CS515 Final Project - Implementing ESCHER in Python

Abigail Calvelli

Rutgers University - Department of Computer Science

December 01, 2021

1 Abstract

ESCHER algorithm is a powerful algorithm which uses input-output examples to synthesize recursive programs. ESCHER adopts an explicit search strategy and creates programs in a bottom-up fashion. The goal of this paper is to implement ESCHER in Python and identify areas of improvement to solidify my understanding of program synthesis and the under workings of this algorithm. I choose ESCHER as my algorithm for multiple reasons. First, ESCHER's use of input-output examples enables non-technical end-users to extend the range of achievable computational tasks. Second, ESCHER is generic and efficient. The algorithm is parameterized by its components, allowing it to be domain agnostic, and alternates between two steps which allow it to search the program space and generate conditionals to improve this search time. Lastly ESCHER has been proven to out-perform SKETCH, another state-of-the-art SAT-based synthesis by leveraging a comprehensive data structure, the goal graph, and the cleverly alternating forward search and conditional inference steps, all of which will be explored in detail later [1].

2 Introduction

My implementation of ESCHER in the Python programming language is guided by the 'Recursive Program Synthesis' article seen in class by Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. The most useful aspects of the paper have been the rules for function definitions and the diagram of the goal graph data structure which make up the logic of the program. A shortcoming of the article I found is its failure to comment on the inefficiencies of the *SATURATE* rule which I elaborate more on later.

$$\frac{}{\langle \mathit{Vars}, (\{\{\mathit{root}\}, \emptyset, \emptyset, \mathit{root}), \mathit{Ex}\rangle}\ \textsc{Init}$$

$$\frac{c \in \mathit{Comps} \quad P_1 \in \mathsf{syn} \quad \cdots \quad P_{\mathsf{a}(c)} \in \mathsf{syn} \\ P = c(P_1, \ldots, P_{\mathsf{a}(c)})}{\langle \mathsf{syn}, \mathsf{goalGraph}, \mathsf{ex}\rangle \to \langle \mathsf{syn} \cup \{P\}, \mathsf{goalGraph}, \mathsf{ex}\rangle}\ \textsc{Forward}$$

$$\frac{\mathit{cond} \in \mathbb{B}^{|\mathsf{ex}|} \quad g \in G \quad r \text{ is fresh} \\ \mathit{bthen} = g|\mathit{cond} \quad \mathit{belse} = g|\neg\mathit{cond} \\ G' = G \cup \{\mathit{cond}, \mathit{bthen}, \mathit{belse}\} \quad R' = R \cup \{r\} \\ E' = E \cup \{(r,g), (\mathit{cond}, r), (\mathit{bthen}, r), (\mathit{belse}, r)\}}{\langle \mathsf{syn}, (G, R, E, \mathit{root}), \mathsf{ex}\rangle \to \langle \mathsf{syn}, (G', R', E', \mathit{root}), \mathsf{ex}\rangle}\ \textsc{SplitGoal}$$

$$\frac{P_1, P_2, P_3 \in \mathsf{syn} \quad r \in R \quad (r, g_1), (r, g_2), (r, g_3) \in E \\ \mathit{match}(\mathsf{eval}_v(P_1), g_1) \quad \mathit{match}(\mathsf{eval}_v(P_2), g_2) \quad \mathit{match}(\mathsf{eval}_v(P_3), g_3) \\ P = \texttt{if } P_1 \texttt{ then } P_2 \texttt{ else } P_3}{\langle \mathsf{syn}, (G, R, E, \mathit{root}), \mathsf{ex}\rangle \to \langle \mathsf{syn} \cup \{P\}, (G, R, E, \mathit{root}), \mathsf{ex}\rangle}\ \textsc{Resolve}$$

$$\frac{P \in \mathsf{syn} \quad \mathit{match}(\mathsf{eval}_v(P), \mathit{root}) \quad \mathsf{ex}{\downarrow_P} \not\subseteq \mathsf{ex}}{\langle \mathsf{syn}, (G, R, E, \mathit{root}), \mathsf{ex}\rangle \to \langle \mathsf{syn}, (\{\mathit{root}'\}, \emptyset, \emptyset, \mathit{root}'), \mathsf{ex} \cup \mathsf{ex}{\downarrow_P}\rangle}\ \textsc{Saturate}$$

$$\frac{P \in \mathsf{syn} \quad \mathit{match}(\mathsf{eval}_v(P), \mathit{root}) \quad \mathsf{ex}{\downarrow_P} \subseteq \mathsf{ex}}{\langle \mathsf{syn}, (G, R, E, \mathit{root}), \mathsf{ex}\rangle \to P}\ \textsc{Terminate}$$

Fig. 3. Escher synthesis algorithm

Figure 1: Rule Definitions

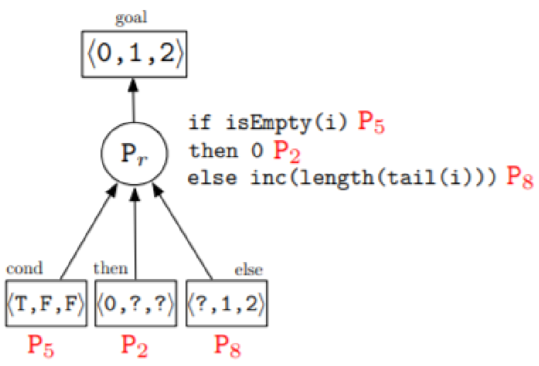


Figure 2: Sample Goal Graph Configuration

The Python implementation of ESCHER is evaluated against similar benchmarks found in the paper. If the Oracle, simulating user evaluation, is given a recursive program and its desired functionality, can

ESCHER synthesize the recursive program solving this task? Ideally, I would have increased the number of programs used as benchmarks, to better analyze the range of the programs ESCHER can handle, and graph them against the performance found in the paper for the OCaml implementation of ESCHER. The programs my implementation of ESCHER does look to synthesize are *Length()* to compute the length of a list mentioned in the paper, *Fib()* to compute the nth fibonacci number, and *Factorial()* to compute the factorial of n. These examples were chosen as they test functionality across multiple domains and test the rules of the program in different ways, for instance, *Fib()* requires multiple saturated examples depending on its set of original input-output examples rather than one saturated example for the *Length()* program.

2.1 Constraints and Search Space

ESCHER assumes an explicit search strategy based on input-output examples. We define the constraints of the algorithm as follows:

- **Behavioural Constraints:** The behavioural constraints are ESCHER are the input-output examples provided by the user and the *SATURATE* rule.
- **Structural Constraints:** The structural constraints of ESCHER are the components, grammar rules, it is parameterized by. For instance, when synthesizing the *Length()* program, the structural constraints are [INC, EMPTY, TAIL, ZERO, SELF, LENGTH]. Because ESCHER utilizes bottom-up enumeration, we combine sub-programs into larger programs using these production rules.

These constraints mean that the Search Space of ESCHER is quite large: it consists of all the possible combinations of productions which increases exponentially. This requires ESCHER to prune the space, through observational equivalence and prioritize the search by exploring more promising candidates via a heuristic function. Both of these concepts will be discussed in more depth throughout the report.

2.2 Project Goals

The limitations of the discussions in the paper created the main goals I had for this project:

1. **Apply program synthesis on a domain other than integer manipulation:** Our exploration of program synthesis in class, mainly via project 2, performed program manipulation on the domain of integers. Since ESCHER synthesizes recursive functions and it is domain agnostic, I chose to add program synthesis of lists to my final project by adding different AST nodes to support that functionality.
2. **Identify areas to improve the efficiency of ESCHER:** The other goal I had for ESCHER would be exploring ways to improve the efficiency of the algorithm. The two most impactful areas I pinpointed for improving efficiency are in the heuristic function and partially evaluating the output vectors of recursive terms in an attempt to minimize the number of additional examples the Oracle must provide. Both of these points will be discussed in further detail later.

2.3 Novelties

We now look at some of the new ideas implemented in this project from the article. The first novelty of this final project is its implementation in Python. The paper discusses techniques to implement ESCHER in OCaml. I chose Python to implement ESCHER for its useful data structures which drive the algorithm and its ability to graph its performance directly from the source code. I made use of the dictionary, lists, and tuple data structures to visualize the input and output examples and implement the goal graph data structure for conditional inference.

The second novelty of my implementation of ESCHER is the computation and implementation of a new heuristic function to guide the explicit search. ESCHER is biased towards synthesizing smaller, less expensive programs first. This greatly improves efficiency and allows the search to avoid synthesizing trivial programs which satisfy only the given input-output examples rather than returning a program representative of the recursive function itself. A popular heuristic employed by ESCHER is to count the number of AST nodes in the synthesized program and synthesize programs iteratively (ex: at iteration 1, programs of size 1 are synthesized and checked, at iteration 3 programs with 3 nodes and so on). While this heuristic is sufficient in synthesizing smaller programs first, I believe it can be improved to make the program more efficient. The different heuristics, their reasoning, and the plot of their performance are examined in the heuristic section of this paper. s

3 Implementation

The main goal of ESCHER is to synthesize a program which satisfies the root goal, the set of outputs from the original and saturated examples provided. Before examining the implementation of ESCHER, we define and illustrate the building blocks of the algorithm.

3.1 Terminology

- **ESCHER Instance:** we define a configuration of ESCHER As a triple $[syn, goalGraph, ex]$
  - **syn:** is a list of synthesized programs
  - **goalGraph:** is the goalGraph
  - **ex:** is a list of input-output examples with **Ex** the original set of input-output examples
- **Components:** we define c to be a component of the list of AST nodes introduced shortly
  - **self:** we define *self* to be a special component which is treated as a recursive call to the synthesized program p
  - **err:** we define *err* to be a special symbol indicating an error value, mainly used in determining if a recursive program terminates or computing undefined functionality such as the tail of an empty list
- **arity (a):** we define the arity to be the number of input parameters a component c takes. For instance, the $a(tail) = 1$, a list. $a(self) = thenumberofinputparametersintheexamplesofEx$

3.2 AST Nodes

We now observe the components and their corresponding AST nodes which can be used in this implementation of ESCHER.

- **Boolean Components:** True, False, NonNeg, Empty, Lt
- **List Components:** Tail, Concat
- **Integer Components:** Num, Var, Zero, Inc, Dec, Neg, Length, Plus, Times, Fib, Fact, Ite

3.3 Evaluation

In order to verify the synthesized program satisfies the original outputs, we define an *eval()* function which takes a component c from the lists above and an input to evaluate the results of both non-recursive and recursive programs. We define two slightly different implementations of *eval* to account for the types of program we are synthesizing. We use (in_i, out_i) to denote the i^{th} input-output example $\in$ Ex, $v[j]$ to denote the j^{th} element of a value vector v and define $eval_v$ as follows

- **Non-Recursive Programs:** we use the function *interpret* defined in project 2 to evaluate the value vector of programs of different component types such that

$$eval_v(x_j)[i] = \text{in}_i\text{[j]}$$
$$eval_v(c)[i] = \text{eval(c,())}$$
$$eval_v(c(P_1, ..., P_n))[i] = eval(c, (eval_v(P_1)[i], ..., eval_v(P_n)[i]))$$
$$eval_v(if P_{cond}\ then\ P_{then}\ else\ P_{else}[i] = \begin{cases} eval_v(P_{then}[i] & \text{if } eval_v(P_{cond}[i] = T \\ eval_v(P_{else}[i] & \text{if } eval_v(P_{cond}[i] = F \\ err & \text{otherwise} \end{cases}$$

- **Recursive Programs:** we utilize a similar $eval_v$ function to evaluate recursive functions but introduce a termination argument. We exemplify this necessity with the following example:

  $a = [[], [2], [1, 2]]$
  $length(a) \rightarrow \{err, err, err\}$ because the three argument lists to $length(a)$ are the same as the original input vector a. On the other hand..
  $length(tail(a)) \rightarrow \{err, 0, 1\}$ since the input vector to length $tail(a) \rightarrow (err, [], [2])$ is smaller in size than the original input vector ([],[2],[1,2])

  Thus, the $eval_v$ function for recursive programs can be defined as

$$eval_v(self(P_1, ...P_{a(self)}))[i] = \begin{cases} eval(self, i) & \text{if self terminates} \\ err & \text{otherwise} \end{cases}$$

3.4 Goal Graph

Now that we can evaluate the output of synthesized programs, we observe data structure driving the logic of ESCHER: the **goalGraph**. I represent the goal graph as a tuple comprised of four components and is used to guide the conditional inference portion of ESCHER which combines programs into if-then-else clauses to satisfy goals. The four components are as shown:

- **G:** a set of value vectors representing goals (outputs) that need to be satisfied
- **R:** a set of resolvers connecting goals to subgoals. Each resolver $r \in R$ consists of an outgoing edge $(r, g) \in E$ and three incoming edges (g_1, r), (g_2, r), (g_3, r) denoting the *cond, then,* and *else* goals that must be satisfied to resolve g. We say that g_1, g_2, g_3 are subgoals of r and g.
- **E:** a set of edges connecting goals to resolvers
- **root** $\in$ **G:** as the distinguised root goal (the value vector of outputs for the input-output examples in Ex)

3.5 Algorithm

With the building blocks of the algorithm formally defined, we look into implementing ESCHER with Python code. The rule definitions from the paper describe the functionality of each part of the ESCHER, I move forward to discuss the ordering of these rules in my implementation of the algorithm.

3.5.1 Rule Scheduling:

1. **INITIALIZE:** we begin the program by calling the initialize rule which creates the first configuration of ESCHER where **syn** = the list of variables and nullary components (Zero(), True(), False(), etc.), a **goalGraph** initialized to ([original outputs], $\varnothing, \varnothing, root$).
2. **Forward Search:** we then apply forward search to synthesize programs having $size = current\_Iteration$. For s $\in$ syn, we apply each of the components to the appropriate typed programs (ex: the component $Tail()$ can only be applied on programs of type $List, Concat, Tail$), and add those to the list of synthesized programs with a heuristic value equivalent to the iteration. Note that the forward search utilizes observational equivalence to exclude any programs which evaluate to the same values of another program already in the set of synthesized programs in an attempt to minimize the set of synthesized programs for efficiency purposes. Then, for each synthesized program, we apply:
3. **Saturate and Terminate:** let s represent the current synthesized program. if $type(s) = type(recursive_{program})$ and s satisfies the root goal ($TERMINATES$), we apply $SATURATE$ to verify that s satisfies all intermediate recursive calls. We illustrate the necessity of the $SATURATE$ rule with the following example: Let inputs $i = [[], [1, 2]]$, outputs $o = [0, 2]$, and program P =

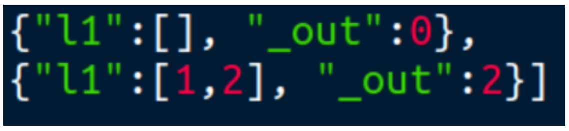
```
if isEmpty(i) then 0
else if isEmpty(tail(i)) then 0
     else inc(length(tail(i)))
```

Observe that P satisfies the given input-output examples, however P is NOT a valid program for synthesizing the *Length()* of a program.

$$Length([1,2] \rightarrow Inc(Length(Tail([1,2]))) \rightarrow Inc(Length([2])) \rightarrow Inc(0) \rightarrow 1 \tag{1}$$

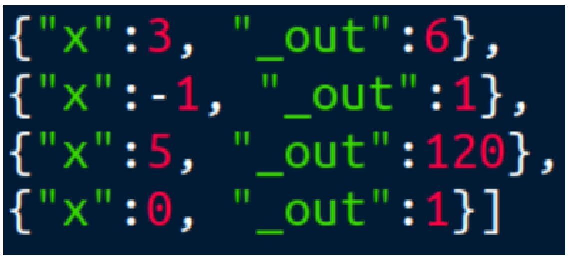
However, when evaluating the correctness of the program, ESCHER incorrectly returns program P as a match for the root goal since recursive evaluation of programs uses the Oracle to evaluate recursive calls, rather than the synthesized program, as required since ESCHER constructs programs in a bottom-up fashions, so $eval_v$ must evaluate **self** before a candidate solution is fully constructed. Thus, the Oracle incorrectly matches Equation(1) with $eval_v(Length(Tail(i)) \rightarrow Length([2]) \rightarrow 1$

To prevent this, we implement the *SATURATE* rule to add "dependent examples" the program is obligated to prove. In the above example, we add the input-out pair $ex_p = \{”i” : [2], ”{}_o ut” :, 1\}$ created from the recursive call on $Length(Tail([1,2]))$ which correctly disproves P as a candidate since $eval_v(P, [2]) \rightarrow 0$ when the Oracle determines it should evaluate to 1. Therefore, P does not satisfy the root goals. The program is then rebooted by creating a new configuration of ESCHER comprised of $[syn, goalGraph, Ex \cup ex_p]$ to accurately expand the root goal and goal graph. The *SATURATE* rule guarantees that if a program is correct on a saturated example set, it is guaranteed to be correct for all examples. The saturated example sets generated by the Python implementation of ESCHER are shown below for the $Length()$ and $Fib()$ programs.



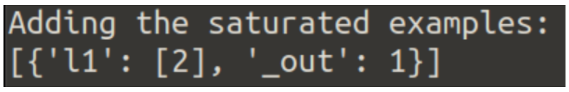
```
{"l1":[], "_out":0},
{"l1":[1,2], "_out":2}]
```

Initial Ex Length



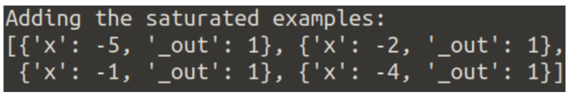
```
{"x":3, "_out":6},
{"x":-1, "_out":1},
{"x":5, "_out":120},
{"x":0, "_out":1}]
```

Initial Ex Fib



```
Adding the saturated examples:
[{'l1': [2], '_out': 1}]
```

Saturated Ex Length



```
Adding the saturated examples:
[{'x': -5, '_out': 1}, {'x': -2, '_out': 1},
 {'x': -1, '_out': 1}, {'x': -4, '_out': 1}]
```

Saturated Ex Fib

4. **Resolve:** if s does not *TERMINATE*, we apply resolve to close as many goals as possible by implementing the conditional inference aspect of the algorithm. *RESOLVE* searches through the list of resolvers and attempts to find three programs $P_1, P_2, P_3 \in syn$ such that each program satisfies a subgoal $(P_{cond}, P_{then}, P_{else})$ for the given resolver. If three programs exist, *RESOLVE* synthesizes the new program $P = \ if\ P_1\ then\ P_2\ else\ P_3$ and appends it to syn. If the subgoals of P satisfy the root goal of the program, then ESCHER has successfully synthesized a program satisfying the synthesis task.

5. **Split Goal:** lastly, we apply *SPLITGOAL* if P matches some positions of an open goal. For instance, if $P = Zero()$ and open goal $g = [0, 1, 2]$, then we apply *SPLITGOAL* to compute the following

vectors $P_{cond} = [T, F, F], P_{then} = [0, ?, ?], andP_{else} = [?, 1, 2]$ where a "?" indicates an arbitrary term. Thus, we append these new goals to **G** , and the appropriate resolver **r** to **R**, and edges to **E** in our **goalGraph**.

While the order in which rules are applied does not have much room for flexibility, it logically makes sense to apply them in this order, it can be beneficial to schedule rules such that more some methods are rarely applied. For instance, the $SATURATE$ rule is expensive as it requires new input-output pairs to be added and an entire reboot of the program, starting from iteration 1 and re-synthesizing all the programs to check them on the saturated example set and recompute their value vectors for the added examples. While the location of the $SATURATE$ rule within ESCHER may not be able to change, we can bias the search away from this rule by assigning a high heuristic value to programs requiring saturation.

3.5.2 Heuristic Formula

As previously mentioned, the explicit search strategy employed by ESCHER allows us to bias our search towards more promising candidates. A typical heuristic for ESCHER counts the number of nodes present in a program. While this simple heuristic does bias the search to smaller, more generic programs, it can be improved. In an attempt to find the best performing heuristic, I plot varying heuristic functions and their performance.

- **H1 (Baseline): Heuristic = Number of AST Nodes** For H1, the typical heuristic, we compute the heuristic as the number of AST Nodes. If $prog = Var("x")$ then the corresponding heuristic value = 1. If nesting occurs such as in $prog = Dec(Dec(Var("x")))$ are associated with the value 3.
- **H2 and H3: Heuristic = Number of AST Nodes + Penalty** We introduce a penalty to the heuristic if nesting programs appear in that which we are synthesizing. The conditional which checks this is $if(type(p) == type(prog))$, thus, programs such as $prog = Dec(Dec(Var("x")))$ will trip this penalty but programs such as $progDec(Var("x"))$ will not.
  - **H2** $Penalty = a(prog) * a(prog)$**:** For H2, I chose to compute the penalty as $a(prog) * a(prog)$. For instance, if $prog = Dec(Dec(Var("x")))$ then the penalty is computed as $1 * 1 = 1$ whereas if $prog = Plus(Plus(p, p))$, the penalty is computed as $(2 * 2) = 4$.
  - **H3** $Penalty = a(prog)*hmap[p]$**:** For H3, I chose to compute the penalty as $a(prog)*hmap[p]$. For instance, if $p = Plus(Dec(Var("x")), Num(5))$ was computed in a previous iteration, it will hold the heuristic value of 4 as no penalty is applied due to nesting terms of the same type, thus the value $= number\ AST\ Nodes$. If $prog$ is synthesized as $prog = Plus(Plus(Dec(Var("x")), Num(5)), Num(2))$, the new heuristic value is computed as $2 * 4 = 8$ as we established the previous value for $p = 8$. Thus, this heuristic greatly penalizes the nesting of the $Plus()$ terms.

My reasoning for introducing a penalty comes directly from observations about the types of programs that ESCHER synthesizes. These programs are often very simple. Thus, combinations of nested terms will be unlikely to appear in the final synthesized program and are penalized to bias the search away from these programs. We observe the graphs below: *For consistency purposes, we run all the heuristic trials on the Fact() function as the average time to run this program with the baseline heuristic (counting the number of AST Nodes) is between 30-40 seconds. We run ten trials for each heuristic and compute the average line of the trials for direct comparison. We refer to p as the current previously synthesized program used to synthesize a new program prog. We refer to hmap as the dictionary which stores the previously computed heuristic value of p if it exists.*

3.6 Reasoning

- why use observational equivalence - why chose the data structure for the goal graph - why chose the saturate rule / how to implement it (talk about rebooting)

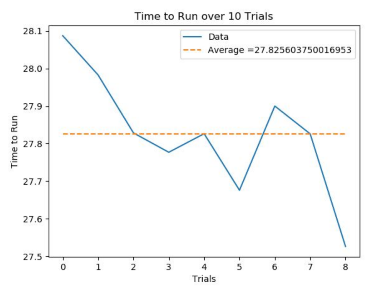


Figure 3: H1 (Baseline)

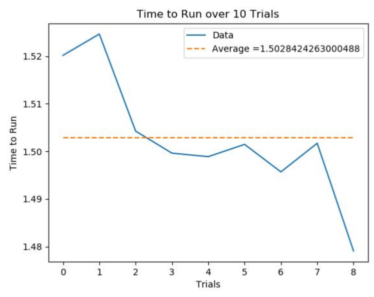


Figure 4: H2

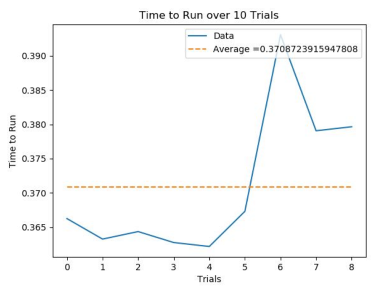


Figure 5: H3

4 Shortcomings

Due to the time constraints and man-power of a single person, there are some shortcomings with my implementation of ESCHER.

- **Small Number of Benchmarks Tested:** As mentioned early, I only found the time to test my program on a few programs. Ideally, I would like to extend the analysis to consider other recursive functions such as reversing a list, removing duplicate elements from a list, checking whether an element exists in a list, getting all nodes at a specific level of a binary tree, and other functional programs which utilize recursion.
- **Improving Implementation for Programs with Multiple Recursive Calls** During implementation, the synthesis of the recursive program for Fibonacci took a much longer time than that to find the length of a list. I predict this is from the double recursive call, used to compute the recursive step, and the nesting, calling $Fib(n-1) + Fib(n-1-1)$ which is penalized heavily by the heuristic. Since the assigned heuristic value is so high for computing the recursive piece of the function, ESCHER must synthesize all the programs, which will be failures, in between the iteration $Fib(x-1-1)$ is found and the assigned heuristic value of $Fib(x-1) + Fib(x-1-1)$ which is computed as 13. Alternatively, if we change the heuristic function to only count the number of AST nodes in a program P, there are significantly more nodes that have to be traversed at each iteration, so it still takes a significant amount of time to synthesize the Fibonacci program.
  - I propose one way to improve the efficiency in a program such as Fibonacci would be to have the heuristic learn from the *SPLITGOAL* function. *SPLITGOAL* adds a new resolver when a program matches some positions of an open goal. Some programs will be closer to the intended outputs and will consequently have more *True* values in its boolean P_{cond} vector. If the heuristic learned the patterns of those with more satisfying values, it may be able to bias the search towards programs with similar functionality. For instance, if the component $Plus()$ yields better results than a component such as $Dec()$, then the search will favour synthesizing terms with a $Plus()$ components before unnecessarily increasing the length of **syn** with $Dec()$ components that do not advance us towards a program satisfying the root goal.
- **Inefficiencies** Another shortcoming of my implementation of ESCHER is its inefficiencies in *FORWARD SEARCH* to synthesize a new program which satisfies the heuristic value and to *RESOLVE* open goals and synthesize an if-then-else program. One way to improve the resolver inefficiency would be to store the value vector and program as a key,value pair in a dictionary since that structure has O(1) search time, but that then requires creating and updating that structure which I have yet to implement. To address the forward search inefficiencies, currently the *FORWARD SEARCH* traverses **syn** once to split programs into lists of their correct evaluation types (Boolean, numerical, lists), and then for each component **c** that the configuration of ESCHER is parameterized by, all the valid programs are traversed, synthesized, and then their heuristic values are checked, observational equivalence is checked (causing another pass of **syn**), and then the program is added to the list of synthesized programs if it passes all the checks. The problem with this implementation is illustrated with an example:

- *Example:* Suppose ESCHER is on iteration 6, thus it is looking to synthesize programs of size 6. Suppose we have some programs that are of heuristic value 2. The current implementation of *FORWARD SEARCH* has those programs still be computing, then checks their corresponding heuristic values when previous iterations have already determined this program will not satisfy the heuristic check because $2 <$ 3,4,5,6, etc.. Some possible solutions to this that I can think of would be adding additional constraints to filter the terms that are being combined to synthesize a new program so that only viable pairs are considered. For instance, if *Plus()* is being synthesized, only consider programs P_1 and P_2 whose heuristic values sum to $iteration - 1$. I have yet to determine how to do this efficiently, but implementing partial evaluation via top-down propagation could be a promising route.

5 Learnings

Implementing ESCHER in Python required me to solidify my understanding of program synthesis to implement an algorithm which works for recursive programs. MY biggest takeaways are as follows:

- **Having a good heuristic dictates the success of your algorithm** Not only does a good heuristic greatly improve the efficiency of a program, it steers the algorithm away from finding trivial programs which satisfy the given examples but not the actual synthesis task. I have realized that finding a heuristic which improves all types of functions can be a hard balance to strike as recursive programs ESCHER aims to synthesize have different characteristics (such as a single or double recursive call in its solution). This leads me to believe that some sort of artificial intelligence in the heuristic formula which learns the behaviour of a program will be most successful.

- **Synthesizing recursive functions requires additional validations** This learning comes specifically from the *SATURATE* rule which I would not have thought to implement without reading this paper [1]. Understanding the need for this *SATURATE* rule required thorough knowledge of how bottom-up synthesis works as this required the recursive call to **self** must be evaluated before the candidate program P is constructed which introduces some logical flaws without implementing the rule. Computing the saturated solution set also gave me a better understanding of how the recursive calls were executing. The hardest part of this program was familiarizing myself enough with the algorithm to understand why steps were being applied in certain ways and how a recursive calls was actually going to be performed. Ensuring that the set of input-output examples contained the subsequent recursive calls (such as $ex_p = \{([2], 1)\}$ on original input [1,2]), helped me visualize and confirm the correctness of the recursive implementation.

6 Conclusion

Overall, my implementation of ESCHER in Python is by no means perfect, but the algorithm is a great example of bottom-up propagation and explicit search for program synthesis. Now that I am familiar with the program, I plan to continue working on my implementation and leveraging powerful Python structures and capabilities to improve the performance of my ESCHER implementation. Since recursive programs are a confusing concept for some people to grasp, I believe ESCHER's ability to generate recursive programs given input-output examples provides an opportunity to be implemented in some sort of auto-grader or learning tool for younger programmers who know the functionality they want to see but are unsure of how to construct the program which achieves this. Overall, the generic nature and elegance of the algorithm provides the opportunity to alter and optimize ESCHER for future synthesis goals such as synthesizing programs with loops or biasing the heuristic specifically towards more complex programs.

References

[1] Aws A. Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. *Microsoft Research*, pages 1–17, 2021.