NeuroSketcher: Synthesizing Non-Differentiable Programs with Non-Differentiable Loss Functions

by

Thirupathi Reddy Emireddy

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Programmatic hypotheses offer valuable properties, such as generalizability and interpretability. However, such hypotheses can be elusive, as one finds them by searching over large spaces of programs. Recent work showed that neural networks can be used to guide the search in the programmatic space by filling in the "holes" of the program sketches considered in the search. Despite its promising results, this approach requires that the synthesized programs and the loss function be differentiable, which can severely limit its use in practice. In this dissertation, we overcome this weakness with a two-step search. The first search uses an abstraction of the original language where non-differentiable operations of the language are replaced with neural networks, thus resulting in a language of differentiable programs. The second search, which is not neural-guided, completes the promising sketches generated in the first search by searching in the original language space while optimizing for any loss function, including non-differentiable ones. We use our approach to synthesize programmatic heuristic functions for two permutation sorting problems: the Pancake and the Topspin puzzles. Our method discovered heuristic functions for the Pancake and Topspin puzzles that outperform some of the best known heuristics for these domains. Due to their programmatic representation, we can prove that search algorithms using our heuristic functions are guaranteed to find bounded-suboptimal solutions.

Preface

The author of this dissertation, in partnership with Levi Lelis, has produced original research. This research is currently undergoing review for potential publication. Because of the collaborative aspect of this endeavor, the term "we" is utilized in this document. Nevertheless, I bear full responsibility for any technical inaccuracies or issues in presentation.

Thirupathi Reddy Emireddy
September, 2023

To my spiritual master, parents, and teachers.

Acknowledgements

I extend my heartfelt gratitude to my supervisor, Dr. Levi Lelis. This journey would not have been as smooth without his constant support and enthusiasm. I hold deep respect for his tolerance of my shortcomings, and I appreciate how he has filled those gaps with his experience and knowledge. From his guidance, I've endeavored to adopt a never-give-up attitude that will serve me well in my future endeavors, whatever they may be. Overall, my time with my supervisor has been a valuable learning experience.

I also express my gratitude to my friends and research colleagues, listed in no particular order: Saqib Ameen, Justin Stevens, Lucas N. Ferreira, Rubens O. Moraes, Tales Carvalho, Kenneth Tjhia, Dr. Elham Parhizkar, Zaheen Farraz Ahmed, Mahdi Alikhasi, Zahra Bashir, Quazi Asif Sadmine, Spyros Orfanos, and Mahdieh Mallahnezhad. Each of you has played a part in my journey, enriching my experience in various ways.

Lastly, I am indebted to my parents and siblings for their unwavering support and patience. To the University of Alberta, I owe a tremendous debt of gratitude for accepting me as a graduate student. This acceptance has significantly altered my life's trajectory and has given me a profound understanding of life's true essence.

Contents

Abstract ii

Preface iii

Acknowledgements v

List of Tables viii

List of Figures ix

List of Algorithms x

1 Introduction 1

1.1 Problem Formulation 2

1.2 Contributions 4

2 Background 6

2.1 Heuristic Search 6

2.2 Iterative Deepening A* (IDA*) 7

2.3 Problem Domains: Permutation Sorting Puzzles 9

2.4 Program Synthesis 10

2.4.1 Bottom-Up Search (Bus) 10

3 Related Work 13

3.1 Symbolic Approaches 13

3.2 Neurosymbolic Approach . . . 13
3.3 Other Approaches . . . 14

4 NeuroSketcher 16
4.1 Neural Language Abstractions . . . 16
4.2 Using Neural Language Abstractions . . . 17
4.3 NeuroSketcher Algorithm . . . 18

5 Empirical Results 20
5.1 Experiment Details . . . 20
5.1.1 Domain-Specific Language and Its Abstraction . . . 20
5.1.2 Loss Functions . . . 21
5.1.3 Baselines . . . 21
5.1.4 Training and Test Instances . . . 22
5.2 Discussion . . . 23
5.2.1 Analysis of h^{AG} for the Pancake Puzzle . . . 26
5.2.2 Analysis of h^{OG} for the Topspin Puzzle . . . 27

6 Conclusions 29

References 30

Appendix 34
A LSTM Implementation Details . . . 34
B Near with Multiple Seeds . . . 34
C Study on the Input Encoding for Neural Models . . . 34

List of Tables

2.1 Illustration of Bottom-Up Search Synthesis 12

List of Figures

1.1 Program Synthesis Components 3

1.2 Domain specific language (left) and the abstract syntax tree for `sum(map(f(x): x^2, v))` (right). 4

2.1 5-Pancake puzzle (left) and the (10, 4)-Topspin puzzle (right). The state on the left shows the goal for both domains, while the state on the right shows a random state. 9

2.2 DSL and AST for `((1+2)+3)`, which produces the output `6`. 10

5.1 Language for permutation sorting problems. 21

5.2 Evaluation of the different heuristic functions on Pancake and Topspin puzzles of different sizes. The y-axis shows the number of problems solved, while the x-axis shows the number of nodes expanded per instance. 24

5.3 FeedForward (NN) and LSTM Study 25

6.1 Study with multiple seeds for NEAR. 35

6.2 Data Representation Study 35

List of Algorithms

1 IDA* 8
2 Bottom-Up Search (Bus) 11
3 NeuroSketcher 19

Chapter 1

Introduction

There has been a growing interest in using programmatic representations of hypotheses in machine learning (Ellis, Ritchie, Solar-Lezama, & Tenenbaum, 2018; Valkov, Chaudhari, Srivastava, Sutton, & Chaudhuri, 2018; Young, Bastani, & Naik, 2019). Such interest is justified since programmatic representations allow the system designer to inject a strong inductive bias through the domain-specific language used to define the space of hypotheses. Moreover, depending on the language used, the learned hypotheses are easier to understand and verify (Bastani, Pu, & Solar-Lezama, 2018).

What limits the use of programmatic hypotheses in practice is that they are often difficult to derive because they require one to search in large spaces of programs. A common approach found in the literature is to use a self-supervised approach in which the learning system exploits the structure of the language to generate training data (Balog, Gaunt, Brockschmidt, Nowozin, & Tarlow, 2016; Odena, Shi, Bieber, Singh, Sutton, & Dai, 2021; Barke, Peleg, & Polikarpova, 2020; Ellis, Wong, Nye, Sablé-Meyer, Cary, Morales, Hewitt, Solar-Lezama, & Tenenbaum, 2020; Ameen & Lelis, 2023). For example, in program synthesis, one needs to find a program that maps a set of inputs to the correct outputs. One can sample programs from the language and generate a set of input values that, when given to the sampled programs, form a set of training problems. These problems are used to learn a function to guide the search in the programmatic space.

This self-supervised approach is feasible only when it is possible to provide the problem specification as input to the guiding function. For example, in the case of program synthesis, the set of input-output pairs is given as input to the guiding function, which is trained to be helpful for a range of different problems (different input-output sets). However, it is not clear how to use this approach to find a program that optimizes loss functions in general. Shah et al. (2020) introduced an alternative approach to guide the search in the programmatic space for *differentiable loss functions*. Instead of training a guiding function ahead of time, NEAR trains one model for

each sketch (Solar-Lezama, 2009) considered in the search. A sketch is an incomplete program, e.g., `sum(map(?))`, where the `?` is a "hole" representing what needs to be finished. NEAR fills holes in programs with neural networks, which are trained with gradient descent. The loss value of a neurosymbolic program provides guidance for the search for symbolic programs.

NEAR requires a differentiable loss function and a language of differentiable programs. This is because the gradients need to flow through the sketches to train their neural networks. In this dissertation, we show how to use neural networks to guide the search in problem domains with non-differentiable programs and loss functions. This is achieved with a two-step search. The first search is performed in an abstract version of the original space where all non-differentiable operations are replaced with neural networks, thus resulting in a language of differentiable programs. Similarly to NEAR, the loss function used in the first search must be differentiable. One then obtains a sketch by removing the neural networks from the program the first search returns.

The holes in the sketch are filled with programs in the second search, which is performed in the original programmatic space. Since this second search is not guided with a neural network, we are able to use even non-differentiable loss functions. We call our approach NEUROSKETCHER, as it uses the guidance of neural networks to learn sketches that are then filled with an uninformed search algorithm. We apply NEUROSKETCHER to the problem of synthesizing programmatic heuristic functions to guide the IDA* search (Korf, 1985), where the objective is to minimize the size of the search tree—a non-differentiable loss function. NEUROSKETCHER discovered provably bounded-suboptimal heuristic functions that outperform some of the best known heuristics for the Pancake and Topspin puzzles.

1.1 Problem Formulation

We consider supervised learning tasks where a training set $T = (X_i, y_i)_{i=1\cdots b}$, with $X_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$, is sampled from a distribution $\mathcal{D}$ and a hypothesis $\rho(\cdot)$ must be learned that maps the vectors X_i to their corresponding y_i in T. The hypothesis ρ is chosen from a (possibly infinite) pool of options H, such that it minimizes a loss function $\mathcal{L}$. The learned hypothesis ρ is expected to generalize to test sets with pairs (X'_i, y'_i), with $X'_i \in \mathbb{R}^m$ and $y'_i \in \mathbb{R}$, sampled from similar but different distribution $\mathcal{D}'$. We consider problems where n and m might be different. We say that ρ generalizes strongly if it is trained on data sampled from $\mathcal{D}$ and presents a similar loss value on data sampled from $\mathcal{D}'$.

In order to achieve strong generalization, we consider programmatic hypotheses. Program Synthesis corresponds to the search for such programmatic hypotheses that satisfy the given specification. It consists of three principal components, namely Domain-Specific Language (DSL), Specification, and Synthesizer, as described in Figure 1.1. The user's intent that a programmatic hypothesis

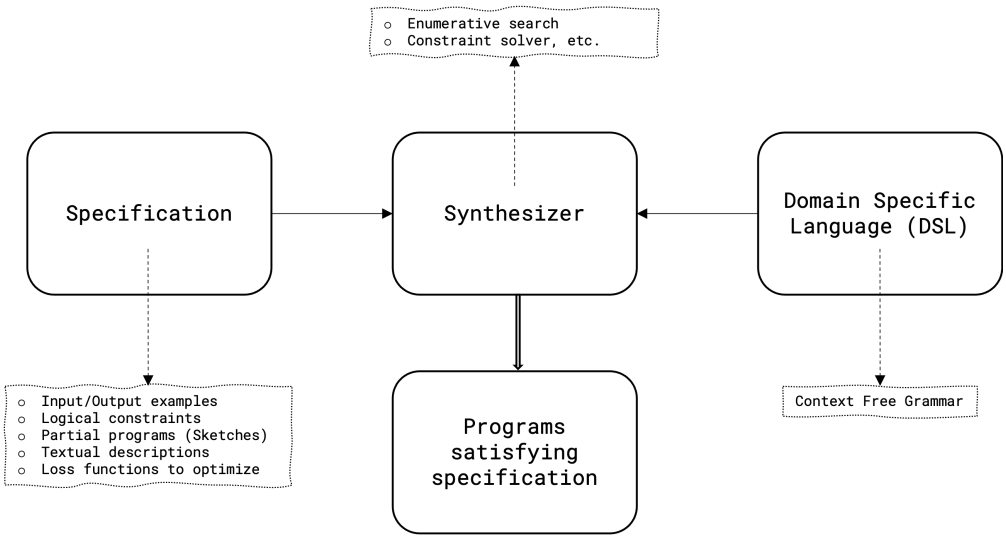

Figure 1.1: Program Synthesis Components

must satisfy is provided in the form of a specification. There are various ways to provide a specification, such as input-output examples, logical constraints, textual descriptions, and loss functions to optimize. Specification in the form of loss functions fits our problem formulation.

Similarly, the Synthesizer corresponds to any search technique that finds the programmatic hypotheses in the program space defined by the DSL, which satisfy the given specification. Enumerative search methods, constraint solvers, etc., are some of the well-known synthesizer techniques.

The DSL is defined as a context-free grammar $\mathcal{G} = (\Sigma, V, R, I)$, where Σ and V are sets of terminal and non-terminal symbols. We denote terminals with lowcase letters and non-terminals with uppercase letters. R defines the set of production rules that can be used to transform a non-terminal symbol into a sequence of terminal and non-terminal ones. Finally, I is the initial symbol of $\mathcal{G}$.

Figure 1.2 shows an example of a DSL (left), where I and B are the only non-terminal symbols, `sum` and `map` are the usual functions for summing all values of a vector and for defining a function that is applied to all values of a vector, respectively. `v` is a vector and `x` is a scalar, and `f(x)` is a lambda function defined by `I`. The language also allows for if-then-else (ITE) instructions, where the Boolean expression is handled with the non-terminal `B`.

The program `sum(map(f(x): x`^{2}`, v))`, which is accepted by this language, sums the squared values of the elements of `v`. The language defines a space of programs that allows for operations that involve squaring and adding values of a vector. For a given language $\mathcal{G}$, we denote by $[\![\mathcal{G}]\!]$ the possibly infinite set of programs that the language accepts.

We represent programs as abstract syntax trees (AST). Figure 1.2 (right) shows the AST for the

```
I → sum(I) | map(I,I)
    | if(B) then I else I
    | f(x):I | I^2 | I+I | x | v
B → I > 0 | I < 0 | I == 0
```

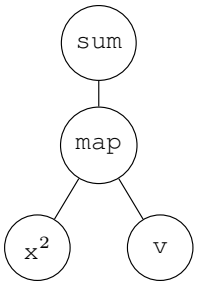

Figure 1.2: Domain specific language (left) and the abstract syntax tree for `sum(map(f(x): x^2, v))` (right).

program `sum (map (f(x): x^2, v))`. Each node represents a production rule (e.g., the root node represents the rule `I → sum(I)`); the number of children of a node is equal to the number of non-terminal symbols on the right-hand side of the rule the node represents. For example, `I → sum(I)` has only one non-terminal, therefore the root has a single child.

A solution to $\arg\min_{p \in [\![\mathcal{G}]\!]} \mathcal{L}(p, T)$ provides a programmatic hypothesis for the supervised learning problem specified by T. In the next sections, we describe different approaches for approximating a solution for this equation.

Thesis Statement In this dissertation, we aim to determine whether the two-step search process adopted by the NeuroSketcher algorithm mitigates the search space explosion problem faced by enumerative search techniques like Bus and the inability of Near to guide the search in cases involving non-differentiable grammars and loss functions.

1.2 Contributions

The dissertation introduces the "NeuroSketcher" algorithm, a novel two-step search approach. This algorithm employs neural networks to tackle the challenge of filling gaps in programs. Its purpose is to guide the search within programmatic spaces, even when dealing with non-differentiable programs and loss functions.

NeuroSketcher has been successfully applied to synthesize heuristic functions. These synthesized heuristics demonstrated superior performance when compared to some of the most effective known heuristics for problems such as the Pancake and Topspin puzzles. The algorithm's synthesized heuristics were compared with those learned using baseline systems. NeuroSketcher's heuristics displayed improved effectiveness and efficiency.

The dissertation illustrates how certain heuristics synthesized by NeuroSketcher can be proven to be bounded-suboptimal. This contribution enhances the understanding of the quality of

these heuristics.

Chapter 2

Background

In this chapter, we delve into the subjects of Heuristic Search, with a particular focus on IDA*. We delve into the problem domains employed in our experimentation, Program Synthesis, and one of its enumerative search algorithms, specifically BUS. Furthermore, we explore the concept of generating hierarchical training data tailored to our problem setting.

2.1 Heuristic Search

A state space search problem is defined by a tuple (s_0, s_g, G, C), where $G = (V, E)$ is a graph with V denoting the set of vertices, which represent the states, and E the set of edges, which represent actions an agent can take from a given state; s_0 and s_g, both in V, define the initial and goal states; C defines a cost function that receives an edge (n_1, n_2) in E and returns the cost of transitioning from n_1 to n_2. The state space, represented by G, does not have to be explicitly defined. That is, an agent knows that it is currently in a state s, and it has access to a successor function that returns all edges that connect s to other states in G. In this way, parts of the graph can be constructed as needed. A solution to a state space search problem is a path $(s_0, s_1, \cdots, s_g)$ such that (s_i, s_j) is in E for any adjacent states s_i and s_j on the path. The cost of a path is the sum of the costs of all the edges connecting states on the path. A solution path π is optimal if there is no other path whose cost is less than π's.

As the search space grows, solving state space search problems becomes computationally infeasible. To address this, heuristic search is a well-known technique used to find solutions in such large search spaces. It involves making informed decisions about which paths or options to explore first, based on heuristic functions. In this context, a 'heuristic' refers to a guiding principle that assists in making decisions when faced with uncertainty or incomplete information. Heuristic functions provide estimates of the cost associated with different choices, enabling the search algorithm to pri-

oritize promising paths and avoid unproductive ones. Therefore, the quality of the heuristic function used plays a crucial role in determining the efficiency and effectiveness of the heuristic search. In our work, we focused on synthesizing heuristic functions for the state space search problems known as Pancake and Topspin. Among the many heuristic search algorithms, notable ones are A* (Hart et al., 1968) and IDA* (Korf, 1985). In our experiments, we utilized the IDA* algorithm.

2.2 Iterative Deepening A* (IDA*)

Let $h(n)$ be a heuristic function that estimates the cost-to-go from a state n to the goal s_g. In addition, $g(n)$ is the cost of the path a search algorithm encounters that connects s_0 to n. IDA* uses the cost function $f(n) = g(n) + h(n)$ to guide its search. IDA* performs a sequence of cost-bounded depth-first searches until it finds a solution path. The first search is performed with the cost value of $h(s_0)$ and all states encountered in the search whose f-value is larger than $h(s_0)$ are pruned. If a solution path is not retrieved, IDA* performs a new cost-bounded search, where the cost of the next iteration is set to the smallest f-value pruned in the previous iteration. This search strategy allows IDA* to find optimal solutions if the heuristic function is admissible, that is, $h(n) \leq h^*(n)$, where $h^*(n)$ is the optimal cost to reach the goal from n. Moreover, IDA* is guaranteed to find bounded-suboptimal solutions, i.e., solutions that are not more expensive than $k \cdot h^*(s_0)$ as long as $h(n) \leq k \cdot h^*(n)$ for all n.

Algorithm 1 illustrates the functioning of the IDA* search algorithm. The algorithm starts by estimating the cost of the solution path, applying the heuristic function h to the start state *root* in Line 1 of the IDA* procedure. The IDA* procedure then utilizes the SEARCH procedure to find the solution cost within the bound. If the solution is not present within the bound, the SEARCH procedure returns the next minimal bound to expand the search tree, and the IDA* procedure continues the search with the new bound

The SEARCH procedure calculates the estimated cost of the path with the current state by adding the current cost g with the heuristic estimate of the current state in Line 1. If the estimated cost exceeds the bound, it prevents further expansion of the current path and returns the estimated cost f of the node as a candidate for calculating the next search bound. Line 4 checks if the current state is the goal state of the problem and returns its cost g. The SEARCH procedure expands the tree in a depth-first fashion, considering each successor state of the current state, and attempts to calculate the next minimal bound value by calling SEARCH with the successor node in Line 8. The procedure returns if it finds a solution within the bound; otherwise, it returns the next minimal search bound.

Algorithm 1 IDA*

Procedure: IDA*($root$, h)
Require: start state $root$, heuristic function h
Ensure: Solution cost or $\perp$
1: $bound \leftarrow$ HEURISTIC($root$,h)
2: **while** not timeout **do**
3: $t \leftarrow$ SEARCH($root$, 0, $bound$, h)
4: **if** t.result equals SOLUTION **then**
5: **return** t.bound
6: $bound \leftarrow t$.bound
7: **return** $\perp$

Procedure: SEARCH($node, g, bound, h$)
Require: current state $node$, path cost g, cost bound $bound$, heuristic function h
Ensure: Search result and new bound (t.result, t.bound)
1: $f \leftarrow g+$ HEURISTIC($node$, h)
2: **if** $f > bound$ **then**
3: **return** $(\perp, f)$
4: **if** GOALTEST($node$) **then** ▷ Check if goal state is reached
5: **return** (SOLUTION, g)
6: $min \leftarrow \infty$
7: **for all** $successor$ **in** GENERATESUCCESSORS($node$) **do**
8: $t \leftarrow$ SEARCH($successor$, $g+1$, $bound$, h)
9: **if** t.result equals SOLUTION **then**
10: **return** t
11: **if** t.bound $< min$ **then**
12: $min \leftarrow t$.bound
13: **return** $(\perp, min)$

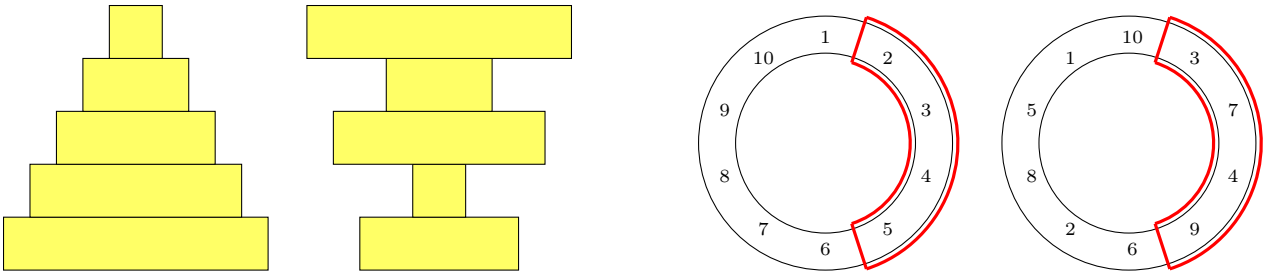

Figure 2.1: 5-Pancake puzzle (left) and the (10, 4)-Topspin puzzle (right). The state on the left shows the goal for both domains, while the state on the right shows a random state.

2.3 Problem Domains: Permutation Sorting Puzzles

We consider the Pancake and Topspin puzzles, two permutation sorting puzzles (Lippi, Ernandes, & Felner, 2016), as problem domains. We chose these domains because they have a similar representation, which allows us to use a single DSL for both problems. Second, the Pancake puzzle has a powerful programmatic heuristic function known as the GAP heuristic (Helmert, 2010) that is effective in state spaces of various sizes. Finally, one can easily change the size of a puzzle to make it arbitrarily easy or arbitrarily hard, which allows us to use easy instances during the synthesis of the heuristics and evaluating them in hard instances.

The Pancake puzzle is represented by a stack of pancakes of different sizes that need to be sorted from largest (bottom) to smallest (top). Figure 2.1 (left) shows the goal state and a scrambled state of the 5-Pancake puzzle. The states can be represented with a vector with integers from 1 to N; the scrambled state shown in Figure 2.1 is $[5, 2, 4, 1, 3]$. In this domain, the agent can flip any top k pancakes of the pile. For example, flipping the top 2 pancakes of the scrambled state, we obtain $[2, 5, 4, 1, 3]$. The state space with N pancakes has $N!$ states and $N - 1$ actions available in each state, which can result in challenging problems for large N.

Topspin is represented by a circle of tokens, which are numbers ranging from 1 to N. The puzzle has a spinning wheel that allows one to flip K adjacent tokens in the circle. We denote by (N, K)-Topspin the version with N tokens and with a spinning wheel of size K. Figure 2.1 (right) shows the goal state, where the numbers are sorted clockwise, and a scrambled version of $(10, 4)$ -Topsin. The spinning wheel is represented by the semicircles around the numbers $[2, \ldots, 5]$ in the goal state. The scrambled state is represented by the vector $[10, 3, 7, 4, 9, 6, 2, 8, 5, 1]$. If we rotate the spinning wheel in the position shown in the figure, then we obtain $[10, 9, 4, 7, 3, 6, 2, 8, 5, 1]$. Rotating the numbers to fit the spinning wheel does not count as an action; thus, the agent can rotate any subsequence of size K. We consider puzzles where both N and K are even, so that all $N!$ states are reachable from the goal (Wilbur, 2001). In all experiments, we use random permutations as initial states.

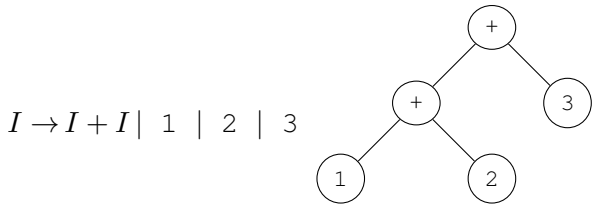

Figure 2.2: DSL and AST for `((1+2)+3)`, which produces the output `6`.

2.4 Program Synthesis

Program synthesis is an active area of research in Artificial Intelligence. It can be considered as a search in the program space to find programs that satisfy a specification λ. The program space is defined by a Domain-Specific Language (DSL). There are different classes of program synthesis search algorithms, and enumerative search algorithms are one category among them. Enumerative search algorithms systematically explore the program space by generating and evaluating all possible programs according to a certain order. In our study, we considered Bus (Albarghouthi et al., 2013), which is one of the enumerative search algorithms for generating programs.

2.4.1 Bottom-Up Search (Bus)

Bottom-up search (Bus) (Albarghouthi et al., 2013; Udupa, Raghavan, Deshmukh, Mador-Haim, Martin, & Alur, 2013) iteratively constructs programs of increasing number of nodes in the AST. The search starts by generating all programs defined by terminal symbols of the language that have an AST size of 1. Using the programs of size 1, Bus generates the programs of size 2 with the combination of the programs generated thus far and the production rules of the language. The search continues using programs of sizes 1 and 2 to generate programs of sizes 3, etc. In case of optimization problems, the search stops when it reaches a time and memory limit and returns the program found that best optimizes the loss.

The advantage of bottom-up over other enumerative approaches such as top-down (Lee, Heo, Alur, & Naik, 2018a; Alur, Radhakrishna, & Udupa, 2017) is that all the programs encountered in the search are complete and can be evaluated. This is in contrast to the programs in the top-down search, which can represent **partial programs**. Partial programs can contain non-terminal symbols in them and for that they might not be executable (e.g., `I+1`, from Figure 2.2).

The ability to execute the programs allows us to perform observational equivalence checks: if programs p_1 and p_2 produce the same output for the specification λ, then the search can discard p_1 or p_2. Observational equivalence checks can drastically reduce the search space and thus justify the popularity of Bus (Odena et al., 2021; Shi, Dai, Ellis, & Sutton, 2022; Ameen & Lelis, 2023) and our choice of Bus in our experiments.

Algorithm 2 Bottom-Up Search (BUS)

Procedure: BUS($\mathcal{G}, \lambda$)
Require: Grammar $\mathcal{G} = (V, \sum, R, I)$, a specification λ
Ensure: Solution program p or $\perp$, program bank B

$s \leftarrow 1$
$B \leftarrow \emptyset$
$\mathcal{H} \leftarrow \emptyset$
while not timeout **do**
    for p in GROW($\mathcal{G}, B, s$) **do**
        $o \leftarrow$ EXECUTE(p)
        if (p, o) satisfies λ **then**
            return p, B
        if $o \notin \mathcal{H}$ **then**
            B[s].add($\{p, o\}$)
            $\mathcal{H} \leftarrow \mathcal{H} \cup \{o\}$
    $s \leftarrow s + 1$
return $\perp, B$

Procedure: GROW($\mathcal{G}, B, s$)
Require: Grammar $\mathcal{G} = (V, \sum, R, I)$, program bank B, and size s
Ensure: program p of size s

for r in R **do**
    if arity(r) = 0 and size(r) = s **then**
        yield r
    else
        for $(p_1, p_2, \ldots, p_k)$ in $B \times B \times \cdots \times B$ **do** ▷ Operation over values of dictionary B
            $p \leftarrow r(p_1, p_2, \ldots p_k)$
            if size(p) = s and type(p_i) = type($r.arg_i$) for i in $\{1, 2, \cdots, k\}$ **then**
                yield $r(p_1, \cdots, p_k)$

Size of Programs	Set of Programs
1	`1,2,3`
2	`1+3, 2+3, 3+3`
3	`1+3+3, 2+3+3, 3+3+3`
...	...

Table 2.1: Illustration of Bottom-Up Search Synthesis

Algorithm 2 illustrates the pseudocode of BUS. It begins with an empty program bank, B (Line 2), and an empty program execution map, $\mathcal{H}$ (Line 3). Subsequently, it employs the procedure GROW (Line 5) to synthesize programs in ascending order of size, which is denoted as s. If it encounters a program, p, that satisfies the provided specification, λ, it returns the solution program along with the program bank. Line 10 executes the observational equivalence check, where it compares the output, o, of the execution with the pre-existing execution outputs stored in $\mathcal{H}$. If p is not observatially equivalent to another program seen in the search, BUS adds it to the program bank and o to $\mathcal{H}$. The procedure EXECUTE on Line 6 runs the program, which in our context represents the result of employing p as a heuristic function in IDA* to solve state-space search problems.

The GROW procedure is given the grammar $\mathcal{G}$, a collection of programs B, and a specified program size s. In the context of GROW, it generates programs of size s using the production rules denoted by $r \in R$ (Lines 1-8). If the production rule r is a terminal rule and the resulting program matches the size s, it is returned (Line 3). On the contrary, if r is not a terminal rule, the procedure performs a Cartesian product on all previously observed programs within the program bank B by utilizing the arity of r. This product results in programs of size s, while ensuring that each subprogram p_i type aligns with the type of arguments arg_i specified by the production rule r, i.e., $\text{type}(p_i) = \text{type}(r.arg_i)$, for i in $\{1, 2, \cdots, k\}$. For example, the production rule $I \rightarrow I + I$ in Figure 2.2 accepts only subprograms of integer type ($\text{type}(I) = \text{integer}$) to be combined into a new program. Function $\text{size}(\cdot)$ indicates the number of nodes in the program's Abstract Syntax Tree (AST) (Line 7), k represents the arity of the production rule r, and the type check guarantees that the types of the subprograms $(p_1, \cdots, p_k)$ correspond to the required argument types of r.

Table 2.1 illustrates an example of programs generated using the DSL in Figure 2.2 with BUS. The set of programs starts with size 1, listing all terminal rules. Subsequently, it combines size 1 programs to create programs of larger sizes through the non-terminal rule $I \rightarrow I + I$. Notably, the program $(1 + 1)$, while potentially generatable, is absent from the program set due to its observational equivalence with the program 2 (they both produce the same output for any input value). Consequently, the observational equivalence check significantly decreases the search space by eliminating potentially redundant programs.

Chapter 3

Related Work

Program synthesis has been applied to various domains, including the synthesis of database transactions (Qian, 1990, 1993), bit manipulation tasks (Solar-Lezama et al., 2005; Gulwani & Venkatesan, 2009), string manipulation tasks (Gulwani, 2011), and logic programming (Kodratoff et al., 1990; Deville & Lau, 1994). Alongside the application to various domains, the search algorithms have evolved to counter various new challenges that arise and to adapt to different domains. In our work, we applied program synthesis techniques to find heuristic functions for the puzzles Pancake and Top-spin. In this chapter, we aim to discuss different search algorithms and various other approaches that focus on synthesizing improved heuristic functions.

3.1 Symbolic Approaches

There are different categories of program synthesis algorithms. Constraint-based search algorithms navigate the search space using constraint solving techniques (Solar-Lezama, 2009). In enumerative search algorithms (Albarghouthi et al., 2013; Lee et al., 2018b; Alur et al., 2017), the search space is explored systematically by generating programs according to a specific order. Bottom-Up Search (BUS) (Udupa et al., 2013) and Top-Down Search (Lee et al., 2018a) belong to this category. We consider the enumerative search algorithms as relevant to the application domain considered for our work. However, both bottom-up and top-down approaches suffer from an explosion in the size of the program space for problems of practical interest.

3.2 Neurosymbolic Approach

Shah et al. (2020) introduced a powerful method known as NEAR, which uses neural networks to guide the search for programmatic hypotheses. NEAR performs an A* search (Hart et al., 1968)

in a top-down fashion, while guided by a heuristic function that estimates, for each node n in the tree, a lower bound on the loss value of the best program in the subtree rooted at n.

The heuristic function NEAR uses is individually trained for each node in the tree during the search. NEAR replaces each non-terminal symbol in the partial program that a node represents with a neural network. Then, it uses a gradient descent algorithm to solve $\arg\min_\Theta \mathcal{L}(p_\Theta, T)$ (from 1.1), where Θ is the set of weights of the neural networks used to replace the non-terminals of the program p and p_Θ is the resulting neural relaxation of p. The value of $\mathcal{L}(p_\Theta, T)$ after training the models in p_Θ is assumed to be a lower bound on the value of $\mathcal{L}(p', T)$ for all complete programs p' that can be derived by continuing searching from the partial program p.

NEAR uses gradient descent to solve $\arg\min_\Theta \mathcal{L}(p_\Theta, T)$, which requires that the programs in $[\![\mathcal{G}]\!]$ are differentiable or have a suitable differentiable approximation that can be used in training and that $\mathcal{L}$ is also differentiable. ITE structures are discontinuous, since small changes to the Boolean expression of the structure are unlikely to change the result of the computation. One can approximate ITE structures such as `if B then A else C` with $\sigma(\texttt{B}) \cdot \texttt{A} + (1 - \sigma(\texttt{B})) \cdot \texttt{C}$, where $\sigma(\cdot)$ is the sigmoid function. Although this approximation is suitable when `A` and `B` are of the same type and can be added, it is unclear how to approximate ITE structures when `A` and `B` are semantically different, e.g., `A` writes in an entry of an array, while `B` returns an integer. Therefore, NEAR fails when $\mathcal{G}$ and $\mathcal{L}$ are non-differentiable. NEUROSKETCHER overcomes this issue through neural language abstraction and two-step search procedure.

3.3 Other Approaches

Nye et al. (2019) use neural networks to generate the sketches. They consider program synthesis tasks where one can learn in a self-supervised manner. Similarly to Shah et al. (2020), we consider the more general case where the problem does not necessarily allow for self-supervised training. Medeiros et al. (2022) also learn sketches, but with imitation learning and in a multi-agent setting; we learn sketches with the guidance of neural networks.

Neural networks can also be used to learn a latent space with helpful properties such as locality (i.e., programs near each other have similar "behavior") for the search for programs (Trivedi, Zhang, Sun, & Lim, 2022; Liu, Hu, Cheng, yi Lee, & Sun, 2023). This idea was applied in the context of finding programmatic policies, and locality in the latent space is defined in terms of the behavior of the policies. It is not clear how to transfer the notion of behavior to problems such as the heuristic synthesis we tackle.

Previous work on learning heuristic functions collects labeled training instances to learn a heuristic function in a bootstrap procedure, where one learns from the easy instances in the data set, which allows the search to solve the harder instances that are added to the training set (Jabbari Arfaee,

Zilles, & Holte, 2011). The programmatic representation of heuristics allows us to follow a very different approach, where we learn from small versions of the problems with the goal of generalizing to larger versions of the problem. The synthesis of programmatic heuristic functions has been applied to polynomial state spaces, where the generalization to exponentially larger spaces cannot be tested (Bulitko, Wang, Stevens, & Lelis, 2022). Furthermore, the search in the programmatic space was not guided, which is similar to our Bus baseline. To our knowledge, this is the first time that synthesized programmatic heuristics are used to guide the search in exponential spaces. Generalized planning uses programs to solve shortest-path problems in the context of classical planning (Bonet, Palacios, & Geffner, 2010; Hu & De Giacomo, 2013; Aguas, Jiménez, & Jonsson, 2018), while we use programmatic solutions to guide the search.

Chapter 4

NeuroSketcher

We present a method that uses neural networks as part of the programs to guide the search even when the language contains non-differentiable operations and the loss is non-differentiable. We create a neural abstraction of the original language where the non-differentiable operators of the language are replaced with neural networks. Similarly to NEAR, we assume that the neural models are powerful enough to be used in lieu of fully symbolic sub-programs of a program. Unlike NEAR, we do so not only to guide the search but also to achieve a fully differentiable language. Another difference from NEAR is that the output of our search in the neural-abstracted language space is possibly a program with neural networks as sub-programs. Since we are interested in fully symbolic programs that are interpretable, we treat the output of this search as a program sketch, where the neural sub-programs are treated as holes (Solar-Lezama, 2009) that need to be replaced with fully symbolic programs in a second step. This second step searches in the space of the original language, which contains non-differentiable symbols. Unlike the first search, this second search is not guided by neural networks.

4.1 Neural Language Abstractions

A key component of our approach is a neural language abstraction, which we define below and provide an example.

Definition 1 (Neural Language Abstraction)**.** *Let $\mathcal{G} = (\Sigma, V, R, I)$ be a context-free grammar defining a programming language. Also, let $\delta\Sigma \subseteq \Sigma$ be symbols related to all differential operators of the language. We define a neural abstraction $\delta\mathcal{G} = (\Sigma', V', R', I)$ of $\mathcal{G}$ where $\Sigma' = \delta\Sigma \cup F_\Theta$. Here, F_Θ is a set of neural models with trainable parameters Θ, with one model for each combination of input and output types of non-differentiable operators in $\mathcal{G}$. The set of production rules $R' \subseteq R$ and non-terminal symbols $V' \subseteq V$ account for all rules and symbols that can be reached from I after*

removing the rules that are not related to Σ'.

Consider the following example for the grammar described in Figure 1.2, which is provided again below for convenience.

$$
\begin{aligned}
\texttt{I} \rightarrow\ & \texttt{sum(I)} \mid \texttt{map(I,I)} \\
& \mid \texttt{if(B) then I else I} \\
& \mid \texttt{f(x):I} \mid \texttt{I}^2 \mid \texttt{I+I} \mid \texttt{x} \mid \texttt{v} \\
\texttt{B} \rightarrow\ & \texttt{I > 0} \mid \texttt{I < 0} \mid \texttt{I == 0}
\end{aligned}
$$

Example 1. *The neural language abstraction* $\delta\mathcal{G}$ *of the language shown in Figure 1.2 is the following.*

$$
\begin{aligned}
I \rightarrow\ & \texttt{sum(I)} \mid \texttt{map(I,I)} \mid \texttt{f(x):I} \mid I^2 \mid I\texttt{+}I \mid \texttt{x} \mid \texttt{v} \\
& \mid f_\Theta(I) \mid g_\Theta(I) \mid z_\Theta(I)
\end{aligned}
$$

Here, the ITE structure of the original language is replaced with three trainable model architectures: $f_\Theta(I)$*,* $g_\Theta(I)$*, and* $z_\Theta(I)$*. We have one model architecture for each possible combination of input-output values. Architecture* $f_\Theta : \mathbb{R}^u \rightarrow \mathbb{R}$ *is recurrent as it receives a sequence of values as input and returns a scalar. This architecture handles programs using the* `sum` *symbol. Architecture* $g_\Theta : \mathbb{R}^u \rightarrow \mathbb{R}^u$ *is a sequence-to-sequence architecture handling program with the* `map` *operation. Finally,* $z_\Theta : \mathbb{R} \rightarrow \mathbb{R}$ *handles programs that receive a scalar and return a scalar to handle operations with variable* `x`*. The non-terminal* `B` *and all production rules related to it are not present in* $\delta\mathcal{G}$ *because they are no longer reachable from* I *once the ITE is removed from it.*

Neural language abstractions can only be helpful if they do not reduce the original language to a trivial language where the initial symbol can only be transformed into neural models. For example, an abstraction would not be helpful for a language of the form `I` $\rightarrow$ `if(B) then C else C`, with non-terminals `B` and `C`, as the initial symbol `I` would only be transformed into a set of neural models. The result of the synthesis would be a trained neural network, as opposed to a neurosymbolic program that represents a sketch that can be completed without neural guidance.

4.2 Using Neural Language Abstractions

We call our approach NeuroSketcher for it uses neural networks to synthesize a sketch that is later completed with a fully symbolic search. NeuroSketcher receives as input a base synthesizer

$\mathcal{S}$, a DSL $\mathcal{G}$ and its neural abstraction $\delta\mathcal{G}$, a training set T, and a loss function $\mathcal{L}$; NeuroSketcher returns a program $p \in [\![\mathcal{G}]\!]$ that minimizes $\mathcal{L}(p, T)$.

NeuroSketcher uses $\mathcal{S}$ to search in the space of $\delta\mathcal{G}$, which returns a possibly neurosymbolic program p_θ that also represents a sketch once the neural networks in p_θ are removed. $\mathcal{S}$ evaluates each program p'_θ it encounters in this neurosymbolic search by computing its loss $\mathcal{L}$ with respect to the training data T. If p'_θ contains one or more neural networks, $\mathcal{S}$ employs a gradient descent algorithm to train the neural models in p'_θ. $\mathcal{S}$ returns the program p_θ encountered in the search in the space $[\![\delta\mathcal{G}]\!]$ that minimizes $\mathcal{L}(p_\theta, T)$. NeuroSketcher then uses $\mathcal{S}$ to search in the space of the original language $\mathcal{G}$ for programs to fill the holes of p_θ. The best program p with respect to $\mathcal{L}$ encountered in this second search is returned as the output of NeuroSketcher.

NeuroSketcher works with any base synthesizer that is able to explore the spaces of $\delta\mathcal{G}$ and $\mathcal{G}$. For example, if using Bus, NeuroSketcher runs the search for a given time and/or memory limit and returns the neurosymbolic program seen in the search that better optimizes the loss. Then, it can rerun the Bus algorithm while using the programs encountered to fill in the holes of the sketch. Similarly, NeuroSketcher can use top-down search algorithms, exactly as described for Bus. NeuroSketcher is able to obtain guidance from neural models for any search algorithm, including bottom-up approaches, because it divides the search into two parts: neural networks determine the most promising sketch, which is completed with a second search. In practice, one might choose to return the top k sketches and try to fill all of them in the second step; we used $k = 1$.

4.3 NeuroSketcher Algorithm

The pseudocode of the NeuroSketcher algorithm is presented in Algorithm 3. It begins by initializing the sketch bank $\mathcal{Q}_k$, which holds the top k sketches on Line 1. Line 2 initializes the bank of best programs, which holds the top n programs for the entire search. The search for the best k sketches proceeds until a timeout is reached on Line 3. NextSketch on Line 4 generates the neurosymbolic program, which acts as a sketch using the base synthesizer $\mathcal{S}$. The sketch is then trained and evaluated using loss function $\mathcal{L}_1$ on Line 5, employing the grammar $\delta\mathcal{G}$. Line 6 inserts the newly found sketch into $\mathcal{Q}_k$, which contains the top k sketches based on the calculated loss l. Line 7 iterates over all the selected best k sketches and fills the neural holes with symbolic programs. The symbolic programs are generated using the grammar $\mathcal{G}$ with the base synthesizer $\mathcal{S}$ on Line 9. The newly generated programs are evaluated on Line 10 with respect to loss function $\mathcal{L}_2$. Line 11 inserts the newly generated program into $\mathcal{H}_n$, which contains the top n programs based on loss l'. Finally, Line 12 returns the top n programs produced using NeuroSketcher. The loss function $\mathcal{L}_1$ must be differentiable, where as the loss function $\mathcal{L}_2$ need not be differentiable.

Algorithm 3 NEUROSKETCHER

Procedure: NEUROSKETCHER($\mathcal{S}, \mathcal{G}, T, \delta\mathcal{G}, \mathcal{L}_1, \mathcal{L}_2, n, k$)

Require: Base synthesizer $\mathcal{S}$, Grammar $\mathcal{G}$, Training set T, Neural abstraction $\delta\mathcal{G}$, Sketch loss function $\mathcal{L}_1$, Program loss function $\mathcal{L}_2$, number of best programs n, number of sketches k

Ensure: Best n programs, $\mathcal{H}_n$

$\mathcal{Q}_k \leftarrow \emptyset$
$\mathcal{H}_n \leftarrow \emptyset$
while not timeout **do**
    $p_\theta \leftarrow$ NEXTSKETCH($\mathcal{S}, \delta\mathcal{G}$)
    $l \leftarrow$ TRAINEVALUATE($p_\theta, T, \mathcal{L}_1$)
    $\mathcal{Q}_k$.insert($\{l, p_\theta\}$)
for *sketch* **in** $\mathcal{Q}_k$ **do**
    while not timeout **do**
        $p \leftarrow$ NEXTPROGRAM($S, \mathcal{G}, sketch$)
        $l' \leftarrow$ EVALUATE($p, T, \mathcal{L}_2$)
        $\mathcal{H}_n$.insert($\{l', p\}$)
return $\mathcal{H}_n$

Chapter 5

Empirical Results

5.1 Experiment Details

In this section, we discuss the Domain-Specific Language used for generating heuristics for both Pancake and Topspin puzzles, the loss functions employed for synthesizing sketches and fully symbolic heuristics, the baselines used for comparing the efficacy of our method, and details about the training and test data used for our experiments.

5.1.1 Domain-Specific Language and Its Abstraction

Figure 5.1 shows the DSL we use in our experiments. The language assumes that the program receives a state as input (e.g., $[5, 2, 4, 1, 3]$). The function `neighbor` receives a vector v and an integer d and returns a vector where the numbers in v are shifted to the right by d positions. For example, `neighbor(state, 1)` returns $[3, 5, 2, 4, 1]$ for our state $[5, 2, 4, 1, 3]$. The language also has discontinuous operations, such as if-then-else structures and modulo (%).

The neural language abstraction we consider replaces all production rules for non-terminal `F` with neural networks that receive a real value and output another real value; the networks replace the operations one can derive with the production rules for `F` in the language. Note that this implies that the non-terminal symbols `ITE` and `B` are no longer reachable and are thus removed from the language. With this, all discontinuous operations of the original language are removed, and the abstraction only accepts strings representing differentiable programs.

```
  I → sum(L) | P+P | M-M
  P → sum(L) | M | C
  M → sum(L) | P | C
  L → state | A+A | A-A | map_k(F,K)
  K → state | A+A | A-A | N
  A → state | N | map_k(F,K)
  C → -1 | 0...9 | len | var1 | var2 | var3
  D → C+C | C-C | C
  N → neighbor(state,D)
  F → F+F | F-F | F%F | F*F | F/F | min(F,F) |
      max(F,F) | abs(F) | ITE | C
ITE → IF(B) THEN (F) ELSE (F)
  B → F<F | F<=F
```

Figure 5.1: Language for permutation sorting problems.

5.1.2 Loss Functions

We synthesize programmatic heuristic functions with the goal of reducing the size of the IDA* search tree. However, the size of the search tree is not a suitable loss function for training neural models in NeuroSketcher's neurosymbolic programs. This is because the computation of the size of the tree requires one to run IDA*, and the gradients do not flow through IDA*'s execution. Instead, we use the mean squared error (MSE) used in previous work (Jabbari Arfaee et al., 2011), which measures the squared difference between the h^*-value and the predicted h-value.

Since the second search NeuroSketcher performs is not guided by neural networks, the loss used in this search does not have to be differentiable. Therefore, we can use the size of the IDA* search tree as the loss. For each programmatic heuristic function h NeuroSketcher considers in its second search, we run IDA* with h with a limit of expansions of $4,000$ nodes for each initial state in a set of training problems T. NeuroSketcher returns the h that expands fewer states while trying to solve the problems in T. Here, we did not aim to optimize any abstract metric, as is the case in the work of Wilt and Ruml (2016), which results in finding better heuristic functions that minimize the mean node expansion. Instead, we considered the mean node expansion itself as the loss function. Additionally, we employed IDA* as the search algorithm rather than greedy best-first search algorithms.

5.1.3 Baselines

The two core baselines we consider are: (i) Bus searching in the original programmatic space while evaluating the programs in terms of the IDA* search tree size and (ii) Bus searching in

NeuroSketcher's neural abstraction space while optimizing for MSE. Our goal with the first baseline is to measure the importance of the guidance the neural networks provide, while our goal with the second baseline is to measure the importance of the second search handling non-differentiable loss functions and languages. Since Near also searches in differentiable spaces and uses differentiable loss functions, we call the second baseline Near.

We also use heuristic functions as baselines in our experiments. In particular, we use the admissible GAP heuristic for the Pancake domain (Helmert, 2010). We also adapt the GAP heuristic to Topspin by dividing the GAP value by 2, so that it remains admissible (in Topspin, each action can fix two gaps at once). A gap refers to a pair of adjacent tokens or numbers in the puzzle that differ by more than 1. For both domains, we train an LSTM (Hochreiter & Schmidhuber, 1997) that encodes a heuristic function. We use a recurrent architecture because it accepts inputs of varied length, so we could use the learned model in puzzles that are larger than those used in training. The details of the parameters used with the LSTMs are provided in the Appendix.

In our experiments, we did not consider heuristic functions that do not generalize strongly. For example, we did not consider fully connected neural networks, as they only generalize to states of the same space used in training. For the same reason, we also did not consider pattern database heuristics (Culberson & Schaeffer, 1998).

We used five seeds for all methods that use neural networks and present the average results and standard deviation, which was zero for NeuroSketcher and LSTMs. For the former because it always finds the same sketches and for the latter because it could not learn effective heuristics.

The Bus search generates a potentially large number of programs in batches. Once the programs are generated, we evaluate them in parallel. For NeuroSketcher and Near, we use 100 CPUs for 0.5 hours of computation for NeuroSketcher's first search and for Near's search; we then use 1000 CPUs for 1.5 hours for NeuroSketcher's second search. For the baseline Bus, we use 500 CPUs for 4 hours of computation. In total, we give the Bus baseline an advantage since NeuroSketcher uses 1550 CPU hours of computation, while the baseline Bus uses 2000 CPU hours. All experiments were run with 12GB of RAM.

5.1.4 Training and Test Instances

NeuroSketcher synthesizes functions for small versions of the problem domain for which the labels (the h^*-values) are readily available by solving instances with uninformed algorithms such as breadth-first search. We do not use training instances from the state space for which we are interested in solving instances. We use training instances of sizes 10, 11, 12, and 14 for the Pancake domain and of sizes $(10, 4)$, $(11, 4)$, $(12, 4)$, and $(14, 4)$ for Topspin.

In our experiments, for both domains, we used 20,000 instances of size 10, 10,000 instances

of size 11, and 8,000 instances of size 12 to train the neural networks in the first search of NeuroSketcher, the neural networks in the Near search, and also the LSTMs. These training instances are generated by performing a breadth-first search from the goal state, which is limited to 1 million expansions. The instances are randomly selected from the last layer of this search. Since the search is performed from the goal, the h^*-values of the states are readily available.

In the second search of NeuroSketcher, as the loss function, we perform IDA* searches in 10 randomly generated instances of size 10, with a limit of 4,000 node expansions, to select the best 100 programs encountered in the search. We then select the 5 programs out of the 100 that minimize the size of the IDA* search tree on 1000 random instances of puzzles of size 14. We use the computationally cheaper IDA* evaluation (size 10) to select a set of promising programs that are evaluated with the computationally more expensive IDA* evaluation (size 14). We perform this step, where we select the most promising programs according to the size of the IDA* search tree in puzzles of size 14, for all baselines: Bus, Near, and LSTMs.

To evaluate how well the programmatic heuristic functions generalize, we test them in much larger state spaces. For the Pancake domain we experiment with problems of size 70 and 90, while for the topspin domain we experiment with problems of sizes $(18, 4)$ and $(20, 4)$.

5.2 Discussion

We hypothesize that NeuroSketcher synthesizes more effective heuristic functions than both the Bus search in the original programmatic space and the Near search in the neural language abstraction space. This is because NeuroSketcher leverages both the guidance that neural networks provide, which Bus misses, and the guidance of the non-differentiable loss function, which Near misses.

Figure 5.2 shows the results for Pancake (top) and Topspin (bottom). We present the best 5 heuristic functions NeuroSketcher returns. They are indicated as NSx, where x is the number of the heuristic. For example, NS1 is the heuristic function that minimizes the IDA* search tree in the 1000 instances of size 14. We only show the best heuristic function obtained with Bus, Near, and LSTM for clarity and because the best 5 heuristics of these methods were all very similar to each other across all domain sizes. The y-axis of the plots shows the number of instances solved from a pool of 1,000 randomly generated instances. The x-axis shows the number of expansions IDA* performs to solve a number of instances. For example, the point y where a line intersects with $x = 250$ tells how many problems IDA* can solve while expanding at most 250 nodes per instance.

We include inflated versions of the GAP heuristic for Pancake and of the adapted GAP for Topspin, which we also refer to as "GAP" in the plots. For example, 2×GAP means that we multiply the h-values by 2. We acknowledge the availability of a simple programmatic policy

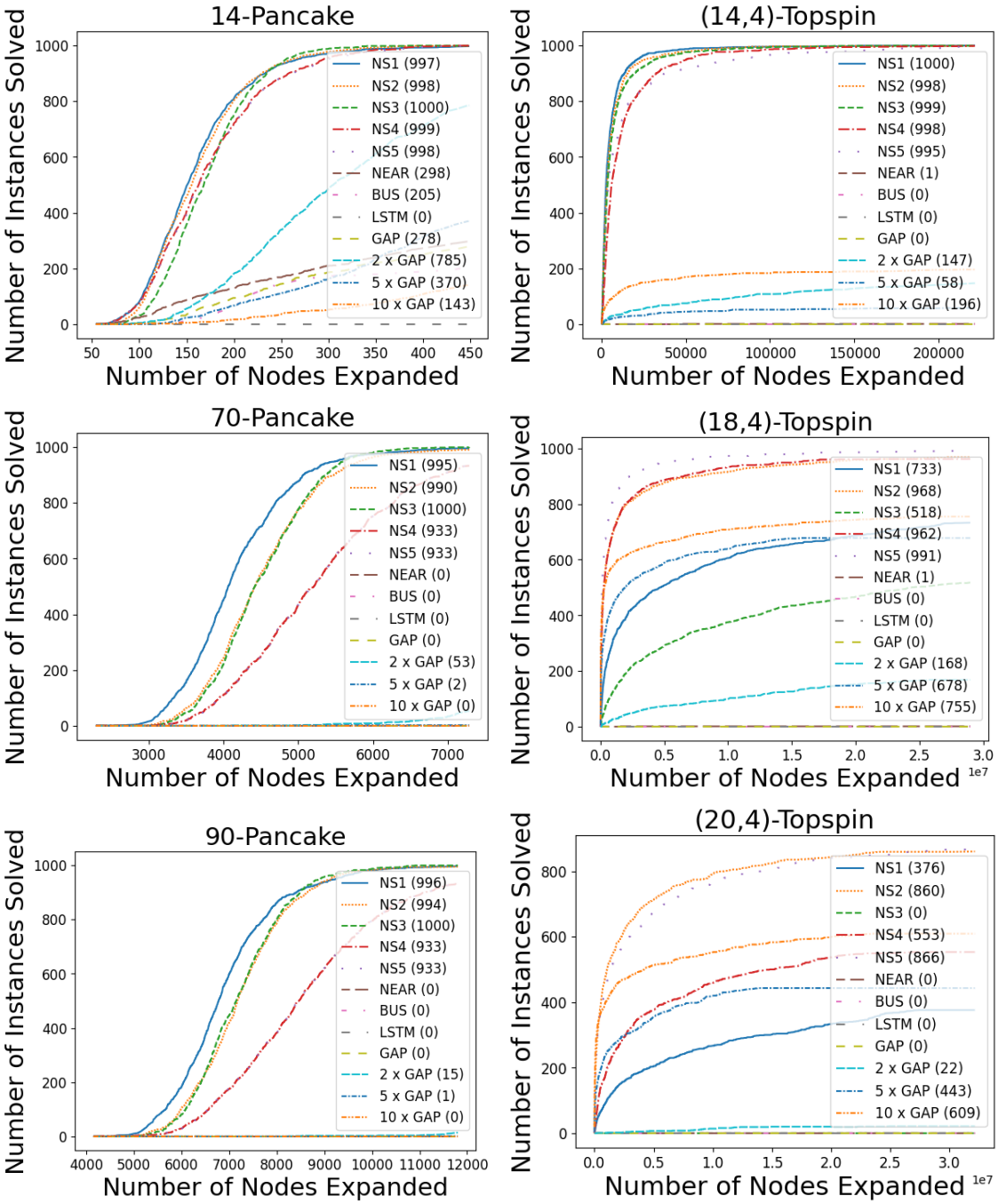

Figure 5.2: Evaluation of the different heuristic functions on Pancake and Topspin puzzles of different sizes. The y-axis shows the number of problems solved, while the x-axis shows the number of nodes expanded per instance.

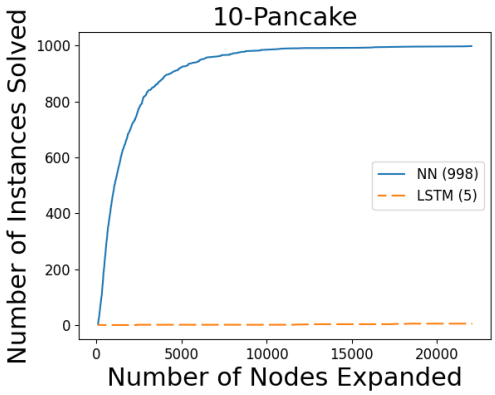

Figure 5.3: FeedForward (NN) and LSTM Study

that can solve any N-Pancake puzzle within 2×N actions. This policy operates by sequentially arranging the largest pancake that is not in its original order, flipping it to the top of the puzzle, and then flipping the remaining pancakes to position it in its corresponding goal state position. We did not include it in our evaluation because our goal was to produce programmatic heuristic functions for IDA* rather than programmatic policies. Moreover, the language we consider does not include the heuristic function that would allow IDA* to simulate such a programmatic policy. Since finding effective programmatic heuristic functions is still challenging for existing program synthesis approaches, the Pancake puzzle is still a relevant benchmark problem to our research question.

The results shown in Figure 5.2 support our hypothesis as the heuristics NeuroSketcher synthesizes outperform the other heuristics. While IDA* can solve a reasonable number of instances of the smaller 14-Pancake puzzle with some of the baseline heuristics, it can solve only a few instances of the larger 70- and 90-Pancake puzzles. The 5 heuristic functions NeuroSketcher synthesized for the Pancake domain allowed IDA* to solve almost all instances in the three sizes tested. It is remarkable that our heuristics outperform even the inflated version of GAP, which prior to our work was the best bounded-suboptimal heuristic function for this domain. Since not all baselines could solve all instances of the puzzles, we did not compare the solution quality the search algorithm returns with a given heuristic function. It would be uncertain to yield any meaningful comparison if there are no common instances solved across all the baselines.

The results in Topspin also support our hypothesis, but not all heuristics NeuroSketcher synthesized generalized to larger puzzles. For example, while NS3 performs well in the (14, 4)-Topspin puzzle, IDA* with it failed to solve any instances of the (20, 4)-Topspin. NS2 and NS5 generalized strongly as it allows IDA* to perform well on all sizes of Topspin tested. In particular, these two heuristics outperform all evaluated baselines.

Our results suggest that, while our programmatic representation allows for strong generalization, the neural networks we considered in our experiment could not learn strong heuristics, let alone

generalize. We also attempted to train fully connected neural networks for puzzles of size 10 in Figure 5.3. In this case, we could learn good heuristic functions. The negative LSTM results can be explained by a mixture of recurrent models being harder to train with their inability to consider non-differentiable loss functions. The latter can also explain the results of NEAR. It is also possible that the neural heuristic functions represented by LSTM and NEAR are causing IDA* to suffer from the problem of quadratic re-expansions with respect to the expansions in its last iteration. This occurs when IDA* expands only one new node at each iteration (Helmert, Lattimore, Lelis, Orseau, & Sturtevant, 2019). This could also place LSTM and NEAR at a disadvantage compared to other heuristic functions.

We conjecture that any system that does not account for the size of the IDA* search tree will not generalize strongly, as MSE might be a poor proxy for the tree size. Although BUS accounts for the right loss, it lacks search guidance and only synthesizes small programs, which result in weak heuristics. The intuition for NEUROSKETCHER's success is that it uses the guidance of the neural networks to learn sketches whose holes are easy to fill, as the sketches are in "promising regions" of the programmatic space.

5.2.1 Analysis of $h^{\mathbf{AG}}$ for the Pancake Puzzle

Due to the programmatic representation of our heuristics, we can formally analyze them. We prove that one of our best performing heuristics (NS3 in Figure 5.2) in terms of number of expansions to solve large instances of the Pancake puzzle is bounded suboptimal. We call it the absolute bounded gap (h^{AG}):

```
sum(map(min(5,(2 * (abs(var1 - var2)))),
                    state, neighbor(state, 1)).
```

This heuristic function sums the absolute difference between adjacent tokens multiplied by 2 and bounded by 5. For example, the h^{AG}-value for state $[5, 2, 4, 1, 3]$ is computed as follows: $\min(5, 2 \times |5 - 2|) + \min(5, 2 \times |2 - 4|) + \min(5, 2 \times |1 - 4|) + \min(5, 2 \times |1 - 3|) + \min(5, 2 \times |3 - 5|) = 5 + 4 + 5 + 4 + 4 = 22$. We can show the following about h^{AG}.

Property 1. *For any state n we have $h^{AG}(n) < 5(h^*(n) + N + 1)$, where N is the number of tokens in a Pancake puzzle.*

Proof. Let $G(n)$ be the number of gaps in the state n, i.e., the number of adjacent tokens in n that differ by more than 1. Also, $n[i]$ be the i-th token in n and the first token is at index 0. $G(n)$ also counts the difference between $n[0]$ and $n[N-1]$ as a possible gap. Since each action in the Pancake puzzle can fix one gap and $G(n)$ counts the gap between the first and last tokens, we have

$G(n) - 1 \leq h^*(n)$. We also have $h^{\text{AG}}(n) \leq 5G(n) + 2N < 5(G(n) + N)$ because the value $h^{\text{AG}}(n)$ adds for each gap is at most 5 and for each non-gap position is 2. Rewriting the last inequality, we obtain

$$G(n) > \frac{h^{\text{AG}}(n)}{5} - N \implies h^*(n) > \frac{h^{\text{AG}}(n)}{5} - N - 1\,,$$

which gives $h^{\text{AG}}(n) < 5(h^*(n) + N + 1)$. □

Since $N \approx h^*(n)$ for many instances, this property guarantees that IDA* searching with h^{AG} does not return solutions for states n with a cost much higher than $10 \times h^*(n)$. The GAP heuristic multiplied by 10 gives a similar property and, as we showed in Figure 5.2 for 90-Pancake, IDA* with h^{AG} solved all 1,000 instances with at most 12,000 expansions per instance, while IDA* with GAP multiplied by 10 did not solve any instances with the same number of expansions.

5.2.2 Analysis of h^{OG} for the Topspin Puzzle

We also prove that one of our best performing heuristics (NS5 in Figure 5.2) in terms of number of expansions to solve large instances of the Topspin puzzle is bounded suboptimal. We call it the ordered gap (h^{OG}) because it penalizes the state based on the number of pairs of tokens that are unordered:[1]

```
sum(map(min(5, abs(var2 % (-1 - var1))
                state, neighbor(state, 1)).
```

This heuristic function sums the absolute value of the modulo between each of the i-th token and the negative value of the $(i-1)$-th token added to 1; this operation is bounded by 5. For example, the h^{OG}-value for state $[5, 2, 4, 1, 3]$ is computed as follows:

$$\begin{aligned}
&\min(5, \text{abs}(5\%(-1-3))) + \min(5, \text{abs}(2\%(-1-5)))+ \\
&\min(5, \text{abs}(4\%(-1-2))) + \min(5, \text{abs}(1\%(-1-4)))+ \\
&\min(5, \text{abs}(3\%(-1-1))) = 3+4+4+4+1 = 16
\end{aligned}$$

We have the following about h^{OG}.

[1]The number of unordered tokens is different from the number of gaps, as defined in the GAP heuristic. For example, the number of gaps in $[5, 4, 3, 2, 1]$ is one, which is given by the difference between token 1 and an imaginary table. The number of unordered tokens is five, one for each adjacent token, including 1 and 5.

Property 2. *For any state n we have $h^{OG}(n) \leq 10 \cdot h^*(n) + 5$, where N is the number of tokens in a $(N, 4)$-Topspin.*

Proof. Let $G(n)$ be the number of unordered pairs in the state n, i.e., the number of adjacent tokens in n that does not resemble their final order in goal state. Since each action in the Topspin puzzle can fix at most 2 unordered pairs, we have $G(n) \leq 2h^*(n)$. We also have $h^{\mathrm{OG}}(n) \leq 5(G(n) + 1)$ because $h^{\mathrm{OG}}(n)$ evaluates to at most 5 for each unordered pair and 0 for each ordered pair except for the pair of tokens $[N, 1]$. For every ordered pair (x, y), we have $y = x + 1$, except for the pair $[N, 1]$. Therefore $h^{\mathrm{OG}}(n)$ evaluates each ordered pair to 0 as shown below

$$\begin{aligned} |y\%(-1-x)| &= |y\%(-1-(y-1))| \\ &= |y\%(-1-y+1)| \\ &= |y\%(-y)| = 0 \end{aligned}$$

Rewriting the last inequality, we obtain

$$G(n) \geq \frac{h^{\mathrm{OG}}(n)}{5} - 5 \implies 2h^*(n) \geq \frac{h^{\mathrm{OG}}(n)}{5} - 5\,,$$

which gives $h^{\mathrm{OG}}(n) \leq 10 \cdot h^*(n) + 5$. □

Chapter 6

Conclusions

In this dissertation, we presented NeuroSketcher, a two-step search algorithm that uses neural networks to fill the holes of programs as a means of guiding the search in programmatic spaces even for non-differentiable programs and loss functions. The first search uses an abstraction of the language where the non-differentiable operations are replaced with neural networks, thus resulting in a language of differentiable programs. This first search must optimize for a differentiable loss function. The program the first search returns can be seen as a sketch, as one creates holes in it by removing its neural models. The holes are filled in a second uninformed search in the original programmatic space. Since the second search does not use neural networks, it can use any loss function, even non-differentiable ones. NeuroSketcher synthesized heuristic functions that outperformed some of the best heuristics known for the Pancake and Topspin puzzles, as well as the heuristics learned with the baseline systems. Finally, we showed how we can prove that some of these heuristics are bounded-suboptimal.

References

Aguas, J. S., Jiménez, S., & Jonsson, A. (2018). Computing hierarchical finite state controllers with classical planning. *Journal of Artificial Intelligence Research*, *62*, 755–797.

Albarghouthi, A., Gulwani, S., & Kincaid, Z. (2013). Recursive program synthesis. In *International Conference Computer Aided Verification, CAV*, pp. 934–950.

Alur, R., Radhakrishna, A., & Udupa, A. (2017). Scaling enumerative program synthesis via divide and conquer. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems conference*, pp. 319–336. Springer Berlin Heidelberg.

Ameen, S., & Lelis, L. H. (2023). Program synthesis with best-first bottom-up search. *Journal of Artificial Intelligence Research*.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., & Tarlow, D. (2016). Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*.

Barke, S., Peleg, H., & Polikarpova, N. (2020). Just-in-time learning for bottom-up enumerative synthesis. *Proceedings of the ACM on Programming Languages*, *4*(OOPSLA), 1–29.

Bastani, O., Pu, Y., & Solar-Lezama, A. (2018). Verifiable reinforcement learning via policy extraction. In *Proceedings of the International Conference on Neural Information Processing Systems*, pp. 2499–2509. Curran Associates Inc.

Bonet, B., Palacios, H., & Geffner, H. (2010). Automatic derivation of finite-state machines for behavior control. In *Proceedings of the AAAI Conference on Artificial Intelligence*, p. 1656–1659. AAAI Press.

Bulitko, V., Wang, S., Stevens, J., & Lelis, L. H. (2022). Portability and explainability of synthesized formula-based heuristics. In *Proceedings of the International Symposium on Combinatorial Search*, Vol. 15, pp. 29–37.

Culberson, J. C., & Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, *14*(3), 318–334.

Deville, Y., & Lau, K.-K. (1994). Logic program synthesis. *The Journal of Logic Programming*, *19-20*, 321–350. Special Issue: Ten Years of Logic Programming.

Ellis, K., Ritchie, D., Solar-Lezama, A., & Tenenbaum, J. (2018). Learning to infer graphics programs from hand-drawn images. In *Advances in Neural Information Processing Systems*, pp. 6059–6068.

Ellis, K., Wong, C., Nye, M. I., Sablé-Meyer, M., Cary, L., Morales, L., Hewitt, L. B., Solar-Lezama, A., & Tenenbaum, J. B. (2020). Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *CoRR*, *abs/2006.08381*.

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Vol. 9 of *Proceedings of Machine Learning Research*, pp. 249–256. PMLR.

Gulwani, S. (2011). Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, *46*(1), 317–330.

Gulwani, S., & Venkatesan, R. (2009). Component based synthesis applied to bitvector circuits. Tech. rep. MSR-TR-2010-12.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, *4*(2), 100–107.

Helmert, M. (2010). Landmark heuristics for the pancake problem. In *Proceedings of the International Symposium on Combinatorial Search*.

Helmert, M., Lattimore, T., Lelis, L. H. S., Orseau, L., & Sturtevant, N. R. (2019). Iterative budgeted exponential search. *CoRR*, *abs/1907.13062*.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735–1780.

Hu, Y., & De Giacomo, G. (2013). A generic technique for synthesizing bounded finite-state controllers. *Proceedings of the International Conference on Automated Planning and Scheduling*, *23*(1), 109–116.

Jabbari Arfaee, S., Zilles, S., & Holte, R. C. (2011). Learning heuristic functions for large state spaces. *Artificial Intelligence*, *175*(16), 2075–2098.

Kingma, D., & Ba, J. (2015). Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, San Diega, CA, USA.

Kodratoff, Y., Franova, M., & Partridge, D. (1990). Logic programming and program synthesis. In *Systems Integration '90. Proceedings of the First International Conference on Systems Integration*, pp. 346–355.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, *27*(1), 97–109.

Lee, W., Heo, K., Alur, R., & Naik, M. (2018a). Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 436–449. Association for Computing Machinery.

Lee, W., Heo, K., Alur, R., & Naik, M. (2018b). Accelerating search-based program synthesis using learned probabilistic models. *ACM SIGPLAN Notices*, *53*(4), 436–449.

Lippi, M., Ernandes, M., & Felner, A. (2016). Optimally solving permutation sorting problems with efficient partial expansion bidirectional heuristic search. *AI Communications*, *29*(4), 513–536.

Liu, G.-T., Hu, E.-P., Cheng, P.-J., yi Lee, H., & Sun, S.-H. (2023). Hierarchical programmatic reinforcement learning via learning to compose programs..

Medeiros, L. C., Aleixo, D. S., & Lelis, L. H. S. (2022). What can we learn even from the weakest? Learning sketches for programmatic strategies. In *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI Press.

Nye, M., Hewitt, L., Tenenbaum, J., & Solar-Lezama, A. (2019). Learning to infer program sketches. In Chaudhuri, K., & Salakhutdinov, R. (Eds.), *Proceedings of the 36th International Conference on Machine Learning*, Vol. 97 of *Proceedings of Machine Learning Research*, pp. 4861–4870. PMLR.

Odena, A., Shi, K., Bieber, D., Singh, R., Sutton, C., & Dai, H. (2021). BUSTLE: Bottom-up program synthesis through learning-guided exploration. In *International Conference on Learning Representations*.

Qian, X. (1990). Synthesizing database transactions. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, p. 552–565, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Qian, X. (1993). The deductive synthesis of database transactions. *ACM Trans. Database Syst.*, *18*(4), 626–677.

Shah, A., Zhan, E., Sun, J. J., Verma, A., Yue, Y., & Chaudhuri, S. (2020). Learning differentiable programs with admissible neural heuristics..

Shi, K., Dai, H., Ellis, K., & Sutton, C. (2022). Crossbeam: Learning to search in bottom-up program synthesis. In *International Conference on Learning Representations*.

Solar-Lezama, A. (2009). The sketching approach to program synthesis. In *APLAS*.

Solar-Lezama, A., Rabbah, R., Bodík, R., & Ebcioğlu, K. (2005). Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, p. 281–294, New York, NY, USA. Association for Computing Machinery.

Trivedi, D., Zhang, J., Sun, S.-H., & Lim, J. J. (2022). Learning to synthesize programs as interpretable and generalizable policies..

Udupa, A., Raghavan, A., Deshmukh, J. V., Mador-Haim, S., Martin, M. M., & Alur, R. (2013). Transit: specifying protocols with concolic snippets. *ACM SIGPLAN Notices*, *48*(6), 287–296.

Valkov, L., Chaudhari, D., Srivastava, A., Sutton, C. A., & Chaudhuri, S. (2018). Houdini: lifelong learning as program synthesis. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 8701–8712.

Wilbur, E. (2001). Topspin: Solvability of sliding number games. *Rose-Hulman Undergraduate Mathematics Journal*, *2*.

Wilt, C., & Ruml, W. (2016). Effective heuristics for suboptimal best-first search. *The Journal of Artificial Intelligence Research*, *57*, 274–306. 2016.

Young, H., Bastani, O., & Naik, M. (2019). Learning neurosymbolic generative models via program synthesis. In *International Conference on Machine Learning (ICML)*.

Appendix

A LSTM Implementation Details

The LSTM uses a fully connected unit with two layers of 128 units with ReLU activation functions. The weights are randomly initialized with the method of Glorot and Bengio (2010) and the model is trained with Adam's optimizer (Kingma & Ba, 2015) with a learning rate of 0.001 for 20 epochs. Similarly to the neurosymbolic models trained in the first search of NeuroSketcher, we use the numbers from 1 to N as input, as a one-hot encoding would require a variable size for puzzles of different sizes.

B Near with Multiple Seeds

We trained the Near program with 5 different random seeds to study the effect of weight initialization on the performance of Near in guiding the IDA* with 1000 random instances of 14-Pancake and (14,4)-Topspin puzzles and no substantial deviation in performance is observed, as shown in Figure 6.1. The standard deviation is almost zero, as the error bands do not appear around the line representing Near.

C Study on the Input Encoding for Neural Models

We trained the LSTM and FeedForward (NN) networks on Pancake data of size 10 and used them to guide IDA* on 1000 random instances of 10-Pancake with integer data input (LSTM, NN) as well as one-hot encoding of the training data (LSTM_E, NN_E). From Figure 6.2, we note that LSTM performed better with the integer representation compared to the one-hot encoding representation. This possibly happens because the input size of the one-hot encoding is longer, making it more difficult for the model to learn. For example, the 3-Pancake state $[2, 3, 1]$, which is given in the integer representation used in our experiments in the dissertation, can be represented as $[0, 1, 0, 1, 0, 0, 0, 0, 1]$ with a one-hot encoding. The one-hot encoding is a factor N longer than

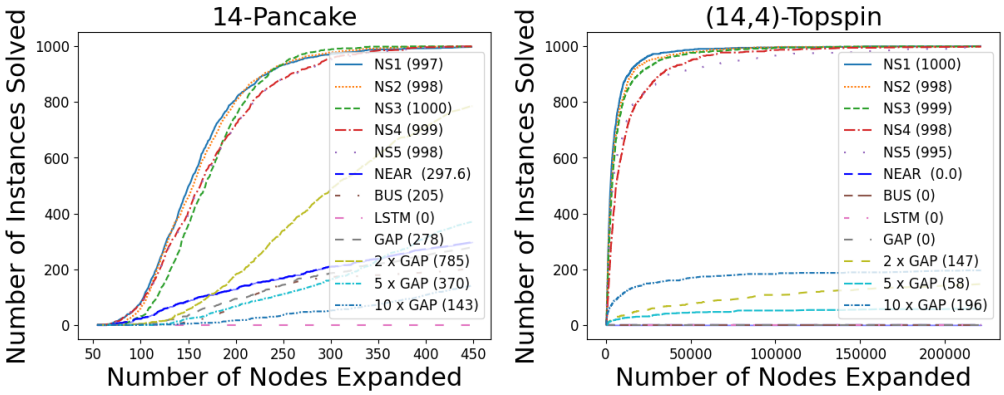

Figure 6.1: Study with multiple seeds for NEAR.

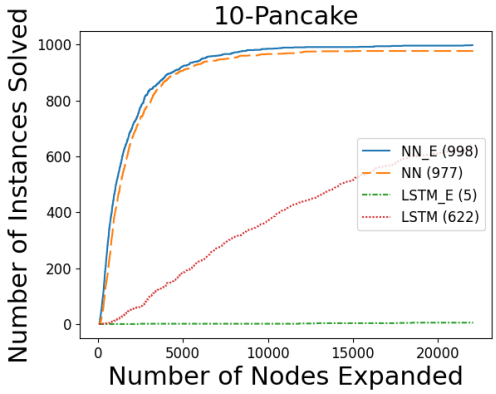

Figure 6.2: Data Representation Study

our integer encoding. These results justified our choice of data representation for our experiments. Although NN_E performed slightly better than NN, NN cannot be used in our experiments because we learn from a small input size and attempt to generalize to larger input sizes, and NN cannot deal with a change in input size.