Combining the Smallest Subset of Programs from Enumerative Search with Decision Trees

Filip Molnár[1]

Supervisor(s): Sebastijan Dumančić[1]**, Reuben Gardos Reid**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**
TU Delft, Van Mourik Broekmanweg 6, 2628 XE Delft, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 26, 2022

Name of the student: Filip Molnár
Final project course: CSE3000 Research Project
Thesis committee: Sebastijan Dumančić, Reuben Gardos Reid, Soham Chakraborty

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

Program synthesis aims to automatically generate programs that fulfil user-specified constraints. The field has developed many different techniques to enable program synthesis in various application domains. This work will focus on the enumerative approach, which iteratively explores solutions in the program space from the smallest programs to larger ones. This technique works well on small to medium-sized problems. With the exponentially growing program space performance of the enumerative solver rapidly declines.

This work aims to generalize the divide and conquer approach to combine partial solutions found by an enumerative solver. Firstly by forming the smallest subset that satisfies every example in a problem and then using decision trees to find the final program that satisfies all input-output examples. Our prototype named SubsetDT will be incorporated into the Julia library Herb.jl which provides a flexible environment for testing and developing program synthesis ideas.

This methodology was evaluated on SyGuS competition benchmarks from two different tracks. Results indicate that the use of a decision tree on top of enumeration search improves solver performance by 33% on the string manipulation track and significantly on the bit-vector manipulation track, though both tracks showed some overfitting. The experiments demonstrate that the SubsetDT algorithm can substantially improve enumeration solvers, its effectiveness highly depends on the iterators used.

1 Introduction

The ultimate goal of program synthesis is to write an algorithm that can automatically find programs that satisfy a user intent specified by some constraints. One of the first applications of program synthesis was a feature incorporated into Microsoft Excel named FlashFill[4]. FlashFill can automatically propose suggestions on syntactic string transformations and fill the rest of the data once a user specifies a few examples. Even for a simple example like extracting a first name and a surname from the email (”john.doe@mail.com” $\rightarrow$ ”john doe”), there exist millions of programs that could satisfy it. The user's intention is usually unknown to the synthesizer making it even more difficult to find the correct program. Moreover, the program space grows exponentially with the program size, making brute-force search infeasible for complex tasks. Despite challenges such as exponentially large program space and ambiguity of user intent, the field of program synthesis has developed many different techniques that enable program synthesis in different real-life application domains[5].

Learning larger or recursive programs is still a significant challenge. However, recent advancements showed promise in the speed of learning complex programs. Alur et al. [2] used decision trees to combine multiple solutions learned on smaller sets of Input-Output examples. Their prototype EUSolver placed in top positions in multiple tracks in the 2016 SyGuS competition[1]. Similarly, the Divide, Constrain and Conquer approach by Cropper[3] was proposed to address the limitations of disability of learning larger and recursive programs. The key idea was to reuse the knowledge from solving smaller programs and thus prune the search space.

Most of the research done in the area of program synthesis specializes in exploring unique and novel algorithms. To do so, researchers often have to write their implementations of program synthesizers from scratch despite similar concepts already being implemented in other projects. The Julia library Herb.jl[1] provides an adaptable environment to explore program synthesis while allowing to test and develop new ideas easily.

This work aims to generalize the divide-and-conquer approach inspired by previous studies[3, 2] and implement the core concept in Herb.jl. More specifically we will focus on extracting the smallest subset of programs which are later combined with decision trees to form one program that supposedly works on all examples in a problem. The divide-and-conquer approach aims to overcome the scalability issues posed by the exponentially growing number of programs with size by splitting the grammar and enumerating the programs while also enumerating boolean and conditional expressions on the side. Once the programs are enumerated they can be combined into one.

The following two questions were formulated to answer the main research question.

- How does a decision tree algorithm improve the basic enumeration solver when it is run after partial programs are generated?
- What is the algorithm's performance on various enumerated programs and predicates?

By doing so, we seek to identify key factors that impact performance and uncover opportunities for optimization in program synthesis.

2 Background and Problem Description

2.1 Programming by Example

In the Programming by Example (PBE) task, we have a set of IO examples(Fig. 1). The goal is to find one final program that matches each example output with the program's output when run on its corresponding input.[6] Usually, the problem comes with context-free grammar to constrain the search space. The domain-specific language (DSL) in Fig.1. is restricted such that only a limited number of expressions can be produced. For example, $x == y$ cannot be produced unless it is constructed from multiple other rules $x \leq y \wedge y \leq x$.

```
S ::= T | if (C) then T else T
T ::= 0 | 1 | x | y | T + T
C ::= T ≤ T | C ∧ C | ¬ C
```

Figure 1: Grammar for linear integer expressions. Figure taken from Alur et al.[2]

[1] https://herb-ai.github.io/Herb.jl/dev/

x	y	output ($max(x, y)$)
1	2	2
3	0	3
-3	0	0

Table 1: Input-Output examples, where x and y are arguments served as an input and output is a basic $max(x, y)$ function.

2.2 Enumerative solver

One of the basic methods for solving a program synthesis task is to enumerate every possible expression in a given grammar and verify it on the examples. That is how an enumerative solver works. It stops only when it finds a perfect solution that satisfies all the examples or when the maximal number of enumerations is reached. In that case, it returns the next best program, which does not fully satisfy all IO examples.

The pseudo-code is depicted in Algorithm 1. At every enumeration, the algorithm proposes a candidate_program generated from the grammar G. Consequently, the function `SATISFIES(candidate_program, IO)` (line 4) is called which returns the percentage of IO examples the `candidate_program` satisfies. Score 1 indicates that every example has been satisfied and in that case, the `candidate_program` is returned as it satisfies all of them. Otherwise, the `best_program` is saved(line 8-9). Once the maximal enumerations are reached the current `best_program` is returned.

The enumerative solving technique is a straightforward but often powerful tool. It effectively solves smaller problems, however, it does not scale well. In our prototype, we make use of a breadth-first enumerator which explores smaller solutions first, thus when it finds a solution that solves every example it is guaranteed that it is the smallest possible program.

2.3 Decision Trees in Related Work

To effectively prune the search space various techniques have been employed. Alur et al.[2] introduced a divide-and-conquer approach in their prototype EUSolver where the grammar is split into two: term grammar, and predicate grammar. Term grammar generates all possible programs of the specified return type. Predicate grammar is responsible for generating boolean expressions (predicates) that can be used as conditional statements for combining the programs. The main idea was to enumerate the grammars separately and combine them with the predicates using decision trees at each enumeration. If the decision tree cannot be constructed more terms and predicates are generated. EUSolver won the SyGuS competition in the programing by example track in 2016[1].

Drawing inspiration from the original idea by Alur et al.[2], we will focus on reusing the smallest number of partial solutions found by the enumerative search and combining them using decision trees.

Algorithm 1: Enumerative solver

Input: Grammar G = $\langle N, S \rangle$
Input: IO_examples IO
Parameter: max_enumerations = ∞
Output: Program expression

```
1:  best_score ← 0
2:  best_program ← NULL
3:  while (i, candidate_program) ∈ ENUMERATE(G) do
4:      score ← SATISFIES(candidate_program, IO)
5:      if score == 1 then
6:          return candidate_program
7:      else if score > best_score then
8:          best_program ← candidate_program
9:          best_score ← score
10:     end if
11:     if i > max_enumerations then
12:         break
13:     end if
14: end while
15: return best_program
```

Our colleague Tudor Andrei[2] is exploring in his work a different approach utilizing the divide-and-conquer technique for synthesizing programs. His work combines multiple programs where individual programs are learned on each example, rather than taking the smallest subset from a larger set of programs. We both test the quality and performance of our approaches on the same set of benchmarks for better comparison.

3 Implementation

The goal of our prototype is to combine the partial solutions produced by any iterator into one program. From the partial solutions, the smallest subset satisfying all examples is selected. For combining the partial solutions we use the divide-and-conquer approach that splits the grammar into terms and predicates from which the predicates are used to fit the decision tree. The idea is to use the decision tree once after the programs are enumerated and the smallest subset is formed.

Therefore, our approach consists of two parts: Finding the smallest subset of programs that satisfies all examples and combining the programs using decision trees. Both parts are standalone and can be used separately or on top of other approaches.

3.1 Finding the smallest subset

We enumerate all possible programs with a specified iterator. The number of maximum enumerations and the maximal enumerating time can all be specified beforehand. Every synthesized program is then run on given examples to test its satisfiability.

- A program satisfies all examples. We return the program.
- A program satisfies only some subset of the examples. We add this program to candidate programs from which we later choose the smallest subset.

[2]His work is not published yet.

- A program does not satisfy any of the examples. We continue iterating.

The algorithm ensures that no two programs satisfy the same subset of examples. There is no need to store two programs that cover the same subset, thus we store the smaller program, which was enumerated first.

The enumeration stops once the number of enumerations exceeds the specified maximum number of enumerations or the time for the search runs out. The smallest subset from the candidate programs is returned.

The set cover problem is known to be an NP-hard problem. For our needs, an approximation algorithm is enough. The greedy algorithm offers a good enough approximation where the solution is at most $log(n)$ times larger than the optimal solution while running in polynomial time. Programs are chosen using only one rule. At each enumeration, the program that covers the largest number of uncovered examples is added to the set. The smallest subset is then passed to the second part where the partial programs are combined into one whole solution.

3.2 Combining the programs

In order to combine two programs together our algorithm requires a starting symbol (bool_symbol) responsible for generating boolean expressions. The algorithm can generate valid predicates from the grammar. If such a symbol is not provided as a parameter, two generic rules are added to the grammar (line 1). These rules are $Start == 1$ and $Start == 0$, where $Start$ indicates start_symbol for generating expressions. These rules were only added for the bit-vector manipulation track as the grammar does not contain bool expression rules. We will discuss this in the section about experimental setup and results. The if-then-else conditional operator is essential for combining the programs. If such a rule is not contained in the grammar we add it manually or just remember its index in the grammar (line 2).

Predicates are generated using an enumerative solver with a starting bool_symbol (line 4). Many generated predicates are considered useless i.e. $1 == 0$ or $arg\_1 \leq arg\_1$. Despite both of them being valid predicates they will always be evaluated as either true or false regardless of the input arguments. We decided to keep only the predicates that contain any argument.

The decision tree by JuliaAI[3] was used as it offers a broad range of functionality and is easy to implement. To train a decision tree (line 6) two arguments must be passed to a model; feature vector matrix and labels.

Feature vectors are created by evaluating an input from the IO examples on predicate expressions. Each predicate returns a boolean value which is collected into a feature vector. This process is done for every IO example, where we end up with a matrix of size $|IO\_examples| \times$ *n_predicates* (line 5). The decision tree uses the classical information gain heuristic to learn which predicate reveals the most information about the labels[2].

Partial programs serve as **labels** (line 3). Every IO example can satisfied by one or multiple programs. This problem thus became a multi-label decision tree classification problem. We assigned unique labels to IO examples corresponding to a unique set of indexed programs. In other words, the label "1-2" indicates that programs 1 and 2 satisfy the given example whereas the label "1" can be satisfied by only program at index 1. By labelling like this, we can isolate examples satisfied by the same set of programs and allow the decision tree to tell those apart from others.

Later, when reconstructing the program (line 7), we can prune the decision tree by merging leaves if they can be satisfied by the same programs. Finally, the reconstructed program is returned.

To demonstrate the efficacy of our method, we next present our experimental setup and results, providing insights into the performance and program quality of our approach.

Algorithm 2: Decision Tree for combining the programs

Input: Grammar G = $\langle N, S \rangle$
Input: IO_examples IO
Input: programs
Input: start_symbol, bool_symbol (recommended)
Parameter: Number of predicates $n = 4096$
Output: Program expression, Grammar

1: G ← ADD_RULES(G)
2: idx ← FIND_COND_RULE(G)
3: labels ← GET_LABELS(programs, IO)
4: predicates ← GET_PREDICATES(G, n, bool_symbol)
5: features ← FEATURES(G, IO, predicates)
6: model = FIT_DT(features, labels)
7: **return** CONSTRUCT_FINAL(model, programs, idx, predicates)

4 Experimental Setup and Results

Our prototype named SubsetDT was evaluated on two different PBE tracks: string transformation benchmark from the SyGuS PBE SLIA Track in 2019 and bit-vector manipulation benchmarks from the PBE BV Track in 2018. Both benchmark tracks are rewritten to Julia programming language using Herb.jl and can be found at HerbBenchmarks github repository[4].

From the SLIA track all 100 problems were used to test our prototype. In the BV track, however, we split the problem set into two and used only 317 problems that have 100 or fewer examples. The rest of the problems (151) having 1000 examples were not used as evaluating them takes much longer and is inefficient in practice.

Since grammars in the BV track do not contain specific boolean rules used to generate predicates, we decided to add two to every grammar: $Start == 1$ and $Start == 0$, where $Start$ indicates start_symbol for generating expressions and if-then-else conditional operator rule for combining partial programs. These rules can be easily accessed by us to generate predicates and combine the programs.

[3] https://github.com/JuliaAI/DecisionTree.jl

[4] https://github.com/Herb-AI/HerbBenchmarks.jl/

Although the grammar already contains similar rules like *if0_cvc* and *im_cvc*, they would be much harder to access and use. *if0_cvc* and *im_cvc* take three $Start$ symbols as arguments and check if the first one equals zero or one, respectively and output the corresponding expression if the condition is satisfied.

All experiments were run on a computer with the following specifications: 6 core 2.6 GHz Intel i7-10750H CPU and 16 GB of RAM. For better comparisons between machines, we decided to run the basic enumeration solver and predicate generator for the decision tree on the number of enumerations rather than time limits. Programs were enumerated for 100, 500, 1000, 10k, 20k, 40k, and 100k after which the subset was formed and predicates were generated with 100, 500, 1000, 8k and 24k enumerations.

Goals: We aimed to evaluate how the SubsetDT compares to the basic enumeration solver in performance on two benchmark datasets from SyGuS. In other words, we are trying to answer the question:

How much can a decision tree help with combining partial solutions when it is run on top of the enumeration solver?

Secondly, we compare the number of predicates that are needed to successfully combine partial solutions.

Finally, we test the quality of the obtained solutions.

4.1 Discussion:

Tables 2 and 3 illustrate the number of solved benchmarks from the two SyGuS tracks by an Enumeration solver and SubsetDT. To interpret the results, the second column corresponds to the performance of a pure Enumeration solver while columns ranging from third to sixth depict the performance when additional predicates expressions were generated and used to combine the partial programs. The last column, marked as NS, shows the number of problems which could not be solved despite the partial solutions satisfying all the examples.

Effect of the number of predicates generated: Table 2 shows that generating just 100 and 500 predicates had the biggest impact on the performance, solving a large majority of problems for which the smallest subset was found. The rest of the problems where a subset of partial programs covered all examples could be solved with the 8000 predicates. The problem (*problem_count_total_words_in_a_cell*), despite covering all the examples could not be combined into one even with 24000 predicates.

Figure 2 illustrates the combined program produced by the SubsetDT that uses 3 subprograms and 2 predicates. This problem required in total of 1000 enumerations whereas the enumeration solver could not solve it within 500k enumerations. Although this example program overfits the IO-examples it shows the efficiency of this approach.

For the BV track the results are quite different as indicated in the table 3. The number of problems solved by SubsetDT is much higher than by enumeration solver. Around 25% of the problems with full coverage of examples could be combined with 500 predicates. With 8000 predicates around 68% problems which could not be solved by enumerative

SLIA	ES	SubsetDT					NS
#Enum	0	100	500	1000	8000	24000	-
100	11	+9	+0	+0	+1	+0	0
500	17	+11	+1	+0	+2	+0	0
1000	19	+12	+1	+0	+2	+0	0
10000	24	+9	+1	+0	+3	+0	1
20000	26	+9	+2	+0	+3	+0	1
40000	29	+7	+3	+0	+3	+0	1
100000	30	+7	+3	+0	+3	+0	1

Table 2: Number of SLIA benchmarks solved (from 100 in total) by Enumeration solver (ES), and an improvement on the solved benchmarks when SubsetDT is used. The first column indicates the base number of enumerations from which a minimal subset was formed. The first row indicates how many predicates were used in SubsetDT. Not solved problems are shown in the last column marked as NS.

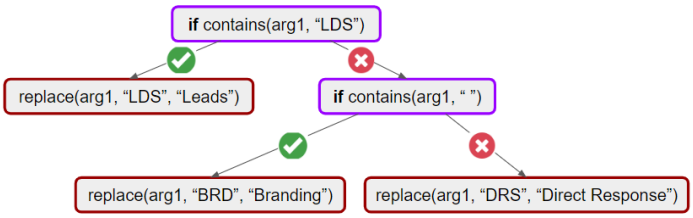

Figure 2: Final program after combining three subprograms (red) using two predicates (purple).

solver were solved by the SubsetDT. There are only 4 problems which cannot be solved with 24000 predicates.

We suspect that the large number of predicates needed to solve a problem compared to the SLIA track is because the boolean rules introduced by us into the grammar are much simpler, compared to the rules from the SLIA track. Thus the algorithm needs to find a complicated predicate under which it can combine partial programs. Moreover, a larger number of predicates may be required since the BV track contains problems with many more IO examples (up to 100). For this reason, we think that both the enumerative solver and SubsetDT with a smaller number of predicates struggle to find one complete solution and with 8000+ predicates SubsetDT overfits the examples.

The bottleneck in the computation time of the decision tree algorithm is its predicate and feature generation. During feature generation, every IO example needs to be evaluated on every predicate. Computation time linearly depends on the number of examples per problem and predicates used. For this reason, we decided not to evaluate SubsetDT on the 150 programs with 1000 IO examples. Despite 1000 examples per problem being rare in program synthesis, when it happens it slows down the entire algorithm.

Performance: We evaluated the performance of SubsetDT compared to the basic enumeration solver by splitting the number of enumerations that go to term and predicate generation. Enumerations were split under ratios of 4:1, 3:2, and 1:1 and on the same benchmarks. Obtained results from

BV	ES	SubsetDT					NS
#Enum	0	100	500	1000	8000	24000	-
100	6	+0	+11	+0	+24	+22	2
500	10	+0	+40	+0	+60	+49	2
1000	10	+0	+50	+0	+71	+54	2
10000	15	+0	+69	+0	+126	+74	4
20000	16	+0	+71	+0	+123	+76	4
40000	19	+0	+73	+0	+121	+75	4
100000	19	+0	+73	+0	+121	+75	4

Table 3: Number of BV benchmarks solved (from 317 in total where the length of examples is at most 100) by Enumeration solver (ES), and improvement on the solved benchmarks when SubsetDT is used.

running the performance experiments are illustrated in the figure 3.

We can see that sacrificing some enumerations for a decision tree on top of the enumeration solver improved the performance of solving the benchmarks on average by 33% for the SLIA track when the maximum of 100,000 enumerations is reached. There is not a big difference in performance when comparing different ratios. The 4:1 ratio of terms to predicates solved one to two more problems than other ratios at every enumeration checkpoint while it struggles when total enumerations are small. There is always the tradeoff of either enumerating more programs or predicates. For some problems, more predicates are needed to successfully combine the partial programs. After all, the predicates only serve to combine the solutions. If the subset of partial programs that satisfy all examples cannot be found any number of predicates will not suffice.

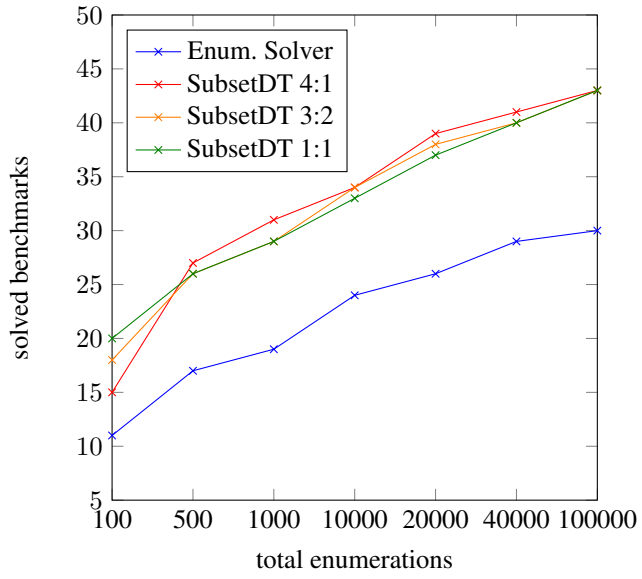

Figure 3: Performance comparison of SubsetDT and enumeration solver with different number of enumerations.

Figure 4. illustrates the performance of our prototype on the BV track. With a set number of enumerations and different ratios for generating programs and predicates proposed our prototype managed to solve at most 292 problems out of 317. However, a close analysis of the final combined programs showed that solutions overfitted the examples.

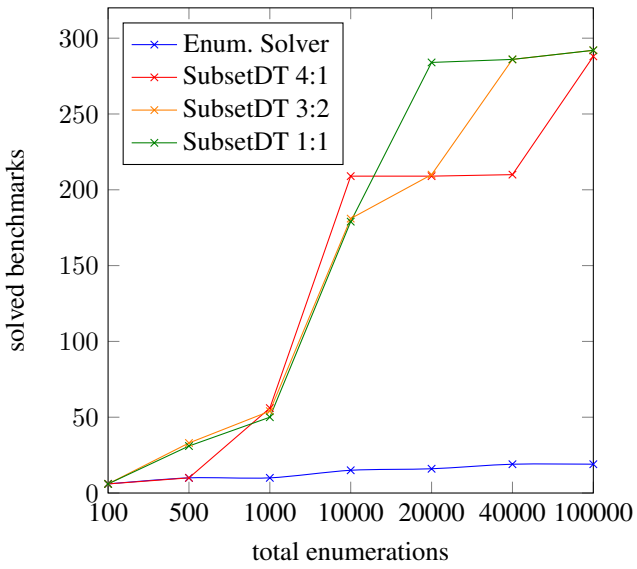

Figure 4: Performance comparison of SubsetDT and enumeration solver with different number of enumerations.

Quality comparison of the solutions can be seen in figure 5. The enumerative solver only found solutions for 15 problems from both the SLIA and BV tracks, where the solution was first found by SubsetDT. This allowed us to compare 8 solutions from the SLIA track and 7 from the BV track. The results clearly show that the SubsetDT produces larger programs. For SLIA track the solution sizes for SubsetDT are about 1.5 times larger than the programs found by enumeration solver. This is because usually two partial solutions of a slightly smaller size than the smallest program found by ES were merged by a predicate.

In the BV track, however, the solution sizes of SubsetDT are about 3 times larger on average despite most of the problems requiring only two programs to satisfy all examples. Meaning that the combined programs overfitted more on the examples and used two splits for two programs.

4.2 Responsible Research

We believe that this work complies with the Netherlands Code of Conduct for Academic Practice.

Reproducibility is an important part of every research. The algorithms used for finding the smallest subset of programs and combining them using decision trees are described in detail. By providing pseudo-code and comprehensive descriptions, we enable other researchers to replicate our methods. Furthermore, the source code is available in the GitHub repository of Herb.jl[5].

[5]https://gitlab.ewi.tudelft.nl/cse3000/2023-2024-q4/Dumancic_Reid/fmolnar-Symphonic-Synthesis-Learning-

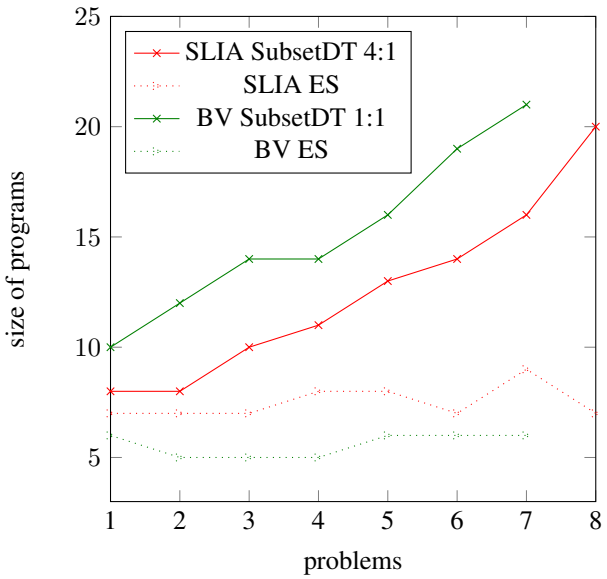

Figure 5: Length of solutions produced by enumerative solver (ES) and SubsetDT 4:1 and 1:1 evaluated on SLIA and BV track, respectively.

All parameters, such as the number of maximum enumerations, maximum enumerating time, and rules added to the grammar, are specified. This ensures that other researchers can replicate our experimental setup precisely.

We utilized the decision tree implementation from an open-source library JuliaAI, which is publicly available on GitHub[6]. The use of open-source tools provides easy access to the same resources used in our research.

SyGuS data sets were taken HerbBenchmarks on 14.5.2024[7]. Detailed documentation of the experimental setup, including the hardware and software environment, is provided.

4.3 Conclusions and Future Work

In this work, we looked at how decision trees can improve the enumeration solver and the quality of the programs they produce. We presented an algorithm that combines the smallest subset of partial solutions into one program to solve the whole problem. The prototype was tested on various benchmarks from past SyGuS competitions and compared to the enumeration solver.

From our experiments, we found that the number of predicates needed to combine programs varies. For the string transformation track the number is relatively small and with a little overhead of generating more predicate expressions it solved 33% more problems. Whereas the bit-vector manipulation track requires many more predicates and produces twice as large programs than programs in the SLIA track.

The SubsetDT is highly dependent on the iterator used in the first place. If the programs it generates cannot satisfy the examples, the decision tree algorithm on top does not help as it cannot create new solutions.

Our decision tree algorithm for combining programs is standalone. It can be used to combine any number of partial programs when the rule for generating boolean expressions is provided. In the future, exploring a generalised predicate grammar that could be applied to every problem could be useful. However, there are limitations to how generalised it can be. Adding more rules would mean that more predicates would need to be enumerated and it would decrease the overall performance. Furthermore, various iterators for both term and predicate generation can be explored.

To conclude, the SubsetDT provides a substantial improvement when it is used as a layer on top of the enumeration solver. For now, it only works for problems containing a predicate subgrammar and a rule for generating those expressions is required.

programs-by-composin

[6]https://github.com/JuliaAI/DecisionTree.jl

[7]https://github.com/Herb-AI/HerbBenchmarks.jl/tree/dev/src/data/SyGuS

References

[1] Alur, R.; Fisman, D.; Singh, R.; and Solar-Lezama, A. 2016. SyGuS-Comp 2016: Results and Analysis. *arXiv (Cornell University)*, 229: 178–202.

[2] Alur, R.; Radhakrishna, A.; and Udupa, A. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In Legay, A.; and Margaria, T., eds., *Tools and Algorithms for the Construction and Analysis of Systems*, 319–336. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-662-54577-5.

[3] Cropper, A. 2021. Learning logic programs through divide, constrain, and conquer. *arXiv (Cornell University)*.

[4] Gulwani, S. 2011. Automating string processing in spreadsheets using input-output examples. *ACM SIGPLAN Notices*, 46: 317.

[5] Gulwani, S.; Polozov, O.; and Singh, R. 2017. *Program Synthesis*.

[6] Odena, A.; Shi, K.; Bieber, D.; Singh, R.; Sutton, C.; and Dai, H. 2021. Published as a conference paper at ICLR 2021 BUSTLE: BOTTOM-UP PROGRAM SYNTHESIS THROUGH LEARNING-GUIDED EXPLORATION.