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.

3 Task 3

3.1 Question 01

In our experiment we have used a version of First cheapest heuristics to select partial programs from the list of the partial programs which can solve at least one input-output pair. In this subsection, at first, we are going to describe a heuristics that performs worse than the first cheapest heuristics. Then in the second part of this subsection, we are going to consider two more heuristics which author has mentioned in their article.

Heuristics with worse performance: One example of a heuristics which have performance worse than the First cheapest strategy is selecting all the partial problems which can solve at least one input-output pair to update the probability distribution.

If we select all the partial programs then there can be potential increase of cost in the search(here cost indicates number of generated program and time of execution). Because in that case many irrelevant programs can be selected to update the probability distribution. Hence the heuristics will increase the probability of unnecessary production rules. Though updating the probability distribution using all the partial programs will result in guiding the search in different direction, but this might not be the expectation. Because we want to

guide our search towards the most potential direction. It can be the case that a subset of the input-output pair can be solved by more than one program and among them one or more doesn't guide the search towards the final solution.

The first cheapest is superior than this heuristics of selecting all the programs, because in case of first cheapest, we are exploring the subset of the generated programs which is lower in cost. Even if the selected program doesn't guide the search towards final solution, at least it explores the potentials programs with lower cost first by increasing the probability of the selected production rule. But if we select all the programs with partial solutions, in that case, the search will be guided towards many direction as all the production rules which can solve at least one input-output pair will get rewards from this heuristics. This will increase the number of generated programs and the execution time.

In conclusion, we can say, though selecting all the programs which can solve a subset of input-output pair is able to rewards different production rules and guide the search in different direction, it can't avoid rewarding irrelevant production rules. For this reason selecting all the programs will perform worse than the first cheapest heuristics.

Other two heuristics from the Paper: The other two heuristics in the paper was selecting largest subset and selecting all cheapest. In largest subset, the program which can solve maximum number of input-output pair will be selected. Though this approach rewards the most promising production rule but it doesn't reward other promising production rules. For this reason, the search is guided towards a single direction. It can be the case that, the production rule which seems most promising initially might not provide the final solution. For this reason, this heuristics might perform worse than the first cheapest heuristics in some cases.

Another approach was to select all cheapest that means if there is a tie in the cost of programs which can solve a unique subset of input-output, select all of them. Though it guide the search towards different promising direction still this approach has the problem of rewarding irrelevant production rules. But the first cheapest heuristics maintain the balance between rewarding different production and not rewarding unnecessary production rules. For this reason, all cheapest heuristics can also have worse performance than first cheapest in different scenarios.

3.2 Question 02

Instead of using the heuristics for selecting the programs which can solve at least one input-output pairs, author could use a different approach to learn an effective probability distribution. In this section we are going to suggest an approach where all the programs who can solve at least one problem will be used to adjust the probability distribution effectively and has the potentiality to perform better than the heuristics. The proposed approach is inspired from the work of Odena, Augustus, et al.[2]

In this approach, we will start generating new programs from cost 1 towards higher cost. And after generating programs to some specific bound(can be cost or size or other metrics), we will use the input-output pairs to generate the signatures for each of the programs. This signature is also inspired from the the work of Odena, Augustus, et al.[2]. In this case, the property signature of a programs will be whether the program could solve all the input-output pairs or not. If the program can solve all the input-output pairs properly, then the signature of that program will be 1 and that will be our solution for the input-output pairs. However, If the program can solve a subset of the input-output pairs, then the signature of that program will be 0 and if the program can't solve any pair of the input-output pairs then it's signature will be -1. That means the property signature will be a vector whose length will be equal to the number of program generated.

After generating the signatures for all the program, we will search whether any program has signature 1 or not. If it is then that will be our solution. But if there is no programs with signature 1, then we will feed the list of generated program and the property signature vector to a machine learning model. The task of the model is to analyze the property signature and predict whether a production rule can be subprogram of the final program or not. The model will make this prediction based on the signature of the programs. A simple neural network model can serve this purpose. Because it will analyze the property signatures of a program and check associated production rule of that program. As the model will have all the programs, their property signatures and whether the program can solve at least one input-output pairs or not, it will be easier for the model to predict whether the final program will contain a production rule or not. Based on the prediction for the production rules, some weight for each of the production rule will be generated and the probability distribution of the production rule will be adjusted with the weights.

This approach can perform better than using heuristics that author mentioned in the paper[1]. There are several reasons behind this. One of the reason is, generating programs and then selecting from them to update the probability increases the cost in terms of time and memory. Another reason is, it is not a easy task to decide which heuristics will perform better for a task. Different heuristics can have better performance for different task. But

using all the programs together to update the probability distribution provides a solution for not going through the experiment of selecting a heuristic for a specific task.

Another problem with the heuristics is, each of the heuristics has their cons. For example, "largest subset" can select a program which can solve the maximum number of input-output pairs initially but it might be the case that this chosen program will not be able to solve all the input-output pairs. In case of "all cheapest" heuristics, many redundant programs can be selected and the heuristics will guide the search towards many unnecessary directions. In case of "first cheapest" heuristics, the search is first guided towards the cheapest programs first and if the final solution is not found, the search will be guided towards costly production rules. So, the first cheapest might be costly for some cases. But in our proposed approach, we will overcome the problems of the heuristics. Thus there will be no complexity of selecting the heuristics still the approach will be able to learn the probability distribution effectively.

From the above discussion, we can understand that our proposed approach can effectively learn the probability distribution using all the programs that can provide partial solution and this approach has the potentiality to perform better than the heuristics. So, we can conclude that, the author of the paper[1] could use this strategy to learn the probability distribution using all the programs that can solve at least one input-output pair.

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).