University of Alberta
Computing Science Department
Program Synthesis in XAI - W21

Bottom-Up Search
Levi Lelis
levi.lelis@ualberta.ca

1 Bottom-Up Search

Let us consider the following DSL as our running example.

$$S \rightarrow x \mid y \mid 1 \mid (S + S)$$

Bottom-up search (BUS) keeps a set of programs that is used to form novel programs. Initially BUS initializes its set of programs with the terminal symbols of the the language. In our running example: $x, y, 1$. Then, BUS iteratively uses this set of programs and the rules of the DSL to generate a larger set of programs. For example, in the second iteration, after generating $x, y, 1, 2$, BUS will use the rule $(S + S)$ with the programs generated in previous iterations to produce other programs such as $(x + x)$, $(x + y)$, $(1 + x)$.

The table below shows the execution of three iterations of BUS.

Iteration	Set of Programs
1	x, y, 1
2	$x + y$, $x + x$, $y + y$, $1 + x$, $y + 1$
3	$x + y + x$, $x + y + 1$, $x + y + y$, $\cdots$

The number of programs can grow very from one iteration to the next. This is because there is a large number of ways one can combine existing programs into novel programs with rules provided by the DSL. One common approach for reducing the number of programs generated during search is by eliminating equivalent programs (e.g., $x + y$ is equivalent to $y + x$, so we don't need consider both while generating new programs). The problem is that it can be hard to prove that two programs are equivalent. In practice we use a weaker notion of equivalence. We consider two programs p and p' to be equivalent if they return the same value for all input-output pairs provided to the synthesizer. For example, if we have the following input-output pairs, where x and y are the inputs and *out* is the desired output,

$$x = 1, y = 0, out = 2$$
$$x = 1, y = 1, out = 3\,,$$

then the programs x and 1 are equivalent because they both produce the same output for all pairs.

We will implement a `grow` function that receives a set of programs (all programs synthesized thus far) and combines them using the rules of the DSL. In our example above we assumed that the `grow` function will generate all possible combinations of programs from one iteration to the next. In practice, however, this function generates all programs that match a given metric. For example, at iteration 1 the function generates all programs with height 1 (their ASTs have height 1). At iteration 2 it will use the programs with height 1 to generate programs with height 2 (all programs with larger heights are discarded).

Another metric that is used is program size. At iteration one we generate all programs with size 1 (their ASTs have a single node). At the next iteration we generate programs with the smallest size larger than 1. For example, if the smallest size larger than 1 is 5, then we generate all programs with size 5 and discard programs with larger size. These metrics guarantee that the first program generated that solves the problem is the program with the smallest metric value (smallest height or smallest size), which is often desirable.

The height-based growth tends to produce a larger number of programs in each iteration. Intuitively, there can be programs of various sizes whose trees are of a fixed height. The size-based growth tends to produce fewer programs in each iteration. A recent work by Barke, Peleg, and Polikarpova[1] suggests that a size-based metric tends to produces better empirical results than the height-based metric.

The pseudocode for BUS is presented below.

```
1 def BUS(CFG G, Specification T, bound d)
2   output = set() # used to verify weak equivalence
3   plist = init_terminals(G) # initialize list plist with the programs with size 1
4   number_eval = 0
5   for i in range(d): # searches up to bound d
6     plist = grow(plist, G, output)
7     for j in range(number_eval, len(plist)):
8       number_eval += 1
9       is_correct(plist[j], T): # true if plist[j] produces the right outputs
10        return plist[j]
11
12 def grow(plist, G, output):
13   nplist = [] # new list of programs
14   for op in G.operations: # in our running example (S + S) is an op
15     nplist.append(op.grow(plist)) # the operation knows how to grow itself
16   for p in nplist:
17     if p not equivalent(output, p): # checks for weakly equivalent programs
18       plist.append(p)
19   return plist
```

For a sufficiently large bound d, the algorithm is **complete**, i.e., it is guaranteed to find a solution if one exists. This is because, eventually, BUS enumerates all programs that can be synthesized with the DSL. The algorithm also encounters the **optimal** solution with respect to the metric used in the `grow` function. For example, if in each iteration we generate programs with the smallest size larger than the size of the ASTs in the previous iteration, then once the solution is found, the solution will have the smallest size.

In addition to being complete and optimal, BUS can be quite effective because it only produces complete programs. That is, all programs added to `plist` in the pseudocode above can be verified against the set of input-output pairs as soon as they are produced. We will see in the next lecture that top-down algorithms produce partial programs that can't easily be evaluated. The ability to quickly evaluate programs is an advantage to learning systems as we learn about the search space as soon as search begins.

One disadvantage of BUS is its expensive program generation function. BUS needs to iterate over large sets of programs to generate the next batch of programs that is added to `plist`. Another disadvantage is that BUS can produce a large number of duplicated programs, i.e., identical programs produced by different combinations of programs and rules. Let's see how duplicated programs can be avoided.

1.1 Duplicated Programs

Once we eliminate for weakly equivalent programs we already eliminate duplicated programs. This is because every duplicated program is also weakly equivalent. The disadvantage of eliminating duplicated programs through the same mechanism used to eliminate weakly equivalent programs is the mechanism's computational

[1] Just-in-Time Learning for Bottom-Up Enumerative Synthesis (2020).

cost. Recall that we compute all outputs of a program to verify for equivalence. If the number of input-output pairs is large or if the program is computationally expensive, then checking for equivalence can be expensive. By contrast, checking for duplicated programs can be much cheaper as one needs to compare the AST with existing ASTs. This procedure can be implemented efficiently with a hash table.

We can implement BUS with such a hash table that stores all programs in `plist`. That way, before checking for equivalence, we check for duplication. If the program is duplicated, there is no need to check for equivalence. It is an empirical question whether duplicate detection can speed up the synthesis process. To the best of my knowledge this was never studied in the literature and for that would be an interesting project.

2 Better Representations to Avoid Equivalent Programs

In this section I will discuss a topic that is marginally related to this lecture, which is the idea of using DSLs that reduce the number of equivalent programs generated.[2] BUS is equipped with a mechanism for detecting weakly equivalent programs, but as we also discussed above, the detection scheme can be computationally expensive. Ideally equivalent programs won't even be generated during search. The design of the DSL might contribute to reducing the number of equivalent programs generated as the example below illustrates.

Consider the following DSL, where var is a variable and num a constant.

$$S \rightarrow var * num \mid S + S$$

As we discussed in our second lecture, the shape of the AST defines where brackets would be placed in expressions generated from the DSL above. Here are a few examples of how $a * 2 + b * 2 + c * 4$ can be represented in this DSL.

$$(a * 2) + (b * 2) + (c * 4)$$
$$a * 2 + (b * 2 + (c * 4))$$
$$(a * 2 + b * 2) + (c * 4))$$
$$\dots$$

If we change the DSL to the one shown below, we have an equivalent search space, but only the second expression from the list above can be generated.

$$S \rightarrow var * num \mid var * num + (S)$$

Representation search is a research area largely neglected in program synthesis. An interesting project will investigate a search procedure that first searches for a good representation (e.g., one that minimizes the generation of equivalent programs) and then searches for a program with the chosen DSL.

[2]The example shown in this section was extracted from Lecture 2 of the MIT course 'Introduction to Program Synthesis' `http://people.csail.mit.edu/asolar/SynthesisCourse/Lecture2.htm`.