University of Alberta
Computing Science Department
Neuro-Symbolic Programming - W23

Assignment 2
20 Marks
Due date: 10/03 at 23:59

## Overview

In this assignment you will implement the Bee Search algorithm for solving string manipulation tasks. It is fine to discuss the assignment with your classmates, but do not share your code and answers to the questions with them. We will use Discord to discuss issues related to the assignment. **You should submit two files: a zip file with your implementation and a pdf with the answers to the questions of this assignment.**

## Dependencies

You will need Python3, Numpy, and Tensorflow installed in your machine to run the starter code.

## Implement Bee Search (5 Marks)

The starter code provides most of the implementation of Bee Search. Your task is to implement the function `search` of the Bee Search algorithm (see `bee.py`), which is invoked in the `synthesize` function, after initializing the grammar used with the trained neural model we provide with the implementation.

The signature of the `search` method is as follows.

```
def search(self, bound, string_literals_list, integer_literals_list,
           boolean_literals, string_variables_list,
           integer_variables_list):
```

The parameter `bound` specifies the number of different costs of programs Bee Search will consider during synthesis. This value is used as a proxy for the time limit of the synthesis: smaller values of `bound` will make the search terminate faster and possibly not find a solution for the problem (if the solution cost is larger than the bound). If the `bound` value is "large enough," which is the case in the starter code as it is set to infinite, then the search will run until a solution is found (or your computer runs out of memory!).

The other input parameters specify the language that should be used in search (i.e., which symbols the search will consider while attempting to find a solution to a problem). The method `search` should return a solution program (it can return `None` if it doesn't find one), the number of programs evaluated in search, and the number of times function `heapify_pqs` was called (you will use these statistics to answer the questions below).

The starter code also provides an implementation for the list of programs we will keep in memory, see class `ProgramList`. This class stores all programs encountered in search and allows for an efficient evaluation of the programs with the neural network (it accumulates all programs generated with a given cost and evaluates all programs in a single batch with the neural model). `ProgramList` has a method called `init_plist` that needs to be called before starting the search. This method initializes the list of programs with the smallest programs possible (just like how you did with BUS in the first assignment). This method also initializes the search in the cost-tuple space—this is all already implemented for you, all you need to do is to call `init_plist` in your implementation of `search`.

In the starter code, `ProgramList` stores one priority queue for each arity of operator in the language. For example, if the language is of the form $S \rightarrow \text{replace}(S, S, S) \mid \text{concat}(S, S)$, then `ProgramList` stores two priority queues, one for cost tuples with arity 3 (for the "replace" operator) and one for cost tuples with arity 2 (for the "concat" operator). In class, we have studied the version of Bee Search that uses a single priority queue for all arities; either strategy for implementing the algorithm is fine.

After initializing the list of programs, `search` will perform a number of iterations that is bounded by the value of `bound`. In each iteration, it gets the list of operations of the language that should be expanded next (i.e., all the cost tuples with cheapest cost). This list of cost tuples can be obtained with the method `get_next_cheapest` of the instance of the `ProgramList` class Bee Search uses. This method receives no parameters and returns a tuple of the form `(list, cost)`, where `list` contains all arities whose cheapest cost-tuple (the cost tuple at the top of its priority queue) has the cheapest value; `cost` stores this cheapest value.

Once the values of `list` and `cost` are obtained from `get_next_cheapest`, `search` can call the `grow` method, which is implemented as an iterator, to obtain the next batch of programs with cost `cost`:

```
for p in self.grow(list, cost):
    # check whether p is a solution to the problem
    ...
```

For verifying whether a program is correct we can use the method `is_correct` from the Bee Search class.

Once the for loop of the grow method is finished, `search` needs to expand the next cost-tuple states for all priority queues whose cheapest cost-tuple state had a cost of `cost`. This is achieved by calling the method `generate_next_set_of_combinations` of class `ProgramList`.

This is all you need to complete the implementation. You can test the implementation on problem 57:

```
python bee.py 57 0
```

The number 0 indicates the "easy" version of the problem 57. If you change from 0 to 1, the search will be performed with a much larger DSL, which normally results in a much more difficult synthesis problem. All experiments will be performed on the easier version of the problems, but you are welcome to try the harder version of them too. If all goes well, Bee Search should solve problem 57 in a few seconds and information about the solution will be added to the log file `bee-search.log`.

## Experiments and Questions (15 Marks)

1. **(4 Marks)** Run Bee Search and BUS on problems 2, 9, 57, 58, 101, 133, and 173; all these problems should be run on the "easy" mode, i.e., with the flag set to 0. The starter code provides an implementation of BUS. For example, for problem 57 you could run BUS with

   ```
   python bus.py 57 0
   ```

   Note that BUS will struggle to solve problem 173; so you might have to add a time limit to your BUS run for this particular problem. Similarly to Bee Search, the results of BUS will be stored in the file `bus.log`. Compare the solutions encountered by the two algorithms in terms of number of programs evaluated, running time, and solution program. Explain the results you observe.

2. **(9 Marks)** Bee Search uses the function `decimal_place_converter` (see `utils.py`). Check where this function is used in the implementation and answer the following questions.

   (a) **(3 Marks)** Explain how `decimal_place_converter` impacts Bee Search's ability of performing synthesis in best-first order (i.e., cheaper programs according to the cost function are evaluated first in search).

   (b) **(3 Marks)** What happens once you change the function to consider a larger number of decimal places? Does Bee Search evaluate more or fewer programs during search? What about the running time of the synthesis, does it increase or decrease in the experiments you performed? Explain your results.

   (c) **(3 Marks)** If Probe used a similar strategy, where we only consider a small number of decimal places (e.g., 2) of the cost of the programs,[1] its synthesis process would become so slow that it would be impractical. Explain why Probe would become so slow that it wouldn't be a practical algorithm. You should also explain why Bee Search doesn't suffer from this problem.

3. **(2 Marks)** Discuss the computational overhead caused by the `heapify_pqs` function in the experiments you performed with Bee Search.

[1]Since Probe truncates the cost values, the trick would have to be implemented differently for Probe. We would have to multiply the costs by a large constant (e.g., 100) and then truncate the cost values.