Enumerative Program Synthesis Techniques for the Synthesis of Magic Card Tricks

Robert Brewer

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

Program synthesis is the task of generating programs which fulfil a specification, and popular synthesis techniques can be adapted for the synthesis of magic card tricks. Previous work has used a constraint-based synthesis strategy for this task, so we implement an enumerative synthesis algorithm, which is more effective for other synthesis problems. We develop grammars for the synthesis of 3 different magic card tricks, then apply our implementation to these tricks, and compare our results to those of the constraint-based method. We also explain and implement 5 search heuristics which are used to guide the enumeration of tricks during the search for solutions. We find that the enumerative algorithm is more effective than the constraint-based algorithm for this task. After comparing the results obtained by using each search heuristic, we conclude that Breadth-First Search, %Correct and Adapted Stochastic Search are the most effective heuristics for the synthesis of magic card tricks.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Robert Brewer*)

Acknowledgements

There are a number of people I would like to thank, without whom this project would not have been possible.

To my supervisor, Elizabeth Polgreen, for your invaluable knowledge, expertise, and your willingness to help me at all times. All are very much appreciated.

To my close family and friends, as well the entirety of Edinburgh University Korfball Club, thank you for your continued love and support, without which I would not have made it through the 4 years it took to reach this point. Special thanks to Ally Day for being there whenever I needed to think out loud or vent my frustrations, please enjoy a well-earned Greggs on me.

To the staff at Pleasance Cafe, thank you for putting up with me over the past year. I may well have spent more time in that building than you have, and your hard work is greatly appreciated.

Finally, thank you to my Granny Irene. You may not be there to see me graduate, but I will be thinking of you when I do. Rest in peace.

Table of Contents

1 Introduction **1**

2 Background **2**
2.1 Program Synthesis 2
2.2 Magic Card Tricks as Programs 4

3 Previous Work **5**
3.1 On $\exists \forall \exists!$ Solving: A Case Study on Automated Synthesis of Magic Card Tricks 5
3.2 Comparison of Synthesis Algorithms 6

4 Implementation **7**
4.1 High Level Overview 7
4.2 Basic CEGIS Algorithm 7
4.3 Synthesis Phase 9
4.4 Verification Phase 9
4.5 Other Procedures 9

5 Enumeration Heuristics **10**
5.1 Breadth-First Search 10
5.2 Depth-First Search 10
5.3 %Correct 11
5.4 Stochastic %Correct 11
5.5 Adapted Stochastic Search 12
5.6 Heuristics Not Implemented 12

6 Magic Tricks **14**
6.1 Baby Hummer 14
6.2 Elmsley Shuffles 15
6.3 "Mind Reading" of Cards 16

7 Results and Evaluation **18**
7.1 Baby Hummer 18
7.2 Elmsley Shuffles 20
7.3 "Mind Reading" of Cards 21
7.4 Discussion 23

8 Conclusions **25**

Bibliography **26**

Chapter 1

Introduction

Program synthesis is the task of generating programs which fulfil a specification. In the 2016 paper "On $\exists \forall \exists$! Solving: A Case Study on Automated Synthesis of Magic Card Tricks" [11], Jha et al. explore the use of a constraint-based program synthesis algorithm for the synthesis of magic tricks. While this provides a good initial baseline for the problem, research has demonstrated that enumerative synthesis is more effective in a number of contexts [1]. We hypothesise that enumerative synthesis will perform better for this particular problem, and implement an enumerative solver using 5 different search heuristics. After comparing our implementation to the results they reported for 3 different magic tricks, we find that the enumerative solver generally outperforms the constraint-based algorithm, which we believe to be a result of the improved control over exploration and exploitation that our implementation allows.

In Chapter 2, we provide background information on the problem of program synthesis and its adaptation for magic tricks, and in Chapter 3, we summarise the findings of previous work in these fields. In Chapters 4, 5 and 6, we will describe and justify our enumerative synthesis algorithm, the search heuristics we have implemented, and the magic card tricks we will explore. Finally, in Chapter 7, we analyse the results of our experiments, comparing them to those reported by Jha et al., and then give our conclusions in Chapter 8.

Chapter 2

Background

2.1 Program Synthesis

Program synthesis is the automated process of generating a program written in some programming language which fulfils a user defined specification. One of the first recorded descriptions of program synthesis is attributed to the American mathematician and computer scientist Alonzo Church who in 1957, expressed the problem of synthesising a circuit from mathematical requirements, with program synthesis sometimes referred to as "Church's Problem". Research in the discipline has significantly increased during the 21st century, however the currently available strategies are only capable of generating relatively small programs. Current applications of program synthesis include bit-vector manipulation, data wrangling and reverse engineering of code.

While different synthesis techniques have unique strategies and associated benefits and drawbacks, the core computational problem remains constant and as a result, these techniques can be generalised as Syntax-Guided Synthesis. Syntax-Guided Synthesis (or SyGuS) requires three user inputs: a background theory; a correctness specification for the problem, often given as a formula in first-order logic; and a set of candidate implementations given as a grammar. Given these inputs, SyGuS then seeks to construct a program from the grammar which is consistent with the background theory and meets the correctness specification for all possible inputs. Alur et al. define the problem as the following [11] :

"Given a background theory T, a typed function symbol f, a formula ϕ over the vocabulary of T along with f, and a set L of expressions over the vocabulary of T and of the same type as f, find an expression $e \in L$ such that the formula $\phi[f/e]$ is valid modulo T."

One of the most prevalent SyGuS techniques is Counterexample-Guided Inductive Synthesis, or CEGIS [15] [14]. CEGIS consists of two main steps: the synthesis step, where candidate programs are generated, and the verification step, where candidates are tested against the specification. In the synthesis step, the CEGIS algorithm searches for candidate programs generated by the grammar which satisfy the specification on a set of sample inputs, which may or may not begin empty. There may be several

candidate programs with this property, and which program is selected for verification is determined by some search strategy. Once a candidate program satisfying these criteria has been selected, it is passed to a verification oracle, which checks whether the candidate is consistent with the background theory and correctness specification. If the verifier finds that the candidate is correct, the CEGIS algorithm terminates, and returns this candidate. Otherwise, the verification oracle generates a counterexample - some input on which the candidate program does not meet the correctness specification. This counterexample is passed back to the synthesis step, where it is added to the set of sample inputs and used for the generation of a new candidate.

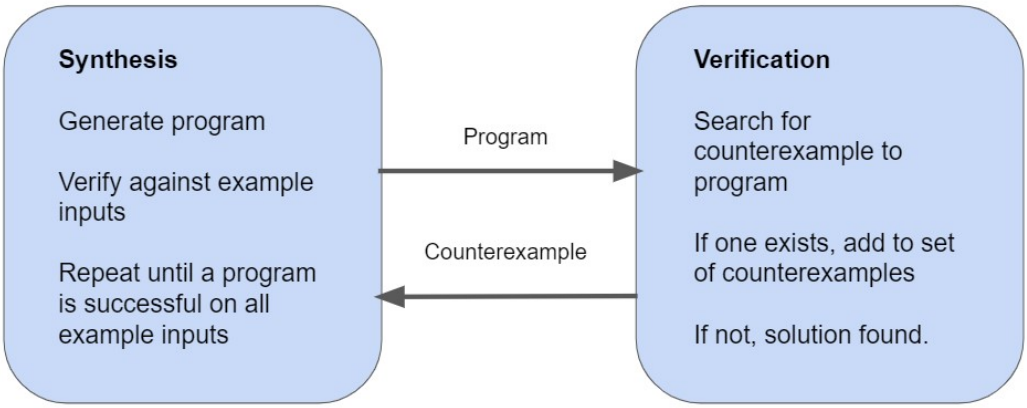

Figure 2.1: A basic CEGIS loop

Satisfiability Modulo Theories (SMT) is the problem of determining whether a formula is satisfiable with respect to a background theory, and SMT solvers, such as Z3 and cvc5, play an important role in a large number of program synthesis algorithms. This occurs primarily during the verification phase, where the correctness specification for a program is written as logical formulae, called constraints, and an SMT solver is used to decide whether a given program satisfies these constraints for all possible inputs.

Constraint-based synthesis algorithms use SMT solvers to a greater extent, employing them for both synthesis and verification. An SMT solver is used during the synthesis stage to generate a program which satisfies the specification constraints for the set of example inputs, and then again in the verification stage to produce counterexample inputs on which a program does not meet the given constraints. Constraint-based synthesis methods are used in some state-of-the-art solvers such as Sketch by Solar-Lezama [15] [14]. Some authors have combined this approach with component-based synthesis, where the basic constituents of a program are placed in a component library, along with a set of specifications explaining the behaviour of each component [8] [10]. Candidate programs are generated by selecting and combining components from the library, allowing the user increased control over the number of instances of each component that can be used.

Another popular strategy used during the synthesis stage of CEGIS is enumerative learning, where a dynamic programming based strategy is used to systematically search through candidates [16]. This begins with the most simple candidates, which are then

stored and used to compose more complex candidates. While a naive enumerative algorithm is effective for simple examples, search heuristics are required to effectively find solutions in more complex search spaces. Assigning a score to previously discovered candidates according to some heuristic informs the search for future candidates. Enumerative search strategies also make great use of the grammar they are provided by using the production rules to extend previously synthesised programs. The majority of contemporary synthesis tools use this approach, including DryadSynth [9], EUSolver [2] and CVC5 [13].

2.2 Magic Card Tricks as Programs

Modern program synthesis techniques can be adapted for solving other problems, such as the synthesis of magic card tricks. In "On $\exists \forall \exists!$ Solving: A Case Study on Automated Synthesis of Magic Card Tricks" [11], Jha et al. adapt the CEGIS algorithm for use in generating magic tricks rather than programs. The similarities between these applications are fairly striking: just as a program is a series of instructions which receives user inputs and returns some desired outputs, a magic card trick can be represented as a sequence of actions which, when carried out, will always result in the magician's desired outcome, regardless of some audience input. For example, in the Face Up/Face Down Mysteries trick described by Jha et al., the audience input is a finite sequence of steps where each step can be either turning over the top two cards, or an audience-directed cut of the deck, and the sequence of actions is the turning over of every other card by the magician. Regardless of which non-deterministic directions were given by the audience, the deterministic outcome will be that half of the cards in the deck are face up, and that half are face down.

The modelling of magic card trick generation as a program synthesis problem is relevant for a number of reasons. The exploration of alternative uses of synthesis techniques could potentially yield results which are informative towards more practical applications and could not be achieved through the study of more typical programs. Additionally, the use of cutting-edge techniques in a context more likely to capture the imagination of children and non-computer scientists leads to unique potential for public engagement opportunities.

Chapter 3

Previous Work

3.1 On $\exists \forall \exists!$ Solving: A Case Study on Automated Synthesis of Magic Card Tricks

In "On $\exists \forall \exists!$ Solving: A Case Study on Automated Synthesis of Magic Card Tricks" [11], Jha et al. use a constraint-based and component-based strategy for the problem of synthesis of magic card tricks from a library of component actions involving non-deterministic choices. They show that this problem can be represented as a logical formula of the form: there exists a composition of actions such that for all non-deterministic choices, there uniquely exists intermediate and final states satisfying a logical specification. While $\exists \forall \exists!$ problems prove challenging for SMT solvers, adapting synthesis techniques for this purpose can generate novel variations of popular card tricks with relative ease. Jha et al. encode the problem as the following:

- Let n be the number of actions in the component library, then add $k \leq n$ copies of an action no-op which does not alter the state. Let k be the upper bound on the length of the magic trick.
- Assume that all tricks have a common initial state s_0
- Let *conn* be the a vector in k dimensions corresponding to a sequence of actions
- Let *choices* be a vector in k dimensions representing the audience input
- Let *states* be a vector in k dimensions where each component s_j is the state after the jth action has been performed
- Let $\phi_{des}\,(conn, choices, s_0, states)$ be a constraint which ensures that the jth component of *states* is obtained by performing the jth action in *conn* on the $j-1$th state in *states* given the jth component of *choices*
- Let ϕ_{states} be the deterministic requirement on the final state for a successful trick
- The problem can be defined as the constraint F_{des} : $\exists conn.\, \forall choices.\, \exists! states.\, \phi_{des}\,(conn, choices, s_0, states) \wedge \phi_{states}$

The large number of choices involved in F_{des} (non-deterministically cutting a 52 card deck has 52 possible outcomes) means that this formula is incredibly challenging to solve. However, using a CEGIS loop to replace *choices* in F_{des} with a subset of all possible choices means that solutions can be generated much more easily. Jha et al. then define a weaker constraint $F_{ver} : \exists conn. \forall choices. \forall states. \phi_{des}\,(conn, choices, s_0, states) \Rightarrow \phi_{states}$.

Since only one possible sequence of states exists for each set of choices, it follows that $F_{des} \Rightarrow F_{ver}$, therefore any solution to F_{des} will be a possible solution to F_{ver}. Candidate solutions were generated by solving F_{des} on only the set of example choices, then counterexamples for these tricks generated by solving $\neg F_{ver}$ were added to the set of example choices. This process was repeated until a candidate was generated for which no counterexamples existed and F_{des} held for all possible choices.

3.2 Comparison of Synthesis Algorithms

There are several areas of the CEGIS algorithm in which advancements are being made, including the search strategy used for the selection of candidate programs. In their 2013 paper "Syntax-Guided Synthesis" [1], Alur et al. implement three such strategies: an enumerative solver following the enumerative synthesis algorithm described in Section 1.1; a constraint-based approach similar to that used by Jha et al. which uses a component library and relies on an SMT solver for both the search for a candidate solution and for the verification of candidates; and a more complex stochastic learning procedure which samples candidates probabilistically from the search space. These strategies were then applied to a variety of benchmarks, ranging from simple toy problems to more complex functions, and the results were analysed. The enumerative solver was found to be comfortably the most successful, followed by the stochastic solver, ahead of the symbolic constraint-based strategy which was the most prone to timing out on each class of benchmark. However, Alur et al. note that the maturity of the solvers was too limited to draw broad conclusions, and more refined implementations of each strategy may yield different results.

Chapter 4

Implementation

4.1 High Level Overview

We implement CEGIS for magic tricks using an enumerative synthesis phase which selects candidate tricks according to some heuristics. These heuristics will be explained in Chapter 5. In order to implement an enumerative CEGIS algorithm, we considered the synthesis and verification phases separately.

In the synthesis phase, a trick is selected from a list of previously discovered tricks and new sequences are composed by adding actions to the sequence according to the context-free grammar. Then the new tricks are compared against the example inputs. Tricks which do not meet the specification on all example inputs are added to the list of tricks and the search continues, while tricks which do are then passed to the verification stage. If at any point it is no longer possible for new tricks to be generated from the existing tricks, the algorithm will terminate and return an empty solution.

In the verification phase, a trick is exhaustively tested against all possible inputs. If it meets the specification for all possible inputs, the algorithm terminates and returns this solution, while if not, a counterexample input for which the trick does not meet the specification is added to the list of examples, the list of tricks is reset, and the synthesis stage begins again with this additional example input. This exhaustive testing approach is different to the typical verification by an SMT solver which is used by Jha et al. [11], and is possible because the input space is finite.

4.2 Basic CEGIS Algorithm

The CEGIS algorithm was implemented as one main function *cegis*(), which makes calls to other procedures *synthesis*(), *verifyOnExamples*() and *verifyOnAll*() representing stages of the CEGIS process. Since the basic CEGIS algorithm does not change for different tricks, *cegis*(), *verifyOnExamples*() and *verifyOnAll*() remain very similar. However, when combined, *synthesis*() (containing the grammar for a particular trick), *performTrick*() and *checkSuccess*() act as the specification for the magic card trick so must be rewritten. While it could be considered undesirable that code must be

rewritten for new applications of the algorithm, modifying these procedures is essentially equivalent to manually writing a specification in first-order logic (e.g. ϕ in Alur et al.'s definition given in Section 2.1), which is the case for other synthesis algorithms.

cegis() begins with an array newTricks containing the empty trick, and loops conditionally until a solution is returned or there are no longer any new tricks to explore. Each new trick is then verified on the example inputs via a call to *verifyOnExamples*(). If a trick is not successful on all example inputs, its score is calculated and it is added to the list tricks. However, tricks which are successful on all example inputs are then passed to *verifyOnAll*(). If this returns true, the trick is returned as a solution. Otherwise, a counterexample input is added to the list examples, and the list of tricks is cleared and the search begins again with this additional example input. Finally, either tricks is empty and newTricks is reset to contain just the empty trick again, or tricks contains tricks to use as the basis for enumeration, in which case a call is made to *synthesis*() and the tricks it selects for exploration are added to newTricks. The pseudocode for this function is given below:

Algorithm 1 Our basic CEGIS algorithm

```
function CEGIS( )
    examples ← [ ]
    tricks ← [ ]
    newTricks ← [ [ ] ]
    while newTricks ≠ [ ] do
        for trick in newTricks do
            results ←verifyOnExamples(trick, examples)
            if all(results) is True then
                cex ←verifyOnAll(trick)
                if cex is None then
                    return trick
                else
                    examples.append(cex)
                    tricks ← [ ]
                    break
                end if
            else
                tricks.append((trick, calcScore(trick, results)))
            end if
            if tricks is empty then
                newTricks ← [ [ ] ]
            else
                newTricks, tricks ←synthesis(tricks)
            end if
        end for
    end while
    return [ ]
end function
```

4.3 Synthesis Phase

synthesis() receives a list of previously discovered tricks sorted by score. For deterministic synthesis heuristics, the best scoring trick is selected, removed from the list, then the trick is extended in each way allowed by the grammar, and these extensions are returned. If for some reason a trick cannot be extended, the next best scoring trick is selected instead and the same process is applied. For stochastic synthesis heuristics, the total score of all the tricks in the list is calculated, and a pseudo-random number between 1 and this total is generated. The range of numbers a trick is associated with is proportional to its score (high scores being good in this case), and the trick associated with the given random number is selected, removed from the list, and explored as previously described.

verifyOnExamples() receives a trick and the set of example inputs, then makes a call to *performTrick*() for each input, returning a list of the results.

Together, *synthesis*() and *verifyOnExamples*() make up the synthesis phase of the algorithm.

4.4 Verification Phase

verifyOnAll() is very similar to *verifyOnExamples*(), but the trick may be performed for the entire input space. If the trick is successful on all possible inputs, it returns None, while if it encounters an input for which the trick is not successful, this counterexample is immediately returned. *verifyOnAll*() is the verification phase of the algorithm and is sufficient because exhaustively testing a trick against the entire finite input space is equivalent to ensuring it meets the $\exists \forall \exists!$ specification according to an SMT solver.

4.5 Other Procedures

performTrick() is a procedure to perform a trick for a given input, returning the deck of cards in its final state.

checkSuccess() is a procedure to receive a final state and assess whether or not it meets the success criteria for that magic card trick.

Chapter 5

Enumeration Heuristics

5.1 Breadth-First Search

Breadth-First Search (BFS) is a basic search algorithm which, after starting at the root of a tree, prioritises exploring all nodes at a particular level before exploring those at greater depths. It can be characterised as adding nodes to a queue in order of discovery and exploring nodes in the order they appear in the queue. One important feature of Breadth-First Search is that even in trees with infinite depth, Breadth-First Search will always find a solution if one exists.

The Breadth-First Search heuristic was implemented as a function *calcScore*() which, given some trick, returns a score which is simply the length of that trick. Each trick is added to a SortedKeyList, which is sorted in ascending order according to the score assigned to each trick, with ties broken by the order of discovery. Given tricks are explored in order of appearance in this list, this algorithm fulfils the properties of Breadth-First Search that all possible tricks with length n will be discovered and explored before any tricks with length n+1 are explored.

calcScore ($trick$, $results$) = len ($trick$)

5.2 Depth-First Search

Depth-First Search (DFS) is another basic search algorithm which, in contrast to Breadth-First Search, prioritises exploring nodes at increasingly large depths until exhaustion before backtracking and choosing a new path. It can be characterised as adding nodes to a stack in order of discovery and exploring nodes in the order they appear in the stack. Unlike Breadth-First Search, Depth-First Search is not guaranteed to find an existing solution in an infinite tree, as it may continue infinitely down a path which never yields a solution.

There are three main traversal methods used in Depth-First Search for binary trees: pre-order, in-order and post-order traversal. Each is differentiated by the order in which they explore a given node N, and its left and right subtrees, L and R. Pre-order traversal

explores these nodes in the order NLR, meaning an ordered list of the nodes visited will be a topological sort, whereas in-order and post-order traversals will have the orders LNR and LRN respectively. Pre-order traversal was deemed the most appropriate for this application due to two main benefits: it is the easiest to implement for non-binary trees, and in the context of program synthesis, simpler and therefore more desirable programs are explored first.

The Depth-First Search heuristic was again implemented as a function *calcScore*(), but with each trick assigned a score equal to its length subtracted from zero. Since the list storing the discovered tricks is once again sorted in ascending order by score, this results in tricks with greater lengths being explored first. Additionally, since a trick is discovered by appending an action to its parent, a trick will never be explored before its parent thus meeting the criteria for a pre-order traversal.

calcScore ($trick$, $results$) $= 0 -$ len ($trick$)

5.3 %Correct

The final deterministic heuristic we implemented was %Correct (%C), which scores tricks based on their success rate on the counterexamples stored during CEGIS. Given both a trick and the results of carrying out this trick for each example input, *calcScore*() returns the number of inputs on which the success criteria is not met. Since the list of discovered but unexplored tricks is sorted in ascending order, this results in tricks with the highest success rate on the example inputs being explored first. As with Depth-First Search, this strategy is not guaranteed to find a solution in an infinite tree, as it may instead eternally search down a high-scoring path in the tree that never actually yields a solution.

calcScore ($trick$, $results$) $=$ map (*checkSuccess*, $results$) .count (False)

One important consideration when developing search strategies is exploration versus exploitation. Exploration is the process of gathering information (literally exploring the search space) while exploitation is using information you have already gathered to pursue a target. A heuristic such as %Correct is very exploitative since every stage of the search is done with the aim of discovering solutions which score as well as possible. While this may result in some exploration as a side-effect, it is never the focus of this strategy and therefore paths which lead to solutions may be overlooked because other paths initially appear more promising.

5.4 Stochastic %Correct

It is often desirable to introduce non-determinism into a search strategy in order to increase exploration. We achieve this by sampling new tricks from a distribution on their scores, rather than always selecting the highest scoring trick. While this may slow down the most effective heuristics in the average-case, as more tricks will be explored during the search, this will significantly improve the worst-case by ensuring

that a solution is always found if one exists, even in an infinite tree, and therefore the worst-case runtime in finding this solution will at least be termination.

Stochastic %Correct (S%C) is an adaptation of the %Correct heuristic with added non-determinism for this purpose. Where previously the tricks to be explored first would receive a low score, *calcScore*() now returns a high score for those more desirable tricks. This allows the *synthesis*() procedure to easily construct a distribution over tricks, where the probability of a trick being explored is equal to its score divided by the sum of all trick scores. Given a score must be non-zero for the associated trick to be explored, when using the Stochastic %Correct heuristic *calcScore*() now returns the number of inputs on which the success criteria is met, plus one.

calcScore ($trick$, $results$) = map (*checkSuccess*, $results$) .count (True) + 1

5.5 Adapted Stochastic Search

In "Syntax-Guided Synthesis" [1], Alur et al. describe a synthesis algorithm called Learning by Stochastic Search where each program e is sampled with probability proportional to Score(e) = exp(-0.5 * C(e)) where C(e) is the number of examples on which e does not satisfy the specification. While Learning by Stochastic Search is more complex than the enumerative search strategies explored in this paper, this scoring strategy was adopted for a new enumeration heuristic, Adapted Stochastic Search (ASS), where the score assigned to a trick is Score(e) rounded to two decimal places and multiplied by 100 to give an integer value. It is unnecessary to consider this scoring function in deterministic environments as it would behave identically to %Correct, since the ordering of scores is preserved.

calcScore ($trick$, $results$) = round (exp (map (*checkSuccess*, $results$) .count (False)), 2) $\times$ 100

This heuristic balances the exploitation and exploration achieved by the previous two heuristics. Just like in Stochastic %Correct, tricks are still sampled from a probability distribution in order to achieve greater exploration of the search space, but here the distribution is more heavily weighted towards tricks which would have scored well on the deterministic %Correct in order to increase exploitation. This should result in a search which can find solutions from paths which do not initially appear promising, while still prioritising paths which look likely to lead to solutions.

5.6 Heuristics Not Implemented

Euphony is a synthesis tool which generates programs from a probabilistic context-free grammar (PCFG) where each production rule is labelled with a probability [12]. We felt that this was not well suited to this problem as the training of the PCFG would require an existing set of solutions to each problem, and the most interesting solutions would likely combine a variety of actions, so bias towards certain actions would make these solutions less probable.

Another common component of contemporary synthesis algorithms is the neural network. DeepCoder uses a neural network to predict the probabilities of programs and uses this to guide enumeration [3], while DreamCoder uses a neural network trained on example programs as well as those it synthesises [7]. Additionally, recurrent neural networks (neural networks where connections between nodes may create cycles) are used in RobustFill to generate programs from given inputs [6], and in Concord to generate probability distributions over future actions from partially complete programs [4]. While effective when used in these algorithms, neural networks are complex and require significant training before use. We chose to focus on the 5 heuristics we have described as these offer a suitable baseline for the problem, and suggest these neural network-based strategies as possible future work.

Chapter 6

Magic Tricks

6.1 Baby Hummer

The Baby Hummer is a magic card trick credited to Charles Hudson as a variation on an existing trick created by 20th century magician Bob Hummer. The trick begins with the magician asking the audience to secretly select and remember a card from a deck of 4, with the chosen card then placed on the bottom of the deck. After the magician performs a sequence of actions, they reveal that the audience's chosen card is now facing the opposite direction to the other 3 cards in the deck. The actions available to the magician are:

- *turntop*: The top card in the deck is picked up, turned over, and placed back on top of the deck
- *turntop2*: The top two cards in the deck are picked up, turned over together such that their order is reversed, and placed back on top of the deck
- *toptobottom*: The top card in the deck is moved to the bottom
- *top2tobottom*: The top two cards in the deck are moved to the bottom, preserving the order they appeared in beforehand
- *cut*: The magician cuts the deck in two at a point directed by the audience and places the bottom part on top of the top part.

The original sequence of these actions is: *toptobottom, turntop, cut, turntop2, cut, turntop2, cut, turntop2, turntop, top2tobottom, turntop*.

In the Baby Hummer trick, the non-deterministic audience input is the position at which the deck is cut. The card selected by the audience is not considered an audience input for this trick as it is always placed at the bottom of the deck at the beginning of the trick, and it is the position of each card at the beginning of the action sequence that determines whether it will be face-up or face-down at the end of the trick. For a trick where a 4 card deck is cut 3 times, as in the original sequence, there will be 64 possible audience inputs, and a successful sequence of actions must result in the desired result (the audience selected card facing the opposite direction to the other 3 cards) for all

of these inputs. Additionally, an interesting sequence of actions must contain at least one audience directed cut of the deck, and it must mix card-turning and card-moving actions. The latter constraint was ensured by the grammar used for the enumeration of tricks, which was as follows:

$$
\begin{aligned}
S &::= \textit{turntop}\ S_{\text{turntop}} \mid \textit{turntop2}\ S_{\text{turntop2}} \mid \textit{toptobottom}\ S_{\text{toptobottom}} \mid \\
&\quad \textit{top2tobottom}\ S_{\text{top2tobottom}} \mid \textit{cut}\ S_{\text{cut}} \mid \varepsilon \\
S_{\text{turntop}} &::= \textit{turntop2}\ S_{\text{turntop2}} \mid \textit{toptobottom}\ S_{\text{toptobottom}} \mid \\
&\quad \textit{top2tobottom}\ S_{\text{top2tobottom}} \mid \textit{cut}\ S_{\text{cut}} \mid \varepsilon \\
S_{\text{turntop2}} &::= \textit{turntop}\ S_{\text{turntop}} \mid \textit{toptobottom}\ S_{\text{toptobottom}} \mid \\
&\quad \textit{top2tobottom}\ S_{\text{top2tobottom}} \mid \textit{cut}\ S_{\text{cut}} \mid \varepsilon \\
S_{\text{toptobottom}} &::= \textit{turntop}\ S_{\text{turntop}} \mid \textit{turntop2}\ S_{\text{turntop2}} \mid \textit{top2tobottom}\ S_{\text{top2tobottom}} \mid \\
&\quad \varepsilon \\
S_{\text{top2tobottom}} &::= \textit{turntop}\ S_{\text{turntop}} \mid \textit{turntop2}\ S_{\text{turntop2}} \mid \varepsilon \\
S_{\text{cut}} &::= \textit{turntop}\ S_{\text{turntop}} \mid \textit{turntop2}\ S_{\text{turntop2}} \mid \varepsilon
\end{aligned}
$$

This grammar seeks to prune the search space by ignoring tricks which are functionally equivalent to some other shorter trick. For example, the *toptobottom* action should never be performed twice in a row, as this would be equivalent to one *top2tobottom* action, while the *turntop* action should never occur twice in a row as this is equivalent to performing no actions at all. Similarly, the actions *toptobottom* and cut should never occur next to each other, either way round, as this would result in an unknown non-deterministic ordering of cards which can be achieved through just one *cut* action. Pruning the search space in this way means the set of tricks discovered by this algorithm will not be as large as if any two actions could follow each other, therefore decreasing the runtime, while simultaneously ensuring that for any given trick, an equivalent trick will be generated. This is equivalent to the constraints implemented by Jha et al. in "On $\exists \forall \exists!$ Solving: A Case Study on Automated Synthesis of Magic Card Tricks" [11], which mix card-turning and cutting actions. Additionally, the number of instances of each component action in a trick was limited to a maximum of 4 to mirror the methods used by Jha et al. (who added 4 instances of each action to their component library) and to force the synthesised tricks to use a variety of actions, and therefore produce more interesting sequences. Additionally, this eased the implementation of the non-deterministic audience input by allowing the example inputs to be modelled as lists of exactly 4 integers without fear of synthesising a trick which contained more than 4 *cut* actions and therefore required more than 4 inputs.

The specification for this trick is that after the audience-selected card is placed at the bottom of the deck and the sequence of actions is carried out, the audience card will be facing in the opposite direction to the other 3 cards in the deck.

6.2 Elmsley Shuffles

Alex Elmsley was a Scottish computer scientist and magician known for his development of mathematical card tricks and work on the mathematics of card shuffling. One such shuffle, known as the Elmsley shuffle, consists of splitting the deck down the middle into two distinct piles, then combining these in such a way that the new deck contains

cards alternately from each pile. The Elmsley shuffle has two variations: in-shuffle and out-shuffle, abbreviated to *inS* and *outS*. During an out-shuffle, the two piles are combined in such a way that the card that was originally on top of the deck returns to that position, whereas for an in-shuffle, the original top card becomes the second card of the new deck. For this magic card trick, the performer will ask the audience to select a card from the deck, reveal it to them but not himself, then place it on top of the deck. The audience will then direct the magician in performing a series of in-shuffles and out-shuffles, and at the end of the trick, the magician is able to find the card in its new position in the deck and reveal it to the audience.

In this trick, the non-deterministic audience input is the sequence of in-shuffles and out-shuffles, and the deterministic result is the position of the audience-selected card given this sequence of actions. However, Jha et al. frame this problem as synthesising a sequence of actions given a desired final position of the audience card [11]. In this case, there is no audience input to model, and a sequence of actions will either always or never result in this final state, meaning synthesis of these sequences does not require the use of counterexamples and can be executed with plain enumeration. Jha et al. make the claim that "more interesting shuffles will not be possible through enumeration", but do not justify this claim.

Since there are only two actions that can possibly be added to an existing trick, and these should not depend on the previous action, the grammar for this trick is simple:

$$S \quad ::= \quad inS\ S \mid outS\ S \mid \varepsilon$$

The specification for this trick is that after the the audience-selected card is placed at the top of the deck and the sequence of actions is carried out, the audience card will be in the position specified by the input to the synthesis.

6.3 "Mind Reading" of Cards

In this trick, the magician presents the audience with a deck of 8 cards, which they are allowed to randomly cut, after which 3 audience members take turns selecting the top card in the pile, hiding their card from the magician. In order to determine which cards the audience members have chosen, the magician asks those with red cards to stand up. With this knowledge, the magician can use their knowledge of the prearranged deck to determine exactly which cards each audience member has picked. This is possible because the deck has been arranged in such a way that any cyclic subsequence with length 3 is unique. For example, the standard colour sequence for this trick is *rrrbbbrb*, where R and B are red and black cards respectively, resulting in 8 unique subsequences: *rrr*, *rrb*, *rbb*, *bbb*, *bbr*, *brb*, *rbr* and *brr*. It is possible to extend this trick by involving 4 or 5 audience members, allowing the number of cards to increase to 16 or 32, since there are more available subsequences with these lengths.

Synthesis for "Mind Reading" of Cards consists of constructing new ways to arrange the deck of cards. Solutions to this problem will be de Bruijn sequences, which have diverse applications including in functional magnetic resonance imaging [5]. The non-deterministic audience input will be the cut of the deck at the beginning of the trick.

The trick is modelled in this way since it is possible for a string which is not itself a de Bruijn sequence to appear as a subsequence in a correct solution (eg. *rrrbbbrb* is a de Bruijn sequence, but the substring *rrrbbbr* is not), and it is important that these subsequences are not prematurely dismissed. The grammar for this trick is as follows:

$$S \quad ::= \quad r\,S \mid b\,S \mid \varepsilon$$

The specification for this trick is that the sequence generated should contain 2^n cards such that all cyclic subsequences with length n are unique, where n is the number of audience members selecting cards.

Chapter 7

Results and Evaluation

7.1 Baby Hummer

We began our investigation by performing a search for the first discovered solution using each heuristic. This resulted in trivial solutions not involving cut actions, so it was decided to add more constraints on solutions. We added the constraints that solutions should contain at least two cut actions, and that these should not appear at the end of the trick as only card turning actions actually change the state relative to the success criteria. The search was then repeated using these additional constraints and the times for both experiments are reported in the table below. Synthesis using stochastic heuristics was carried out using 10 randomly generated seeds, and the mean time and standard deviation are reported.

Heuristic	First Solution	First Interesting Solution
BFS	0.001	1.338
DFS	0.048	188.954
%C	0.001	37.124
S%C	0.0012 ± 0.0004	42.1438 ± 75.8887
ASS	0.0014 ± 0.0007	9.0601 ± 8.6477

Table 7.1: Time to find solution (seconds) for Baby Hummer using our search heuristics

Comparing the performance of the heuristics when searching only for the first solution yields fairly uninteresting results, with the algorithm terminating in 0.001 seconds for 4 of the 5 heuristics, while Depth-First Search takes significantly longer, at 0.048 seconds. This occurs because the 4 heuristics which find solutions in 0.001 seconds return very short sequences. Rather than encountering a short solution early on, as a result of the pre-order traversal and order of operations in our grammar, DFS begins by searching through sequences which start with exclusively card-turning rather than card-moving actions. This area of the search space is much more sparse with solutions so it takes longer to find a successful sequence.

After the additional constraints were added to find more interesting solutions, we also

found more interesting results. Depth-First Search was once again the slowest method for the reasons explained above, however Breadth-First Search was the fastest method, beating all 3 of the more complex heuristics intended to target areas of the search space with more solutions. This could possibly be explained by it exploring tricks which involve cut actions earlier in the process, while the other heuristics may instead opt to explore sequences involving only deterministic actions which are therefore more predictable in their success. An alternative hypothesis could be that the correlation between the success of a trick on the example inputs and the success of a future trick on all inputs is fairly weak, although analysis of the results from the other three heuristics does not necessarily support this. The %Correct and Stochastic %Correct heuristics prioritise exploitation and exploration respectively, and yet both were slower than the Adapted Stochastic Search method, suggesting the latter finds a good balance between the two. The fact that ASS fairs better than S%C indicates that there is value in prioritising the exploration of tricks which perform well on the set of examples. ASS outperforming %C proves that, for this specific search space, increased exploration is more effective than pure exploitation.

In their paper "On $\exists \forall \exists!$ Solving: A Case Study on Automated Synthesis of Magic Card Tricks" [11], Jha et al. report the times for synthesis of 6 tricks, as shown below:

	Action Sequence	Time
Trick 1	*toptobottom, turntop, cut, cut, turntop, toptobottom, turntop2, toptobottom, turntop, toptobottom*	12m 23sec
Trick 2	*turntop2, turntop, toptobottom, toptobottom, cut, turntop2, toptobottom, toptobottom, turntop, cut, turntop2, turntop2*	12m 04sec
Trick 3	*turntop2, toptobottom, toptobottom, turntop, cut, toptobottom, cut, turntop2, turntop2, cut, turntop, turntop*	10m 30sec
Trick 4	*toptobottom, turntop, cut, toptobottom, toptobottom, turntop2, cut, cut, turntop, toptobottom, turntop, turntop2*	5m 43sec
Trick 5	*toptobottom, turntop, cut, toptobottom, toptobottom, cut, turntop2, cut, turntop, toptobottom, turntop, turntop2*	11m 11sec
Trick 6	*toptobottom, toptobottom, toptobottom, turntop, cut, top2tobottom, cut, turntop2, turntop2, cut, turntop, turntop*	11m 39sec

The trick sequences generated by Jha et al. were converted to equivalent sequences which could be generated from our grammar, and then we found the time taken to discover these solutions using Breadth-First Search. The times taken ranged from 0.357 seconds to 89.318 seconds, meaning in all instances, the enumerative algorithm with the BFS heuristic was significantly faster than the constraint-based method.

Finally, we report the number of distinct solutions discovered by our algorithm in 10 minutes for each heuristic. This experiment was repeated 10 times for the stochastic heuristics with the same random seeds as above, so in these cases the mean number and standard deviation are reported:

Heuristic	Number of Solutions
BFS	30540
DFS	22855
%C	17123
S%C	3386.2 ± 1311.3
ASS	4373.8 ± 1299.8

Table 7.2: Number of solutions for Baby Hummer found in 10 minutes using our search heuristics

While Breadth-First Search gives the fastest time to find one successful sequence, Stochastic %Correct and Adapted Stochastic Search find more solutions within 10 minutes. It may be the case that solutions occur more sparsely at higher depths in this search space meaning BFS finds fewer solutions when it reaches these depths. As before, ASS is more effective at this task than the more exploitative %Correct and the more exploratory Stochastic %Correct so appears to find a good balance between the two. The fact that %Correct is clearly the least successful heuristic here supports our earlier hypothesis that success of a trick on the example inputs is not a strong indicator that tricks enumerated from this will be successful on all inputs, which would explain why pure exploitation of this heuristic is not effective here. Through the combined results of these two experiments, we would conclude that BFS and ASS are the most effective heuristics for enumerative synthesis of Baby Hummer tricks.

We cannot draw direct comparisons with Jha et al.'s implementation regarding the number of solutions found in a limited time as they do not provide this information. However, we hypothesise that it would be less effective at searching for multiple solutions as well. This is because the only way to search for multiple solutions using constraint-based synthesis is to add constraints which explicitly prohibit previously discovered solutions, and SMT solvers generally take longer to solve problems with more constraints. This means the time to find each additional solution will often be greater than the time taken to find the previous one. This is in contrast to our enumerative synthesis algorithm which can simply continue its exploration of the search space after finding a solution.

7.2 Elmsley Shuffles

Since the Elmsley Shuffles trick involves some user input between 0 and 7 to use as the final position of the audience selected card, each heuristic was tested for every possible input position, in addition to the repeated experiments for stochastic heuristics. A search for the first discovered solution produced identical results for each heuristic: the time taken was always 0.000 seconds (to 3 decimal places). As a result it is impossible to draw any meaningful conclusions from this data.

In their paper, Jha et al. report times for the synthesis of 4 specific tricks as follows [11]:

	Action Sequence	Input	Time
Trick 1	*inS,inS,outS*	6	6m 42sec
Trick 2	*inS,outS,outS,outS,inS,outS*	6	8m 52sec
Trick 3	*inS,outS,inS*	5	5m 18sec
Trick 4	*inS,outS,outS,outS,outS,inS*	5	6m 08sec

Each heuristic was applied to each of these tricks, and the times for synthesis of each trick are displayed in the table below. For the stochastic heuristics for which experiments were repeated, the mean time and standard deviation are reported.

Heuristic	Trick 1	Trick 2	Trick 3	Trick 4
BFS	<0.001	0.001	<0.001	0.001
DFS	<0.001	0.009	0.005	0.010
%C	<0.001	0.001	<0.001	0.001
S%C	<0.001 ± 0.000	0.002 ± 0.003	<0.001 ± 0.000	0.001 ± 0.000
ASS	<0.001 ± 0.000	0.002 ± 0.001	<0.001 ± 0.000	0.002 ± 0.000

Table 7.3: Time to find solution (seconds) for Elmsley Shuffles using our search heuristics

The most obvious conclusion that can be drawn through comparison of the enumeration heuristics is that Depth-First Search was the worst performing. However, it is not necessarily true that the method which finds these specific solutions last is worse in the general case, especially given DFS performs as well as the other 4 heuristics when simply searching for one solution. The other 4 heuristics performed similarly in the search for specific solutions, never with a worse average time than 0.002 seconds. This led us to conclude that the search for a trick without non-deterministic audience input was too easy for the enumerative strategies to make any meaningful comparisons between the heuristics.

Overall, our enumerative synthesis implementation never takes longer than 0.010 seconds when searching for these 4 tricks, clearly outperforming the constraint-based method implemented by Jha et al. which had a minimum time of 5m 18sec. While it is difficult to explain such a vast difference in results without access to their implementation, we hypothesise that this is a result of the SMT solver Jha et al. use to generate candidate solutions taking significantly longer than the time in which sequences can be constructed by enumeration. While Jha et al. state that more interesting shuffles will not be possible through enumeration, they do not substantiate this claim. Given our implementation generates the same sequences they provide in a mere fraction of the time, and can exhaustively explore the search space, we would dispute the validity of this claim.

7.3 “Mind Reading” of Cards

We tested our implementations of enumerative synthesis against the variants of this trick involving both 3 and 5 audience members, which require decks of 8 and 32 cards

respectively. The results of this are displayed in the following table.

Heuristic	3 Audience Members	5 Audience Members
BFS	0.017	TIMEOUT
DFS	0.002	TIMEOUT
%C	0.013	24.730
S%C	0.006 ± 0.006	TIMEOUT
ASS	0.006 ± 0.005	678.2 ± 385.3

Table 7.4: Time to find solution (seconds) for "Mind Reading" of Cards using our search heuristics

Depth-First Search is the most effective heuristic for 3 audience members, but reaches the 30-minute timeout for the variant with 5 audience members. This makes a lot of sense: solutions have a specified length and DFS is the fastest to reach this length whereas other heuristics will build up more slowly, however for the variant requiring a longer solution, as a result of the pre-order traversal, DFS will begin by constructing a sequence of 32 red cards and then spend a lot of time backtracking before reaching a shorter sequence which could possibly be part of a valid solution. On the other hand, BFS is the least effective heuristic for 3 audience members and also times out for 5 audience members, because it insists on discovering all possible sequences of n-1 cards before discovering any with the specified length n. While %Correct is the next least effective for 3 audience members, it outperforms the other 4 heuristics by a significant margin for 5 audience members, terminating in under 25 seconds. The Stochastic %Correct and Adapted Stochastic Search heuristics resulted in similar times for 3 audience members, however ASS was more effective for 5 audience members with a mean time of 11m 18sec whereas S%C timed out after 30 minutes with every random seed. These results are consistent with the relationship between the success of a trick on the set of example inputs and on the entire input space, which is much stronger than for the Baby Hummer trick, as well as the aforementioned build up towards sequences with the required length. The most efficient of these search methods is the one most reliant on exploitation of previous results as this will quickly identify search paths with high potential for solutions, while the least effective is the heuristic most weighted towards exploration which will continue to explore tricks at shorter lengths and with low potential even when a promising path has been discovered.

Jha et al. provide times for 2 trick sequences discovered by their implementation [11]:

	Action Sequence	Time
Trick 1	*RRRRRBRBRRBRRRBBBBBRBBBRRBBRBRBB*	8m 34sec
Trick 2	*RRRRRBRRBRBBRRBBBBBRRRBBRBBBBRBRB*	9m 18sec

Attempts to discover these tricks using our implementation resulted in reaching the 10 minute timeout in all cases. In order to still make comparisons, we report instead the number of solutions found by our algorithm within these 10 minutes:

Heuristic	Number of Solutions
BFS	0
DFS	0
%C	3052
S%C	0 ± 0
ASS	2.1 ± 1.5

Table 7.5: Number of solutions for "Mind Reading" of Cards found in 10 minutes using our search heuristics

These results are once again consistent with the conclusions drawn from synthesising one trick with 5 audience members: %Correct is comfortably the most successful method, significantly outperforming Jha et al.'s constraint-based method; the Adapted Stochastic Search heuristic performs fairly comparably to the constraint-based method; and the other 3 heuristics for our enumerative strategy all fail to find solutions inside 10 minutes.

7.4 Discussion

Our results suggest that our enumerative algorithm is at least as effective for the synthesis of magic tricks as the constraint-based method. The enumerative algorithm was significantly more successful for the Baby Hummer and Elmsley Shuffles tricks, and also outperformed the constraint-based method for the "Mind Reading" of Cards trick with 5 audience members when the %Correct heuristic was used.

For our experiments involving the Baby Hummer trick, Breadth-First Search was the most effective heuristic, followed by Adapted Stochastic Search. %Correct and Stochastic %Correct perform somewhat similarly, while Depth-First Search is the least effective by a fair margin. For "Mind Reading" of Cards, %Correct performed the best, followed by Adapted Stochastic Search. None of the other three heuristics found a solution inside half an hour. The Elmsley Shuffles trick did not lead to conclusive results, although Depth-First Search seemed to be the least effective strategy once again. We are able to provide reasonable hypotheses for the relative performance of each heuristic on each trick given our understanding of the problem.

These results lead us to conclude that the most effective heuristic for each trick depends on the unique nature of the problem, and as a result no heuristic can be definitively considered the most effective for the synthesis of magic tricks. While %Correct is the most successful for the "Mind Reading" of Cards trick where pure exploitation of a trick's performance on example inputs was effective, DFS and ASS outperformed it on the Baby Hummer trick where an increased emphasis on exploration was preferable. One notable finding was that Stochastic %Correct almost never performed better than BFS, %C or ASS, indicating that it is outclassed regardless of a problem's position with respect to exploration and exploitation. Depth-First Search also proved ineffective in the majority of circumstances. In order to draw further conclusions on the effectiveness of the Breadth-First Search, %Correct and Adapted Stochastic Search heuristics for the

synthesis of magic card tricks, it would be necessary to perform experiments on a larger number and variety of problems.

Chapter 8

Conclusions

We have shown that our CEGIS implementation using enumerative synthesis with exhaustive verification is able to outperform the constraint-based implementation by Jha et al., with the fastest search heuristic completing the search for a solution close to or more than 20 times faster for each magic card trick. This is consistent with previous work on program synthesis, as described in Section 3.2, where enumerative synthesis has been found to be the most effective synthesis strategy across a variety of applications [1].

Based on the work carried out, we have identified Breadth-First Search, %Correct and Adapted Stochastic Search as the heuristics with the most potential for general success. Future work should apply these heuristics to a greater selection of magic tricks in order to draw further comparisons across a wider variety of problems. This could also involve attempts to find or develop algorithms that combine the strengths of these heuristics, for example by focusing on exploration of the search space during the early synthesis stages before prioritising exploitation after a certain portion of the space has been explored. Finally, future work could also involve experiments using the full Learning by Stochastic Search algorithm developed by Alur et al. which inspired our Adapted Stochastic Search heuristic [1], or combining our enumerative synthesis algorithm with constraint-based verification using an SMT solver and comparing the results to those achieved using our exhaustive verification method.

Bibliography

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013.

[2] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, March 2017.

[3] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *ICLR (Poster)*. OpenReview.net, 2017.

[4] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. Program synthesis using deduction-guided reinforcement learning. In *CAV (2)*, volume 12225 of *Lecture Notes in Computer Science*, pages 587–610. Springer, 2020.

[5] Ching-Shui Cheng, Ming-Hung Kao, and Federick Kin Hing Phoa. Optimal and efficient designs for functional brain imaging experiments. *Journal of Statistical Planning and Inference*, 181:71–80, 2017.

[6] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. *arXiv preprint arXiv:1703.07469*, 2017.

[7] Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Lucas Morales, Luke B. Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *PLDI*, pages 835–850. ACM, 2021.

[8] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 62–73, New York, NY, USA, 2011. Association for Computing Machinery.

[9] Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. Reconciling enumerative and deductive program synthesis. In *PLDI*, pages 1159–1174. ACM, 2020.

[10] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, page 215–224, New York, NY, USA, 2010. Association for Computing Machinery.

[11] Susmit Jha, Vasumathi Raman, and Sanjit A. Seshia. On $\exists \forall \exists!$ solving: A case study on automated synthesis of magic card tricks. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, FMCAD '16, page 81–84, Austin, Texas, 2016. FMCAD Inc.

[12] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In *PLDI*, pages 436–449. ACM, 2018.

[13] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark W. Barrett, and Cesare Tinelli. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In *CAV (2)*, volume 11562 of *Lecture Notes in Computer Science*, pages 74–83. Springer, 2019.

[14] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, USA, 2008. AAI3353225.

[15] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 404–415, New York, NY, USA, 2006. Association for Computing Machinery.

[16] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. Transit: Specifying protocols with concolic snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 287–296, New York, NY, USA, 2013. Association for Computing Machinery.