1 Task 1

In task 1, we experimented with a PCFG to find a suitable expression which can solve the given input output. The PCFG is provided in Figure 1.

$$S \rightarrow \text{“”} \mid < \mid > \mid \text{arg} \mid \text{Replace}(S, S, S) \mid \text{Concat}(S, S)$$

Fig. 1. PCFG

The input output set which we have used in our experiment is provided in Figure 2 and the probability of the production rules are provided in Figure 3. We have taken this PCFG, Input-output and the probability distribution from the article *Just-in-Time Learning for Bottom-Up Enumerative Synthesis*[1]. In the implementation, we made a small change in the grammar. Instead of the blank string ”” we have used ”*” and made necessary modification in the input output accordingly. We considered ”*” as blank string. We have kept all other information same as the aforementioned article.

Input (arg)	Output
a < 4 and a > 0	a 4 and a 0
<open and <close>	open and close
<change> <string> to <a> number	change string to a number

Fig. 2. Input-output pairs

Production Rule	Probability
arg, “”, “<”, and “>”	0.188
Replace	0.188
Concat	0.059

Fig. 3. Probability Distribution

For task 1 at first we run the bottom-up search using the probabilities of the PCFG. Then again we run the bottom-up search using uniform probability distribution on the production rule. As there are 6 production rules, each of the production rules have probability 0.166 in the uniform probability distribution. In our program implementation we have given two constraints. The constraints are given below:

1. **Concat:** Blank string shouldn't be added to any strings or any strings shouldn't be added to blank string. This won't make any change.
2. **Replace:** In Replace function there are three variables. The first variable is the string on which the replace will be done. The second variable is the string which is to be changed and the last variable which will be placed instead of second variable. In replace we put two constraints. The first variable can't be "<",">" or "" because in that case single character can be replaced with an entire string which is very unusual for our test case. Another restriction is the third variable can't be the input string. Because in that case, in a string, a single character can be replaced by the input string again which is also unlikely for our case. We also put a general constrain that is the second and third variable can't be the same. Because that won't change anything of the first string.

Table 1. Number of programs generated

Cost	With Given probability distribution	With uniform Probability
2	4	4
6	0	9
8	15	6
10	0	27
12	0	96
14	168	207
16	0	591
18	0	952
20	1594	3084
22	0	5686
24	0	15201
26	10794	29604
28	0	74986
30	0	161391
32	79905	388609
34	0	831158
36	0	1941452
38	530762	4301733

We have provided the result of the two approaches in table 1. The second column is the number of program generated with the associated cost using provided provided probability

distribution and the third column represents the number of program generated with uniform probability distribution. In the table we can see that there are some costs where there is no generated program with the given probability distribution but with uniform distribution, some programs are generated at that cost. The reason behind that is the variation in the probability of the production rules. The program which can solve all the input-output pairs is provided bellow and this expression is found by all the approaches in our experiment.:

args.replace("<", "").replace("<", "").replace("<", "").replace(">", "").replace(">", "").replace(">", "")

We have provided the total Number of programs generated, number of programs evaluated and time to find a suitable expression which can solve the input output pairs in table 2

Table 2. Total Number of programs and execution Time

Approach	Generated program	Evaluated Program	Execution time
Given Probability	623238	108543	7.315 sec
Uniform probability	7754792	3453633	192.37 sec

From table 1 and 2, we can clearly observe, grammar with probability distribution which is biased towards the solution can find a solution within 7.315 seconds where program with uniform distribution takes more than 3 minutes for the same task. The number of generated programs, evaluated programs is also much higher in case of uniform distribution. This huge number of program generation and evaluation also creates higher memory requirements. So, from this discussion, we can conclude that, using PCFG with bias towards the solution is more efficient than PCFG with uniform probability distribution in terms of required time and memory.

2 Task 2

In task 2, we again search for a suitable expression for the input output pairs provided in Figure 2 using the grammar of Figure 1. But in this case we start with uniform probability distributions over the production rules and if we find any program which can partially solve a unique combination from the input-output pairs and m satisfies the First Cheapest heuristic, we update the probability distribution of the grammar. Here we started with probability of 0.166 for each of the production rules. The number of program generated with their cost is provided in Table 3. In this table we showed a comparison with the number of program generated for starting with uniform probability and with no learning.

An important point regarding the information of the table is, for learning approach, we only showed the number of generated program with different cost after the last update of the probability distribution that is table 3 shows only the generated programs with the final probability distribution for the learning approach. In that table we can see the costs of the programs are slightly different from those with uniform probability without learning or from the programs of task 1. This happened because the probability of the operations are changed with learning thus the cost of the production rules also changed.

Table 3. Number of programs generated

Cost	With Learned probability	With uniform Probability
2	0	4
3	4	0
6	0	9
8	0	6
9	15	0
10	0	27
12	0	96
14	0	207
15	168	0
16	0	591
18	0	952
20	0	3084
21	946	0
22	0	5686
24	0	15201
26	0	29604
27	5226	0
28	0	74986
30	0	161391
32	0	388609
33	24938	0
34	0	831158
36	0	1941452
38	0	4301733
39	204994	0

For better understanding of the big picture, in Table 4, we provide the total number of programs generated, evaluated and the execution time for both the approaches. Here it is to be mentioned that in table 3, we provided the number of programs for the learning approaches only with the final probability distribution but in table 4, we have provided the total number of generated and evaluated program from the very beginning of the search.

Table 4. Total Number of programs and execution Time

Approach	Generated program	Evaluated Program	Execution time
With learning	286503	104006	5.216 sec
Without learning	7754792	3453633	192.37 sec

From table 4, we can find a very important observation. In the approach of learning the probability while running the program, when the probability distribution is updated we have to restart the search from the beginning. This sounds a very costly step. But from the output of the programs we can find that, though we have to restart the search, it can find the suitable solution much faster. With updating the probability distribution and restarting the search, a total of 286503 programs were generated and 104006 programs were evaluated and the execution time was 5.216 second. In this whole search the probability distribution was updated 3 times that means the search was restarted thrice. But in case of uniform distribution without updating probabilities and restarting the search, the number of generated program, evaluated program and execution time were much much higher. From this we can conclude that, though restarting the search is a costly procedure, but in the bigger picture it can guide the search towards a suitable solution and it requires generating smaller number of programs in less time than the search without learning.

References

1. 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.
2. Odena, Augustus, et al. "BUSTLE: Bottom-up program-Synthesis Through Learning-guided Exploration." arXiv preprint arXiv:2007.14381 (2020).