagery Reasoning Language-I (VIMRL). These also double as production rules for generating code during program synthesis.

$\langle instruction \rangle ::= \langle assignment \rangle$
 $\quad \quad \quad | \quad \langle operation \rangle$

$\langle assignment \rangle ::= \langle identifier \rangle '=' \langle operation \rangle$

$\langle operation \rangle ::= \langle identifier \rangle '(' \langle arguments \rangle ')'$

$\langle arguments \rangle ::= \langle argument \rangle$
 $\quad \quad \quad | \quad \langle arguments \rangle ',' \langle argument \rangle$

$\langle argument \rangle ::= \langle identifier \rangle$
 $\quad \quad \quad | \quad \langle number \rangle$
 $\quad \quad \quad | \quad \langle operation \rangle$

$\langle number \rangle ::= ('-')?[0-9]^+$

$\langle identifier \rangle ::= [a-zA-Z][a-zA-Z0-9]^*$

the data structures backing these data types were determined by the interpreter of the underlying Python language in which VIMRL programs were executed.

Values typed as `image` were used to represent grids from the ARC task. An `image` typed value had a fixed number of rows and columns, and each cell of the grid (also considerable as a pixel of the image) held a symbol valued between -1 and 9. The symbols 0 through 9 corresponded with symbols from ARC tasks, while the -1 symbol was used by VIMRL to represent transparent sections in images.

Symbols stored in grid cells had the `color` type. Internally, the `color` type was represented as an integer, and its values were passed as arguments to operations that required colour information. For operations that returned or consumed actual numerical values, like when counting, the `number` type was more appropriate since that could represent a wider range of values.

The `object` type was for sections of grids that were isolated to represent individual “objects”. Any `object` value internally contained an `image` typed value representing the

object's grid, and two `number` typed values for the top-left row and column coordinates of the original image from which the object was isolated. Segmentation operations used this type to represent any objects that were extracted. Typically, values of object type were dealt with in groups represented by the `list` type. Objects could be arbitrarily shaped, and in such cases a colour value of -1 was used for cells meant to be transparent.

Some tasks in the ARC had grids whose images also depicted grids (for example, see Figure 2.7(c)). I referred to these as “grid of grids”¹. Cells of these “grid of grids” could either contain a single solid colour, or they could also contain an entire image. To effectively deal with tasks that had such grids, I incorporated VIMRL values that were typed as `grid`. Every `grid` typed value contained information about the number of rows and columns in a grid, the image stored in each cell, as well as, the grid's outline colour.

4.2.2 Execution of VIMRL Programs

A VIMRL program under execution had the following state:

1. The set of all variables that had been defined throughout the program's lifetime.
2. A related set of all the values associated with the defined variables.
3. The current line of instruction being executed.
4. An error state flag, which was set to true whenever an operation resulted in an error.

Whenever a program was running, two main levels of instructions could be executed. I named these low-level and high-level instructions. Low level instructions in VIMRL performed simple operations, and any arguments (operands) required by the operation had to be explicitly specified. Conversely, operations performed through high-level instructions had the opportunity to analyse all the training items from the task being solved to determine any desired characteristics that could be exploited to solve a task item through a local search. High level operations could take a single explicit argument, which was typically the image being processed or a derivative of it (such as a list of objects extracted from the image).

¹From this point onward, the terminology around grids in this dissertation may appear a bit confusing when referring to ARC task items. To keep things simple, unless explicitly specified, the term image will refer to any of the input and outputs of ARC tasks items, or any segments of those inputs and outputs that are extracted as objects, while the term grid will refer to the concept of “grid of grids” introduced here.

Local searches executed by high level operations could fail. Failure typically occurred because certain characteristics required by the operation were not available in the images. Whenever a local search failed, the program was put into an error state and the execution of any subsequent instructions was aborted. In the context of a broader program search (when the solver is trying to solve a given task) failures can be taken advantage of to prune out programs that were not going to yield valid solutions. Because the error flag was used as a signal in most of the search algorithms, high level operations performed preliminary checks, which caused instructions to fail fast if they had to, preventing precious time from being spent on invalid inputs.

In addition to the two main categories of operations, I added the ability to derive mapped operations. These operations were created from existing low level operations that had a single image input and a single image output. As their name suggests, these mapped operations were built around a mapping function—akin to the high-order map procedure found in most parallel and functional programming environments—to compose low level operations to operate on multiple items at a time. Inputs to mapped operations were of the `list` type, and they equally always returned `list` typed outputs. Any low level operation that took a single `image` typed value as argument and equally returned a single `image` typed value could produce a derived mapped function. Such operations were thus considered *mappable*.

4.2.3 Details on VIMRL Instruction Types

Consider the tasks displayed in Figure 4.1 and the programs listed in Table 4.2. Each program in Table 4.2 was produced as a possible VIMRL solution to a corresponding task in Figure 4.1. Programs (a) through (c) each had a single instruction, and program (d) had multiple lines of instructions.

Program (a) provided an example of a simple low level operation. Here, `trim` removed any extra surrounding cells in the input image to create a bounding box around the depicted object. The image section enclosed by the bounding box was returned as the final output.

Program (b) showed an example of a high level operation, `attract`. When executed, `attract` used simple naive physics rules and simulations to solve the problem of objects

(a) <code>output = trim(input)</code>	(b) <code>output = attract(input)</code>
(c) <code>output = recolor(input)</code>	<code>enclosed = find_enclosed_patches(input)</code> <code>recolor = recolor_objects(enclosed)</code> <code>output = draw(input, recolor)</code>

Table 4.2: VIMRL program listings for possible solutions to the ARC tasks displayed in Figure 4.1

being attracted to each other. This internal search tried to find objects that remained stable, those that moved, as well as the directions in which objects moved, when input and output images were considered.

Just as with program (b), program (c) also employs a high level function. This time, however, the internal search observed how colours were transformed between input and output images to help solve tasks that involved colour changes.

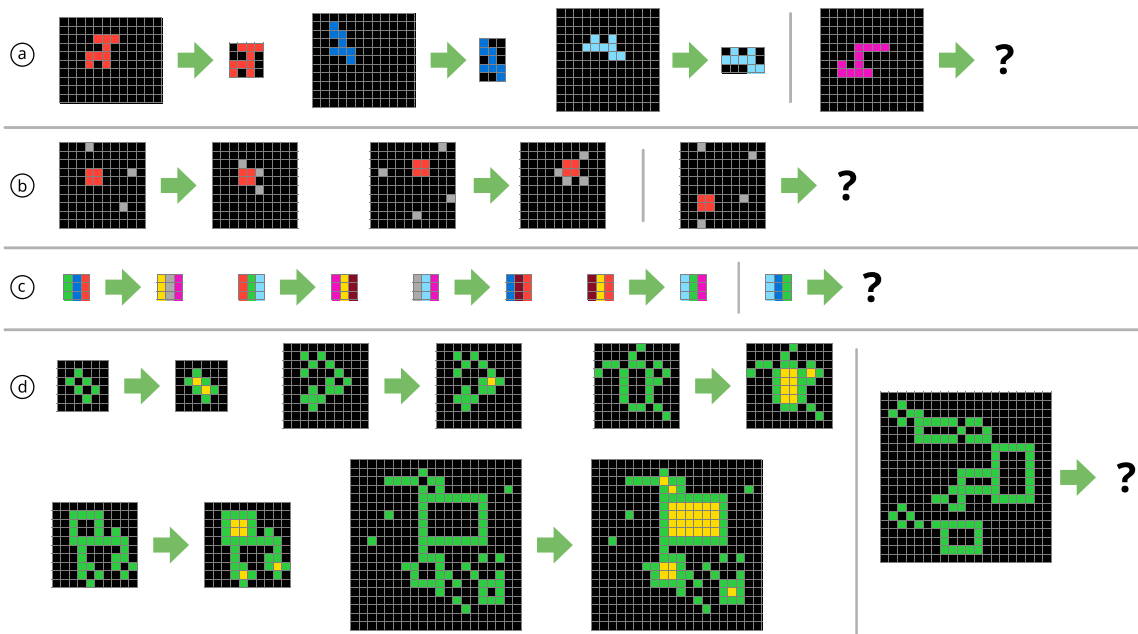


Figure 4.1: Tasks for the programs displayed in Table 4.2.

Program (d) demonstrated how programs could be crafted with multiple instructions, including mixing both high level and low level ones. With multiple instructions, complications arose when a high level instruction was executed right after a low level one. This was due to the fact that executing a high level instruction required an instance of the

task’s training items to be analysed. And in the case where other instructions had already executed, there was a possibility that the input images, I^e , from the input-output pairs of the tasks training items, $\{\langle I_1^t, O_1^t \rangle, \dots, \langle I_n^t, O_n^t \rangle\}$, no longer corresponded with the current execution state, considering the modifications prior operations may have already made.

To solve this problem, before any high level instruction was executed, all prior instructions issued were re-applied to the input-output training pairs to create a modified version of the task.

This worked as follows: Consider the sequence of all instructions performed before a high-level operation is executed as a partial program, $\varphi'(x)$, then generate a modified task, T' , with training items, $\{\langle I_1^t, O_1^t \rangle, \dots, \langle I_n^t, O_n^t \rangle\}$, such that $\forall_{i \in \{1, \dots, n\}} \langle I_i^t, O_i^t \rangle \rightarrow \langle \varphi'(I_i^t), \varphi'(O_i^t) \rangle$. This modified task, T' , can be passed to the high level operation as an intermediate version of the task that is representative of all prior operations.

4.2.4 A Walk-through of High Level Function Execution

To further illustrate how partial programs modified a task’s training items before they were analysed by high level instructions, consider the programs from cells (c) and (d) of Table 4.2 (which were solutions for their respective tasks in Figure 4.1). As shown earlier, the program in cell (c) used a single call to a high level function, `recolor`, which learned the rules by which colours in an image were transformed, to solve the task in Figure 4.1 (c).

A similar colour changing operation, `recolor_objects` was used in Program (d). But before this operation was ever executed, `find_enclosed_patches` operation would have already been executed. At this point, the input being passed for recolouring was no longer be representative of the input image. Instead of a full image, it was instead be a list of objects extracted by the `find_enclosed_patches`. To make it possible for the call to `recolor_objects` to access the correctly typed training items, a modified task would be generated, as demonstrated in Figure 4.2.

From the walk-through illustration (Figure 4.2) we observe that the initial state had a single variable, `input`, with the input image of the test item as its value. The first instruction, `enclosed = find_enclosed_patches(input)`, analysed the input image and extracted all patches of the grid’s image that were enclosed within the green lines. In the

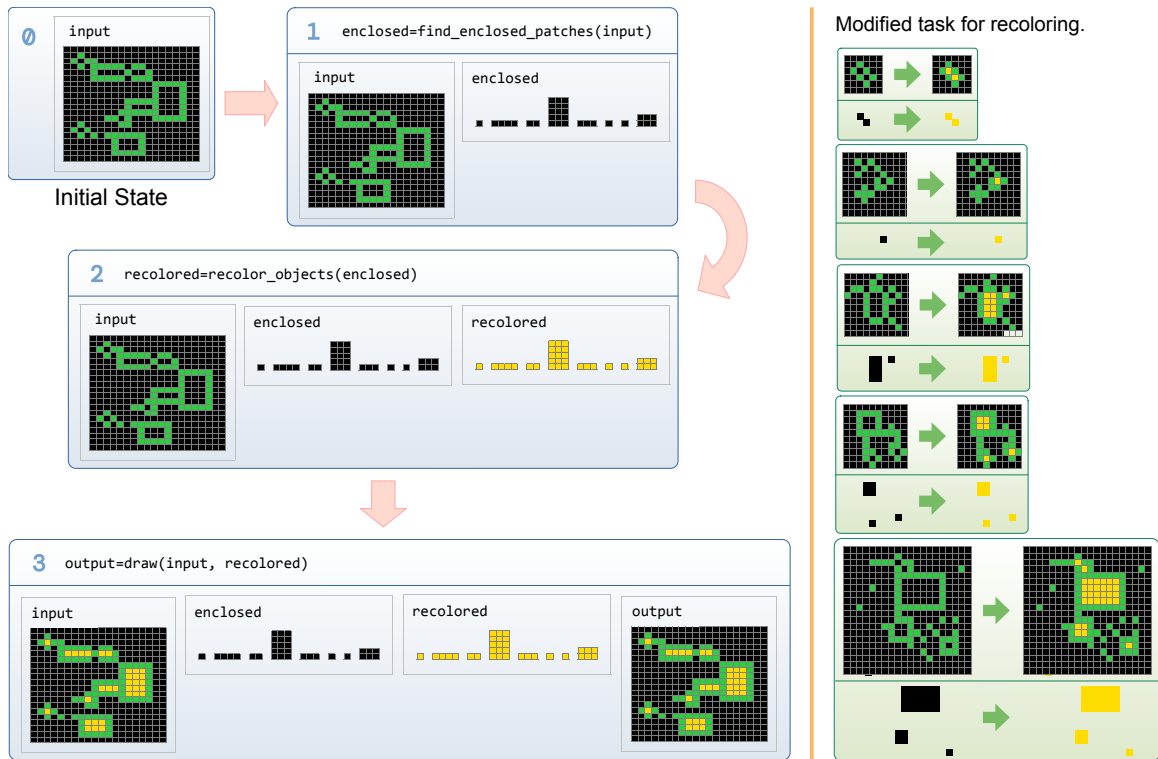


Figure 4.2: A walkthrough of the execution of Program (d) from Table 4.2 on the task from Figure 4.1 (d). On the left side of the image, each box shows a time step in the execution of the program. The instruction executed is on the top of each box, and the bottom of each box shows the state of all variables available in the runtime environment. The left section shows the modified task passed to the `recolor_objects` operation. For each task item, the top section shows the original task, and the bottom section shows the list of objects passed on to the `recolor_objects` operation.

case of this particular example, the call to `find_enclosed_patches` returned a list value with 8 objects and their locations.

The next instruction, `recolor_objects`, which was a high-level function, took this list of objects as its only explicit argument. In addition to this list, the `recolor_objects` function also received a copy of the task, a copy that had already been modified to reflect any changes the earlier call to `find_enclosed_patches` may have made to the input image. To build this modified task, the partial program, which contained only a single call to `find_enclosed_patches`, was executed on all the inputs and output images of the task.

In the case of this example, when the partial program was executed on the inputs and outputs of the training set, the corresponding image (input or output) in the task was replaced with the value associated with the value of the `enclosed` variable (see Figure 4.2).

add	(N)(O)	attract	(G)(O)	change_color	(O)	color_sorted	(N)(O)
complete_pattern	(T)	connect_grid_cells	(G)	connect_pixels	(G)	copy	(O)
create	(T)	difference	(N)	draw	(T)	find_central_object	(O)
find_enclosed_patches	(O)	find_objects	(O)	find_objects_in_context	(O)	find_odd_object	(O)
first_image	(O)	first_object	(O)	flip_horz	(O)(T)	flip_vert	(O)(T)
get_color	(T)	get_grid	(T)	get_grid_cells	(T)	grid_as_image	(T)
grid_cells_as_image	(T)	grid_get_color		invert	(O)	last_image	
last_object		map	(N)	map_image	(N)	recolor_image	
recolor_objects	(O)	repeat	(G)(O)	reset_background	(T)	rotate_180	(O)(T)
rotate_270	(O)(T)	rotate_90	(O)(T)	scale_2x	(O)(T)	scale_3x	(O)(T)
scale_4x	(O)(T)	scale_5x	(O)(T)	scale_half	(O)(T)	scale_quart	(O)(T)
scale_third	(O)(T)	segment_objects	(O)	self_scale	(O)(T)	sort_by_color	(N)(O)
sort_by_color_frequency	(N)(O)	sort_by_holes	(N)(O)	sort_by_size	(N)(O)	trim	(O)(T)

Legend (N) Numbers and counting (O) Objectness (G) Goal Directedness (T) Geometry and Topology

Table 4.3: A list of selected ARC operations and their assumed core knowledge priors.

The enclosed variable was chosen as the replacement value because the `recolor_objects` operation receives it as an argument during execution.

It is also worth noting that in this case, `enclosed` was of a `list` type, which led to a situation in which all the training images (input and output) were replaced with lists. This did not cause any problems because the `recolor_object` operation required a `list` typed value. In its operation for this particular task, it compared the lists of objects in the input and outputs of the modified task’s items to detect that everything coloured black (0) was switched to yellow (4).

After the execution of the `recolor_objects` operation, the `draw` operation was finally used to paint all the recoloured objects back to the input grid, and the results were assigned to the output variable as a solution to the task.

4.3 Operations for Reasoning about the ARC

In all, there were 20 high level, 56 low-level, and 29 mappable operations available to VIMRL for solving ARC problems. This gave a total of 105 operations. The internal implementation of all these operations were inspired by the core knowledge priors (as discussed in Section 2.5.4.2) suggested for the ARC. Moreover, most of these operations were also chosen because of similarities and common themes that were observed in tasks from the public ARC dataset. A sampling of operations and the core priors by which they may likely be operating are listed in Table 4.3, with details in Appendix A.

All operations designed for solving ARC tasks with VIMRL were put together only after

observing the 400 training tasks from the public ARC dataset. I decided not to consider any tasks from the evaluation section in order to give me a better basis for evaluating how well concepts between tasks in the evaluation and training sections were separated.

4.4 Overview of Selected Operations

Before explaining how the different search experiments were performed, I intend to use this section of the dissertation to provide some details on how a few selected operations worked internally. Although this section explains only four operations out of possibly 76 (if we choose to ignore the derived mappable operations) the ideas around their implementation could easily be transferred to other operations. In this section, I will be providing details on the `solve` and `connect_pixels` operations, which were high level ones; and the `find_eclosed_patches` and `self_scale` operations, which were low level ones. For brief summaries on how these and all the operations work, see Appendix A.

4.4.1 The `self_scale` Low Level Operation

The `self_scale` operation was one of the simplest operations implemented for VIMRL. It took two input images a and b , with which it produced an output whose size was determined by scaling a 's dimensions (in both axes) by factors of b 's dimensions (in corresponding axes to a 's) and every corresponding scaled pixel from the original input image, a , was replaced by the entire image in b .

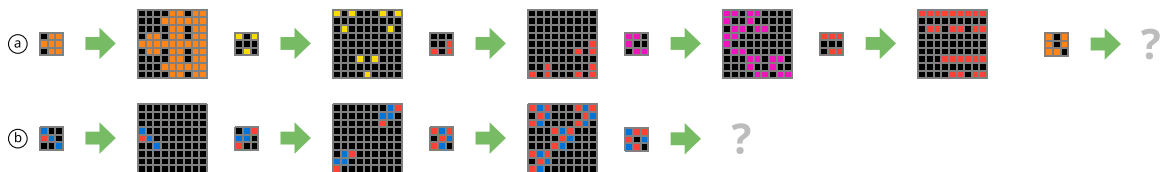


Figure 4.3: Two tasks from the ARC that make use of the `self_scale` function. (a): `007bbfb7`; (b): `cce03e0d`.

Considering the tasks in Figure 4.3. Task (a) represents a straight forward use of the `self_scale` operation. Here, the task's input image was passed as both parameters to the operation, essentially causing the image to scale itself², while replacing the pixels scaled in

²This operation was named `self_scale` because its original use case was for situations in which images scaled themselves, like in Figure 4.3 (a). However, through search, the situation in Figure 4.3 (b) was discovered.

the output pixels with its unscaled self. When Figure 4.3(b) is considered, however, a much more complex case, where the input pixels were initially filtered to remove some colours, before scaling was applied, to ensure the final image had the original unfiltered image tiled in the pattern of the filtered image. VIMRL programs that solved both cases are listed in Table 4.4.

Table 4.4: Listings for VIMRL programs that solve tasks in Figure 4.3

(a) <code>output = self_scale(input, input)</code>	(b) <code>var1 = filter_color_pass(input, 2)</code> <code>var2 = self_scale(var1, input)</code>
---	---

4.4.2 The `find_enclosed_patches` Low Level Operation

Some tasks in ARC required solvers to reason about arbitrarily shaped patches. For example, the task in Figure 4.1 (d) required a solver to extract rectangular shaped patches bordered by “green” cells. To find enclosed patches, the operation cycled through all colours available in the input image, and in each cycle assumed the active colour to be the border colour. A depth first search in which individual cells were treated as search nodes extracted all patches that were enclosed by the active border colour for the cycle. At the end of all cycles, all enclosed patches isolated were returned. For patches that were not perfect rectangles, the areas that either formed the borders of the patch or were outside the enclosed regions were given a cell value of -1 to indicate their transparency.

4.4.3 The `solve` High Level Operation

The `solve` operation took as input, a list of same sized images and found the rules by which these inputs could be combined into a single same sized image. Inputs to `solve` came in the form of `list` typed values that were intended for storing lists of objects. To find the relationship between input images and the output, `solve` used a simple multi-layer perceptron to learn the spatial relationships between the image pixels. Because the `solve` operation was a high level one, it had the opportunity to use the task’s training images as a basis to train this internal neural network.

As a demonstration of how the `solve` function worked, consider the task in Figure 4.4.

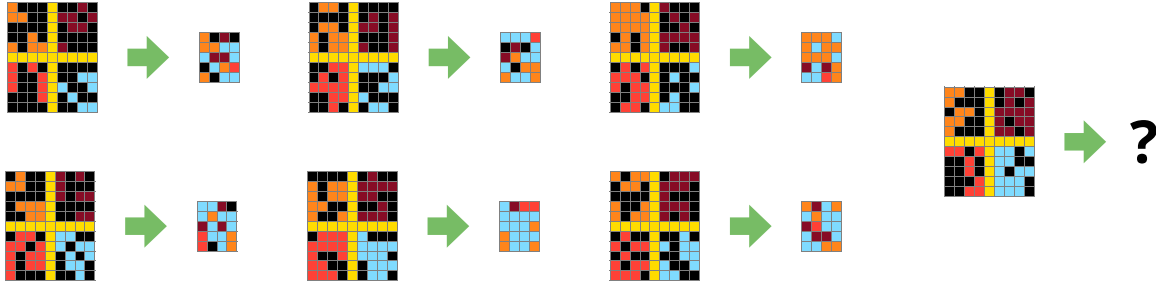


Figure 4.4: Task e999362f0 from the Abstract Reasoning Corpus.

This task was solved with the VIMRL program listed in Table 4.5. In this program, the `get_grid` operation extracted a grid typed value containing the “grid of grid” observed in the input. Then, the individual cells of this grid value were further extracted as a list of objects to be passed on to the `solve` function.

Table 4.5: Listing for a possible VIMRL solution to the task described in Figure 4.4

```

var1 = get_grid(input)
var2 = get_grid_cells(var1)
output = solve(var2)

```

Once passed to the `solve` operation, an internal local search to find how pixels were transformed was executed. Because the `solve` operation expects a list value, the ARC task’s training items were all converted to lists through the partial program of already executed instructions (see Section 4.2.3). Pixels from the ARC item’s training were then used to construct a new data set for training the neural network. This new data set was constructed as follows:

- For each object in the input list, iterate over each pixel in the object’s image grid.
- Given that each grid is of the same size as the output grid, construct a single training record for the neural network by considering each pixel on the output as the neural network’s output and the corresponding pixels on each cell of the grid’s images as the neural network’s input vector.

With the dataset constructed, a simple feed forward neural network with three layers and 5 hidden nodes on each layer was prepared for the number of inputs and a single output. This neural network was trained till convergence, after which the network could be used to predict the individual pixels of the output image (see Figure 4.5.) If the network failed to converge, the error flag was set and the operation failed.

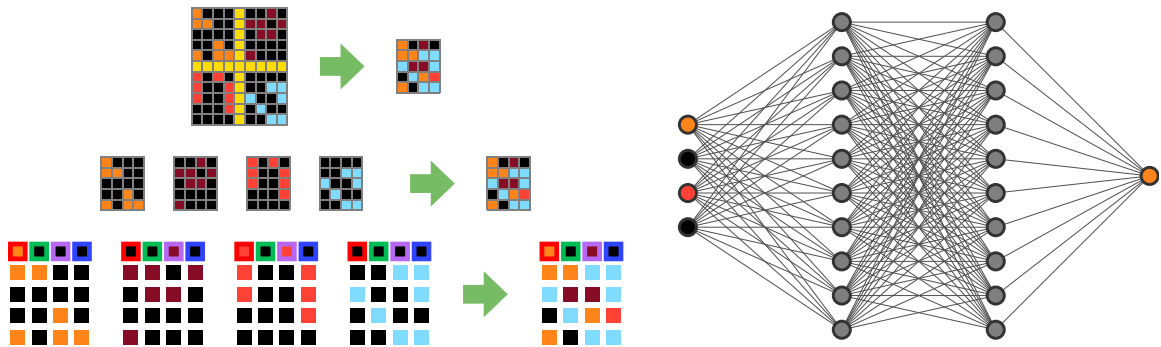


Figure 4.5: A visual demonstration of how data items from tasks are converted into the dataset for the neural network.

4.4.4 Title `connect_pixels` High Level Operation

The `connect_pixels` operation connected pixels in an image with simple line segments. As a high level operation, it worked by analysing the training items to determine the underlying rules by which connections were made in the training items. Connections could either be horizontal, vertical, or diagonal (always with a gradient of 1). See Figure 4.6 for an example of a task whose solution involves the `connect_pixels` operation.

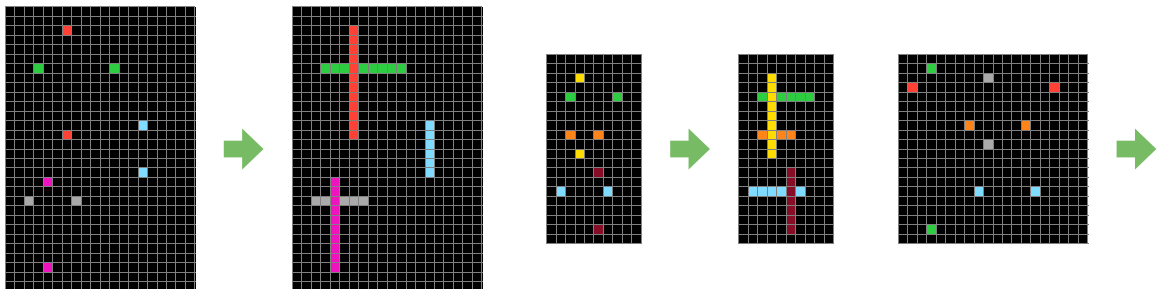


Figure 4.6: A sample task for demonstrating the `connect_pixels` operation

When presented with an input, the `connect_pixels` operation first attempted to find

isolated pixels in the input images, then it checked for connections between these isolated pixels in the output images. If any connected pixels were found, the directions of the connections (diagonal, vertical, or horizontal), the colours of the connecting and connected pixels, and also the order of connections were then checked. If no connected pixels were found, then the error flag was set and the operation terminated.

Even when connected pixels were found, the final behaviour of the operation was decided only if the connection rule was consistent for all the task's training items.

For a walk-through of this operation, consider the program listed in Table 4.6. This program offered a straight forward call to the connect pixels operation to solve the task presented in Figure 4.6. The task had two training items and a single test item.

Table 4.6: Listing for a possible VIMRL solution to the task described in Figure 4.4

```
output = connect_pixels(input)
```

When presented with the test item, the connect_pixels operation performed a search on the training items. From the search, it was detected that isolated pixels on these items could be connected such that pixels sharing a common row or column that were also of the same colour were linked. When it came to ordering the connections, the rule for this task was that horizontal connections were established before vertical ones.

4.5 Synthesizing VIMRL Strategies for ARC Tasks

ARC strategies were synthesized through a search process that had the goal of finding programs that solved a given number of items from the training section of any given ARC task. I broke the search algorithm's design down into the following three main components:

1. Exploration: Successor generation and node pruning.
2. Optimization: Selecting the least number of operations that provide the best results possible.

3. Selection: Deciding on what the final three programs will be for any search.

The final search algorithm could be described as follows:

Given an instance of the ARC task, $T = \{ \langle I_1^t, O_1^t \rangle, \dots, \langle I_n^t, O_n^t \rangle; I_0^e, \dots, I_m^e \}$, generate and collect candidate programs, $\varphi_i(x)$, which satisfy $\left(\frac{\sum_{i=1}^n \lambda(\varphi(I_i^t), O_i^t)}{n} > \alpha \right)$, where $\lambda(x, y) = \begin{cases} 1 & x = y \\ 0 & \end{cases}$, for some threshold $\alpha > 0$.

This general search algorithm was executed in an enumerative *generate-execute-test* cycle until a given number of programs were found, or a pre-determined time-out was reached.

Whenever the search ended, regardless of the terminating conditions, the best performing programs were selected from the set of candidate programs whose score exceeded the threshold, α . Because external ARC evaluators expected three predictions from a solver, a selection algorithm was used to filter all but three of the candidate programs, to obtain the three final proposals.

In my experiments, I investigated searching the space of VIMRL programs with either classical tree search algorithms (where I looked at some informed and uninformed approaches involving with breadth-first and depth first traversal), and the Monte-Carlo Tree Search algorithm. I implemented a couple of rules for pruning nodes that appeared likely to fail. And I implemented two different selection algorithms, one that selects the three smallest best performing programs, and another that selects the three best performing programs with unique outputs.

4.5.1 Tree Traversal with Classic Search Algorithms

Exploration with the classical search algorithms always started from an empty program, and with successor generators adding on one instruction at a time, potential programs were built up for evaluation. There were two main approaches for generating successors: there was a brute-force approach, which attempted to exhaustively find all possible programs, and there was a stochastic approach that tried to probabilistically generate the next best program.

When successors were generated through brute-force exploration, programs were gen-

erated as if a production system was used to extensively evaluate VIMRL's grammar to produce all possible subsequent instructions when given the last instruction in a program. But with the potential to add over 100 operations at each step, the branching factor grew rapidly at every step of the search, which led to a quick exponential explosion in the search space. Regardless of the space's size, a full brute force search served as a good starting point to help in understanding the dynamics of program generation in VIMRL.

When successors were stochastically generated, the probabilities of possible successor nodes were computed from a corpus of hand-coded ground truth programs that were generated for a subset of the 400 training tasks on the ARC. This corpus of hand coded programs was used to build models that had potential intrinsic knowledge on how operations in VIMRL interacted with each other. During search, a fixed number of new instructions were sampled from a set of all possible instructions, according to probability estimates computed from the ground truth programs. Sampling a fixed number led to a constant branching factor that could easily be managed during search.

4.5.2 General Structure of ARC Search

The classical search techniques used in this solver could be described by a standard framework. Search always started with an initial node that contained an empty program, then proceeded to build up the program one instruction at a time. The choice for subsequent instructions were determined by the current successor generator algorithm (brute-force or stochastic) and the direction of exploration was determined by the data type of the frontier from which the next nodes to be expanded are selected. In accordance with standard tree exploration, a queued frontier produced a breadth-first-search, where all successors of a node were evaluated before progressing, and a stacked frontier produced a depth first search, where the first successor of every node was evaluated until a terminal node was reached before node evaluation backed up the search tree.

While exploring the search tree, programs at each node were evaluated on the training items of any task being considered, and programs that had accuracy values greater than the α threshold were kept as potential candidate programs. Search was terminated only when a set number of candidate programs were produced or a pre-determined timeout

period elapsed. Once the search ended, the best three programs were selected with one of two selection algorithms.

The first of these algorithms selected the three smallest programs that had the highest α score. Here, the size of a program was determined by the number of instructions it contained, and the α score was determined by the number of test items the program solved. This selection algorithm, inspired by Occam's Razor, favoured simplicity. During evaluation, however, it turned out that most of the programs that were selected produced the same outputs.

In an attempt to increase the solver's odds of performing better, I implemented a second selection algorithm, which placed emphasis on producing outputs that were unique. This algorithm first grouped all candidate programs by their outputs, and for each group the smallest program was selected to form a new collection of candidate programs that produced unique outputs. Of these new candidates, the top three smallest programs with the highest scores were then selected as final outputs.

4.5.3 Estimating Probabilities for Stochastic Successor Generation

With VIMRL programs lacking branching instructions, it was possible to capture the complete logical structure of any program in just the plain sequence of instructions. This simplicity allowed me to apply a Markov Model, where the probability of an instruction occurring at any point in the sequence was determined by its predecessor. To properly account for the beginning and end of programs, I introduced special marker instructions, which marked the beginning and end of programs, to be used only when computing probabilities.

Instructions required arguments for which values had to be provided during successor generation. Instead of directly sampling these arguments together with their instructions—such as predicted through a joint probability distribution of instructions and arguments—they were sampled separately after each instruction was selected. Thus, for any instruction, the probability of an argument being assigned was based entirely on the instruction and nothing else. During execution of search, after an instruction was selected, a fixed number of random samples were taken for arguments according to distributions estimated from

the corpus of ground truth programs.

Probability distributions for both the transitions of instructions and the relationships between instructions and their arguments were estimated using either a Maximum A Posteriori (MAP) estimation or a Maximum Likelihood Estimation (MLE) based on counts from the ground truth program corpus. Details on how sampling worked follows.

4.5.3.1 Details on Sampling Instructions

Let A be a VIMRL program that contained N instructions. If each instruction was represented by a random variable X_i , then the set of A 's instructions could be represented by the set $\{X_1, \dots, X_N\}$ and the set of values for X_i was the set of all instructions available to the VIMRL system. The probability of each instruction X_i is conditioned on its predecessor such that $X_i = P(X_i|X_{i-1})$, and the probabilities of X_1 and X_N are predicated on X_{begin} and X_{end} such that $X_1 = P(X_1|X_{begin})$ and $X_{end} = P(X_{end}|X_N)$, where X_{begin} and X_{end} were special marker instructions marking the beginning and end of programs respectively.

4.5.3.2 Details on Sampling Arguments for Instructions

Values used as arguments in the VIMRL programs could either be literal or variable. When it came to solving ARC tasks, literal were always color typed, and they were extracted from grid cells. Because the choice of colours varied among tasks, any probabilities estimated from literal colour values in the ground truth corpus were not expected to be reliable at sampling time. As such, whenever an argument required a literal value, one of the colours from the unique set of all colours in the current task was picked at random according to a uniform distribution.

When it came to variable values, I considered them in two categories: the input variable and other defined ones. The input variable was treated as a special case because it was the entry point for most programs (almost always the first to be used).

The second category of variables I considered were those defined within a program. Because the probabilities of these were not estimated at runtime, the actual values of these variables could not be factored into probability estimates. Instead, the function that yielded the value for the variable—a property that could be observed from just the list

of instructions—was used. This concept can be referred to as sampling on the *variable's provenance*, since the originating function provided a concrete value on which variables could be classified when determining proportions for probability estimates.

With three categories of arguments (literals, the special input variable, and other defined variables) each with their own set of rules and limitations, I implemented a two-step sampling system for arguments. In this system, the first step sampled whether the argument was going to be one of either category (input, literal, or variable) before performing a second category specific sampling. Probability distributions for sampling were separately estimated for both tiers.

In the first tier, selecting the variable category was based on the distribution:

$$S = \{\forall_{category \in \{\text{input, literal, variable}\}} P(V_i^c = \text{category} | X)\}$$

Here, V_i^c represented the category of the i^{th} value in the operation's arguments.

In the second tier, if the choice was for the input variable, there were no options to sample because there was always a single input variable. In the case of a literal variable, however, values were sampled from the set of all unique colours in the task according to a uniform distribution. Finally, for variable values, sampling selected a candidate from already defined variables according to a distribution constructed around the function responsible for defining the variable. Details on how this variable sampling works is as follows:

Let X be a random variable representing the current instruction, and let V_i represent a random variable for the variable value required as the i^{th} argument to the instruction. Given that the ultimate value for V_i was expected to be of a given type, let the set $U = \{u_1, \dots, u_M\}$ represent all values already defined in the program with the required type. If we have a function $originator(u)$, which takes a variable and returns the instruction that originated it, then a value for V_i can be selected from the set, U , according the probability $V_i = P(originator(V_i) | X \wedge i)$. Figure 4.7 provides a flow diagram of the entire two-step sampling flow.

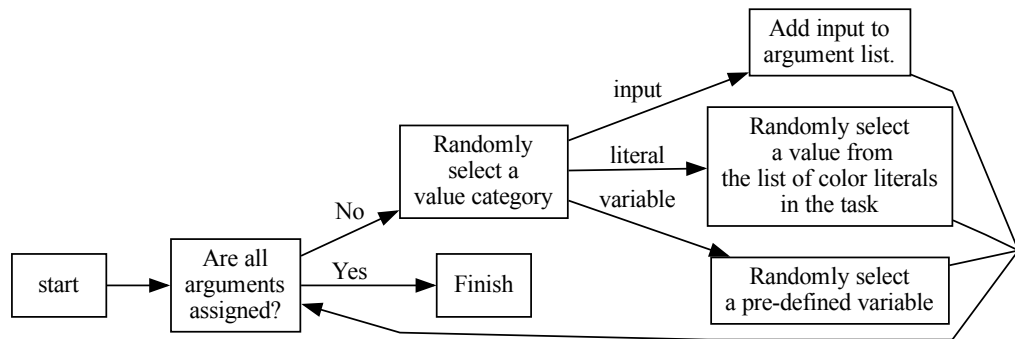


Figure 4.7: A flow diagram of how instructions and their arguments were sampled in a multi-tier sequence.

4.5.4 Ground Truth Programs for Stochastic Search Methods

To build the probability models for stochastic successor generation, a total of 363 ground truth VIMRL programs were produced for tasks in the ARC’s public training set. Some of these programs were coded by hand, with a significant number of them obtained through the brute-force search methods described in Section 4.5.1.

In all, there were a total of 363 ground truth programs, covering 177 unique ARC tasks. Of these programs, 231 were found through search and the other 132 were produced by hand. Search-found programs covered 115 unique tasks (corresponding with the solver’s best performance) while hand coded programs covered 129 unique tasks, with 67 tasks having shared solutions from both search and hand coding. On average, there were 2.1 ground truth programs per task, with a maximum case involving 4 programs per task, and as far as program sizes went, there were between 1 and 7 instructions per program in the ground truth. Programs were also well represented when the grid sizes of task items were concerned. Figure 4.8 shows charts that demonstrate some key characteristics of the ground truth dataset.

4.5.5 Applying Monte-Carlo Search Techniques

Apart from the traditional tree traversal algorithms, I also worked on searching the program space with the Monte Carlo Tree Search (MCTS) algorithm (Browne et al., 2012). As its

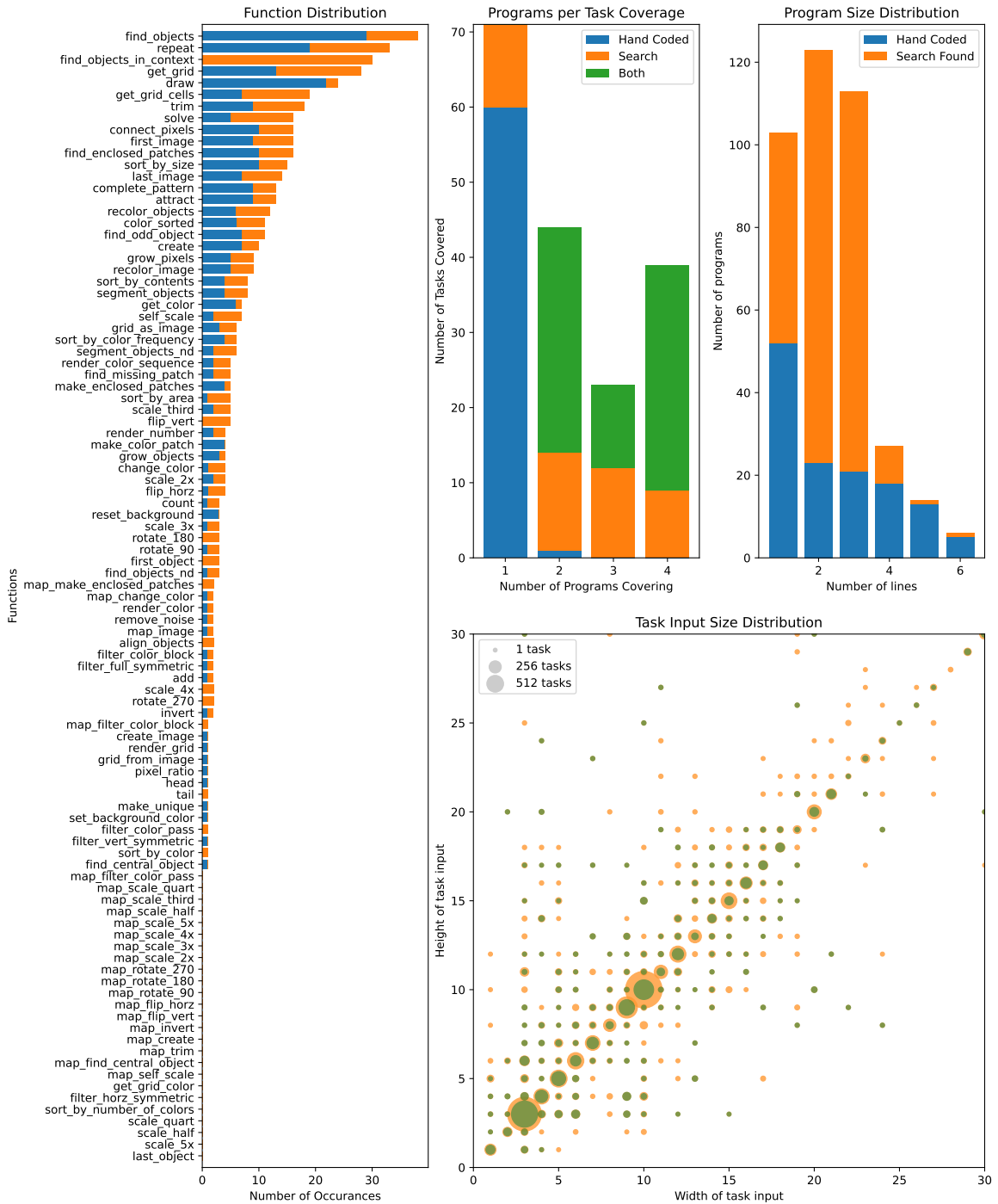


Figure 4.8: Characteristics of the ground truth dataset. The chart on the left shows the distribution of operation occurrences across all programs in the ground truth dataset. The upper-middle chart shows the distribution of the number of ground truth programs per task. The upper-left chart shows the distribution of program sizes across the entire ground-truth dataset. And the lower left chart shows the distribution of task sizes in the full dataset versus those that have been solved in the ground truth dataset.

name suggests, MCTS takes inspiration from Monte Carlo methods that are typically used for the approximation of complex mathematical functions (Robert & Casella, 2005). MCTS explores search spaces by sampling search nodes to build a tree that potentially leads to an optimal decision. One advantage MCTS brings is its non-heuristic nature, which allows it to work provided the problem’s rules are properly defined. The random exploration, which is directed by a mathematical model that directs a balance between exploration and exploitation, leads to a search that can be both efficient in nodes explored and in covering nodes that are most likely to be successful.

The basic structure of the MCTS algorithm is defined as a cycle over four different stages. These stages—selection, expansion, simulation and back propagation—can be implemented differently to create flavours of the MCTS algorithm (Świechowski et al., 2021). Selection is concerned with selecting the next best node to expand, expansion expands any selected nodes, simulation evaluates the selected node to yield a reward, and back propagation backs up the reward from the selected node to the root. The selection and expansion sections together are known as the tree policy of an MCTS implementation, and the simulation part is known as the default policy.

For the work in synthesizing state machines, I tried out the most widely implemented tree policy for MCTS, the Upper Confidence Bounds applied for Trees (UCT) (Browne et al., 2012; Kocsis et al., 2006; Kocsis & Szepesvári, 2006). When using UCT, the successor of each explored node is evaluated through the default policy, at least once. UCT’s tree policy selects nodes to be expanded according to the following formula:

$$\operatorname{argmax}_{\varphi' \in G(\varphi)} \frac{Q(\varphi')}{N(\varphi')} + c \sqrt{\frac{2 \ln N(\varphi)}{N(\varphi')}}$$

Here, $G()$ is a successor generator function, which takes a VIMRL programs and returns a list of its successors using the same techniques as used in standard tree search (see Section 4.5.1). $Q()$ returns the reward value for a given node, $N()$ represents the number of times a node has been visited, and φ represents a state machine. On the successor generator, G , because MCTS inherently builds a tree as it explores the environment, in cases where a stochastic successor generator was implemented, the children of a given node were sampled

once and fixed throughout the rest of the search.

The default policy phase is where the selected search node is simulated to compute a reward value. In game play scenarios, for which MCTS is mostly used, this phase of the algorithm will have the entire game played out to the end, with rewards computed accordingly.

For the purposes of synthesizing VIMRL programs, I tried out two different default policies. The first policy simply evaluated the program at the current selected node and returned the α value as the score, and my second policy went further by randomly adding successors to a node, and evaluating it at each stage until a leaf node is reached (in the case of the stochastic successor generator) or a pre-defined depth is reached (in the case of the default successor generator). Once a terminal node was reached, the best α score observed was backed up. It is also worth noting that, any program that had an α score greater than the search threshold was saved as a potential goal from which the best three would be selected according to the selection algorithm in use.

4.5.6 Search Space Pruning and Optimization

Even when potential successors were stochastically sampled, the search was still expected to grow. Although this inevitable explosion in the search could not be entirely prevented, steps could be taken to ensure only meaningful programs were explored and expanded, leading to a diminished search space.

For the work described in this dissertation, I considered three primary forms of node pruning to control search space expansion. These techniques were: depth limiting, reference gaps, input repetition, pruning of logically equivalent programs, early program evaluation.

4.5.6.1 Depth Limiting

My first attempt at search space management was to limit the maximum depth of search to a fixed number of instructions per program. This step placed a hard limit on the entire search space, making it finite. For a fewer number of instructions (up to about 2 instructions per program), the entire search space of all programs could be covered without any changes to

the search algorithm, given the number of operations currently in use.

4.5.6.2 Reference Gaps

Reference gaps were strict requirements enforced on programs to ensure that any variables declared were used within a given number of instructions. For example, if search was restricted to a reference gap of 2, then a variable defined must be used within two instructions or the program violated the restriction. During search, any nodes representing programs that failed to conform with the reference gap were pruned.

4.5.6.3 Logically Equivalent Programs

There were also cases where two programs had the same instructions, but were presented in different sequences. These programs could be considered logically equivalent because they produced the same final execution state and output. During search, because these programs had instructions that were sequenced differently, they were treated as separated distinct nodes. Executing these logically equivalent programs introduced inefficiencies in search, especially when you consider that every logically equivalent program was further expanded.

To prevent duplicated, logically equivalent programs from being executed multiple times, I implemented a sorting algorithm that ensured all logically equivalent programs were sequenced the same. This sorting algorithm involved a dependency tree in which the locations of instructions that consumed variables were child nodes to the locations of instructions that created the variables. When this tree was topologically sorted, the sequence of instructions were re-ordered such that an instruction was used almost as soon as it was defined. After sorting, all the variables were renamed through an enumeration scheme to ensure that variables had the same name across all sorted programs. During search, every search node was preemptively sorted before it was entered into search.

4.5.6.4 Early Program Evaluation

When evaluating VIMRL programs on ARC tasks, the last variable written to was considered as the final output of the task. This meant that a program's output was to be

considered valid for evaluation only when it was written as an image typed value. In some cases, however, to prevent searching for extra programs, values that were typed as a list of objects were rendered unto an empty input-sized image for evaluation. Because some tasks in the ARC required objects to be manipulated and rendered back to input, this approach pre-emptively caught some correct programs before any extra instructions were added on.

4.6 Implementation Details of VIMRL ARC Experiments

All search algorithms and techniques discussed in this chapter were implemented in the Python programming language (Version 3.11; Van Rossum and Drake, 2009). For parsing the ground truth programs, the python client for ANTLR4—ANother Tool for Language Recognition—by Parr (2013) was leveraged to build a language parser. And finally, for handling image data and implementing most of the functions underlying VIMRL’s operations, I used the NumPy numerical computation library (Harris et al., 2020).

As already evident from earlier discussions in this chapter about optimizations (see Section 4.5.6), managing the size of the search space was a major challenge. Even with all the optimizations in place, a single computer system failed to keep up. Fortunately, program searches for tasks in the ARC were independent of others. It was, thus, possible to run searches in parallel to improve throughput.

Parallel searches were performed in two domains: directly on a single computing device through python’s multiprocessing API to leverage multiple cores on the CPU and, even more effectively, through an ad-hoc cluster of computer systems, each running the aforementioned multiprocessing enabled client.

The ad-hoc cluster was based on plain consumer grade workstations connected over the Internet. To keep the system’s development simple, all systems were configured to run on the same version of the Fedora Server Operating System (version 35).

The backbone of the cluster was a self-hosted WireGuard VPN, to which all computer systems participating in the cluster were connected. For the roles played by connected systems, there was a central Redis server that stored a job queue of tasks to be processed and kept track of results, there were publisher machines that managed that pushed tasks unto the queue, the bulk of machines acted as workers that performed the actual search

operation, and finally there was a single machine that provided a web based user interface for performing cluster operations. The web portal allowed jobs to be scheduled and provided a centralized destination for viewing results (which included the programs generated, outputs of all programs regardless of accuracy, timing information, and the final accuracy scores).

Whenever an experiment was to be performed, the publisher machine put together a file that listed all the tasks to be evaluated (using their hexadecimal identifiers), as well as a pointer to the dataset from which the tasks could be found. For convenience, and to reduce network traffic, as well as to simplify the software implementation, a copy of all datasets was made available on each worker system. Workers were constantly connected to the publisher through a blocking Publisher-Subscriber (Pubsub) link. This meant that as soon as a job was placed on the queue, a random worker could pick it up and start executing it immediately. Once a worker's search was complete, it pushed its results back to the Redis server and whenever all tasks in a job were complete, the publisher collated results and updated the web portal.

Through this ad-hoc cluster I was able to improve search times, which in turn increased my experimentation turnaround time. But, although this cluster worked well, I had a few blocking challenges to resolve along the way. The first major issue I faced came from software synchronization. As the number of workers increased, there were times I forgot to update some of the workers, and this led to jobs having different tasks solved by workers with different implementations of the search algorithm. I resolved this issue by requiring each worker to pre-emptively self-update before executing search for tasks in a new job.

Another issue I had to deal with came from clients getting disconnected (for various other reasons) before they completed and had the opportunity to submit results for any jobs they were solving. To fix this issue, each task sent to a worker was given a timeout duration, and whenever this timeout expired, the task was pushed back to the head of the queue. Resolving connection issues was a major routine maintenance task I had to perform because connections occasionally went down.

A final major issue I had to deal with came from the mismatched performance provided by the different computer systems used as the workers. In dealing with this issue, I explored

assigning weights to different systems such that the amount of time the machine was expected to spend was scaled by its weight. With this solution, slower machines had the opportunity to spend a longer time searching when compared to their faster counterparts. The weights for the machines were arbitrarily assigned according to my subjective judgment of the relative performance differences between the machines.

4.7 Experiments

I performed two experiments to evaluate the ability to solve ARC tasks by searching a space of strategies specified as VIMRL programs. After producing a suite of ground truth programs to verify the sufficiency of VIMRL for representing ARC strategies, the first group of experiments were meant to explore how well the different tree traversal algorithms worked, while the second group were meant to investigate how effective the different techniques adopted for pruning and managing the search space's size were.

4.7.1 Experiment I: Tree Traversal

For the tree traversal experimental runs, I varied parameters that drove the search, including the choice of a tree traversal algorithm, successor generation algorithm, branching factor (where applicable) and maximum search depth. The goal was to find the combination of parameters that worked best according to the number of tasks solved from the public ARC tasks.

Tree-traversal was handled by either a Breadth First Search, a Depth First Search, or a Monte Carlo Tree Search exploration. Successor generation was as choice between a full brute-force search, or a probabilistic sampler that relied on either a uniform distribution, MLE estimated distributions, or MAP estimated distributions (see Section 4.5.3). In all experiments involving stochastic methods of successor generation, I fixed the branching factor to either 5 or 10 and kept the maximum depth of search between 3 and 10. All experiments were executed with a timeout of 800 seconds per task on an ad-hoc computer cluster with 22 worker nodes. Experiments reported in this dissertation were performed on the publicly available ARC dataset.

Other than removing logically equivalent programs and early evaluation of some pro-

grams, none of the pruning rules were activated for the trials in this experiment.

4.7.1.1 Results

Results for the best performing search runs are shown in Table 4.7. For these results, an experimental run was considered to be the best if it had the highest recorded evaluation score for a particular tree traversal and successor generator combination, while having the lowest branching factor and the lowest maximum depth setting.

Table 4.7: Best results observed for the different forms of search traversal. For stochastic methods, these were results taken over several experimental runs.

#	Tree Traversal	Successor Generator	Branching Factor	Max. Depth	Train Score	Evaluation Score
1	BFS	Brute-Force	-	3	106/400	44/400
2	BFS	Uniform	5	5	34/400	9/400
3	BFS	MLE	5	5	70/400	17/400
4	BFS	MAP	5	5	46/400	9/400
5	DFS	Brute-Force	3	3	46/400	12/400
6	DFS	Brute-Force	5	5	31/400	12/400
7	MCTS	Brute-Force	10	10	40/400	6/400
8	MCTS	Uniform	10	10	36/400	6/400
9	MCTS	MLE	5	10	42/400	6/400
10	MCTS	MAP	5	10	36/400	6/400

4.7.1.2 Discussion

The best performing tree traversal combination (#1) was the breadth-first search, coupled with a brute-force successor generator. Due to the high branching factor obtained from expanding all possible successors, this search could only progress to a depth of 3 nodes, leading to programs with a maximum size of 3 instructions. Even with a depth limit of 3, the entire space was hardly ever completely searched, with all unsuccessful searches timing out.

When BFS is coupled with any of the stochastic successor generators, it does not perform as well. The selective nature of picking successors led to a limited search of the larger tree. When uniformly sampling successors, which was an uninformed strategy, the stochastic generators got the worst results.

The only DFS results reported has it coupled with a Brute-Force search. Compared to

the BFS implementation, it performed significantly worse, and this was expected. Searching with a DFS approach was not bound to explore a wider range of programs. It was however able to search deeper, although it never found any meaningful programs beyond a depth of 3. Nonetheless, it appeared test scores from the DFS search runs were both the same, due to the programs being closer to the root of the search tree with either 1 or 2 instructions.

On the trials that relied on coupling stochastic successor generation with BFS, MAP and MLE based distributions performed better. After all they were influenced by patterns in the ground truth dataset in making their predictions, unlike the uniform distribution.

Interestingly, performance on MCTS search was close regardless of the successor generation algorithm. This could probably be due to how the MCTS algorithm inherently selects nodes to expand. Also, all the results obtained on the evaluation data for MCTS were on the same set of tasks.

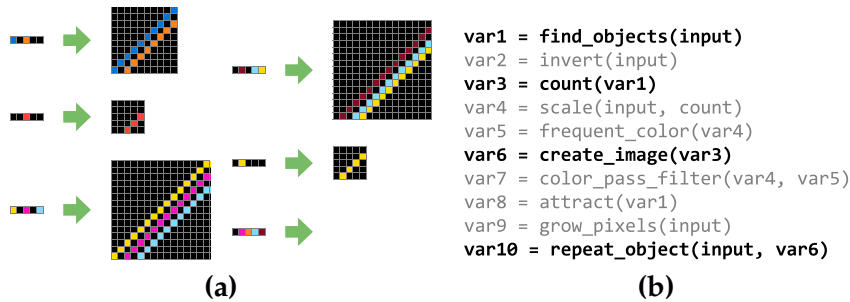


Figure 4.9: A task and its MCTS generated solution. Instructions shown in bold were actually responsible for solving the problem. The others greyed out instructions had no effect on the final solution.

On the size of programs, it was interesting to note that having larger programs did not necessarily lead to successful programs. All across the stochastic search algorithms, the longer programs were typically filled with dummy instructions that had no effect on the actual logic of the program. For example, the program in Figure 4.9 (b) was generated by MCTS to solve 4.9 (a). Although this particular program was 10 instructions long, only 4 of the instructions actually worked towards solving the problem. Here is what these instructions did:

1. The first instruction finds all objects in the input image.

- The second instruction counts the number of objects found in the list of objects obtained in item 1.
- The third instruction, which is a high-level one that determines how numbers affect the size of an image, creates an output image whose size is a fixed factor of the number of objects.
- The final instruction repeats the input image to fill the newly created output image.

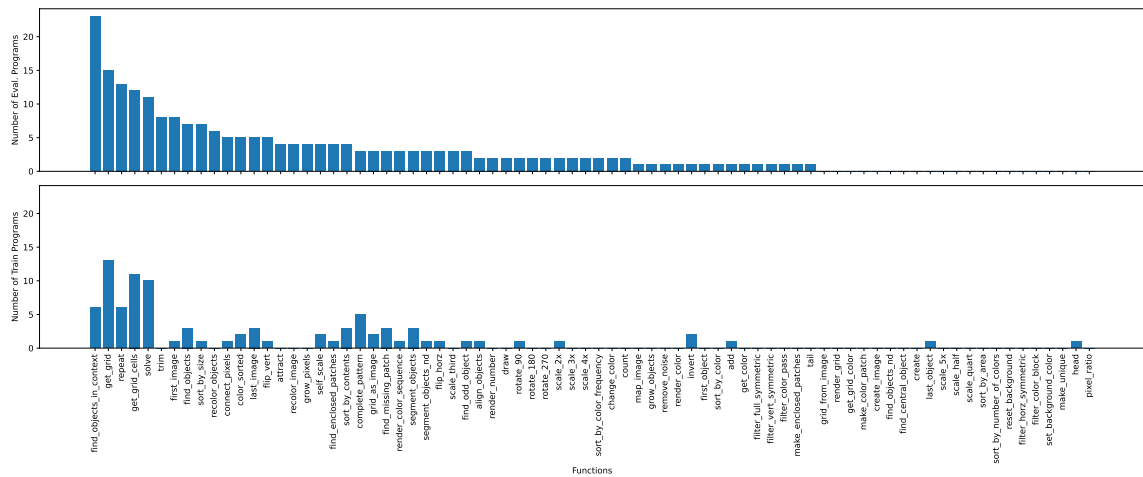


Figure 4.10: Function distribution

4.7.2 Experiment II: Pruning and Selection

In my second experiment, I investigated the effects of pruning rules and final result selection on had on the final accuracy for the solver. For these experiments I ran the search with tree traversal fixed to BFS and a brute-force successor generator working to a maximum depth of 3 (in line with the best performing configuration from Experiment I).

Along with the fixed parameters, I varied the pruning rules, alternating whether repeated inputs were either enabled or not, while varying the reference gap between 0 (not being enforced) and 1 through 3, I altered the program selection algorithm, choosing between either the unique program and smallest three best-scoring programs algorithms. Because of the large search spaces involved, the logically equivalent and early execution pruning rules were necessary if any of these experiments had to achieve a decent score.

Considering all the fixed and varied parameters, there were a total of 16 unique configurations in this experiment.

4.7.3 Results

Scores on the public ARC dataset’s training and evaluation tasks, as well as the total number of nodes executed through the search, are displayed in Figure 4.11 for the 16 configurations tested. Performance on training ranged from 106 out of 400 to 115, and on the evaluation side, it ranged from 42 to 49, corresponding with a peak performance of 28.75% on training and 11.25% on evaluation respectively.

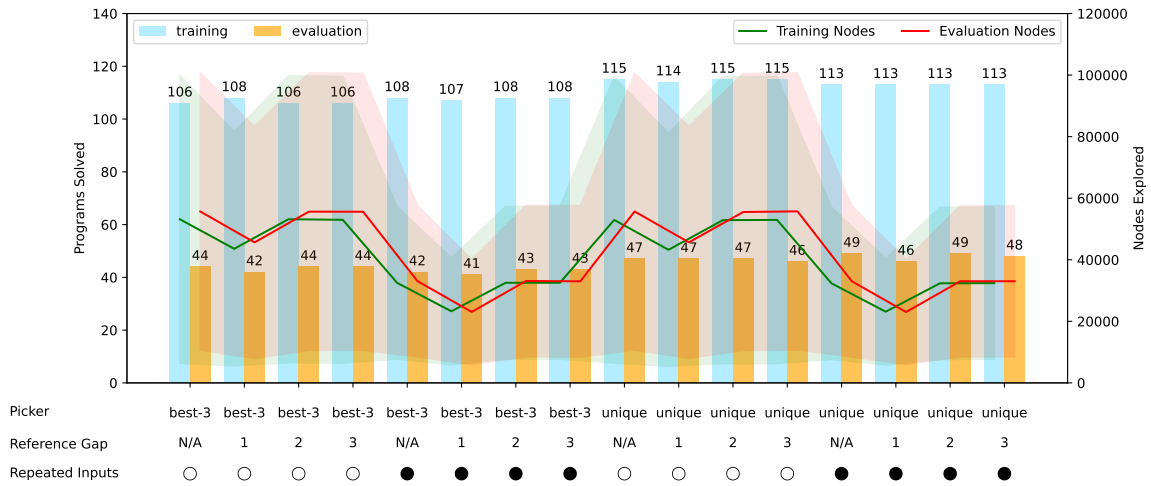


Figure 4.11

4.7.4 Discussion

Across the board, there were three best performing configurations when considering the training set. These results were all from searches where a unique result was picked and the repeated inputs pruning rule was not active. For the best performing configurations on the test set, there were 2. These best performing configurations still used the unique result picker, but had the repeated inputs pruning rule activated.

Overall, whenever the repeated inputs pruning rule was active, significantly fewer nodes were evaluated. In fact, the fewest nodes were expanded whenever repeated inputs were combined with a reference gap of 1. This was because with a reference gap of 1, a

variable had to be used immediately it was defined, and several potential programs would not meet that requirement. This aggressive pruning may not have been as effective, since most of the runs configured with a reference gap of 1 performed worst when compared to runs with similar configurations.

When it came to the final selection of outputs, the results show that picking a set of unique programs was a better choice for a picker algorithm. Across the board, configurations that chose unique programs scored above 113/400 on training and 46/400 on evaluation, whereas the best-3 picker peaked at 108/400 on training and 44/400 on evaluation.

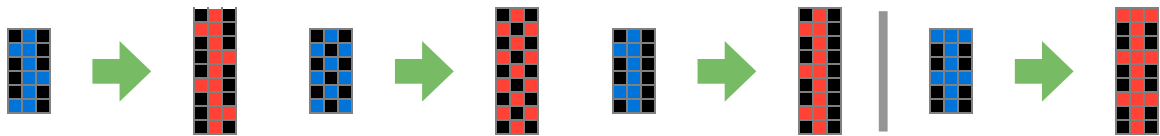








Figure 4.12: A sample task from the training section of the ARC

For an illustration of the kinds of choices both pickers made, consider the task displayed in Figure 4.12. In this task, it appears the solver was expected to vertically repeat the input image with a 3 pixel overlap. All input images were 3x6 pixels and all outputs were 3x9. The operations available to my solver could not have produced a valid program—when considering one that solves all train items, including the test. There were, however, operations that could solve some of the train items. Table 4.8 shows the three final choices made by the two different pickers for this problem.

From the table, it is obvious how the best-3 picker honed in on choices that that scored highest on the training items, although they were not able to make the right prediction for the test item. All the choices made were programs that produced the same output and were merely simple copies of each other with minor modifications. Allowing for a diversity of strategies through the unique picker, however, saw final choices that scored on just a single training item being selected to ultimately make the right prediction.

Table 4.8: The sets of three programs generated by both the Unique picker and the Best 3 picker.

Unique	Best 3
<p>SCORE 0.67</p> <pre>var1 = change_color(input, 2) var2 = repeat(var1)</pre> 	<p>SCORE 0.67</p> <pre>var1 = change_color(input, 2) var2 = repeat(var1)</pre> 
<p>SCORE 0.33</p> <pre>var1 = flip_vert(input) var2 = change_color(var1, 2) var3 = repeat(var2)</pre> 	<p>SCORE 0.67</p> <pre>var1 = trim(input) var2 = change_color(var1, 2) var3 = repeat(var2)</pre> 
<p>SCORE 0.33</p> <pre>var1 = change_color(input, 2) var2 = make_enclosed_patches(var1) var3 = repeat(var2)</pre> 	<p>SCORE 0.67</p> <pre>var1 = change_color(input, 0) var2 = change_color(input, 2) var3 = repeat(var2)</pre> 

4.8 2022 ARCATHON Run

In addition to the results shared here, the best performing algorithm from Table 4.7 (#1) was entered into the 2022 International ARCATHON competition, where it tied for 4th place after solving 2 tasks from the hidden private test set. This competition was organized by LAB42, a Swiss based AI research organization.

Submissions for the ARCATHON were sent to the LAB42 team as docker images that were executed on n1-standard-4 virtual machines from the Google’s cloud platform. These machines had access to 4 CPU cores and up to 15 GB of random access memory. During evaluation, each submission had up to 9 hours to solve 100 tasks from the private ARC dataset. Submissions that required GPUs were given up to 3 hours and access to an NVIDIA T4. My submission executed solely on the CPU. Table 4.9 shows the competition’s final leader board.

4.9 Conclusion

Generally, the ARC is considered a significantly challenging problem for AI systems. Its limited training items and lack of language use, reduces the effectiveness or even the ability of most current AI systems to solve this task. Regardless of this difficulty, program synthesis approaches seem to be one of the ways to make meaningful progress in solving tasks on

Table 4.9: Final leader board for the 2022 International ARCATHON Competition showing the best entries. The entry from this dissertation is shown in boldface.

Text	Country	% of Tasks Solved	Score	Entries
pablo	Switzerland	6%	0.94	5
Mirus	Switzerland/Slovakia	3%	0.97	4
notXORdinary	Denmark	3%	0.97	1
AIVAS	USA	2%	0.98	3
MADIL	France	2%	0.98	3
Zeroone	France	1%	0.99	2
ARGA	Canada	0%	1	7

the ARC.

The techniques described in this chapter was successful at solving tasks from the ARC mainly because, in the best case, a space of potential domain specific programs was exhaustively searched till ones that solved the task was found. This exhaustive search property, however, acts against when it comes to solving a larger variety of ARC tasks. With ARC tasks having varying concepts, a solver that searches exhaustively will require a large set of operations, which in-turn makes search more expensive.

To cut back on this search cost, informed approaches that selectively search the space of programs may be necessary. But, as the results from this chapter show, those did not work as well. And that is mainly because informed approaches require knowledge to allow discernment of what a better potential program is. Building this knowledge will nonetheless require a full corpus of ground truth programs and effective algorithms that can learn to generalize knowledge from those ground truth programs to any potential unseen ARC tasks.

CHAPTER 5

Exploring Strategy Differences on the Block Design Task

In continuing with the theme of synthesizing programs to represent strategies, the work described in this chapter was centered on applying the program synthesis techniques explored in Chapter 4 to the Block Design Task. I extended the language with new ways of expressing control structures to allow for the representation of more fluid reasoning techniques.

5.1 Extending VIMRL with Control Structures

For the work discussed in this chapter, I will be describing a space of programs that were represented as state machines. In these programs, operations were grouped into states, and transitions between these states controlled the program's flow. It is worth noting that using state machines for control flow made the language extremely expressive. With this expressiveness came the challenge of large search spaces that were expensive to search.

Regardless of the limitation in the cost of search, I still considered the choice of state machines since it allowed the transformation of VIMRL into a representation sufficient for human-like fluid decision-making. After all, as humans we do not always reason through tasks in a fixed sequence of steps; we tend to take in information about the task, and we make decisions about our potential next steps by choosing from a set of potential options.

Throughout the rest of this chapter, I will explain the state machine representation for programs and its associated language, which is an extension of VIMRL. Then, I will go on to describe the different experiments I performed on the Block Design Task (Kohs, 1920) towards applying this representation to a real world task.

5.2 An Overview of State Machines

State machines are a model of computation built around the interaction of distinct states, directed by a transition function and an input alphabet. Whenever a state machine is operated, there is a single active state, and based on the inputs received by the machine, this

active state is switched according to the transition function. State machines are widely used to describe and model the behaviour of complex systems. A commonly used specialization of the state machine, in which there are a fixed number of states, is the finite state machine.

For a practical example of a state machine model, consider a simplified version of a car's trunk door. In simple terms, the door could either be open or closed. These two properties represent the distinct states in our door model. Our door can only be open or closed at any time, but can never be in both (closed and open) states at the same time. The actions we can perform to switch the active state will be to press a button that opens the trunk door or push the trunk door shut. These two actions can be considered as inputs to the machine. We can visualize this simplified model as the state machine displayed in Figure 5.1.

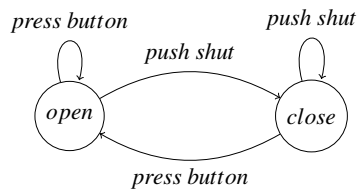


Figure 5.1: A state machine model for a simplified trunk door with two states and two inputs. The circles represent the states and the arrows represent the transitions that take place on specific actions.

We can take this model a step further by adding a state to represent a locked trunk door. In this case, we will need additional actions for locking and unlocking the door. With this extension, the new model will become as described in Figure 5.2.

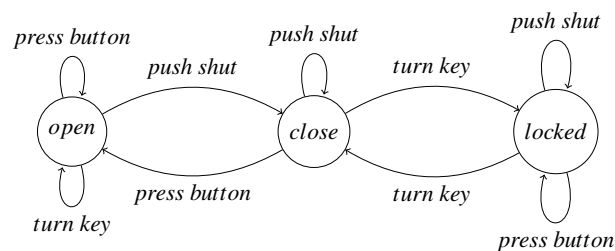


Figure 5.2: An extended version of the state machine described in Figure 5.1. This extension adds an extra state to represent a locked door and an extra input for locking and unlocking the door.

5.2.1 Formal Representation of State Machines

Formally, a state machine can be defined as: $\varphi = (\Sigma, S, s_0, \delta, F)$. Where Σ is the input alphabet, S is the set of states, s_0 is the initial state, which is a member of the set of states S , F is a set of final states and δ is the transition function. The transition function is defined over states and inputs, such that $\delta : S \times \Sigma \rightarrow S$.

The nature of the transition function makes the state machine either deterministic or non-deterministic. Deterministic state machines have transition functions that provide a one-to-one mapping. As such, an input will always switch the machine between two specific states. In non-deterministic machines, an input alphabet could potentially switch the machine from one into multiple states, and a final decision must be made through a tie-breaking procedure.

In our simple trunk door model, our finite state machine will have three states: *open*, *close* and *locked*. And there will also be three inputs: *press button*, *push shut*, and *turn key*.

5.2.2 Prior-work in Synthesizing State Machines

State machines are heavily utilized in automata-theory, a research area that has been active since the very early days of computer science (e.g. see Minsky, 1967; Shannon and McCarthy, 1956). Automata-theory also has a direct relationship with formal languages, and formal languages form the basis of how we communicate with computers (Hopcroft et al., 2006).

Biermann (1978) presented early work on the synthesis of state machines from sets of input-output pairs. With foundations laid by Nerode (1958), who discussed how state machines could be represented as a system of linear equations, Biermann's synthesizer took as input of a finite set of input-output pairs and a parameter k to synthesize a state-machine. As expected, the input-output pairs provided information on the behaviour of the system, and the k parameter determined how precisely the final synthesized state machine characterized the underlying function of the input-output pairs. With lower values of k , the machines tended to be non-deterministic and had fewer states. As k increased, however, the number of states also increased and the machine became more deterministic. In some ways, the parameter k could be considered as the number of degrees of freedom the synthesizer

had; higher values of k led the synthesizer to overfit its input.

Several practical applications of state machine synthesis can be found in the field of microelectronics. In the design of semiconductor chips, state machines may be used to communicate how some of the operational logic on the chips are represented. Because of this key role state machines play, efforts to build synthesizers that convert logic represented in more descriptive higher level languages to state machines have been of great interest in the field of microelectronics. ESTREL (Berry, 2000) and LUSTRE (Halbwachs et al., 1991) are both examples of higher level programming languages that have the capability to synthesize their outputs to state machines. Both languages are modelled around the idea that data flows through a system in the form of signals that other components need to respond to.

Another area in which state-machines play a role—much similar to what happens in this work—is in the implementation of Non Player Characters (NPCs) in video games (Hy et al., 2004). The behaviours of these characters, their animation, as well as, how they interact with other characters are generally implemented through state machines. Although most of these state machines are hand designed by the builders of the video games, some of these can be built to self modify and dynamically adapt to other in game elements.

5.3 Modelling Reasoning with State Machines

In several ways, the state machines used for the work in this chapter fit into the formal description provided in Section 5.2.1. The behaviour being modelled by these machines were that of an agent reasoning through tasks in a virtual environment. Each state had a list of instructions, in the form of a VIMRL program called the state script, which was executed whenever the state was active. Instructions in these state scripts could create and assign variables (similar to the process described in Section 4.2), which become globally accessible to all other scripts, and can call agent specific operations (like getting an agent to pay attention to an object) through a mechanism similar to a function call in a conventional computer program. Transitions between states are triggered by conditional statements that are defined on the variables found in the state scripts.

Formally, the state machine structure used for this work was defined as: $\varphi = (S, s_0, V, \delta)$. Here, S was a set of program states, s_0 was the initial program state, V was the set of

variables assigned within the state scripts, and δ was the transition function for program states. The state script for a given program state, s_i could also be represented as an ordered set of instructions, $s_i = \{x_1, x_2, \dots, x_n\}$.

Transitions in the state machines were non-deterministic. Thus, in cases where a condition had multiple potential target states, the ultimate state was selected at random. With this, the transition function could be defined over states and variables as: $\delta : S \times V \rightarrow P(S)$. The choice to have non-deterministic transitions was made to eliminate the extra overhead incurred from ensuring that all conditions from a given state were mutually exclusive during search.

A bulk of my work described in this chapter went into exploring the structure of state-machine based search spaces and dealing with its related issues. It is worth noting that using state machines for control flow made programs extremely expressive. With this expressiveness came the challenge of larger search spaces that were expensive to search.

5.4 Integrating VIMRL and State Machines

As explained in Section 5.3, each state of the machine had a list of instructions, which was expressed as a VIMRL program, and states could have transitions to other states that were conditioned on boolean expressions defined on variables declared in the state scripts. Conditional expressions for transitions were specified in a VIMRL subset that allowed the definition of single boolean expressions. (See Table 5.1 for the grammar of this subset).

The design choice to limit instructions in state scripts to VIMRL and transitions to single boolean expressions was made to ensure that the flow of logic in a state machine was driven solely by transitions between states. Constructs from traditional programming languages, like if statements could be implemented through states transitioning to other states, and loops could be implemented with states transitioning unto themselves. These limitations in language features also meant searches for scripts in machines could be conducted over a smaller grammar.

For the purposes of executing state scripts, the VIMRL language itself remained the same as described in Section 4.2. Operations invoked special actions defined in the task's environment (for example, the agent in a VREE environment). Also, how variable values

were stored depended on the environment in which programs were executed. For example, machines that were executed in VREE had their variables stored in the agent's short-term memory, and the values stored there were subject to any limitations of the storage medium (such as simulated forgetfulness in short-term memory).

Table 5.1: Grammar for language used in specifying state operations.

Grammar for the language used in state scripts

$\langle instruction \rangle ::= \langle assignment \rangle$
 $\quad \quad \quad | \langle operation \rangle$

$\langle assignment \rangle ::= \langle identifier \rangle '=' (\langle number \rangle | \langle operation \rangle)$

$\langle operation \rangle ::= \langle identifier \rangle '(' \langle arguments \rangle ')'$

$\langle arguments \rangle ::= \langle argument \rangle$
 $\quad \quad \quad | \langle arguments \rangle ',' \langle argument \rangle$

$\langle argument \rangle ::= \langle identifier \rangle$
 $\quad \quad \quad | \langle number \rangle$
 $\quad \quad \quad | \langle operation \rangle$

$\langle number \rangle ::= ('-')?[0-9]^+ \langle identifier \rangle ::= [a-zA-Z][a-zA-Z0-9]^*$

Grammar for the language used in transition conditions

$\langle condition \rangle ::= 'not' \langle operation \rangle$
 $\quad \quad \quad | \langle operation \rangle$
 $\quad \quad \quad | \langle identifier \rangle '==' \langle identifier \rangle$
 $\quad \quad \quad | \langle identifier \rangle '!=' \langle identifier \rangle$
 $\quad \quad \quad | \langle identifier \rangle '>' \langle identifier \rangle$
 $\quad \quad \quad | \langle identifier \rangle '<' \langle identifier \rangle$
 $\quad \quad \quad | \langle identifier \rangle '>=' \langle identifier \rangle$
 $\quad \quad \quad | \langle identifier \rangle '<=' \langle identifier \rangle$
 $\quad \quad \quad | \langle condition \rangle 'and' \langle condition \rangle$
 $\quad \quad \quad | \langle condition \rangle 'or' \langle condition \rangle$

5.5 Synthesizing State Machines

In line with other forms of synthesized programs, state machines were generated through search. I investigated a suite of search algorithms that mainly employed an enumerative style of program search. These algorithms proceeded in a continuous *generate-and-test* cycle, where a state machine was generated according to a set of rules and evaluated against the specifications at hand. Exploration of the program space occurred through tree traversal, allowing for the use of traditional tree search algorithms, like breadth-first or depth-first search.

At the barest minimum, a search algorithm exploring the space of state machine programs needed as part of its inputs, the list of all operations supported by the agent and a list of any literal values the agent needed to be made aware of (for use as a form of knowledge in long term memory, or as inputs to the program).

5.5.1 Exploring the Space of State Machines

For the purpose of synthesis, I considered any state machine as a tree node. Then, I considered any duplicate of the machine with a single extra element as a successor (or child) of the node. Specific elements that could be added to generate successors included a new program state, a new instruction appended to the state script of any program state, or a new transition between states. Using these rules, it was possible to explore the space of state machines with conventional search algorithms.

Formally, searching the space of state machines started with a machine, φ_0 , which contained a single state, s_0 , that was considered to be the initial state. When φ_0 was expanded, its successors built on the state, s_0 , by extending the machine with extra elements. For example, a successor machine of φ_0 could be generated by adding another state, or by adding instances of possible instructions to the state script of s_0 .

When implemented, this tree traversal technique led to large exploding search spaces. Whenever a new machine was added, the potential to extend it with extra elements increased the branching factor of the tree. For example, when there was just a single state in a machine, successors that added operations only had the single state as an option. When extra states were available, however, potential instructions for these multiple states had to

be considered. This scenario repeated when considering instructions: assignment operations created variables that any subsequently generated instructions could consume. As the number of variables in a machine increased, the possible combinations of operations that could consume these variables also increased. This whole processes led to a branching factor that grew exponentially with increasing tree depth.

With this issue of space explosion in mind, I experimented with four different strategies for generating the successors when given machine during search. These were:

1. An unrestricted exploration of all possible machines, which led to a complete brute-force search when combined with traditional tree traversal algorithms.
2. A heuristic enhanced exploration in which potentially faulty machines were pruned from consideration in search.
3. A stochastic exploration, which relied on a probabilistic model estimated from a set of ground truth programs in a similar task.
4. A version of stochastic exploration that used heuristics to remove any potentially faulty machines.

5.5.2 Unrestricted exploration

Unrestricted exploration was an attempt to exhaustively generate all possible state machines for a given task. It was a direct implementation of the tree traversal process described in Section 5.5.1, without any modifications. Although ineffective at yielding solutions for anything but the smallest problems, the exhaustive nature of this exploration provided a great way to obtain some insights about the search space's nature.

When exploring without restrictions, successors for a given machine could be generated by the three different actions described below:

1. A successor could be generated by adding a single empty state to the machine.
2. For each state in the machine, a successor could be generated for each valid instruction that was added to any of its state scripts. A valid instruction in this case was one whose

operation had all the required arguments available in the state machine (either as variables, or literals). In cases where multiple candidates for an operation's arguments existed, successors were generated for every unique combination of arguments.

3. For any two program states in the machine, a successor could be generated for each valid transition between the two states. A valid transition in this case was one whose conditional expression had all the required arguments available in the state machine. Because transitions could start and end on the same program state (in the case of a self transition for a loop), the two program states under consideration for this action could be the same. Just as it was with operations, when multiple candidates existed for a condition's operation, successors were generated with transitions for each unique candidate condition.

5.5.3 Heuristics for Improving Search

Exploring every possible state machine made the unrestricted search strategy impractical for most problems, except for the very simple ones. One way to work around this problem was to employ heuristics that cut off search nodes which were likely to reach faulty state machines or ones that may not meet the task's requirements. In an attempt to provide some of these heuristics, I defined a couple of pruning rules for removing search nodes whose machines met certain given criteria.

During search, anytime a state machine was generated, pruning rules were applied, and machines that matched any of the pruning rules were promptly eliminated from the search process. Removing faulty machines saved time in the evaluation of successors and also prevented non-compliant nodes from getting expanded through search.

Algorithms that used these heuristics to improve search took as input an ordered list of pruning rules, which implicitly provided a hierarchy that gave some rules priority over the others. In all, there were six rules (single blank state, reversed machine, unreachable states, one-way true conditions, executable machines, and reference gaps) by which machines could be pruned in my experiment. The following sections describe these pruning rules.

5.5.3.1 Single Blank State

This pruning rule ensured that a machine always had a maximum of one blank state (a state with an empty state script) at a time. By pruning any nodes with machines that had two blank states, extra blank states that contributed nothing to reasoning were avoided. Having just a single blank state, however, provided an opportunity to grow the state machine, through instructions added to this blank state, as search progressed.

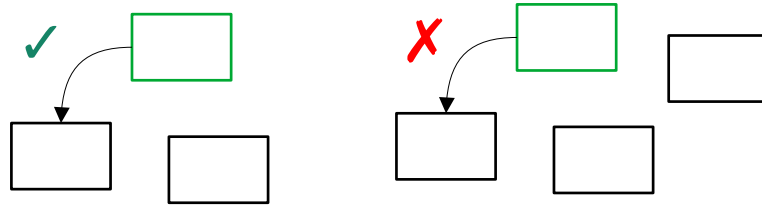


Figure 5.3: A depiction of state machines that has a single state (left) and a state machine that has multiple empty states (right).

5.5.3.2 Reversed Machine

A reversed machine is one in which there were no transitions originating from the initial state. Because the search algorithm tried to create every possible transition between states, machines with such configurations were sometimes generated as successors. Whenever executed, such a machine obviously stayed in the initial state throughout its execution, making the other connected states useless.

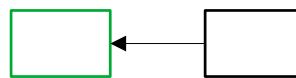


Figure 5.4: A reversed machine in which a state transitions into the initial state.

5.5.3.3 Unreachable States

Just as with reversed machines, sometimes machines had states that could not be reached during execution. This heuristic removed such machines, unless there was an empty, single unreachable state. This exception for single, unreachable states was made to ensure machines could be extended during search whenever a successor was generated to connect

the single unreachable state to the rest of the machine.

Because of its definition, the *Unreachable States* heuristic could also prune nodes that were candidates for the *Reversed Machine* heuristic. However, because the search for unreachable states tested all states, instead of just the initial state, the *Reversed Machine* heuristic was more efficient to execute in the specific case where the initial state was the only reachable state. Since pruning rules were applied according to a hierarchy, it was therefore more efficient to execute the reversed machine rule before the unreachable state rule.

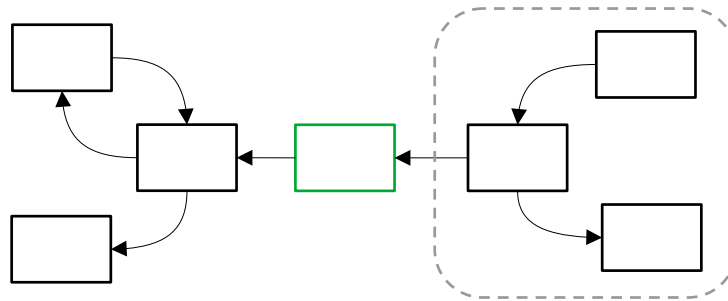


Figure 5.5: A state machine with an unreachable cluster (existing in the dotted line).

5.5.3.4 One-way True Conditions

With machines transitioning on boolean expressions, sometimes the triggering condition was a literal *True* value, or an expression that always evaluated to *True*. In such cases, it was not necessary for a return transition to exist from a destination state with a similar, always true condition. This situation created a cycle of true values that led to an implicit infinite loop. Because of how this rule was defined, it had the potential to affect machines that had self looped always true transitions.

5.5.3.5 Executable Machine

During search, when a transition was established between any two states, the order in which variables were defined (in the source state) and accessed (in the target state) was not verified. This sometimes caused transitions to originate from states that consumed variables and end in the states that defined the variables, effectively reversing the order in which a variable was defined and accessed. In such a situation the machine would not be

executable, since accessing an undefined variable was bound to produce an error.

5.5.3.6 Reference Gaps

Given that state scripts in the machines were written in VIMRL, it was possible to apply the reference gap pruning technique (see Section 4.5.6.2). This pruning rule ensured that variables in states were referenced within a given number of instructions after they were defined. Considering that state machines had scripts spread out over different states, for variables defined in the tail end of a state's script, checks were made within transitions originating from the state or the list of instructions in connecting states.

5.5.3.7 Logically Equivalent Machines

Similar to reference gaps described above, logically equivalent programs, as discussed in Section 4.5.6, could also be applied. With state machines, however, each state script was first individually sorted, before variables were renamed, then the individual states and their associated transitions were compared one-to-one to ascertain a match.

5.5.4 Stochastic Successor Generation

Even when the heuristics discussed were applied, running a search exhaustively ultimately led to an explosion in search states. This made it difficult to find machines that could represent complex reasoning strategies. Another way of exploring search spaces that could limit this exponential growth was through a stochastic search processes. Here, only successors that were likely to yield successful state machines were generated.

Because stochastic search was an informed process, a set of ground truth state machines were manually defined for instances of sample problems in the task. These instances were then used to generate a model of how state machines were typically constructed. From these ground truth programs, a probability distribution of the number of states in a machine, how transitions occurred between states, and how instructions for state scripts were sequenced was estimated. During search, these probability distributions were then sampled to generate just a limited number of potential successor nodes.

Stochastic successor generation worked in three phases, just like the unrestricted ver-

sion. The only difference here involved generating only a fixed sample of probable successors instead of all possible ones. This way, search proceeded with samples taken in the first phase for whether a state should be added, in the second phase for the instructions to be added, and in the third first for transitions that should be established.

5.5.4.1 Sampling Successors that Added States

In the first phase of search, states were added to the machine. The probability of adding a new state, s' , to an existing machine, ϕ , to increase the number of states in the machine to a value k was the probability that a machine taken from the set of ground truth programs had k or more states in its machine, $P(n_s > k)$, where n_s represents the number of state machines. Because every single machine in the ground-truth had at least one state, the probability of adding a state to a blank machine—which was the root node—was always one. This probability dropped to zero after the number of states in the synthesized machine exceeded the maximum number of states from machines found in the ground truth dataset.

5.5.4.2 Sampling Successors that Added Operations

In the second phase, instructions were added to states in the machine. Here, for successor nodes in which operations were added to states, candidate operations were modelled as a Markov chain, such that the probability of adding an instruction x_i is conditioned on its preceding instruction x_{i-1} , represented as $P(x_i|x_{i-1})$ (as also described in Section 4.5.3). The chain was initiated by a special opening instruction x_{open} , which represented the beginning of the instruction list, and was similarly terminated by another special x_{close} instruction. Transition probabilities for the instructions were computed from the ground truth programs through maximum a posteriori (MAP) estimation. It must be noted that the operations sampled at this stage were incomplete, since they did not have any arguments. Similar to the work described in Section 4.5.3.2, arguments were sampled separately in a bid to simplify the computation of probability estimates.

5.5.4.3 Sampling Arguments for Operations

Arguments were sampled separately as part of the third phase. The value, v_i , for the i^{th} argument of an operation x_j was sampled from the probability distribution, $P(v_i|x_j, i)$. This distribution was also computed from the ground truth programs using maximum a posteriori estimation. Values for arguments could be variables or literals. In the case of literals, the probabilities were computed from actual values within the ground truth programs.

For variables, the probabilities were based on the variable's provenance (see Section 4.5.3.2). A variable's provenance kept track of the operation from which the value of a variable was assigned. Since the intrinsic value of variables were only meaningful at runtime, estimating the probability that a particular argument was a variable did not provide any extra information when a machine was not in execution. Instead of just the fact that a variable existed as an argument, estimating the probabilities of variables defined by the operation from which the variable got its value allowed for the selection of candidate variables that were likely to contain the right kind of value.

5.5.4.4 Sampling Transitions between States

The probability of adding a transition between states during synthesis was equal to the probability at which a transition existed between any two states in machines from the ground truth dataset. This probability, $P(\text{transition})$, was computed from the dataset as:

$$P(\text{transition}) = \frac{\sum_i^M T_i}{\sum_i^M C(S_i, 2) + S_i}$$

, where M was the number of items in the dataset, T represented the number of transitions in a state machine, and S represented the number of states in a state machine.

Whenever it was predicted that a transition could exist between any two arbitrary states during synthesis, the condition governing the transition was also sampled from a probability conditioned on the last instruction from the transition's originating state script. Conditions for transitions were sampled from $P(c|x_{last})$ where c was the condition on the transition, and x_{last} was the last instruction of the originating state's script. Each condition

sampled could be considered as an operation, which needed arguments, and the sampling mechanism used for operation arguments was also applied to the process of sampling arguments for conditions (see previous section).

5.6 Experiments on Synthesizing Strategies for the Block Design Task

The experiments described in this section were concerned with the synthesis of state machines for studying strategy differences in the block design task. Unlike earlier work described in this dissertation (in Section 2.5.2) I eschewed imagery based representations in favour of symbolic ones for the work described in this chapter. This choice was made in favour of speeding up initial experimentation and reducing the extra complexity imagery representations brought.

Also, due to this use of symbolic representations, the VREE environment (as described in Section 3.3) could not be used for the experiments in this chapter. Instead, a simplified version that equally relied on symbolic representations to convey information about objects was built. In this simplified VREE instance, the concept of agents and objects remained, and affordances still allowed agents to interact with objects, but the selective visual attention mechanism was replaced with simple symbolic assignments.

5.6.1 Simplifying the Block Design Task

In addition to the changes in representations used for reasoning, the block design task was also simplified in various ways to increase the feasibility of initial experiments. Most of these simplifications involved combinations of reducing the task's scale, and reducing the degrees of freedom agents had to manipulate blocks. For instance, tasks could require fewer blocks or the manipulation of blocks could be restricted to specific actions. The plan for this phase of simplified task experiments was to find strategies for simpler related tasks and gradually build up complexity until the full block design could be solved by completely synthesized programs.

The first simplified task, which was also intended as the first test for program search, was set up such that there was a single block in the block bank and a single block in the design. Additionally, the block's face in the block bank was selected to be the same as that

on the design. In this particular configuration, the agent's job was to only pick the block from the block bank and place it in the construction area. I called this the *single block task*.

The second simplified task took the same approach as the first—there was a single block in the construction area and a single block on the design. The difference here, however, was that the design on the block was chosen such that a rotation over a single axis was enough to find the face required by the design. I called this second task the *single block single axis task*.

The third simplified task extended the second task by allowing multiple (two or more) degrees of freedom on the blocks. Of course, a few constraints were put in place to ensure the degrees of freedom were not direct complements of each other, such that blocks were inadvertently limited to a single axis. I called this variant the *single block multi axis task*.

5.6.2 The Single Block Task

For the single block task, the search algorithm was set up to use a heuristic search, with an evaluation function that compared the face of the block in the construction area to the design. Because the task's setup was symbolic, these faces were represented by their symbolic variants (as shown in Figure 3.15).

The experimental environment was set up with three objects that represented the design, the construction area and the single block to be manipulated. Additionally, two affordances, one for picking blocks and another for dropping blocks, were added to the environment. For task specific operations, since the agent was not expected to perform any other form of block manipulation, a single operation to simulate visual attention, *get_item(id)*, was added. The *get_item(id)* operation took the unique ID of an object in the environment as an argument, and returned an instance of that object.

To make the agent aware of the objects it could potentially access through the *get_item(id)* operation, the literals "block_1", "design", and "construction", which were the unique identifiers of objects in the environment, were added as seed values to be used in the search. Just as with the imagery based implementation of VREE, the two affordances were also converted into operations, *affordances.PickUp(x)* and *affordances.Drop(x)*. This conversion led to a total of three operations over which the agent could search (see Sections

3.3.3 and 3.3.4 for details on how affordances were used internally as operations).

5.6.3 The Single Block Single Axis Experiment

After running the single block experiment, I performed a slightly more complex, “single block, single axis” variant of the experiment. In this variant, the task was still limited to a single block, but instead of just picking and placing the block into the construction area, the agent was expected to flip the block over a single axis in order to find a matching face.

This experiment was conducted in an environment similar to that of the single block experiment, with the following changes:

- An additional affordance to allow the block to be flipped in just a single axis (for example, *affordances.flip,ight(x)*) was introduced. It did not matter the direction in which the flip occurred; the only requirement was that the face on the design and that on the construction block should exist on the same axis of rotation.
- An additional literal value, "design", which referred to the ID of the design area was added.
- An additional operation *state_of(x)*, which returned the state of any object *x*, was added.

The evaluation function used for this task checked all face combinations to guarantee the machine could solve any problem in the single axis task space. Since there are 4 faces on any given axis of the block, the problem space always contained 4 possible problems. As such, during search, the evaluation function run 16 instances of the problem for all the possible face combinations between the block and design. By limiting the axis of rotation, some problems became inherently invalid because the target face did not share an axis with the initial face of the block.

5.6.4 The Single Block, Multiple Axes Experiment

In continuing with the theme of increasing complexity with every experiment, I moved on to a “single block, multiple axes” variant of the block search experiments. Theoretically,

the agent could be given up to six degrees of freedom to manipulate the block. Thus, the goal of this subsequent experiment was to investigate strategies that were possible as the agent's ability to manipulate the blocks increased. For this experiment, there was still a single block available, and the agent's ability to manipulate blocks was constantly increased as the experiment progressed. The environment for this experiment remained unchanged from the previous case, except for the extra affordances that were added to improve the agent's ability to manipulate the blocks.

Depending on the number of axes required for a given phase of the experiment, the environment was configured accordingly to have the same number of block manipulation affordances. In the case where an agent had two degrees of freedom to manipulate the block, two affordances for the corresponding selected axes were placed in the environment, and so on. Also, for the special case of having two affordances in the environment, a constraint was put in place to ensure that the two affordances were not direct opposites of each other (for example, flipping left and right caused the block to remain on the same axis), essentially limiting the block to be manipulated on a single axis.

5.7 Results for Strategy Search Experiments

After running searches for all the different BDT sub-tasks, some of the strategies obtained are shown in Table 5.2. Results shown in this table were obtained from heuristic searches with environments setup as described in the respective sections above. For the Single Block task, the optimal result obtained is shown. For the other experiments, however, variations that use different interesting strategies are also shown. Due to the ever-growing search space (with respect to number of operations added) the single block, multi axes experiment was successful up to only two degrees of freedom (out of a possible six).

5.8 Discussions

Except for the single block task, all other experiments yielded multiple solution strategies. In the following sections, I will be discussing each of these solutions. Since the single block task was extremely simple to solve, I will use its results to provide a walk-through of how searching for state machines worked for most experiments described in this chapter.

Table 5.2: A sampling of state machines obtained for all experiments.

Experiment	Sample Machines Obtained	
Single Block	<pre> var0 = get_item("block_1") affordances.pickup(var0) affordances.drop(var0) </pre>	
Single Block, Single Axis	<pre> var0 = get_item("block_1") affordances.pickup(var0) var1 = get_item("design") true affordances.flip_right(var0) state_of(var0) != state_of(var1) state_of(var0) == state_of(var1) affordances.drop(var0) </pre> <p>(a)</p>	<pre> var0 = get_block("block_1") affordances.pickup(var0) affordances.flip_right(var0) affordances.drop(var0) state_of_block(var0) != state_of_design() </pre> <p>(b)</p>
	<pre> var0 = get_item("block_1") affordances.pickup(var0) var1 = get_item("design") true affordances.flip_right(var0) true state_of(var0) == state_of(var1) affordances.drop(var0) </pre> <p>(c)</p>	<pre> var0 = get_item("design") var1 = get_item("block_1") affordances.pickup(var1) true affordances.flip_right(var1) state_of(var0) != state_of(var1) state_of(var0) == state_of(var1) affordances.drop(var1) </pre> <p>(d)</p>
	<pre> var0 = get_item("block_1") affordances.pickup(var0) true affordances.flip_right(var0) var1 = get_item("design") true state_of(var0) == state_of(var1) affordances.drop(var0) </pre> <p>(e)</p>	<pre> var0 = get_item("design") var1 = get_item("block_1") true affordances.pickup(var1) affordances.flip_right(var1) affordances.drop(var1) state_of(var0) != state_of(var1) state_of(var0) == state_of(var1) </pre> <p>(f)</p>
	Single Block, Double Axis	<pre> var0 = get_item("block_1") affordances.pickup(var0) var1 = get_item("design") true affordances.flip_left(var0) true state_of(var0) != state_of(var1) state_of(var0) != state_of(var1) affordances.flip_up(var0) state_of(var0) == state_of(var1) affordances.drop(var0) </pre>

5.8.1 Exploring Single Block Search Tree

The single block task resulted in a strategy whereby the agent simply picked the block and placed it in the construction. Whenever the block face and the design did not match, the search operation never terminated as a correct flip program would not be found. Because of how simple the single block search was, its entire search tree could be captured on a single page. Figure 5.6 shows this search tree, along with a description of how search proceeded.

5.8.2 Single Block, Single Axis

The main goal of this task was to have the agent pick up the block and flip it a single direction until it found a face that matched the design. When the match was found, the agent dropped the block into the construction area.

Six interesting strategies obtained through search for this task are displayed in Table 5.2. In strategy (a), the agent picks up the block and flips it around in a single direction, while checking for matches after each flip, until a face whose design matches the target design is found. When the match is found, the block is dropped in a construction area.

Several other strategies that followed a similar strategy were also found. From the results, we can see how these strategies vary. Strategy (c) was almost identical to (a), except that the condition on which the flip action occurs had a literal true value. This inherently made the strategy inefficient, since it was forced to randomly choose between flipping again or dropping the block whenever its face does not match that of the design. This is unlike strategy (a) where no more flips occurred after the target face was found.

A strategy of putting an always-true condition on the transition that flips blocks can be seen in strategy (c). This strategy increased the inefficiency by unnecessarily obtaining the instance of the design after every flip of the block. This operation could be considered analogous to looking at the design after every flip. Strategy (d) took this a step further by moving the pickup and drop operations into the same state where the flipping took place. As such this strategy continuously picked the block, flipped it, and put it back until a face that matched the design was reached.

Strategy (d) represents a special case; it was logically equivalent with strategy (a), although the order of instructions were different. Solutions of this type were generated whenever the procedure for comparing logically equivalent programs was not applied. Before this procedure was applied to search, most searches were limited to three instructions per state. This restriction was partially influenced by the initial “hand-coded” solution, and the aim to find programs that spanned multiple states. However, with the improvements gained from pruning logically equivalent programs, I attempted to search for larger programs, and the number of instructions per state was increased to 4. With 4 instructions per state, strategy (f), which was significantly more optimal when compared to both the

original handwritten solution and all the other strategies obtained from searches limited to 3 instructions per state, was generated.

5.8.3 Single Block Multiple Axis

Of all the different configurations that were considered for the experiment, only the case of having two affordances led to successful strategies being formed. Beyond 2 degrees of freedom, search became intractable. Although this limitation significantly impeded the progress of my experiments, it provided a window to how complex the block design is: even in such a simplified form, a brute-force search of strategies just fail after a significant number of rules were added.

5.9 Conclusion

The goal of this chapter's experiments was to progressively increase the block manipulation operations while increasing the number of blocks, until the full version of the block design task could be solved. Unfortunately, this goal was not entirely met: experiments proceeded to the stage where there was a single block that could be manipulated with two degrees of freedom (such that it only moved along two of its axes in a single direction per axis). Regardless of this inability to reach a stage of solving the full block design task, the strategies obtained for the stripped down versions of the test were interesting to study, and showed early signs of how strategies represented as state machines could be insightful in helping to explain how an intelligent system may be making its strategy choices.

Possibly, the most interesting observation from these sets of experiments was the optimal program generated for the single block, single axis task (See machine (b) on the Single Block, Single Axis row of Table 5.2). With the search algorithm originally biased toward a particular structure of programs, its ability to find such an optimal solution was certainly hindered. But, when all the restrictions were removed, and the right conditions were created, an optimal program was synthesized. Removing restrictions came at the cost of limiting how far search could go, and this represents the balancing act between search expressiveness and achieving the right solutions.

CHAPTER 6

Analysing Human Strategies on the Block Design Task

Another approach to studying strategy differences in intelligent systems is to record the performance of these systems and analyse. In this chapter, I describe work I contributed toward various systems that allow for the measurement of human performance on the block design task.

6.1 The Block Design Task

When first encountered, the block design task may appear simple. After all, the only explicit requirement is to replicate a geometric design with a set of blocks that look similar to items in the design. All information needed to work through a task comes from the design and is made available to the subject and kept open throughout the test. Nothing is hidden from the test subject. Also, the mode of reproducing the design with blocks evokes a sense of childish playfulness which makes the test appear less intimidating. In fact, Kohs (1920) discusses how children and adults alike were excited to take the test, when presented with the colourful blocks.

Yet as simple as it may appear, puzzles from the block design can be challenging to solve. To be successful, a person may have to fall on their visuospatial reasoning skills; correctly coordinate the interaction between their perception, gaze and motor actions; and also be able to form and execute strategies. These cognitive requirements of the block design makes it an interesting and informative test for evaluating human intelligence.

Originally the block design was intended to be scored by factors including a subject's response time, accuracy and the total number of moves made (Kohs, 1920). However, tallying moves appeared to complicate the work of clinicians and test administrators, and its use was later dropped in clinical practice (Hutt, 1932). But from block design research literature, it can be seen that factors other than response time and accuracy provide deeper insights into a subject's cognition.

For example, features of the block design performance such as the types of errors

subjects make (Joy et al., 2001; Rozenchwajg & Corroyer, 2002; Toraldo & Shallice, 2004), the way subjects divide their attention between the target design and block construction area (Rozenchwajg et al., 2005; Rozenchwajg & Corroyer, 2002; Rozenchwajg & Fenouillet, 2012), incorrect placements of blocks (Ben-Yishay et al., 1971; R. S. Jones & Torgesen, 1981; Joy et al., 2001; Schatz et al., 2000) and other qualitative errors (Akshoomoff & Stiles, 1996; Joy et al., 2001; J. Kramer et al., 1991; J. H. Kramer et al., 1999; Schatz et al., 2000; Zipf-Williams et al., 2000) are all factors researchers have considered as indicators of a subject's cognitive state. But, although clinicians that administer the block design test may be well-trained at evaluating their subjects, without the assistance of technology it may be impossible to detect all the behaviours that may provide a rich stream of information about a subject's performance. In an attempt to make it easier for test administrators and clinicians who administer the block design task to measure some of these features, my research group has been focused on creating tools that automate the measurements of these difficult to observe metrics in block design performance.

Our work in this area has primarily centred on automating the process of tracking block placements and collecting information about a subject's attention while solving the task. Additionally, in response to restrictions imposed by the global pandemic, we had to look at ways of virtually administering the block design task. In this chapter, I will be discussing work done towards this project, including emphasizing my contributions, and elaborating on how data collected from this work fit into the larger framework of this dissertation.

6.2 Automatically Scoring the Block Design Test

For our block design scoring system, when a subject took the test, we focused on collecting two primary streams of data: block placements and attention through gaze. With the block placement data we were able to get information about accuracy, the types of errors people made, steps people took to correct their errors, and any strategy patterns a subject exhibited. Through gaze data, we were able to study how attention was distributed, and we could implicitly deduce how a person may have relied on their working memory while reasoning about the puzzle.

So far, there have been three main iterations of our automated evaluation system. The

first version relied on an overhead Microsoft Kinect RGB-D camera for tracking block placements and hand movements, while gaze was tracked in two different ways with a wearable eye tracker and an experimental corneal imaging system, which was being evaluated at the time. In the second iteration, we traded the overhead Kinect RGB-D camera for a GoPro Camera, and we introduced a subject facing Intel Realsense depth camera, along with a Pupil Labs eye tracker for gaze and head tracking.

All the work described in this chapter was the result of a large, multi-year group effort by members of my research team and other contributors. Throughout the rest of this chapter, I will be highlighting the roles played by myself and that other team members as I discuss the work.

6.2.1 Overview of the Three Iterations

The first version of the system was originally developed by a team of researchers at the Georgia Institute of Technology, and it was used in a study involving 14 adult computer science undergraduates who had to solve puzzles from the block design test (Cha et al., 2020). Of these students, 7 had their gaze recorded through a head mounted gaze tracker and the other 7 used an experimental corneal imaging tracker. I was not involved in either the design or any of the data collection activities performed with this system, but I played a key role in analysing the data from this study.

The second system was developed at the Vanderbilt University, and it was not used in any major studies due to disruptions from the COVID-19 lock-downs and the general concerns about the safety of in-person activities. It was, however, used in a number of internal evaluation runs, and it was featured prominently in an appearance on CBS's 60 Minutes news magazine program (A. Cooper, 2021). I was fully involved in the design and testing of this system.

The third iteration, a virtual instance of the block design, built as one solution to address the challenges of in-person use, departed from physical interaction by presenting the block design task through a virtual web interface. Here, participants manipulated blocks on screen and attention was gleaned from mouse movements and block interactions.

Most of the results discussed in this chapter come from the experiments performed

on the first study. However, I will share some of the results from our CBS 60 Minutes appearance, and later in the chapter I will provide some details about the online system and some of the early data that has been collected through it.

6.2.2 Setup for Testing

For both in-person BDT evaluation systems, subjects took the test seated at a table. To help with computer vision, the table's surface was covered with a green material that provided enough contrast against the colour of the blocks. Additionally, the blocks themselves were about three times larger than those typically used in the real-world block design task to make them more visible to the cameras.

Also, to provide a simple scheme for tracking attention, the test environment was broken down into three zones. Two of these zones were on the table. The first of these table-top zones, which was directly in front of the subject, was considered as the construction area where the design was put together. The second zone, which was found to the subject's side, was considered as a block bank for holding unused blocks. Finally, seated across the table, a research assistant (or clinician) administering the test held up the easel containing the target design, which was considered as another attention zone.

Although this setup remained largely unchanged across both instances of the in-person systems (as can be seen in Figure 6.1), there were a few subtle differences in how the construction area was presented and how the eye tracking equipment was set up. In the earlier design, a blue outline was drawn on the table's surface to mark the spot where the final design was to be placed. This blue outline was put in place to help with the computer vision routines. In the second implementation, however, blocks could be constructed anywhere on the table's surface. When it came to segmenting the blocks for later analysis, a suite of visual clustering algorithms and other computer vision techniques were used.

It is also worth noting that for subjects that were evaluated on the corneal imaging system, a chin rest was provided since subjects were required to keep their head sturdy to help their eyes stay in the focus of the corneal imaging camera while they solved their puzzles (see Section 6.3.) In cases where the other forms of gaze measurements were employed, the chin rest was not used.

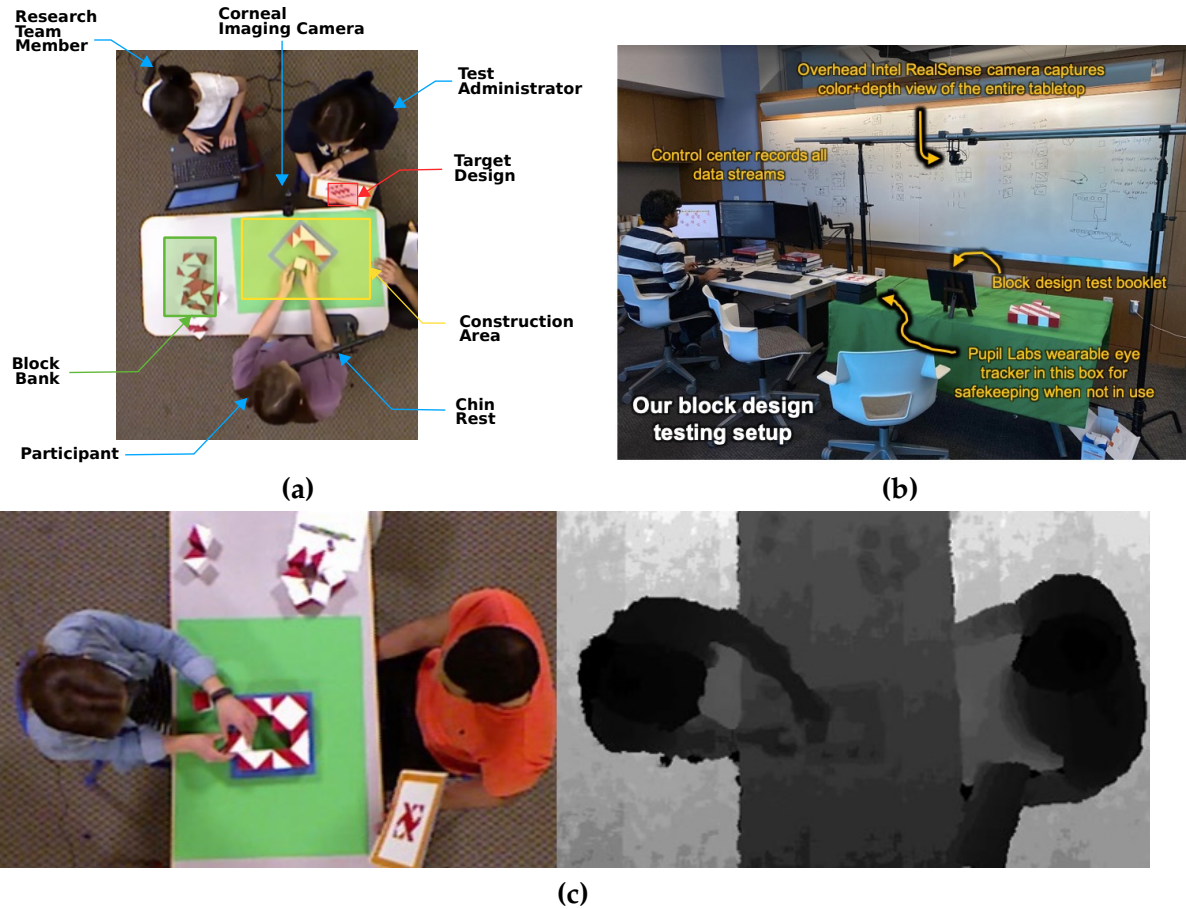


Figure 6.1: A setup of the block design task, showing the differences between sensor setup in the different iterations, as well as, how attention zones were defined. (a) Shows an overhead view of the setup in our first iteration, as captured by the overhead camera, and (b) Shows the setup for our second iteration.

6.2.3 Measuring Block Placements

One of the main information streams of the evaluation system came from the placement and movement of blocks. An overhead video camera, oriented to capture the tabletop, acted as the sensor for block placement and movement. For the first iteration of the system, a Microsoft Kinect camera was used, but we switched to a GoPro camera in the second iteration. This switch was mainly due to the Microsoft Kinect being discontinued by its manufacturer, and also the resolution of images captured through the depth camera was found to be practically useless when it came to isolating individual blocks; relying on colour information was simpler to implement and much more efficient.

Measuring block placements broadly involved detecting and isolating the individual

blocks, and determining the design on the block's face. Because the hands were constantly interacting with blocks on the table, they obscured some of the blocks at times. This situation introduced an additional step of detecting and filtering out hands to the block detection pipeline. The following sections provide details on how block isolation was performed in both iterations of our system.

6.2.3.1 Isolating Individual Blocks

In our initial implementation, we placed a blue outline on the table to make subjects arrange their designs within the outline's bounded space. Because these outlines had the same size and orientation as the design to be solved, the outlines may have influenced the choices subjects made about their block placements, especially when diagonal puzzles were concerned. The outlines were, of course, intended to be temporarily used in the early exploratory experiments.

With the outlines in place, isolating individual blocks was extremely simplified. We mainly had to detect the blue outline from the input image using its distinct colour characteristics, and cropped out the image sections enclosed by the outline. Because the space bounded by the outline was the same as that occupied by the final design, subdividing the cropped interior image into $n \times m$ parts, where n and m are the number of blocks required for both sides of the design, helped split out the locations of individual blocks.

This process was not always smooth. Since the sizes of designs changed as subjects worked through puzzles, the blue outlines on the table had to be swapped out as puzzles progressed. The continual replacement of the outlines, led to the outlines getting rotated differently from puzzle to puzzle, such that a pre-processing step of adjusting outlines into their upright position was required as part of block detection.

In the second iteration, to improve over the first iteration's block detection approach, we instead used the YOLO (Redmon et al., 2016) off-the-shelf object detection algorithm to detect blocks. Due to this change, subjects could place blocks anywhere on the table, and without the blue outlines, subjects were not clued in on what the sizes or orientations of the final designs were going to be. This made testing much closely aligned with the real world version of the test.

Although this improvement took us closer to a more realistic test administration process, it also came with its own technical challenges. In this case, unlike the bounded nature of the earlier approach where we had the outlines as guides for determining block locations, detecting the exact locations of the blocks was not as straight forward. Except for the blocks themselves, there was nothing to guide us in isolating the blocks. To solve this, we relied on clustering algorithms which took as input the locations of all blocks detected by YOLO and computed the coordinates of the area that bounded the design. From here, the rectangle formed by the blocks could be aligned and segmented into blocks just as they were in the first iteration. This process was performed in reverse, so the final state of the blocks were used to isolate where the grid for detecting blocks went on the table, and the sequence in which individual blocks were added was inferred.

6.2.3.2 Detecting the Face of the Blocks

After individual blocks were isolated, the actual design on the face of each block also had to be identified. The approach for this did not change between the two iterations of our in-person testing system. The same techniques used in the first iteration worked equally well in our second implementation. The following paragraphs describe our block face detection algorithm.

Faces of blocks could either have a single uniform colour, or they could have two colours split in any way across the two diagonals on the block's face. Since each block may already be isolated into its own image, a quadrant based feature detector that took advantage of the diagonal nature of the block's faces, was used alongside different classifiers to detect the designs. See Figure 6.2 for a visualization of this feature detector.

The feature detector worked by sampling pixels from each of the four quadrants that form when diagonals are drawn across the face of a block. While analysing the data from our block design study, we evaluated four different algorithms for classifying the block faces according to data from the quadrant feature detector. These algorithms were as follows:

- An *RGB Averaging* technique where the average RGB value of all pixels in a given quadrant (which represents a crude way of computing the luminance of the quadrant)

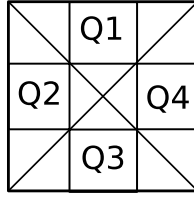


Figure 6.2: Feature detector for quadrant based block face detection. Samples taken from the quadrants labelled Q1 through Q2 are each guaranteed to contain a uniform colour. By combining these in multiple ways, the design on the face of the blocks can be found.

were computed and compared to a threshold value. From our experiments, when RGB values were taken over a range of 0 to 255, a threshold value of 140 gave the best results.

- A *K Means* clustering technique, where the individual RGB values of pixels were clustered to obtain the dominant values across the three channels. A final classification decision was made based on the dominant clusters.
- A KNN Classifier, trained on hand-picked ground truth colours from reasonable hue, saturation, value (HSV) regions and evaluated on pixels collected from the quadrants of the feature detector.
- A Multi Layer Perceptron (MLP), trained on a set of images captured during the study from the different quadrants, and equally evaluated on other images from the study.

Results from evaluating these face detection algorithms are shown in Table 6.1. From these results, it should not be surprising that the MLP classifier worked best. When compared to the other techniques, which used either hand-picked thresholds or synthetic colours as input, the MLP was trained on actual data from images captured by the overhead cameras.

6.2.3.3 Filtering Out Hands

Another issue we had to deal with concerned hands occluding the camera's view of the blocks as subjects worked through puzzles. Since detection was performed on a frame-by-frame basis, block placement data was lost on frames where this occlusion happened. We

Table 6.1: Accuracy results for different approaches.

Method	Accuracy
RGB Averaging	0.68
K-Means Clustering (k=1)	0.68
K-Means Clustering (k=4)	0.64
K-Nearest Neighbors	0.67
Multi-Layer Perceptrons	0.96

fixed this with a filtering mechanism that detected the hands (using the same YOLO object detection algorithm) and used information from earlier frames to fill in missing information from parts that a hand may have occluded. This technique was used in both the first and second iterations of our in-person scoring systems.

6.2.4 Measuring Gaze and Attention

Generally speaking, our eyes play an important role when we interact with our environment. Through vision, we are able to acquire much of the information we need to perform many everyday activities and also special tasks like the BDT. Studying visual attention—how and where people look, and what they decide to focus on while performing a task—can help us gain valuable information about cognitive processes (Land & Tatler, 2009). A person’s gaze is one of the few externally observable components of the cognitive process. For this reason, gaze tracking has been a very important technique when it comes to the study and understanding of human cognition (Just & Carpenter, 1976). Gaze tracking technologies measures the eyes’ (and sometimes the head’s) position with respect to external stimuli, thus providing a measure of overt visual attention. ¹

6.2.4.1 A Brief Background on Gaze Tracking

When considered broadly, you can categorize gaze trackers as being either wearable or remote. For real-world interaction and real-world activity tracking in three-dimensions, like in the case of the block design task, head-mounted wearable trackers are considered some of the best and most accessible options. Although wearable trackers work well in

¹Visual attention, in general, also includes covert attention shifts, which are not externally observable, like triggers in the peripheral vision.

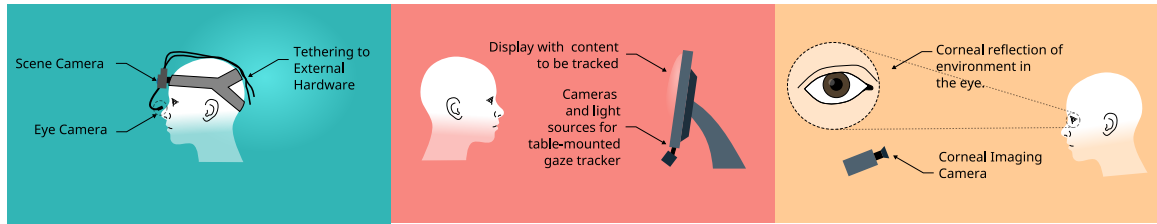


Figure 6.3: A visualization of the two main classes of gaze tracking technology widely available today (wearable on the left and remote in the middle), and the corneal imaging technique (right).

most settings, they present special challenges to certain populations, like children, people with sensory sensitivities, or people with certain physical or cognitive disorders. Their wearable nature may also make them distractions for some use cases.

Remote trackers, as their name suggests, track the gaze of subjects remotely, without any contact with the subject. These trackers are more challenging to implement, since systems must remotely infer gaze by “observing” the subject—much like a person would. The most popular remote tracker implementation may be on screen gaze trackers that measure a subject’s attention to content displayed on a screen. Screen based gaze trackers are simpler to implement when remote gaze trackers are considered. Because the content to be tracked is displayed over a fixed, relatively small two-dimensional plane instead of the real-world three-dimensional space, computations to localize gaze fixations are simpler on screen based gaze trackers. In recent times, the use of deep learning based computer vision systems have also made it possible to produce effective remote gaze trackers for estimating people’s gaze when the performance of certain real world tasks are considered (see Shehu et al., 2021).

6.2.4.2 Our Approaches to Gaze Tracking

In our studies on the block design task, we considered two different types of eye tracking. For some of the subjects in our earlier study, and for all our recent work, we relied on head mounted gaze trackers. For certain subjects in our earlier study, however, we tried a relatively recent form of eye tracking that relied on estimating gaze by capturing images of the world as reflected on a person’s cornea, known as corneal imaging (Nishino &

Table 6.2: A brief comparison chart of various forms of gaze tracking.

Gaze Tracker	Head Movements	Body Movement	Invasive	Tethered to Body
Table-Mounted	From totally restrained to the limits of camera's depth of field	None	No	No
Corneal Imaging	Restrained to the limits of camera's depth of field	None	No	No
Head-Mounted	Full	Full, within limits of tethered equipment	Yes	Yes

Nayar, 2004). The specific corneal imaging system used in this work was implemented and executed by Eunji Chong, who was a doctoral student at the Georgia Institute of Technology.

6.3 Measuring Gaze Through Corneal Imaging

In Figure 6.4 we see an image of how the world is reflected on a person's cornea while staring at a setup of the block design task. The goal of corneal imaging is to use this exact image as a means of tracking a person's gaze. Corneal imaging brings several advantages when compared to other forms of gaze tracking.



Figure 6.4: A reflection of the world as seen on the cornea of a person solving a block design task.

Most importantly, being a remote tracking approach means corneal imaging does away with the issue of discomfort wearable trackers bring. Also, since the images are taken directly off the cornea, corneal imaging allows for real world three-dimensional tracking,

with need for little to no calibration. The corneal imaging approach, in fact, seemed well suited to the table-top nature of the block design task.

A few downsides of the corneal imaging approach comes from the complexity involved in setting it up, and the complicated processing required to extract the images and gaze targets off the cornea. For a respectable corneal image, a high resolution camera combined with a high quality zoom lens with autofocusing capability may be required. Apart from the technical challenges, the colour of a person's eyes also present another challenge for corneal imaging: corneal images are much more visible on darker eyes than lighter ones.

6.3.1 Gaze Estimation with Corneal Images

The setup for corneal imaging had a high resolution camera pointed directly at one eye of a subject, from which we recorded the corneal image. This camera was fitted with an autofocusing telephoto lens that ensured the subject's eye was fully in the frame, while being far away enough from the camera to be distracted by it. Although corneal imaging was meant to be used without any physical restraints, for our experimental setup, participants were given a chin rest to help them stay in the image frame with their eyes always in focus.

Estimating the gaze in corneal imaging involved capturing the corneal image, unwarping the image to remove the eye's spherical distortion, and computing the gaze point with a three-dimensional eye model fitted to the corneal image. Effectively, anyone studying the tracking data obtained through corneal imaging would be looking at images that were captured directly off the subject's cornea, overlaid with the estimated gaze point.

Computing the gaze point from the corneal images required fitting an ellipse around the limbus of the eye in the image frame. An ellipse p was represented by 5 parameters, such that $p = (r_{max}, r_{min}, x, y, \Theta)$. Where r_{max} and r_{min} represent the major and minor axes of the ellipse, x and y represent the centre of the ellipse with respect to the entire image frame, and Θ is the angle of tilt. Actual ellipse fitting was performed with Hough transforms (Duda & Hart, 1972) using code provided by Eunji Chong.

Given that the distance from the camera to the eye was known—thanks to the chin rest—the fitted ellipse could be projected unto a three-dimensional, spherical model of the eye to represent the limbus. Through this projection, the orientation of the eye would be

estimated, and the gaze point would be assumed to be the point where the axis that runs through the centre of the eye model intersects with the surface through the pupil. When this point of intersection is projected back to the two-dimensional image of the eye, the exact gaze point can be estimated.

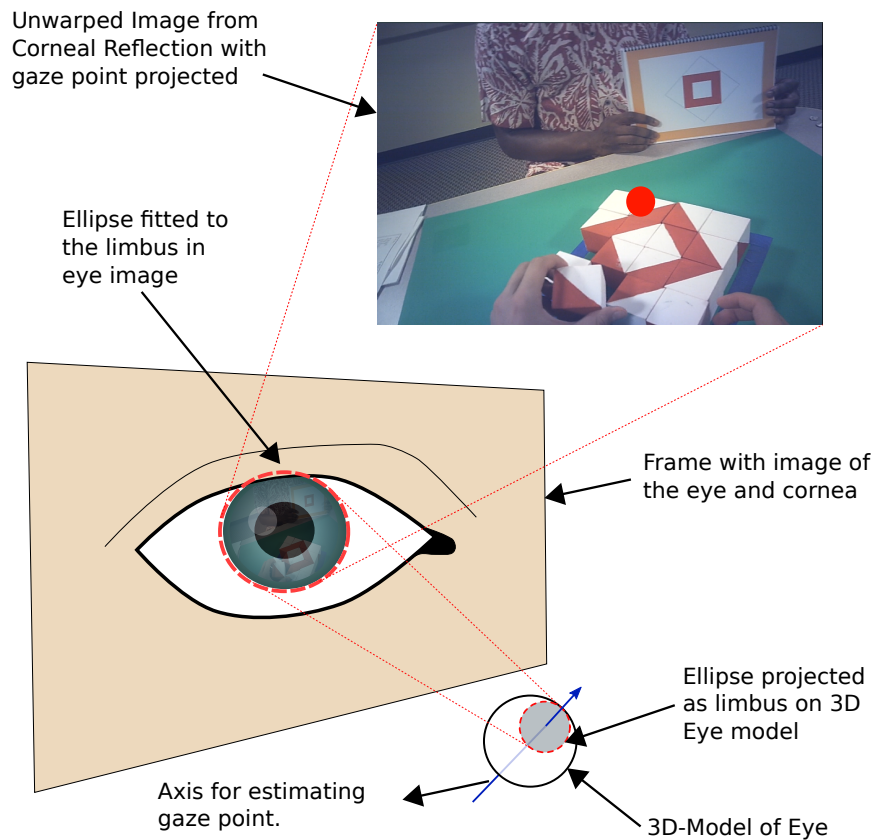


Figure 6.5: Fitting an eye model to the limbus for obtaining the exact gaze point with respect to the corneal image.

6.3.2 Collecting and Analysing Corneal Imaging Data

The work on corneal imaging was merely a proof of concept implementation, with a primary goal of evaluating how effective this form of gaze tracking would be in a clinical setting. As such, we sought to understand the following:

1. Would the images recorded off the cornea be of a fidelity high enough that human

raters would be able to reasonably recognize what the person was looking at?

2. If the ratings supplied by human raters (through the experiment in the point above) were fed to machine learning algorithms, would they be able to determine the gaze targets from other corneal imaging videos.

To answer these questions, I performed a couple of experiments.

6.3.3 Experiment 1: Evaluating Image Fidelity by Human Annotation

After participants in the first study had solved the block design puzzles, the corneal image recordings (in video format) were annotated with the ELAN annotation tool (Wittenburg et al., 2006) by myself and a team of undergraduate researchers. Annotations involved marking video frames with the gaze target observed in the corneal image. As stated earlier, in order to simplify annotation, gaze targets were broadly categorized according to the experimental setup as follows: the block bank, the construction area, the target design, and other (for any other gaze target). Additionally, the blink state of the eye (blink vs. non-blink), and the type of gaze shift (saccade or smooth pursuit) was also annotated.

Making these annotations were quite time-consuming. On average, it took about an hour of real time to annotate a minute of recorded corneal image video. And when solving puzzles, subjects typically took anywhere from 1 to 3 minutes. Considering how the physical space was set up for the experiments, there was some redundancy in the content of corneal images and the position of a subject's pupil in the overall frame. For views to the construction area, the eye remained in the centre of the frame, whereas for views to the target design the eye moved left, and for views to the block bank the eye moved right. Although it may appear human raters relied on this information to identify gaze targets, initial analyses suggested this information was not factored into selecting gaze targets for annotation.

Since our goals were to measure how reliably human observers could distinguish images off the cornea to detect gaze targets, we measured how closely annotators agreed with each other when annotating the same video. Although it would have been ideal to have every annotator rate each video, so we could have an across-board comparison between raters,

the time-consuming nature of the annotation process made this unattainable. Instead, we selected five particular videos, and had them annotated by various subsets of our primary four annotators, to be used as a measure of reliability.

6.3.3.1 Results and Discussion of Experiment 1

I used the Cohen’s Kappa metric (Cohen, 1960) to compute the agreement between raters. The results of this are shown in Table 6.3. From the inter-rater reliability measure, we could conclude that images on the cornea were of a fidelity high enough that humans could reasonably differentiate between the gaze targets. That said, the results in Table 6.3 are not perfect, and this could have been due to the factors discussed below.

Table 6.3: Cohen’s Kappa scores for the inter-rater reliability among the four raters on our research team.

Rater Pairing	Puzzle	κ Score
$A_1 - A_2$	Puzzle 1	0.77
$A_1 - A_2$	Puzzle 2	0.78
$A_1 - A_3$	Puzzle 1	0.79
$A_1 - A_3$	Puzzle 3	0.69
$A_1 - A_3$	Puzzle 4	0.72
$A_1 - A_4$	Puzzle 2	0.80
$A_1 - A_4$	Puzzle 5	0.66
$A_2 - A_3$	Puzzle 1	0.74
$A_2 - A_4$	Puzzle 2	0.75

First, while the reliability metric was computed on a frame by frame basis, the actual annotations were performed over continuous video. Sampled at 15 frames per second, the average puzzle contained almost 1000 individual frames. Annotators had to literally drag a pointer along a timeline to select frames with a computer mouse through the ELAN tool’s user interface. When annotations that rated around 0.70 for their Cohen’s metric score are analyzed visually with bar graphs placed side by side, the similarities are much more apparent than the differences, and at least some of the differences appear to be due to discrepancies in the start/end times of various gaze events (see Figure 6.6). Using an event-based reliability metric instead of a frame-based one may provide a more useful estimate of overall reliability in the context of the desired quality of gaze annotations for

our task.

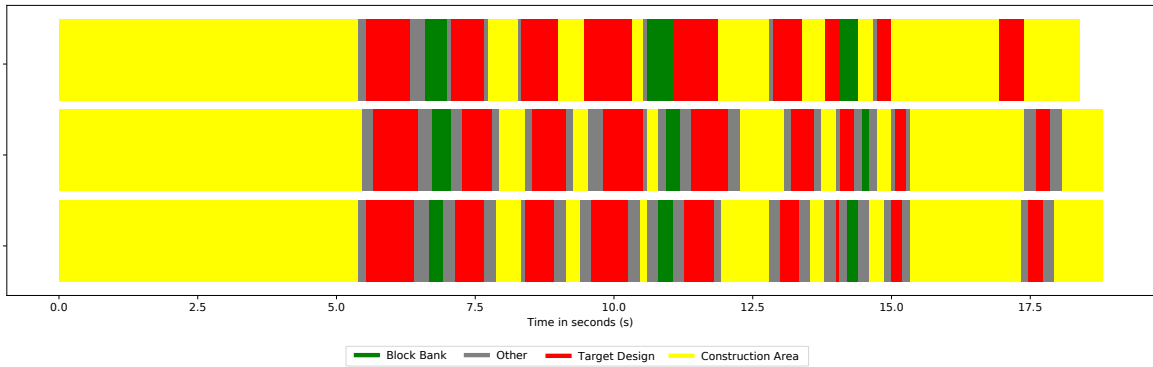


Figure 6.6: A visual comparison of annotations produced by three different raters on the same block design puzzle.

Second, our annotation process proceeded mostly in a feed-forward fashion, with little revision of the annotation scheme once the process was underway. Revising the basic scheme based on our observations from the initial attempt, and also providing more training to annotators, could have gone a long way towards increasing reliability.

6.3.4 Experiment 2: Automating Corneal Image Analysis

After the human annotation experiment, I attempted to automate the analysis of our corneal image data using machine learning. Ground truth classification for this experiment was obtained from the human annotation process described above.

My first attempt involved running a pre-trained convolutional neural network on the images obtained directly from the corneal imaging camera. The images fed to the classifier still had the eye's spherical distortion, and the other visible parts of the eye (like eyelids and lashes) in the frame remained intact.

For the second set of classifiers, I directly considered the geometric information extracted when the ellipse model was fitted onto the frame. As stated earlier, and as shown in Figure 6.5, the ellipse parameters consisted of five separate values that define the geometry of an ellipse.

I evaluated both classifiers with two different training and testing approaches. First, we employed a within-participant, across-puzzle leave-one-out strategy where we trained

the classifier on all but one puzzle videos for a single participant and tested on the video for the excluded puzzle from that same participant. This was done for all the different puzzles, and the scores obtained were averaged.

Second, I employed an across-participant leave-one-out strategy where we trained the classifier on all puzzles from all participants except one participant, for whom all puzzle data was used for testing. Again, this was repeated across all participants and averaged.

While in general, evaluating within a single participant yielded better results (as described below), the use case of testing across participants more closely matches likely applications of this approach in clinical settings. It is unlikely that annotations for individual clinical patients could be provided on a regular basis, but it is more plausible that a large sample could be collected and annotated up front, and then algorithms trained with these data applied to new patients.

6.3.4.1 Results of Experiment 2

Two different types of classifiers were trained on the images: for predicting gaze target locations and for detecting when a person was blinking. All classifiers took in the raw image pixels as captured from the corneal imaging camera without any modifications. A pre-trained copy of the Inception v3 (Szegedy et al., 2016) convolutional neural network running on Tensorflow v1.9 (Abadi et al., 2016) was used as the classifier.

The blink classifier obtained a final average accuracy of **81%** when evaluated across puzzles for individual participants, and an average accuracy of **61%** when evaluated across participants. For the gaze classifier, we obtained an average accuracy of **69%** when evaluated on an individual's puzzles and an average accuracy of **51%** when evaluated across participants. The results for the blink classifier are shown in Table 6.4, and for the gaze classifier in Table 6.5.

Figure 6.7 shows a visualization of the gaze target classification accuracy using the within participant testing approach.

Additionally, we trained classifiers using data from the ellipse model described in Section 6.5. We were able to obtain ellipse fit data for 28 individual puzzle attempts that we could use to test a classifier trained purely on geometric features from the ellipse model.

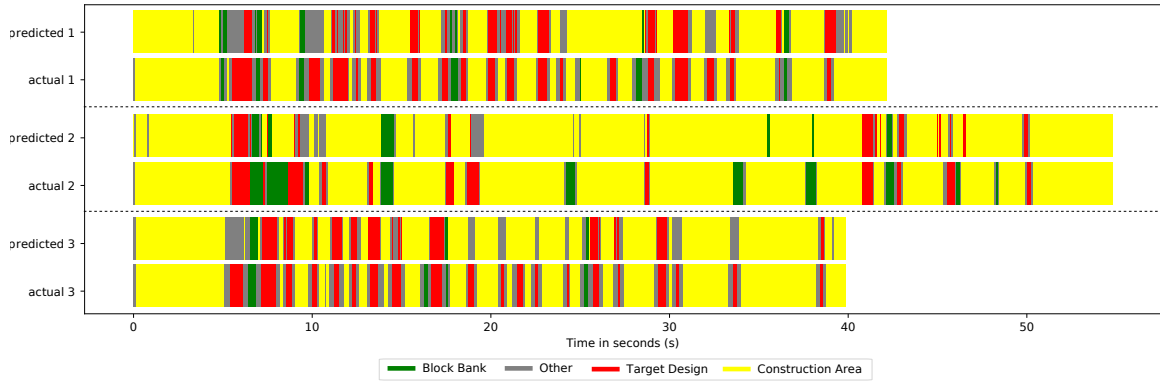


Figure 6.7: A visual comparison between annotations generated by human raters and the neural network algorithm for the same puzzles.

Table 6.4: Average accuracy and precision scores for the neural network blink classifier

Task	Accuracy	Precision
Evaluated by puzzles	0.81	0.59
Evaluated by participants	0.65	0.41

We used the same two train/test paradigms as described above.

For these analyses, we fed the ellipse parameters directly into the classifiers as feature vectors. We evaluated two different classifiers: k -nearest neighbours and multi layer perceptrons. Our k NN classifier had an n value of 5 and our MLP was configured with three hidden layers having 50, 100 and 50 nodes respectively. The k NN implementation was obtained from the scikit-learn toolkit (Pedregosa et al., 2011) and the MLP implementation was from Tensorflow (Abadi et al., 2016). Results for the k NN classifier are shown in Table 6.6, and results for the MLP classifier are shown in Table 6.7.

6.3.4.2 Discussion of Experiment 2

Without any explicit calibration, the image based classification algorithms were able to predict the gaze targets with some (though far from perfect) accuracy.

However, as described previously in the context of human generated annotations (see Section 6.3.3), it is worth investigating to what extent the image classification algorithms were working based directly on the image of the cornea versus on the position of the pupil

Table 6.5: Average accuracy and precision scores for the neural network gaze target classifier

Task	Accuracy	Precision
Evaluated by puzzles	0.69	0.55
Evaluated by participants	0.51	0.45

Table 6.6: Average accuracy and precision scores for the kNN gaze target classifier

Task	Accuracy	Precision
Evaluated by puzzles	0.77	0.66
Evaluated by participants	0.61	0.54

within the frame. The consistency of the positioning of the gaze targets meant that the pupil was in very distinct regions when fixations were on particular targets of interest, in the context of our block design task setup. The performance of the classification based on the ellipse model, which inherently used only the geometric position of the eyes, suggested that geometric positioning may have been a strong predictor of gaze target in our study.

6.4 Analysing Block Placement Sequences

Automating the scoring of block design provides the opportunity to extract high level strategy patterns that subjects may be using. For example, in our study, we observed that various participants exhibited some specific block placement strategies. Some participants preferred to complete puzzles systematically, placing blocks row by row, while others worked in a disorderly fashion, placing blocks in a disjoint manner. We therefore sought to classify the individual styles each subject used for block placement. Work on this section was mainly performed by Seunghan Cha, with my help on some machine learning pipeline design.

The first step we took was to define five general block placement strategies that we observed across our participant data. These were as follows: row by row, column by column, subsection, perimeter complete and vertices first. See Figure 6.8. These strategies were defined over specific regions subjects may be working on when the grid along which

Table 6.7: Average accuracy and precision scores for the MLP gaze target classifier

Task	Accuracy	Precision
Evaluated by puzzles	0.71	0.41
Evaluated by participants	0.64	0.37

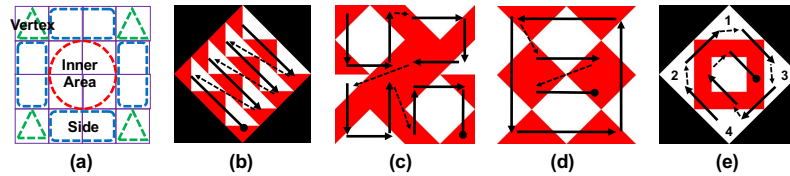


Figure 6.8: **(a)** Regions of the block construction area, labeled as vertices, sides, and the inner area. **(b-e)** Examples of spatial block placement strategies, with arrows representing the order of blocks placed: **(b)** row-by-row, **(c)** subsection, **(d)** perimeter-complete, and **(e)** vertices-first.

the blocks are to be placed is concerned.

These generic strategies, as defined, were not “purely” adhered to by subjects in their solutions. And even when the “pure” forms were considered, there was a significantly large set of variations (too many to enumerate in most cases) in how the blocks could be placed. As such, our strategy for classifying the styles of individuals was performed empirically, through comparisons with a computationally generated sample set of concrete sequences for

Generally, the similarity scores for a subject’s run was computed by comparing their sequences on a puzzle with all the sequences in the sample set, and finding the maximum similarity values that are computed for each strategy. We used Kendall’s tau coefficient, which evaluates the similarity between two sets of ranked lists, as our similarity metric. This similarity is computed as follows:

$$\tau = \frac{P - Q}{\sqrt{(P + Q + T) \times (P + Q + U)}}$$

If we consider x and y to represent the two different lists being compared, then P is the number of matching pairs in both lists, Q is the number of non-matching pairs in both

lists, T is the number of ties only in x , and U is the number of ties only in y . Kendall's tau computes numbers between -1 and 1, with -1 showing strong disagreement.

Figure 6.9 shows some of the results obtained when these analyses are performed on four block design puzzles by our seven subjects. In this chart, the green gridded area represents the sequence in which a participant placed blocks, with darker greens representing later placed blocks. Charts under the green grids show the similarity of the placement strategy to those from our sample library, with the highest scoring strategy in red. Finally, the leftmost column represents the puzzle under consideration.

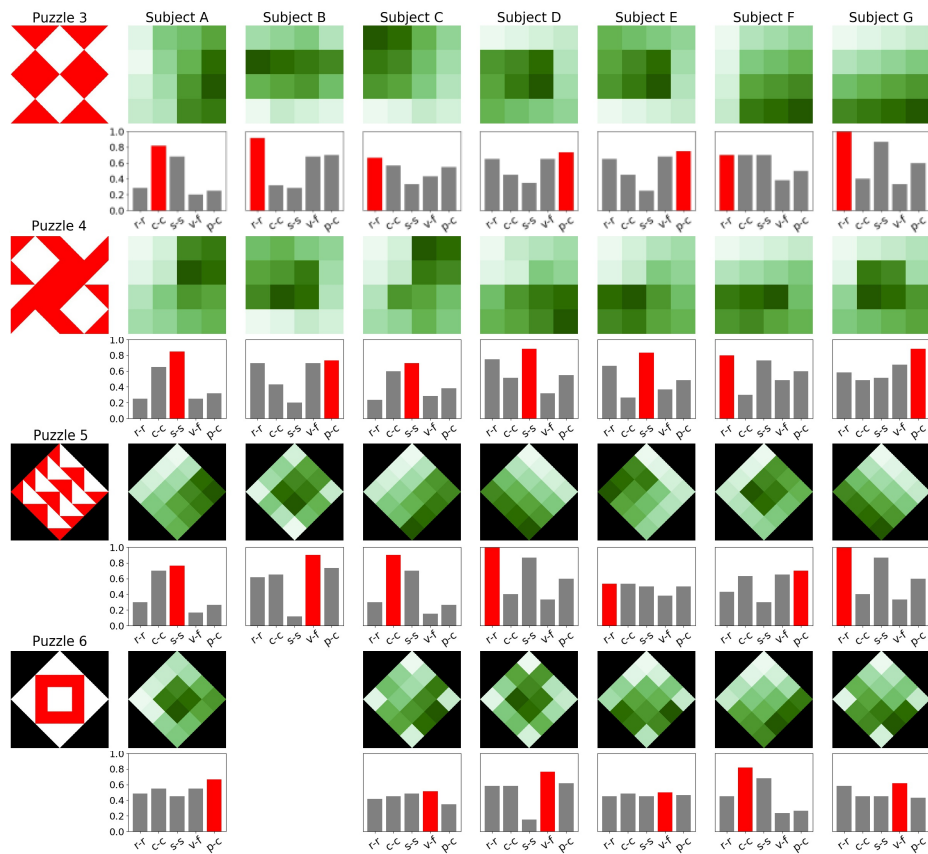


Figure 6.9: A visualization of sequences in which subjects from our study responded to puzzles from the block design task, and the predictions made by our system on strategy. For the green patches, earlier blocks are represented by lighter greens and later blocks are represented by darker greens. The following abbreviations are used on the charts: (r-r) for row by row, (c-c) for column by column, (s-s) for subsections, (p-c) for perimeter complete, and (v-f) for vertices first.

Just by looking at the charts, it is easy to appreciate the similarities and differences

between the strategies exhibited by participants. For example, we can see how participants C and G used very similar strategies on the 6th puzzle. The charts show cases in which people strictly stuck to one of the five pre-defined strategies, such as participant G and D using the row by row strategy on puzzle 5.

Given that our work on strategy classification was just a proof of concept, the sequences we defined for this work were based on our intuition, and it was not in any way informed by actual metrics clinicians use. In other words, these five strategies were not set in stone, and new sequence strategies could always be defined.

But then, it is also worth noting that sequences of block placements are not the only means by which strategy can be analysed on the block design test. Even from our system, it is possible to extract several low level features that may be of interests to neuropsychologists and other clinicians alike. For example, we can extract:

- The number and types of errors.
- Intermediate and final reaction times.
- Spatial distance between consecutive block placements.
- Progression tendencies, eg. left-to-right or right-to-left.
- Single versus multiple simultaneous block placements.
- Block pair swapping versus in-place block changes.
- Correlations among combinations of some, or even all the features above.

For example, see Figure 6.10 for a scatter plot of the reaction time against the number of errors participants made per puzzle.

6.5 Combining Gaze and Block Placement Data

So far, we have looked at the types of information we could obtain when each individual stream of data from one of our sensors was analyzed—attention from gaze, and strategies and other low-level features from block placements. With both data streams recorded on

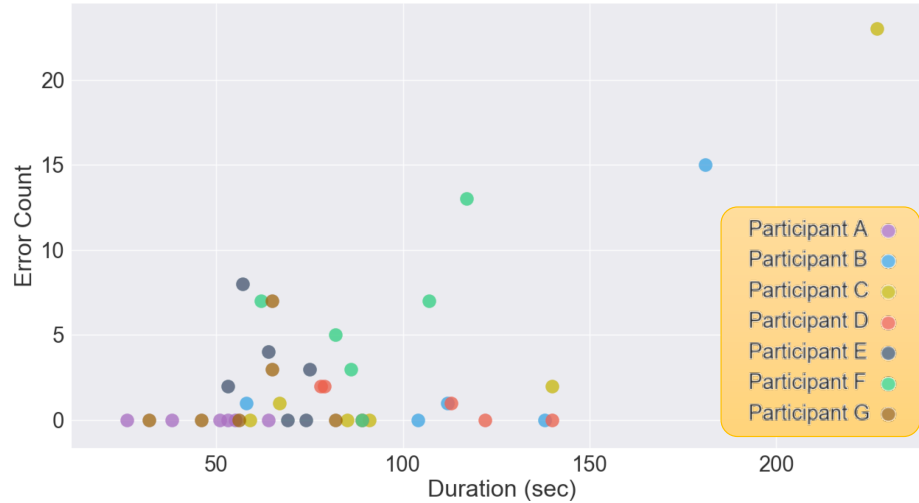


Figure 6.10: A plot of reaction time against the number of errors made.

the performance of same task, it was possible to perform analyses that rely on the combined data stream.

Before this data integration could be possible, however, the different streams of data had to be integrated. The cameras used in our experiments were mainly consumer grade cameras that lacked inputs for clock synchronization. This meant that the actual frames of video collected from the different cameras, all run according to different clocks. We manually set synchronization points using clapper boards, and to actually synchronize all the video data, I wrote a couple of python scripts, which used audio cues and other external signals, to synchronize all the videos recorded from the different cameras.

The chart in Figure 6.11 shows a visualization of the overall performance by three different subjects on a single task from the block design task. In each chart, the topmost horizontal bar represents the gaze fixations that subjects had on different areas of the experimental setup, and the second bar shows the instances where the subject was moving blocks. The bottom section of the charts show the time specific blocks were placed.

6.6 Introducing Novelty with Blue Blocks

Ideally, the block design task must be novel to any subject who takes it. However, given how widely the BDT is used in intelligence and cognitive testing, this cannot be guaranteed for all our potential subjects. To guard against this, and also to provide some form of

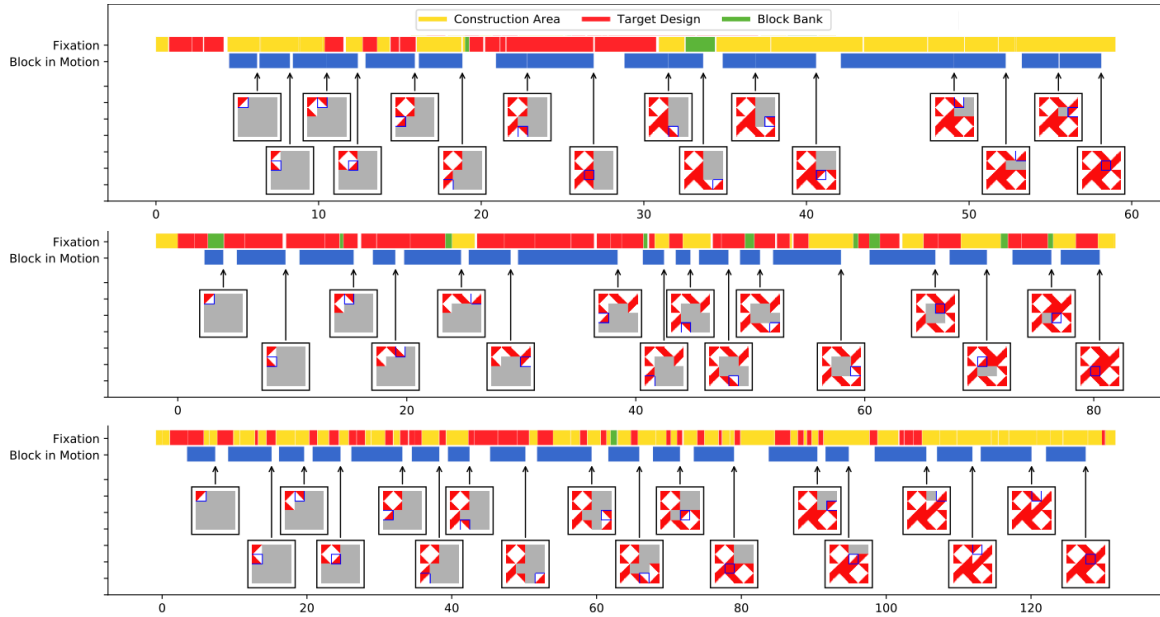


Figure 6.11: A synchronized plot showing gaze data, along with block movement and placement, on a single plot.

novelty for participants who have already experienced the BDT, we designed a new set of blocks that follow in the traditional block design's spirit, while being significantly different in certain other aspects.

In our blocks, faces had curves instead of straight lines, and they featured a blue and white colour scheme instead of the traditional red and white. From some preliminary data, we had already observed that the curves on block faces made puzzles a little harder to solve than usual. This may either be because the curves on the faces deviated from the straight edge nature of the blocks, or because the blue blocks had more distinct individual states when compared to the red ones. Technically, the red and white blocks have a total of 12 distinct states, when all the faces and their potential rotations are considered, and the blue and white blocks have twice as many with 24 states under the same considerations. See Figure 6.12 for a comparison of the block faces and Appendix B for a state diagram of both blue and red block states in full.

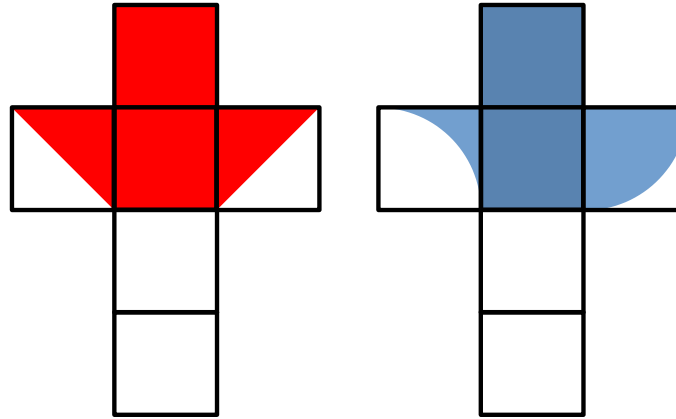


Figure 6.12: Net diagrams of the blocks showing the red blocks (left) and the blue blocks (right).

6.7 Case Study: The Block Design Task as Featured on CBS' 60 Minutes

Although the second iteration of the in-person BDT scoring system was not used in a fully sanctioned study (due to disruptions caused by the 2020 global pandemic), it was featured in a segment on CBS' 60 Minutes news magazine program (A. Cooper, 2021; Kunda et al., 2020). The episode, which was entirely focused on work being done by different organizations to provide work opportunities for autistic individuals, shed light on work that was being done by Vanderbilt University's Frist Center for Autism and Innovation. As part of that work, our group's work on automating block design evaluation and its potential for providing effective non-verbal assessments was featured.

One of the episode's main highlights had the show's host, Anderson Cooper, take tests from the block design alongside Dan Burger, an autistic data scientist with a record of good visual reasoning ability, evidenced by his work on the FilterGraph data visualization tool (Burger et al., 2012). Both subjects worked through a total of 14 puzzles, 8 of them from the red blocks and another 6 from the blue blocks. Tests from the red blocks were taken in one sitting, after which participants had a brief break before proceeding to those on the blue block.

6.7.1 Results and Analyses: Anderson Vs. Dan

When the block design test is administered in clinical settings, the metric typically used for evaluation is the response time. Figure 6.13 shows a chart of the response times measured

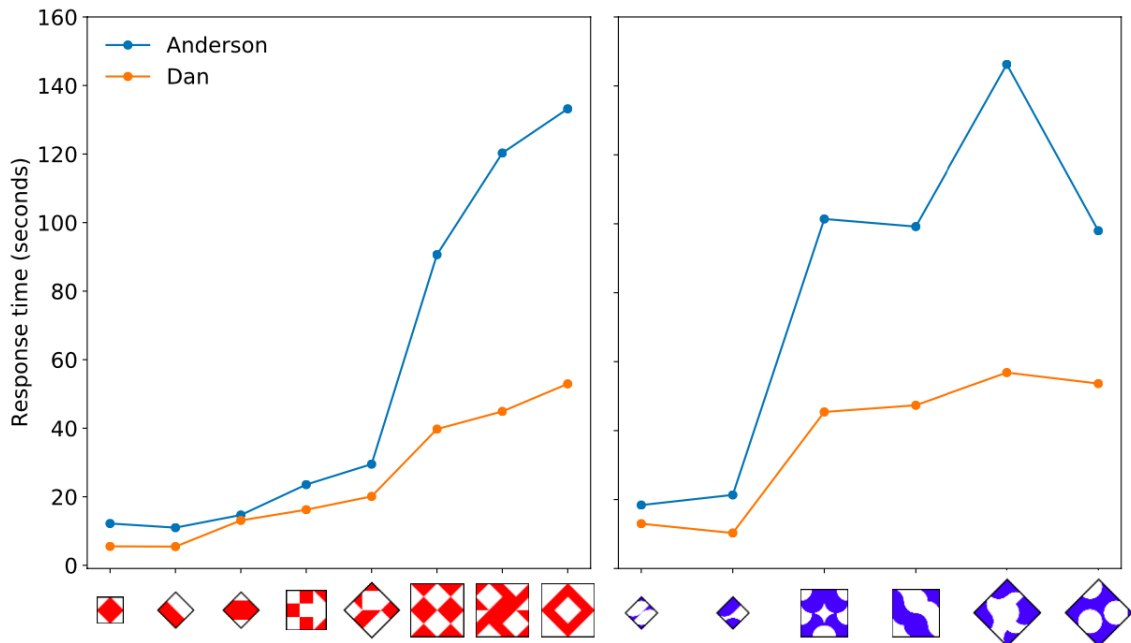


Figure 6.13: A chart of Anderson’s and Dan’s the response times on all the blue and red block tasks.

between Anderson and Dan. From this chart, we can clearly see that both participants started out with similar response times on the easier tasks, but Dan’s performance significantly improved as the tasks became more difficult.

Although this chart tells us that Dan performed better than Anderson, it does not give us information about why this is so. To get such insights into each participant’s strategy choices, we may need to observe other performance metrics. One way we could infer a participant’s strategy would be to use data from the overhead camera and gaze sensors. From the overhead camera data we could analyse the sequence in which participants placed blocks, and we could also use data from the gaze tracker to observe how participants paid attention to different regions while reasoning through the task.

Figures 6.14, and 6.15 show two different charts that provide some insights into why Dan may have performed better than Anderson. Data for these charts were recorded while both participants attempted the 7th red block puzzle (see the x-axis of the chart in Figure 6.13 for a reference of the puzzle’s design).

The first chart, Figure 6.14, shows the gaze and attention patterns both participants

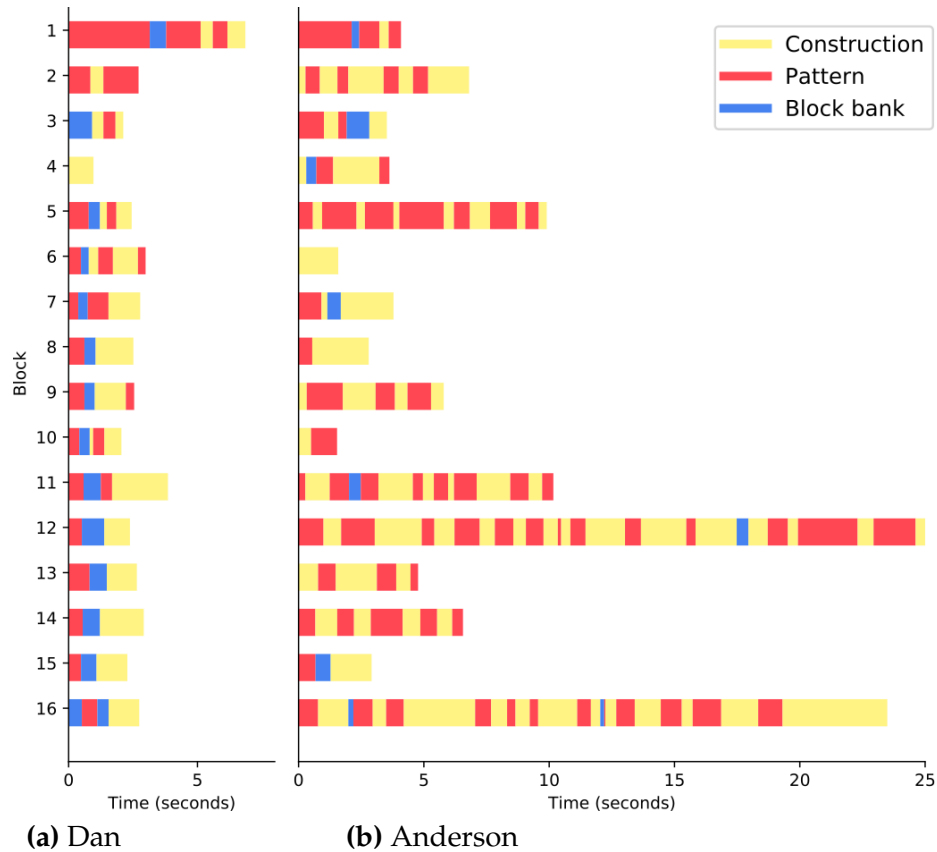


Figure 6.14: A visualization of the gaze and attention patterns exhibited by Dan and Anderson as they reasoned through a single instance of the block design task.

exhibited. Here, we see how Dan relied consistently on the strategy of looking at the pattern briefly before picking up a block from the bank and ultimately making a decision about block placements. Although he deviated from this strategy a few times, in most cases the deviation was because he looked back to confirm the design after finally placing the block. Anderson, on the other hand, spent much time shifting attention between the construction and the pattern, probably because he kept forgetting the design and had to remind himself.

The block placement charts, labeled (a) and (b) in Figure 6.15, additionally give us some more information about the strategy choices of both participants. In (b) we have Dan's block placement, and from it, we see how he methodically placed blocks in a row-by-row fashion. When compared to Anderson's (in (a)), we see a much more segmented approach where the figure is broken up into quadrants, and the quadrants with similar designs are

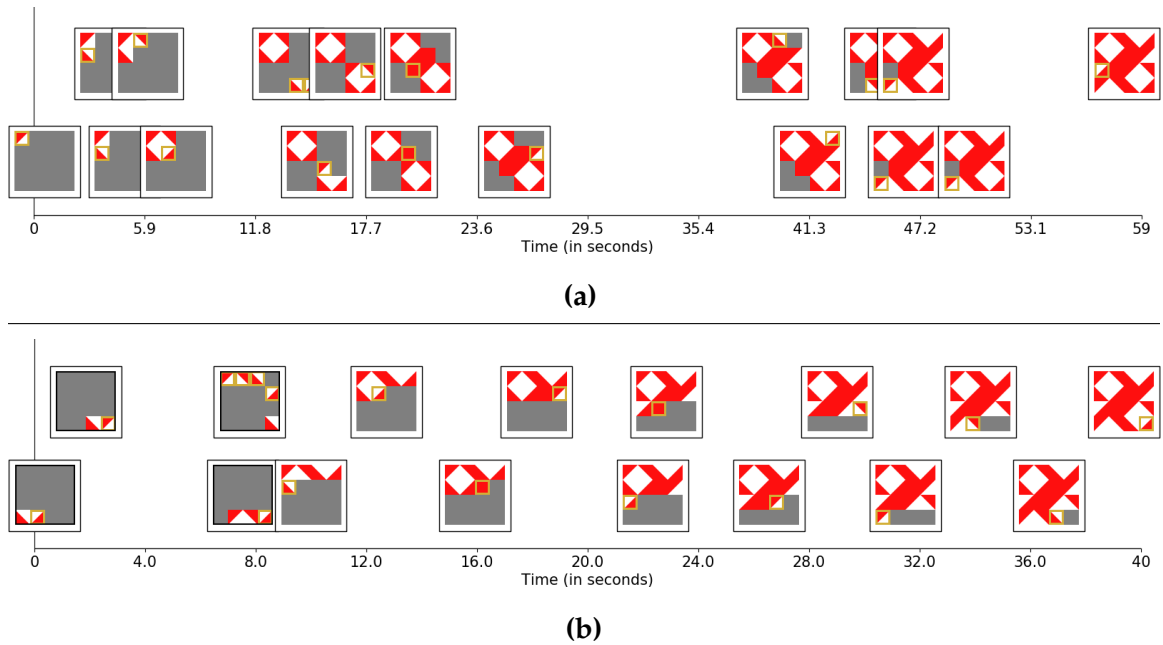


Figure 6.15: Block placement charts that show how Anderson (a) and Dan (b) placed their blocks respectively, while solving the 7th red puzzle from Figure 6.13

solved one after the other.

6.8 Taking the Block Design Test Online

Early in 2020, plans were quite advanced in our research group for a larger study based on the improved second iteration of the block design evaluation system. However, due to the global pandemic, all plans were shelved, and we instead took the opportunity to pivot to an online evaluation system. The goal was to administer block design studies through a web interface. This led to the development of the Web-based Online Measurement of Block Arrangement Task (WOMBAT) system.

WOMBAT provided an experience of the BDT that was similar in most ways to the physical one, sans the physical tactile blocks. Its design was heavily influenced by our physical test environments, with a lot of the original task's characteristics transferred over and even reinforced in some cases.

As with in-person BDT instances, participants are presented with a work environment that has a block bank, a designated area for displaying the design, and a construction area. Blocks are manipulated with the mouse through traditional mouse gestures, like dragging

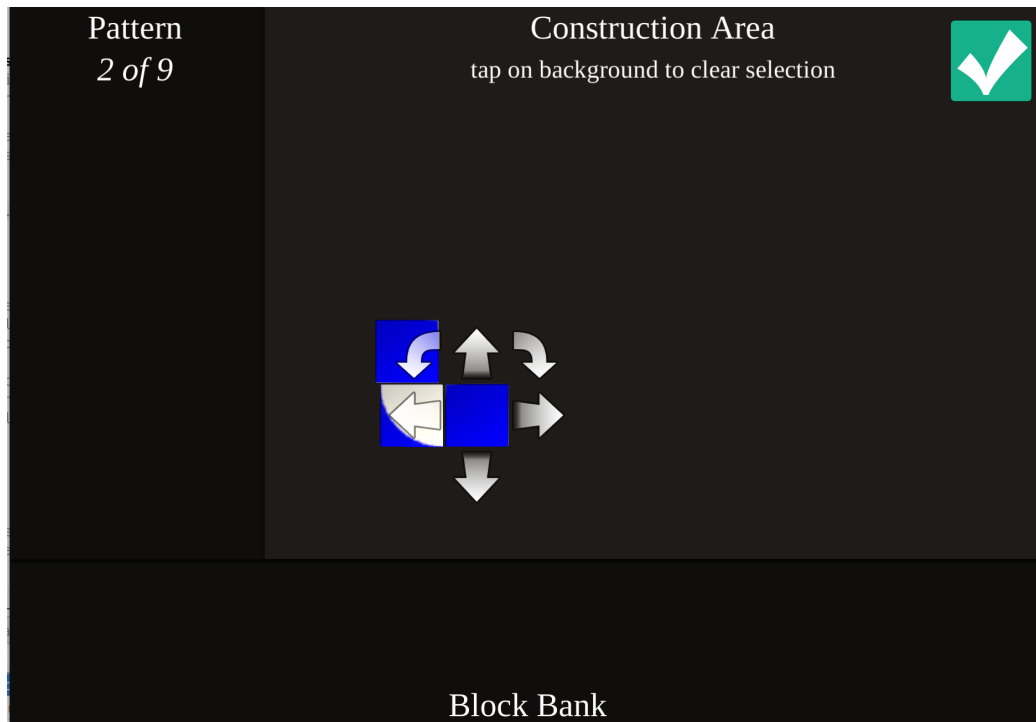


Figure 6.16: A screenshot of the WOMBAT system showing a block design task session in progress. In this view, the user is manipulating a block in the construction area with the pattern and block bank obscured.

blocks across the screen or using on-screen arrow buttons to rotate the blocks.

A participant's attention is gleaned through the mouse pointer, which also happens to be the same tool with which blocks are manipulated. Just as with physical block design instances, the task environment was split between the construction area, the block bank, and the design. Further, to strictly ensure that participants were actually paying attention to these specific regions, the respective gaze target was blanked out when the mouse was not hovering over. When participants were taking tests, wombat recorded every mouse movement, every block placement, and also every attention shift.

6.8.1 First Participant Study

So far we have conducted a single major study with WOMBAT. For this study, participants were recruited through Prolific (Palan & Schitter, 2018), a platform for performing behavioral research studies. The online nature of WOMBAT made it best suited to participant sourcing platforms like Prolific. In all, a total of 80 participants, 40 female and 40 male,

with 20 self-identifying as autistic individuals, took part in the study. In addition to Pro-lific's filtering, we deployed a demographic questionnaire that had a question to clarify the clinical diagnosis of autistic participants. Specifically, the questionnaire asked "Have you received a formal clinical diagnosis of Autism Spectrum Disorder?", for which participants could answer as any of the following: (a) Yes, as an adult; (b) Yes, as a child; (c) I am in the process of receiving a diagnosis; (d) No, but I identify as being on the autism spectrum. Apart from the demographic questionnaire, participants also worked through the Autism Quotient, completed a Spacial Habits Survey, and a Paper folding Task.

Of the 80 participants who took part in the study, we were only able to use data from 39. Most participants' data were excluded from analyses because they failed to properly complete the block design task. Some participants simply submitted empty responses, while others only completed the other questionnaires without touching the block design task. Additionally, participants who declined to answer any questions about their clinical autism diagnosis were also excluded. In our final population of 39 participants, 15 self-identified as autistic individuals.

6.8.2 Assessing Strategy Differences on The study

Staying on the theme of exploring strategy differences between autistic and non-autistic individuals, we performed analyses on the data to find out how well participants from each population group (ASD and Non-ASD) performed. With response times being one of the main metrics by which the block design is typically evaluated, we initially visualized the response times of autistic and non-autistic participants on all items on the task. Figure 6.17 shows a box plot of all the response times of the different populations on all the items on the task.

The results fall in line with observations from the literature, whereby autistic individuals typically perform better than their non-autistic peers in the block design (Shah & Frith, 1993). In a bid to find out the specific strategy differences that autistic individuals may have employed, we further analysed how attention was distributed by participants on tasks. We performed these analyses on a task by task basis, and for the purpose of this dissertation, I will be using the results from the 6th puzzle—the puzzle with the highest

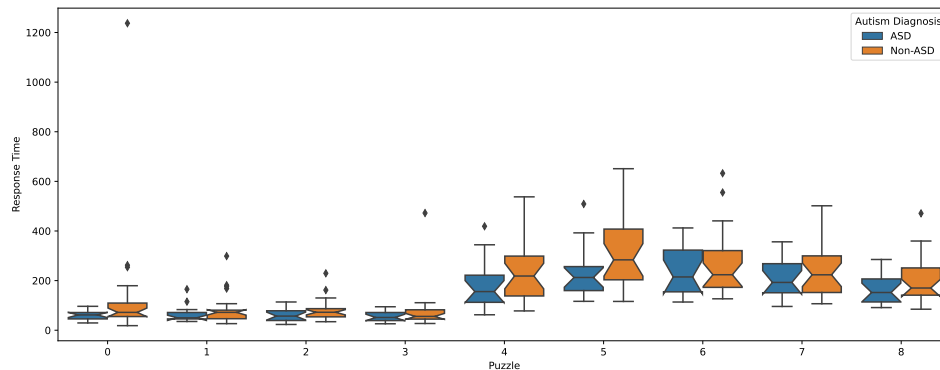


Figure 6.17: A box plot showing the response times of the autistic and non-autistic participants.

variation in response times when averages of the different populations were considered.

Figure 6.18 shows how the different participants distributed their attention when working through this 6th puzzle. Although there were no significantly discernable differences in the attention patterns of both population groups, it can be seen that autistic participants spent less time than their non-autistic participants switching attention between the design and construction area. This could be a signal that non-autistic participants may have had the need to constantly refresh their memory about the design. Interestingly, the best performing participant on this task was from the non-autistic population.

Also, when the attention charts in Figure 6.18 are analysed, two strategy trends are seen to be played out in some of the best performing participants, most of whom were in the autistic group. In the first of these strategies, participants quickly assembled all the blocks into a grid and proceeded to perform all the rotations needed to complete the design. This strategy limited switching of attention between the design and the construction area, and significantly improved performance.

The second of these strategies, which was quite similar to the first one, involved the participants moving blocks from the block bank into the construction area, without necessarily putting them in a grid. Both strategies served the participant with one particular advantage: removed the restrictions placed by panels that obscure the block bank when the mouse is hovering over. This way, the block bank is continuously available, and participants

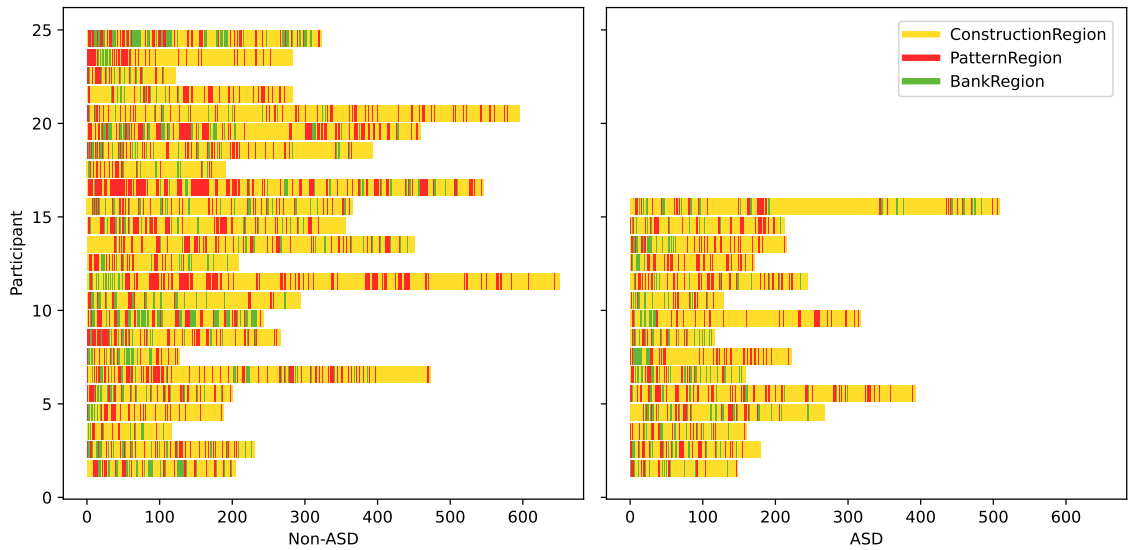


Figure 6.18: Attention charts for all participants on the 6th task from the BDT. The graph on the left shows the attention of all Non-ASD participants, and the graph on the right shows the attention of all ASD participants.

can quickly attend to them when needed.

It is worth noting that both of these strategies only be advantageous when the block design is administered in a virtual form like WOMBAT. In a real life block design task, gaze shifts between the block bank, design, and construction area tend to be larger. In WOMBAT, having blocks displayed on a relatively smaller screen means participants can simultaneously attend to the unused block and the construction at the same time. Even if participants had to make gaze shifts, these shifts would be so small it is safe to assume that participants had a complete view of the entire problem at the same time.

Taking things a step further, some participants went ahead to combine both of these strategies, by first moving all blocks into the construction area, before proceeding to construct a grid, and ultimately rotate the blocks in place. Some of the blocks were placed directly in the correct orientation without any rotation, and sometimes rotations were made before blocks were placed in the grid.

6.8.3 Future Work on WOMBAT

Although there was not enough time for us to perform numerous experiments on WOMBAT before the completion of this dissertation work. The data from this initial pilot study was enough to show how different environments could lead people to adopt specific strategies when solving the same task. The two strategies of in-place rotation and moving blocks into advantageous positions will not be practical in real world implementations of the block design task. Blocks placed together will be too clumsy to move about when the in-place rotation strategy is used in the physical block design, and moving blocks out of a designated block bank will not reduce the costs of gaze shifts in any significant way.

After the first participant study, a lot was learned from how participants interacted with the system, and several changes are being worked on for a second iteration. For example, we are working on having blocks snap together to ease grid construction. In the current WOMBAT system, with the blocks being free to move about, a number of participants spent a significant portion of their response times ensuring that blocks are displayed in a well-formed grid. In physical versions of the task, users do not have to perform this step, since participants typically square up their grids by pressing them together. Other improvements are with respect to how instructions are given to participants about the process of taking the test, with interactive tutorials and a few task comprehension questions. These would ensure that people will not just skip through the task, but actually make the effort to complete the items. With more people completing items, we are guaranteed to have more data for deeper analyses.

CHAPTER 7

Conclusions and Future Work

Work on this dissertation was directed at studying and exploring different ways in which strategies are implemented and used in intelligent systems. One primary goal was to study how strategies could be formalized, so they could be systematically synthesized, scrutinized and transferred. The thrust of this work was split three ways: There were investigations into strategies that were hand-coded by the designers of a system, there was an exploration of how program synthesis could be leveraged by AI systems to generate their own strategies, and there was a study of ways in which human performance on tasks can be recorded and analysed to identify and extract a person's strategy.

The tasks I studied in this work were visuospatial in nature, and for most of my experiments and studies, an emphasis was placed on using imagery as a knowledge representation.

Broadly, the questions I pursued in this work led to a couple of interesting observations. But there were other things I would have liked to explore further. For the rest of this chapter, I will be discussing contributions made in each of the research directions, and I will be sharing my thoughts on possible future directions the work described in this document could take.

7.1 On Hand Coded Strategies

While investigating hand coded strategies, I studied the VZ-2 Paper Folding Task, the Leiter-R tests, and the Block Design Test. One important contribution from this thrust of the work was that imagery representations are sufficient for—and in some cases present a more efficient way of—reasoning about all these task areas.

The strategies explored in this study fell in two categories: Those that were human inspired and those that attempted to express strategies in computer code. In building systems explored hand coded strategies, I took a “work at all cost” approach, whereby as long as a strategy can be represented in computer code, it was acceptable. I used this

approach of non-human inspired strategies to build models for the VZ-2 Paper Folding Task and a solver that reasons on a number of tasks from the Leiter-R.

To solve items from the VZ-2, a recursive strategy that relied on a three-dimensional model of the paper, and recursively folded the paper was implemented. And for the Leiter-R, programs were written to solve a number of sub-items with a small set of 6 imagery operations.

When it came to exploring human-inspired strategies, I put together a framework within which experiments were conducted. This framework, which I called Visuospatial Reasoning Environment for Experimentation (VREE) is a contribution of this dissertation work. VREE simulates a virtual environment in which virtual agents can interact with virtual objects. Agents have the ability to “look” at objects in the environment with a gaze window, through which agents receive imagery inputs. I evaluated the block design task, along with selected items from the Leiter-R on reasoning agents in VREE.

Work on generating hand coded strategies made it possible to identify operations that were both sufficient and necessary for reasoning about the tasks of interest. The knowledge gained from doing this work fed directly into the design of experiments on machine synthesized strategies.

7.2 On Machine Generated Strategies

After establishing that tasks in the domain of this dissertation could be solved through programs, the next question of whether strategies could be synthesized, given a set of operations, was answered through program synthesis. Program synthesis allows us to build systems that generate programs to meet given task specifications. The approach taken in this work involved searching a space of potential programs, which were meant to represent strategies, for those that are meaningful to the tasks at hand. The domain specific language designed for defining strategies in this dissertation, was the Visual Imagery Reasoning Language (VIMRL).

VIMRL programs can be presented as a straight listing of instructions, or they could be composed into state machines where every state has an assigned program. Another property of VIMRL execution is how instructions can infer their arguments at runtime

through local searches on the tasks being solved.

In a first test for VIMRL, I built a solver for the Abstract Reasoning Corpus. The process of building this solver served as a valuable testing ground for developing the language and its associated search techniques. The solver performed well on the public ARC dataset, and tied for 4th place in the 2022 International ARCATHON where the private hidden ARC dataset was used.

With the language sufficiently developed, I leveraged it in a state-machine representation to define strategies for reasoning about the Block Design Task. In working towards synthesizing strategies, this state machine structure was used as a state space within which to search for BDT strategies. I made progress in building an agent that forms strategies for stripped down versions of the BDT.

The stripped down BDT had a limited number of blocks and the agent's freedom to move were restricted. Although in all my experiments I was only able to work with a single block, I was able to progressively increase the degrees of freedom until the agent could move the block between the bank and the design, and also flip the block along two axes. Adding any more blocks or degrees of freedom led to search spaces that were too large to explore.

Regardless of the challenges in scaling strategy synthesis to the full block design task, observations from the simplified BDT provided a concrete way of visualizing how the sequencing of instructions affected an agent's efficiency in completing the task.

7.3 On Investigating Human Strategies

Ultimately, a better view of how humans form strategies and how these strategies can be analysed is a key step towards understanding human reasoning in general. To that effect, a significant portion of my work on this dissertation was spent in helping build systems for quantifying human performance on the block design task. This portion of the work involved producing tools that allowed for the capture, visualization, and analysis of human performance on the BDT.

For the work in this dissertation, I relied on two classes of human performance data: those collected through in-person test sessions and those collected through online test

sessions. Three different systems were used to obtain this data. There were two iterations of in-person performance recording systems, and a third online system. All three systems represented progressive improvements in evaluating human performance on the BDT.

Work performed on the first in-person BDT system helped lay foundations for effective ways in setting up such designs. It showed the importance of collecting good gaze data, and it provided a good proving ground for developing techniques for synchronizing data streams. With some of the kinks in the first in-person system removed, the second one allowed for more efficient data collection, and it improved the test taking experience for participants. The final online system

Unfortunately, none of these systems were used in full-blown studies. They were, however, used in effective pilot studies which yielded meaningful data that was worth reporting on. Data from the first in-person system showed how some specific of participants exhibited consistent strategies, like looking once at the design. Data from the second system gave us a case study of why autistic individuals may be outperforming others. And data from the first only pilot showed how people could adapt strategies to minimize the cost of taking the test, such as evading gaze restrictions by repositioning blocks.

In the end, it is worth noting that work done with these systems provided a solid foundation on which larger studies can be performed in future.

7.4 Summary of Contributions

Overall, the work on this dissertation led to four main contributions:

1. I was able to demonstrate sufficiency of imagery as a representation for reasoning about four main tasks: The VZ-2 Paper Folding Task, the Leiter Intelligence Scale, the Block Design Task, and the Abstract Reasoning Corpus. As part of this contribution, I developed the Visuospatial Reasoning Environment for Experimentation, which provided a framework for performing visual reasoning experiments on virtual agents for evaluating human-like reasoning experiments.
2. I developed the Visual Imagery Reasoning Language (VIMRL), which was used by the AI systems in this dissertation for representing reasoning strategies. While

developing VIMRL, I tested its features on a solver for the Abstract Reasoning Corpus, which tied for 4th in the 2022 ARCATHON.

3. I demonstrated the synthesis of strategies on modified versions of the Block Design Task. The strategies synthesized were expressed in state machines driven by VIMRL programs. This ability to synthesize strategies provides a foundation on which strategy differences on visual reasoning tasks can be studied.
4. I contributed to the design of systems for collecting data about human performance on the block design task, from both in-person and online evaluation sessions. I additionally worked on a slew of visualization and analytical tools to help in identifying and extracting potential strategies people may be using on the Block Design Task.

7.5 Future Work

There were a few lines of work I would have preferred to pursue if I had more time. Particularly, I would have favoured honing the search algorithms by working towards increased performance on the ARC benchmark, pushing the work on synthesizing strategies further into larger instances of the BDT, and exploring ways of interpreting human BDT performance data with synthesized strategies.

7.5.1 The Abstract Reasoning Corpus and Search

The Abstract Reasoning Corpus is a difficult benchmark. Its lack of language use and reliance on abstract concepts captured as few-shot image training pairs puts effective solutions out of the range of most traditional machine learning algorithms. While working on this dissertation, the ARC served as good proving ground for developing search algorithms. The inherent difficulty led me to devise creative optimizations and workarounds that ultimately led to better search performance in other tasks.

Although the current performance of the VIMRL solver on ARC is acceptable, there are a few things that could bring some potential improvements. First, a complete corpus of ground truth programs that cover most, if not all, of the publicly available tasks will do a better job of capturing the knowledge required to solve ARC tasks. This way, the

solver could rely more on a knowledge driven search that could explore more potential programs, instead of the current best performing brute-force approaches described in this dissertation. Just as large image datasets like ImageNet were instrumental to the computer vision in driving innovation, the ARC benchmark has the potential to do the same for the program synthesis and visual reasoning community.

7.5.2 Synthesizing and Understanding Strategies

As it currently stands in this dissertation, the work on synthesizing strategies demonstrates the feasibility of the approach, along with showing the usefulness of representing strategies as programs. This usefulness comes from programs having the ability to encapsulate strategies in a way that allows them to be applied in multiple instances of problems from a given class (such as different instances of a BDT puzzles being solved by a single strategy represented as a program.)

Future work on synthesizing strategies could be focused on simpler visual reasoning tasks, for which the space of strategies will be smaller. Although, such a task may also exhibit a limited diversity of strategies, it will provide a good proving ground for the techniques described in this dissertation. Additionally, when richer streams of human performance data is collected on the block design task, strategies could be synthesized and mapped to this human performance data to obtain better insights about people's choices.

7.6 Final Thoughts

In this dissertation, I worked to explore different ways in which strategy differences can be expressed and studied in intelligent systems. I hope the work described here provides another small piece of the puzzle being worked on towards the grand quest of unearthing the mechanisms behind human cognitive abilities. As valuable research in this area increases, it will not only improve life for us as humans, but it can also better equip us to build efficient AI systems that reason much like humans do.

Bibliography

- Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. *OSDI*, 16, 265–283.
- Ackerman, P. L. (1988). Determinants of individual differences during skill acquisition: Cognitive abilities and information processing. *Journal of Experimental Psychology: General*, 117(3), 288–318.
- Ainooson, J., & Kunda, M. (2017). A computational model for reasoning about the paper folding task using visual mental images. *CogSci*.
- Ainooson, J., Michelson, J., Sanyal, D., Palmer, J. H., & Kunda, M. (2020). Strategies for visuospatial reasoning: Experiments in sufficiency and diversity. *Proceedings of the Eighth Annual Conference on Advances in Cognitive Systems (ACS)*, 20. <https://par.nsf.gov/biblio/10209971>
- Akshoomoff, N. A., & Stiles, J. (1996). The influence of pattern type on children's block design performance. *J. International Neuropsychological Society*, 2(5), 392–402.
- Albers, A. M., Kok, P., Toni, I., Dijkerman, H. C., & De Lange, F. P. (2013). Shared representations for working memory and mental imagery in early visual cortex. *Current Biology*, 23(15), 1427–1431.
- Anderson, J. R. (1993). *Rules of the mind*. Lawrence Erlbaum Associates, Inc.
- Baddeley, A. D., & Hitch, G. (1974, January). Working Memory. In G. H. Bower (Ed.). Academic Press. [https://doi.org/10.1016/S0079-7421\(08\)60452-1](https://doi.org/10.1016/S0079-7421(08)60452-1)
- Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., & Tarlow, D. (2017). DeepCoder: Learning to write programs. *arXiv:1611.01989 [cs]*
Comment: Submitted to ICLR 2017.
- Ben-Yishay, Y., Diller, L., Mandleberg, I., Gordon, W., & Gerstman, L. J. (1971). Similarities and differences in block design performance between older normal and brain-injured persons: A task analysis. *Journal of Abnormal Psychology*, 78(1), 17.
- Berry, G. (2000). The foundations of estereel. In *Proof, language, and interaction: Essays in honour of robin milner* (pp. 425–454).
- Bertel, S., Barkowsky, T., König, P., Schultheis, H., & Freksa, C. (2006). Sketching mental images and reasoning with sketches: NEVILLE—a computational model of mental & external spatial problem solving.
- Biermann, A. W. (1978). The Inference of Regular LISP Programs from Examples. *IEEE Transactions on Systems, Man, and Cybernetics*, 8(8), 585–600. <https://doi.org/10.1109/TSMC.1978.4310035>
- Birchall, D. (2015). Spatial ability in radiologists: A necessary prerequisite? *The British Journal of Radiology*, 88(1049). <https://doi.org/10.1259/bjr.20140511>
- Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

- Bringsjord, S., & Schimanski, B. (2003). What is artificial intelligence? psychometric AI as an answer. *Proceedings of the 18th international joint conference on Artificial intelligence*, 887–893.
- Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., . . . Amodei, D. (2020). Language models are few-shot learners. <https://doi.org/10.48550/ARXIV.2005.14165>
- Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P. I., Tavener, S., Perez, D., Samothrakis, S., Colton, S., & et al. (2012). A survey of monte carlo tree search methods. *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI*.
- Burger, D., Stassun, K. G., Pepper, J., Siverd, R. J., Paegert, M. A., & Lee, N. M. D. (2012). Filtergraph: A flexible web application for instant data visualization of astronomy datasets.
- Butler, E., Torlak, E., & Popović, Z. (2017). Synthesizing interpretable strategies for solving puzzle games. *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 1–10. <https://doi.org/10.1145/3102071.3102084>
- Campbell, M., Hoane, A. J., & Hsu, F. (2002). Deep blue. *Artif. Intell.*, 134, 57–83.
- Cha, S., Ainooson, J., Chong, E., Soulières, I., Rehg, J. M., & Kunda, M. (2020). Enhancing cognitive assessment through multimodal sensing: A case study using the block design test.
- Chollet, F. (2019). On the Measure of Intelligence.
- Chown, E. (2014). Cognitive modeling.
- Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1), 37–46.
- Cooper, A. (2021, July). Recruiting for talent on the autism spectrum [(Correspondent)]. *CBS News*.
- Cooper, L. A., & Shepard, R. N. (1973). CHRONOMETRIC STUDIES OF THE ROTATION OF MENTAL IMAGES. In W. G. CHASE (Ed.), *Visual information processing* (pp. 75–176). Academic Press. <https://doi.org/10.1016/B978-0-12-170150-5.50009-3>
- Cowan, N. (2001). The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, 24(1), 87–114. <https://doi.org/10.1017/S0140525X01003922>
- Doumas, L. A., & Hummel, J. E. (2012). Computational models of higher cognition. *The Oxford handbook of thinking and reasoning*, 19.
- Duda, R., & Hart, P. (1972). Use of the hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15, 11–15.

- Ekstrom, R. B., & Harman, H. H. (1976). *Manual for kit of factor-referenced cognitive tests, 1976*. Educational testing service.
- Evans, T. G. (1964). A heuristic program to solve geometric-analogy problems. *Proceedings of the April 21-23, 1964, Spring Joint Computer Conference*, 327–338. <https://doi.org/10.1145/1464122.1464156>
- Funt, B. V. (1980). Problem-solving with diagrammatic representations. *Artif Intell*, 13(3), 201–230. [https://doi.org/10.1016/0004-3702\(80\)90002-8](https://doi.org/10.1016/0004-3702(80)90002-8)
- Gardin, F., & Meltzer, B. (1989). Analogical representations of naive physics. *Artificial Intelligence*, 38(2), 139–159. [https://doi.org/10.1016/0004-3702\(89\)90055-6](https://doi.org/10.1016/0004-3702(89)90055-6)
- Giaquinto, M. (2007, July). *Visual Thinking in Mathematics*. Clarendon Press.
- Glasgow, J., & Papadias, D. (1992). Computational imagery. *Cognitive Science*, 16(3), 355–394. [https://doi.org/10.1016/0364-0213\(92\)90037-U](https://doi.org/10.1016/0364-0213(92)90037-U)
- Goldsmith, L. T., Hetland, L., Hoyle, C., & Winner, E. (2016). Visual-spatial thinking in geometry and the visual arts. *Psychology of Aesthetics, Creativity, and the Arts*, 10(1), 56–71. <https://doi.org/10.1037/aca0000027>
- Grandin, T. (1995). *Thinking in pictures: And other reports from my life with autism*. Doubleday.
- Gulwani, S. (2016). Automating String Processing in Spreadsheets using Input-Output Examples. Retrieved July 2, 2021, from <https://www.microsoft.com/en-us/research/publication/automating-string-processing-spreadsheets-using-input-output-examples/>
- Gulwani, S., Hernández-Orallo, J., Kitzelmann, E., Muggleton, S. H., Schmid, U., & Zorn, B. (2015). Inductive programming meets the real world. *Commun. ACM*, 58(11), 90–99. <https://doi.org/10.1145/2736282>
- Gulwani, S., Polozov, O., & Singh, R. (2017). Program synthesis [ISBN: 9781680832921 Num Pages: 126 Place: Hanover, MA Delft Publisher: Now Publishers Series Number: 4.2017, 1-2]. *Foundations and trends in programming languages*.
- Halbwachs, N., Caspi, P., Raymond, P., & Pilaud, D. (1991). The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9), 1305–1320. <https://doi.org/10.1109/5.97300>
- Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., . . . Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825), 357–362. <https://doi.org/10.1038/s41586-020-2649-2>
- Hegarty, M. (2004). Mechanical reasoning by mental simulation. *Trends in Cognitive Sciences*, 8(6), 280–285. <https://doi.org/10.1016/j.tics.2004.04.001>

- Hernández-Orallo, J., Martínez-Plumed, F., Schmid, U., Siebers, M., & Dowe, D. L. (2016). Computer models solving intelligence test problems: Progress and implications. *Artificial Intelligence*, 230, 74–107. <https://doi.org/10.1016/j.artint.2015.09.011>
- Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2006). *Introduction to automata theory, languages, and computation (3rd edition)*. Addison-Wesley Longman Publishing Co., Inc.
- Hutt, M. L. (1932). The kohs block-design tests. a revision for clinical practice. *Journal of Applied Psychology*, 16(3), 298–307. <https://doi.org/10.1037/h0074559>
- Hy, R. L., Arrigoni, A., Bessi re, P., & Lebeltel, O. (2004). Teaching bayesian behaviours to video game characters. *Robotics and Autonomous Systems*, 47(2-3), 177–185. <https://doi.org/10.1016/j.robot.2004.03.012>
- Johnson, A., Vong, W. K., Lake, B. M., & Gureckis, T. M. (2021). Fast and flexible: Human program induction in abstract reasoning tasks
Comment: 7 pages, 7 figures, 1 table.
- Jonassen, D. H., & Grabowski, B. L. (1993). *Handbook of individual differences, learning, and instruction*. Lawrence Erlbaum Associates, Inc.
- Jones, R. M., Laird, J. E., Nielsen, P. E., Coulter, K. J., Kenny, P., & Koss, F. V. (1999). Automated Intelligent Pilots for Combat Flight Simulation. *AI Magazine*, 20(1), 27–27. <https://doi.org/10.1609/aimag.v20i1.1438>
- Jones, R. S., & Torgesen, J. K. (1981). Analysis of behaviors involved in performance of the block design subtest of the WISC-R. *Intelligence*, 5(4), 321–328.
- Joy, S., Fein, D., Kaplan, E., & Freedman, M. (2001). Quantifying qualitative features of block design performance among healthy older adults. *Archives of Clinical Neuropsychology*, 16(2), 157–170. [https://doi.org/10.1016/S0887-6177\(99\)00063-3](https://doi.org/10.1016/S0887-6177(99)00063-3)
- Just, M. A., & Carpenter, P. A. (1976). Eye fixations and cognitive processes. *Cognitive Psychology*, 8(4), 441–480. [https://doi.org/10.1016/0010-0285\(76\)90015-3](https://doi.org/10.1016/0010-0285(76)90015-3)
- Just, M. A., & Carpenter, P. A. (1987). *The psychology of reading and language comprehension*. Allyn & Bacon.
- Kasneci, E., Sessler, K., K uchemann, S., Bannert, M., Dementieva, D., Fischer, F., Gasser, U., Groh, G., G nnemann, S., H llnermeier, E., Krusche, S., Kutyniok, G., Michaeli, T., Nerdel, C., Pfeffer, J., Poquet, O., Sailer, M., Schmidt, A., Seidel, T., . . . Kasneci, G. (2023). Chatgpt for good? on opportunities and challenges of large language models for education. *Learning and Individual Differences*, 103, 102274. <https://doi.org/https://doi.org/10.1016/j.lindif.2023.102274>
- Katayama, S. (2010). Recent improvements of magichaskeller. In U. Schmid, E. Kitzelmann, & R. Plasmeijer (Eds.), *Approaches and applications of inductive programming* (pp. 174–193). Springer Berlin Heidelberg.
- Keehner, M. M., Tendick, F., Meng, M. V., Anwar, H. P., Hegarty, M., Stoller, M. L., & Duh, Q.-Y. (2004). Spatial ability, experience, and skill in laparoscopic surgery. *American Journal of Surgery*, 188(1), 71–75. <https://doi.org/10.1016/j.amjsurg.2003.12.059>

- Kelley, A., Athy, J., King, M., Rickson, B., Chiaramonte, J., Vasbinder, M., & Thompson, A. (2011, August). *Think before You Shoot: The Relationship between Cognition and Marksmanship* (tech. rep. No. 2011-23). United States Army Aeromedical Research Laboratory.
- Kocsis, L., Szepesvari, C., & Willemson, J. (2006). Improved monte-carlo search.
- Kocsis, L., & Szepesvári, C. (2006). Bandit based monte-carlo planning. In J. Fürnkranz, T. Scheffer, & M. Spiliopoulou (Eds.), *Machine learning: Ecml 2006* (pp. 282–293). Springer Berlin Heidelberg.
- Kohs, S. C. (1920). The block-design tests. *Journal of Experimental Psychology*, 3(5), 357–376. <https://doi.org/10.1037/h0074466>
- Kolev, V., Georgiev, B., & Penkov, S. (2020). Neural abstract reasoner. *arXiv:2011.09860 [cs]* Comment: 12 pages, 8 figures.
- Kosslyn, S. M., & Shwartz, S. P. (1977). A simulation of visual imagery. *Cognitive Science*, 1(3), 265–295. [https://doi.org/10.1016/S0364-0213\(77\)80020-7](https://doi.org/10.1016/S0364-0213(77)80020-7)
- Kosslyn, S. M., Thompson, W. L., Klm, I. J., & Alpert, N. M. (1995). Topographical representations of mental images in primary visual cortex. *Nature*, 378(6556), 496–498.
- Kosslyn, S. M. (1973). Scanning visual images: Some structural implications. *Perception & Psychophysics*, 14(1), 90–94. <https://doi.org/10.3758/BF03198621>
- Kosslyn, S. M. (1980). *Image and mind*. Harvard University Press.
- Kotseruba, I., & Tsotsos, J. K. (2018). A Review of 40 Years of Cognitive Architecture Research: Core Cognitive Abilities and Practical Applications. *arXiv:1610.08602 [cs]* Comment: 74 pages, 10 figures.
- Kozhevnikov, M., Hegarty, M., & Mayer, R. E. (2002). Revising the visualizer-verbalizer dimension: Evidence for two types of visualizers. *Cognition and Instruction*, 20(1), 47–77. https://doi.org/10.1207/S1532690XCI2001_3
- Kramer, J., Blusewicz, M., Kaplan, E., & Preston, K. (1991). Visual hierarchical analysis of block design configural errors. *J. Clin. and Exp. Neuropsychology*, 13(4), 455–465.
- Kramer, J. H., Kaplan, E., Share, L., & Huckleba, W. (1999). Configural errors on WISC-III block design. *J. International Neuropsychological Society*, 5(6), 518–524.
- Kunda, M. (2017). Understanding the Role of Visual Mental Imagery in Intelligence: The Retinotopic Reasoning (R2) Cognitive Architecture. *2017 AAAI Fall Symposium Series*. Retrieved June 13, 2021, from <https://www.aaai.org/ocs/index.php/FSS/FSS17/paper/view/16013>
- Kunda, M. (2018). Visual mental imagery: A view from artificial intelligence. *Cortex*, 105, 155–172.
- Kunda, M., Ainooson, J., Elliott, F., Fallon, C., & Park, S. (2020, October 1). Our research on the block design test, as featured on cbs 60 minutes with anderson cooper. *Frist Center for AutismInnovation*. Retrieved May 1, 2023, from <https://www.vanderbilt>

edu/autismandinnovation/our-research-on-the-block-design-test-as-featured-on-cbs-60-minutes-with-anderson-cooper/

- Kunda, M., Banani, M. E., & Rehg, J. M. (2016). A computational exploration of problem-solving strategies and gaze behaviors on the block design task. *CogSci*.
- Kunda, M., McGreggor, K., & Goel, A. K. (2013). A computational model for solving problems from the Raven's Progressive Matrices intelligence test using iconic visual representations. *Cognitive Systems Research*, 22-23, 47–66. <https://doi.org/10.1016/j.cogsys.2012.08.001>
- Land, M., & Tatler, B. (2009). *Looking and acting: Vision and eye movements in natural behaviour*. Oxford University Press.
- Larkin, J. H., & Simon, H. A. (1987). Why a Diagram is (Sometimes) Worth Ten Thousand Words. *Cognitive Science*, 11(1), 65–100. <https://doi.org/10.1111/j.1551-6708.1987.tb00863.x>
- Lathrop, S. D., Wintermute, S., & Laird, J. E. (2011). Exploring the Functional Advantages of Spatial and Visual Cognition From an Architectural Perspective. *Topics in Cognitive Science*, 3(4), 796–818. <https://doi.org/10.1111/j.1756-8765.2010.01130.x>
- Lázaro-Gredilla, M., Lin, D., Guntupalli, J. S., & George, D. (2018). Beyond imitation: Zero-shot task transfer on robots by learning concepts as cognitive programs. *arXiv:1812.02788 [cs]*.
- Lovett, A., & Forbus, K. D. (2013). Modeling Spatial Ability in Mental Rotation and Paper-Folding. *CogSci*.
- Luck, S. J., & Vogel, E. K. (1997). The capacity of visual working memory for features and conjunctions. *Nature*, 390(6657), 279–281. <https://doi.org/10.1038/36846>
- Lufler, R. S., Zumwalt, A. C., Romney, C. A., & Hoagland, T. M. (2012). Effect of visual-spatial ability on medical students' performance in a gross anatomy course. *Anatomical Sciences Education*, 5(1), 3–9. <https://doi.org/10.1002/ase.264>
- Marewski, J. N., & Link, D. (2014). Strategy selection: An introduction to the modeling challenge. *WIREs Cognitive Science*, 5(1), 39–59. <https://doi.org/https://doi.org/10.1002/wcs.1265>
- Marr, D. (1982). *Vision: A computational investigation into the human representation and processing of visual information*. Henry Holt; Co., Inc.
- Marschark, M., Spencer, L. J., Durkin, A., Borgna, G., Convertino, C., Machmer, E., Kronenberger, W. G., & Trani, A. (2015). Understanding Language, Hearing Status, and Visual-Spatial Skills. *The Journal of Deaf Studies and Deaf Education*, 20(4), 310–330. <https://doi.org/10.1093/deafed/env025>
- Mayer, R., & Massa, L. (2003). Three Facets of Visual and Verbal Learners: Cognitive Ability, Cognitive Style, and Learning Preference. *Journal of Educational Psychology*, 95, 833–841. <https://doi.org/10.1037/0022-0663.95.4.833>

- McCarthy, J., Minsky, M. L., Rochester, N., & Shannon, C. E. (1955). A proposal for the dartmouth summer research project on artificial intelligence.
- Miller, G. A. (1956). The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychological Review*, 63, 81–97.
- Minsky, M. L. (1967). *Computation: Finite and infinite machines*. Prentice-Hall, Inc.
- Mumford, M. D., Baughman, W. A., Threlfall, K. V., Uhlman, C. E., & Costanza, D. P. (1993). Personality, Adaptability, and Performance: Performance on Well-Defined Problem Solving Tasks. *Human Performance*, 6(3), 241–285. https://doi.org/10.1207/s15327043hup0603_4
- National Research Council. (2009). Mathematics learning in early childhood: Paths toward excellence and equity. *National Academies Press*.
- Nerode, A. (1958). Linear automaton transformations.
- Newell, A., & Simon, H. A. (1976). Computer science as empirical inquiry: Symbols and search. *Commun. ACM*, 19(3), 113–126. <https://doi.org/10.1145/360018.360022>
- Nishino, K., & Nayar, S. K. (2004). *The World in an Eye* (tech. rep.).
- Norman, D. A. (2013). *The design of everyday things*. Basic Books.
- Paivio, A. (1990). *Mental representations: A dual coding approach*. Oxford University Press.
- Palan, S., & Schitter, C. (2018). Prolific.ac—a subject pool for online experiments. *Journal of Behavioral and Experimental Finance*, 17, 22–27. <https://doi.org/https://doi.org/10.1016/j.jbef.2017.12.004>
- Palmer, J. H., & Kunda, M. (2018). Thinking in polar pictures: Using rotation-friendly mental images to solve leiter-r form completion. *Thirty-Second AAAI Conference on Artificial Intelligence*.
- Parr, T. (2013). *The definitive antlr 4 reference* (2nd ed.). Pragmatic Bookshelf. <https://www.safaribooksonline.com/library/view/the-definitive-antlr/9781941222621/>
- Pearson, J., & Kosslyn, S. M. (2015). The heterogeneity of mental representation: Ending the imagery debate. *Proceedings of the National Academy of Sciences*, 112(33), 10089–10092. <https://doi.org/10.1073/pnas.1504933112>
- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.
- Petre, M., & Blackwell, A. F. (1999). Mental imagery in program design and visual programming. *International Journal of Human-Computer Studies*, 51(1), 7–30. <https://doi.org/10.1006/ijhc.1999.0267>
- Polyshyn, Z. W. (1973). What the mind's eye tells the mind's brain: A critique of mental imagery. *Psychological bulletin*, 80(1), 1.

- Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., Chen, M., & Sutskever, I. (2021). Zero-shot text-to-image generation. <https://doi.org/10.48550/ARXIV.2102.12092>
- Raven, J., & Raven, J. (2003). Raven Progressive Matrices. In R. S. McCallum (Ed.). Springer US. https://doi.org/10.1007/978-1-4615-0153-4_11
- Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 779–788. <https://doi.org/10.1109/CVPR.2016.91>
- Robert, C. P., & Casella, G. (2005). *Monte carlo statistical methods (springer texts in statistics)*. Springer-Verlag.
- Roid, G. H., & Miller, L. J. (1997). Leiter international performance scale-revised (leiter-r). *Wood Dale, IL: Stoelting*.
- Rosenbloom, P. S., Laird, J., & Newell, A. (Eds.). (1993, July). *The SOAR papers: Research on integrated intelligence*. The MIT Press.
- Roy, D., Hsiao, K.-Y., & Mavridis, N. (2004). Mental imagery for a conversational robot. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34(3), 1374–1383. <https://doi.org/10.1109/TSMCB.2004.823327>
- Rozencwajg, P., Cherfi, M., Ferrandez, A. M., Lautrey, J., Lemoine, C., & Loarer, E. (2005). Age related differences in the strategies used by middle aged adults to solve a block design task [PMID: 15801388]. *The International Journal of Aging and Human Development*, 60(2), 159–182. <https://doi.org/10.2190/H0AR-68HR-RRPE-LRBH>
- Rozencwajg, P., & Corroyer, D. (2002). Strategy development in a block design task. *Intelligence*, 30(1), 1–25. [https://doi.org/10.1016/S0160-2896\(01\)00063-0](https://doi.org/10.1016/S0160-2896(01)00063-0)
- Rozencwajg, P., & Fenouillet, F. (2012). Effect of goal setting on the strategies used to solve a block design task. *Learning and Individual Differences*, 22(4), 530–536. <https://doi.org/10.1016/j.lindif.2012.03.008>
- Rudolf Burggraaf, I. T. C. H., Maarten A. Frens, & van der Geest, J. N. (2016). A quick assessment of visuospatial abilities in adolescents using the design organization test (dot) [PMID: 25551357]. *Applied Neuropsychology: Child*, 5(1), 44–49. <https://doi.org/10.1080/21622965.2014.945114>
- Schatz, A., Ballantyne, A., & Trauner, D. (2000). A hierarchical analysis of block design errors in children with early focal brain damage. *Dev. Neuropsychology*, 17(1), 75–83.
- Schmid, U., & Kitzelmann, E. (2011). Inductive rule learning on the knowledge level. *Cognitive Systems Research*, 12(3), 237–248. <https://doi.org/Schmid2011>
- Schunn, C. D., & Reder, L. M. (1998). Strategy adaptivity and individual differences. In *The psychology of learning and motivation: Advances in research and theory* (pp. 115–154, Vol. 38). Academic Press.

- Shah, A., & Frith, U. (1993). Why do autistic individuals show superior performance on the block design task? *J. Child Psychology and Psychiatry*, *34*(8), 1351–1364.
- Shannon, C. E., & McCarthy, J. (1956). *Automata studies. (am-34) (annals of mathematics studies)*. Princeton University Press.
- Shaw, D. E., Swartout, W. R., & Green, C. C. (1975). Inferring LISP Programs from Examples. <https://doi.org/10.7916/D89K4K6X>
- Shehu, I. S., Wang, Y., Athuman, A. M., & Fu, X. (2021). Remote eye gaze tracking research: A comparative evaluation on past and recent progress. *Electronics*, *10*(24). <https://doi.org/10.3390/electronics10243165>
- Shepard, R. N., & Feng, C. (1972). A chronometric study of mental paper folding. *Cognitive Psychology*, *3*(2), 228–243. [https://doi.org/10.1016/0010-0285\(72\)90005-9](https://doi.org/10.1016/0010-0285(72)90005-9)
- Shepard, R. N., & Metzler, J. (1971). Mental rotation of three-dimensional objects. *Science*, *171*(3972), 4. <https://doi.org/10.1126/science.171.3972.701>
- Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489. <https://doi.org/10.1038/nature16961>
- Silvia, P. J. (2008). Another look at creativity and intelligence: Exploring higher-order models and probable confounds. *Personality and Individual Differences*, *44*(4), 1012–1021. <https://doi.org/10.1016/j.paid.2007.10.027>
- Simon, H. A., & Newell, A. (1971). Human problem solving: The state of the theory in 1970. *American Psychologist*, *26*, 145–159. <https://api.semanticscholar.org/CorpusID:14405801>
- Slotnick, S. D., Thompson, W. L., & Kosslyn, S. M. (2005). Visual mental imagery induces retinotopically organized activation of early visual areas. *Cerebral cortex*, *15*(10), 1570–1583.
- Solar-Lezama, A. (2008). *Program synthesis by sketching* [Doctoral dissertation] [AAI3353225]. University of California at Berkeley.
- Stewart, M. E., Watson, J., Allcock, A.-J., & Yaqoob, T. (2009). Autistic traits predict performance on the block design [PMID: 19261684]. *Autism*, *13*(2), 133–142. <https://doi.org/10.1177/1362361308098515>
- Stokes, M., Thompson, R., Cusack, R., & Duncan, J. (2009). Top-down activation of shape-specific population codes in visual cortex during mental imagery. *Journal of Neuroscience*, *29*(5), 1565–1572.
- Summers, P. D. (1977, January). A methodology for lisp program construction from examples. Association for Computing Machinery. <https://doi.org/10.1145/321992.322002>

- Sun, R., Zhang, X., & Mathews, R. (2006). Modeling meta-cognition in a cognitive architecture. *Cognitive Systems Research*, 7(4), 327–338. <https://doi.org/10.1016/j.cogsys.2005.09.001>
- Świechowski, M., Godlewski, K., Sawicki, B., & Mańdziuk, J. (2021). Monte carlo tree search: A review of recent modifications and applications.
- Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2818–2826.
- Tabachneck-Schijf, H. J. M., Leonardo, A. M., & Simon, H. A. (1997). CaMeRa: A computational model of multiple representations. *Cognitive Science*, 21(3), 305–350. [https://doi.org/10.1016/S0364-0213\(99\)80026-3](https://doi.org/10.1016/S0364-0213(99)80026-3)
- Thagard, P. (2005). *Mind: Introduction to cognitive science* (Vol. 17). MIT press Cambridge, MA.
- Toraldo, A., & Shallice, T. (2004). Error analysis at the level of single moves in block design [PMID: 21038226]. *Cognitive Neuropsychology*, 21(6), 645–659. <https://doi.org/10.1080/02643290342000591>
- Torlak, E., & Bodík, R. (2013). Growing solver-aided languages with rosette. *Onward!* <https://doi.org/10.1145/2509578.2509586>
- Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, 59(236), 433–460. <http://www.jstor.org/stable/2251299>
- Tversky, B. (2005). Visuospatial Reasoning.
- Uttal, D. H., Meadow, N. G., Tipton, E., Hand, L. L., Alden, A. R., Warren, C., & Newcombe, N. S. (2013). The malleability of spatial skills: A meta-analysis of training studies. *Psychological Bulletin*, 139(2), 352–402. <https://doi.org/10.1037/a0028446>
- Van Rossum, G., & Drake, F. L. (2009). *Python 3 reference manual*. CreateSpace.
- Wai, J., Lubinski, D., & Benbow, C. P. (2009). Spatial ability for STEM domains: Aligning over 50 years of cumulative psychological knowledge solidifies its importance. *Journal of Educational Psychology*, 101(4), 817–835. <https://doi.org/10.1037/a0016127>
- Wanzel, K. R., Hamstra, S. J., Anastakis, D. J., Matsumoto, E. D., & Cusimano, M. D. (2002). Effect of visual-spatial ability on learning of spatially-complex surgical skills. *The Lancet*, 359(9302), 230–231. [https://doi.org/10.1016/S0140-6736\(02\)07441-X](https://doi.org/10.1016/S0140-6736(02)07441-X)
- Wechsler, D. (1974). *Wechsler intelligence scale for children—revised*. Psychological Corporation.
- Wechsler, D. (2008). *Wechsler adult intelligence scale—fourth edition*.
- West, T. G. (2009). *In the mind's eye: Creative visual thinkers, gifted dyslexics, and the rise of visual technologies*. Prometheus Books.

- Winnerman, L. (2018). Making a thinking machine. *Monitor on Psychology, 49*(4). Retrieved July 20, 2021, from <http://www.apa.org/monitor/2018/04/cover-thinking-machine>
- Winston, P. H. (1992). *Artificial intelligence (3rd ed.)* Addison-Wesley Longman Publishing Co., Inc.
- Wittenburg, P., Brugman, H., Russel, A., Klassmann, A., & Sloetjes, H. (2006). Elan: A professional framework for multimodality research. *5th International Conference on Language Resources and Evaluation (LREC 2006)*, 1556–1559.
- Wright, R., Thompson, W. L., Ganis, G., Newcombe, N. S., & Kosslyn, S. M. (2008). Training generalized spatial skills. *Psychonomic Bulletin & Review, 15*(4), 763–771. <https://doi.org/10.3758/PBR.15.4.763>
- Yang, Y., Mcgreggor, K., & Kunda, M. (2020). Not quite any way you slice it: How different analogical constructions affect raven's matrices performance (null, Ed.). *Proceedings of the Eighth Annual Conference on Advances in Cognitive Systems (ACS)*. <https://par.nsf.gov/biblio/10209953>
- Young, R. M. (2001). Production systems in cognitive psychology. *NJ Smelser and PB Baltes, eds*, 12143–12146.
- Young, R. M., & O'Shea, T. (1981). Errors in children's subtraction. *Cognitive Science, 5*(2), 153–177. Retrieved June 3, 2021, from <https://www.sciencedirect.com/science/article/pii/S0364021381800304>
- Zhai, X., Kolesnikov, A., Houlsby, N., & Beyer, L. (2021). Scaling Vision Transformers [arXiv: 2106.04560 version: 1]
Comment: Xiaohua, Alex, and Lucas contributed equally.
- Zhang, J., & Zong, C. (2020). Neural Machine Translation: Challenges, Progress and Future
Comment: Invited Review of Science China Technological Sciences.
- Zipf-Williams, E., Shear, P., Strongin, D., Winegarden, B., & Morrell, M. (2000). Qualitative block design performance in epilepsy patients. *Archives of Clinical Neuropsychology, 15*, 149–157.

Appendix A

List of VIMR operations for ARC

The VIMRL based ARC solver had access to a total of 105 operations. The table below provides an extensive list of all these operations along with their parameters.

Name	Execution	Parameters	Type
add	Low Level	('objects',)	image
align_objects	High Level	('objects',)	objects
attract	High Level	('objects',)	objects
change_color	Low Level	('image', 'color')	image
color_sorted	High Level	('objects',)	objects
complete_pattern	High Level	('image',)	image
connect_pixels	High Level	('image',)	image
count	Low Level	('objects',)	number
create	Low Level	('image',)	image
create_image	High Level	('number',)	image
draw	Low Level	('image', 'objects')	image
filter_color_block	Low Level	('image', 'color')	image
filter_color_pass	Low Level	('image', 'color')	image
filter_full_symmetric	Low Level	('objects',)	objects
filter_horz_symmetric	Low Level	('objects',)	objects
filter_vert_symmetric	Low Level	('objects',)	objects
find_central_object	Low Level	('image',)	image
find_enclosed_patches	Low Level	('image',)	objects
find_missing_patch	High Level	('image',)	image
find_objects	Low Level	('image',)	objects
find_objects_in_context	Low Level	('image',)	objects
find_objects_nd	Low Level	('image',)	objects
find_odd_object	Low Level	('objects',)	image
first_image	Low Level	('objects',)	image
first_object	Low Level	('objects',)	objects
flip_horz	Low Level	('image',)	image
flip_vert	Low Level	('image',)	image
get_color	Low Level	('image',)	color
get_grid	High Level	('image',)	grid
get_grid_cells	Low Level	('grid',)	objects
get_grid_color	Low Level	('grid',)	color
grid_as_image	Low Level	('grid',)	image
grid_from_image	Low Level	('grid', 'image')	grid
grow_objects	High Level	('objects',)	objects
grow_pixels	High Level	('image',)	image
head	Low Level	('objects',)	objects
invert	Low Level	('image',)	image
last_image	Low Level	('objects',)	image

Name	Execution	Parameters	Type
last_object	Low Level	('objects',)	objects
make_color_patch	High Level	('color',)	image
make_enclosed_patches	Low Level	('image',)	image
make_unique	Low Level	('objects',)	objects
map_change_color	Low Level	('objects', 'color')	objects
map_complete_pattern	Low Level	('objects',)	objects
map_connect_pixels	Low Level	('objects',)	objects
map_create	Low Level	('objects',)	objects
map_filter_color_block	Low Level	('objects', 'color')	objects
map_filter_color_pass	Low Level	('objects', 'color')	objects
map_find_central_object	Low Level	('objects',)	objects
map_find_missing_patch	Low Level	('objects',)	objects
map_flip_horz	Low Level	('objects',)	objects
map_flip_vert	Low Level	('objects',)	objects
map_grow_pixels	Low Level	('objects',)	objects
map_image	High Level	('image',)	image
map_invert	Low Level	('objects',)	objects
map_make_enclosed_patches	Low Level	('objects',)	objects
map_map_image	Low Level	('objects',)	objects
map_recolor_image	Low Level	('objects',)	objects
map_remove_noise	Low Level	('objects',)	objects
map_repeat	Low Level	('objects',)	objects
map_rotate_180	Low Level	('objects',)	objects
map_rotate_270	Low Level	('objects',)	objects
map_rotate_90	Low Level	('objects',)	objects
map_scale_2x	Low Level	('objects',)	objects
map_scale_3x	Low Level	('objects',)	objects
map_scale_4x	Low Level	('objects',)	objects
map_scale_5x	Low Level	('objects',)	objects
map_scale_half	Low Level	('objects',)	objects
map_scale_quart	Low Level	('objects',)	objects
map_scale_third	Low Level	('objects',)	objects
map_self_scale	Low Level	('objects', 'image')	objects
map_trim	Low Level	('objects',)	objects
pixel_ratio	Low Level	image	number
recolor_image	High Level	('image',)	image
recolor_objects	High Level	('objects',)	objects
remove_noise	High Level	('image',)	image
render_color	High Level	('color',)	image
render_color_sequence	High Level	('objects',)	image
render_grid	Low Level	('grid',)	image
render_number	High Level	('number',)	image
repeat	High Level	('image',)	image
reset_background	Low Level	()	color
rotate_180	Low Level	('image',)	image
rotate_270	Low Level	('image',)	image

Name	Execution	Parameters	Type
rotate_90	Low Level	('image',)	image
scale_2x	Low Level	('image',)	image
scale_3x	Low Level	('image',)	image
scale_4x	Low Level	('image',)	image
scale_5x	Low Level	('image',)	image
scale_half	Low Level	('image',)	image
scale_quart	Low Level	('image',)	image
scale_third	Low Level	('image',)	image
segment_objects	Low Level	('image',)	objects
segment_objects_nd	Low Level	('image',)	objects
self_scale	Low Level	('image', 'image')	image
set_background_color	Low Level	('color',)	color
solve	High Level	('objects',)	image
sort_by_area	Low Level	('objects',)	objects
sort_by_color	Low Level	('objects',)	objects
sort_by_color_frequency	Low Level	('objects',)	objects
sort_by_contents	Low Level	('objects',)	objects
sort_by_number_of_colors	Low Level	('objects',)	objects
sort_by_size	Low Level	('objects',)	objects
tail	Low Level	('objects',)	objects
trim	Low Level	('image',)	image

Appendix B

State transitions of the two different Blocks for the BDT

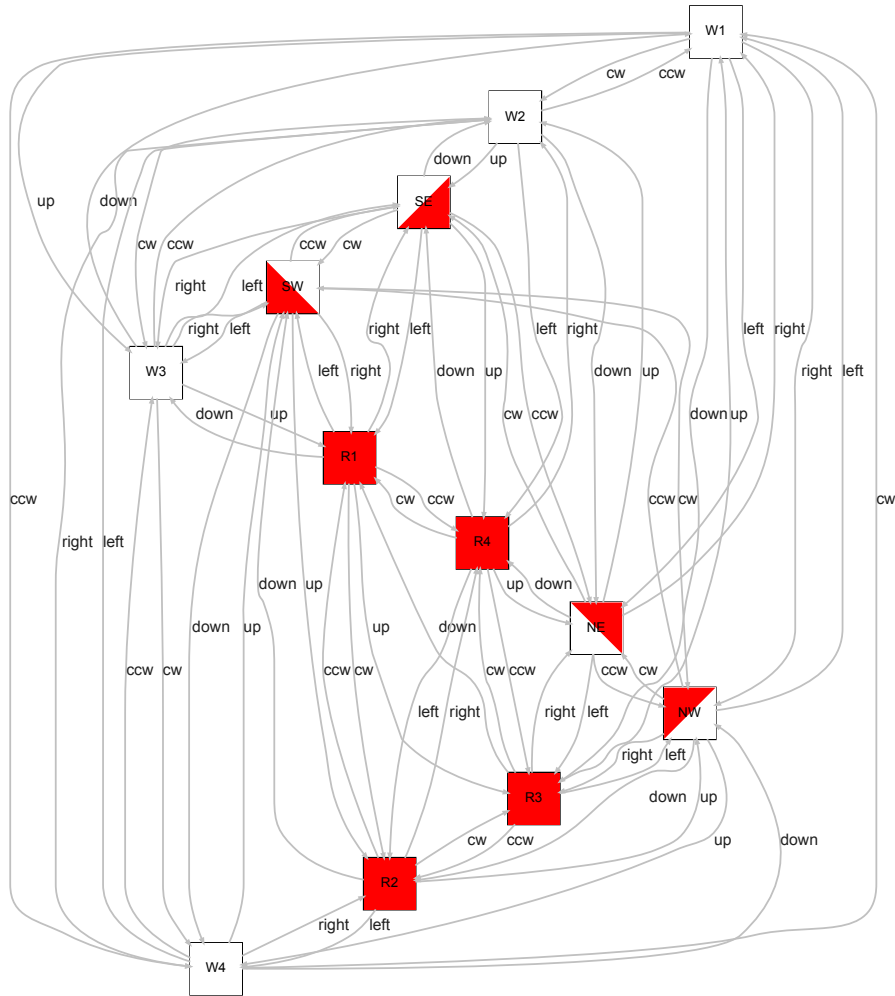


Figure B.1: All the states of the RED block design Chart

