# Bottom-up enumeration with egg

YORK LIU, RUOCHEN WANG, and JIARUI HAN, University of California San Diego, USA

In this project we implemented bottom-up enumeration with observational equivalence using egg. Instead of storing one program per equivalence class, we store all of the explored inside an e-graph. We implemented our method in a tool called EgSolver and compared our method with a baseline solver that only stores one program per equivalence class.

Additional Key Words and Phrases: Program Synthesis

**ACM Reference Format:**
York Liu, Ruochen Wang, and Jiarui Han. 2022. Bottom-up enumeration with egg. *J. ACM* 1, 1, Article 4 (January 2022), 6 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

Bottom-up enumeration is a classic program synthesis algorithm. It starts from small programs and combines explored programs into larger programs according to the production rules. In programming-by-example (PBE) problems, Bottom-up enumerator uses observational equivalence to prune the search space by discarding programs that yield the same output as an explored program. However, the discarded programs may be useful, despite the fact that storing all the equivalent programs is challenging because of the exponential program space. Luckily, e-graph provides us with a compact structure for storing equivalent programs. In this paper, we use the egg[2] library as the e-graph implementation and explore the feasibility of storing all explored programs in bottom-up enumeration in an e-graph. We also explore the use cases of those additional programs. E-graph enables the synthesizer to store all the equivalent programs while keeping track of the best program. Another limitation of the bottom-up enumeration is it does not support the customization of cost function. Usually bottom up enumerator searches program by size. If a user wants to customize the cost function, they need to redesign the baseline algorithm. In our implementation, EgSolver, we solved the two limitation from baseline bottom-up enumeration listed above by using the e-graph analysis feature to group the equivalent programs and extract multiple candidates after the search process.

## 2 USE CASES OF ADDITIONAL PROGRAMS

### 2.1 Output multiple programs

Sometimes the user may want multiple programs that they can choose from. The baseline solver cannot satisfy this requirement because for every equivalent set of programs, it only stores the best one. If we want to find $k$ best candidates using baseline solver, we must keep it running until $k$ satisfied programs are found. Since the size of programs grows exponentially, baseline solver would be very inefficient if user specified $k$ to be a large number. Because EgSolver returns an

Authors' address: York Liu, yil173@ucsd.edu; Ruochen Wang, wangrc@ucsd.edu; Jiarui Han, jhan@ucsd.edu, University of California San Diego, USA.

0004-5411/2022/1-ART4 $15.00
https://doi.org/XXXXXXX.XXXXXXX

e-class as the synthesize result, we have access to all the equivalent programs. We can then run an extractor to find the $k$ best programs according to a cost function.

## 2.2 Customize cost function

The user may also want different cost functions when enumerating the program. For example, the user may assert that some operators are more efficient than others, so the user decides to give higher cost to operators which consume more time. In addition, some equivalent programs are more readable than others. For example, $x \ll 1$ is equivalent to $x \times 2$. But $\times$ is more commonly used than $\ll$ operator. The user may want to assign a higher cost to $\ll$. In EgSolver the user has much freedom in designing cost functions. Moreover, storing additional programs enables a more global perspective when choosing the best program. With this in mind, more interesting cost function can be developed. If we only keep one program per equivalence class, we have to rerun the enumeration process with a different cost function. For programs with huge size, it takes exponential amount of time to rerun the synthesizer. However, if we store all equivalent programs, we can run multiple extractors after the enumeration and get different results within linear time (See section 4.2).

## 3 METHODOLOGY

Our synthesizer has two phases, an enumeration phase and an extraction phase. In the enumeration phase, we explore the program space and store all discovered programs in the e-graph. In the extraction phase, we extract multiple programs base on a cost function that is customizable by the user.

## 3.1 Enumeration phase

We make use of the “e-class analysis” feature of egg which allows us to store additional information with each e-class. As an e-class represents a group of observationally equivalent programs, we can store in the e-class analysis the *output vector* of those programs when given the PBE inputs. To implement the enumeration by size, we also store the size of the smallest program of the e-class (the *min size*) in e-class analysis.

The enumeration algorithm is shown in Algorithm 1. We enumerate the programs by size. Whenever we discover a program, we calculate its output vector. If some previous e-class has the same output vector as the new program does, then we union the new program with this e-class, so that they become one bigger e-class. If this output vector is not seen before, we record it and add the program to the program bank. After we find the correct program, we return the e-class ID of that program as the result. We can then run an extractor on that e-class to find $k$ best programs in this e-class.

## 3.2 Extraction phase

The egg library allows us to define custom cost functions for the extractor. As an example, we have defined a cost function that assigns different costs to different operators. Because branching operations typically take more time than arithmetic operations, we assign a cost of 100 to the if-then-else operator, and a cost of 10 to arithmetic operators. To extract multiple programs from the e-graph, we keep track of the programs that we have extracted before in a HashSet. Later when calculating the cost, if the cost function encounters an extracted program, it will output an infinite cost, so that extracted programs will not be chosen by the extractor.

The algorithm for marking a program as extracted is shown in Algorithm 2. We will not delete nodes from e-classes whose min size is less than or equal to 2 because e-classes like this is likely to be referenced by other e-nodes. For example, we don’t want to delete nodes from the e-class containing the program $x$. We first find the e-node with the smallest cost in the e-class. This e-node

**Algorithm 1:** `enumerate`: Enumeration algorithm with egg

```
Input   : maxsize: the maximum program size to enumerate
Output  : an e-class representing the found program, or nil if not found
initialize e-graph with programs of size 1;
for size := 2 to maxsize do
    for prod in productions do
        for args in gen_args(size - 1, prod.arity, size_bank) do
            new_eclass = egraph.add(prod(args));
            outvec = egraph[new_eclass].outputvec;
            if outvec in classmap then
                egraph.union(classmap[outvec], new_eclass);
            else
                classmap[outvec] = new_eclass;
                size_bank[size] = size_bank[size] $\cup$ {new_eclass};
            end
            if outvec = goal_output then return egraph.find(new_eclass);
        end
    end
end
print("not found within size", maxsize);
```

**Algorithm 2:** `delete_best`: Algorithm to mark a program as extracted

```
Input   : eclass: the e-class to perform the remove
InOut   : removed: the set of removed programs
Output  : a boolean value indicating whether something has been removed
if eclass.minsize $\leq$ 2 then return false;
best_node = find_best_node(eclass);
for child in best_node.children() do
    if delete_best(child, removed) then return true; // removed something in the child
end
to_delete = nodes in eclass that has the same operator as best_node;
if to_delete $\neq$ eclass.nodes then
    removed = removed $\cup$ to_delete;
    return true;
else // all the e-nodes has the same operator, don't remove
    return false;
end
```

represents the extracted program. We then try to delete nodes from the children of the e-class by recursively calling the delete algorithm. We don't directly delete the best node since there may not be any equivalent programs left in the e-graph if we do. If none of its children can be deleted and there exist operators different from the best node, we remove the best node and all the e-nodes with the same operator from the e-class. This way, more than one programs are removed in one call to the delete function. We can also choose to remove just one program, but during testing, we found that removing multiple programs led to more diverse and meaningful outputs, as opposed to commutative terms like $x + 1$ and $1 + x$.

## 4 EVALUATION

We have implemented our algorithm as a tool called EgSolver. Our tool is available at https://github.com/weirane/egsolver. For comparison, we also implemented the baseline bottom-up enumeration by size with observational equivalence in Rust.

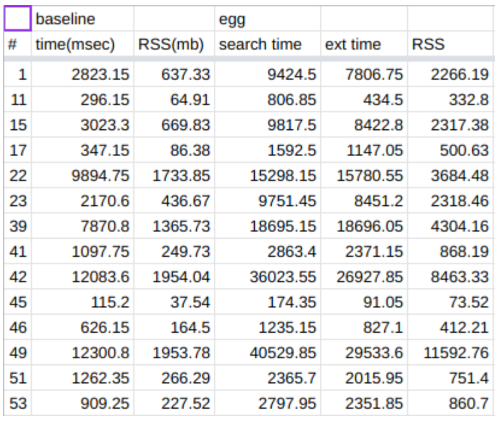
In this section, we compare the baseline solver and EgSolver on a selected subset of the PBE-BitVector track of the 2018 SyGuS[1] competition and answer the following research questions:

(1) Does egg provide enough compression for us to store all the programs?
(2) How does the runtime of EgSolver compare to the baseline solver?

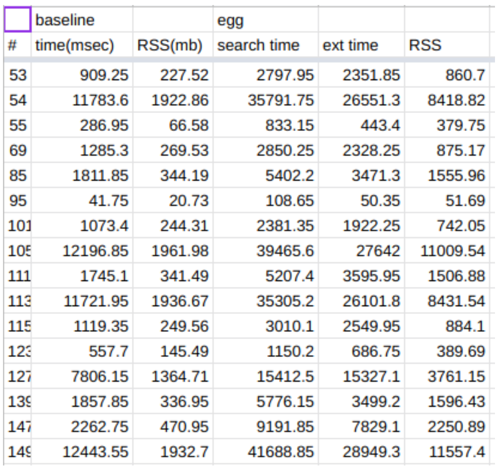
All benchmarks were conducted on one humble laptop with 16GB of RAM and 4GHz CPU clock. Because EgSolver needs to store all equivalent programs, whose number grows exponentially fast, the benchmark is extremely memory-bound. As a result, we were only able to collect data for 29 problems whose answer size is less than or equal to 10; When the answer size exceeds 10, EgSolver uses (or exceeds) SWAP which skews the measurement. We ran each data point 20 times and took the average, in order to rule out the randomness in the hashing library used in egg.

Table 1 shows all the data we collected.

Table 1. All data points for the benchmark.
Time measurements are in milliseconds, and memory measurements (RSS) are in megabytes



| | baseline | | egg | | |
|---|---|---|---|---|---|
| # | time(msec) | RSS(mb) | search time | ext time | RSS |
| 1 | 2823.15 | 637.33 | 9424.5 | 7806.75 | 2266.19 |
| 11 | 296.15 | 64.91 | 806.85 | 434.5 | 332.8 |
| 15 | 3023.3 | 669.83 | 9817.5 | 8422.8 | 2317.38 |
| 17 | 347.15 | 86.38 | 1592.5 | 1147.05 | 500.63 |
| 22 | 9894.75 | 1733.85 | 15298.15 | 15780.55 | 3684.48 |
| 23 | 2170.6 | 436.67 | 9751.45 | 8451.2 | 2318.46 |
| 39 | 7870.8 | 1365.73 | 18695.15 | 18696.05 | 4304.16 |
| 41 | 1097.75 | 249.73 | 2863.4 | 2371.15 | 868.19 |
| 42 | 12083.6 | 1954.04 | 36023.55 | 26927.85 | 8463.33 |
| 45 | 115.2 | 37.54 | 174.35 | 91.05 | 73.52 |
| 46 | 626.15 | 164.5 | 1235.15 | 827.1 | 412.21 |
| 49 | 12300.8 | 1953.78 | 40529.85 | 29533.6 | 11592.76 |
| 51 | 1262.35 | 266.29 | 2365.7 | 2015.95 | 751.4 |
| 53 | 909.25 | 227.52 | 2797.95 | 2351.85 | 860.7 |



| | baseline | | egg | | |
|---|---|---|---|---|---|
| # | time(msec) | RSS(mb) | search time | ext time | RSS |
| 53 | 909.25 | 227.52 | 2797.95 | 2351.85 | 860.7 |
| 54 | 11783.6 | 1922.86 | 35791.75 | 26551.3 | 8418.82 |
| 55 | 286.95 | 66.58 | 833.15 | 443.4 | 379.75 |
| 69 | 1285.3 | 269.53 | 2850.25 | 2328.25 | 875.17 |
| 85 | 1811.85 | 344.19 | 5402.2 | 3471.3 | 1555.96 |
| 95 | 41.75 | 20.73 | 108.65 | 50.35 | 51.69 |
| 101 | 1073.4 | 244.31 | 2381.35 | 1922.25 | 742.05 |
| 105 | 12196.85 | 1961.98 | 39465.6 | 27642 | 11009.54 |
| 111 | 1745.1 | 341.49 | 5207.4 | 3595.95 | 1506.88 |
| 113 | 11721.95 | 1936.67 | 35305.2 | 26101.8 | 8431.54 |
| 115 | 1119.35 | 249.56 | 3010.1 | 2549.95 | 884.1 |
| 123 | 557.7 | 145.49 | 1150.2 | 686.75 | 389.69 |
| 127 | 7806.15 | 1364.71 | 15412.5 | 15327.1 | 3761.15 |
| 139 | 1857.85 | 336.95 | 5776.15 | 3499.2 | 1596.43 |
| 147 | 2262.75 | 470.95 | 9191.85 | 7829.1 | 2250.89 |
| 149 | 12443.55 | 1932.7 | 41688.85 | 28949.3 | 11557.4 |

Note that to support equality saturation, EgSolver does strictly more work than the baseline by storing all equivalent e-nodes and maintaining the e-graph. As a result, EgSolver is bound to be slower and to consume more memory than the baseline, as shown in the table. We will measure how EgSolver's time and memory scale compared to that of the baseline.

### 4.1 Memory

Figure 1 compares the memory consumption of EgSolver to that of the baseline.

The fitted model is $755.3 + 145.354 \cdot 1.00212^x$, which is exponential in terms of the memory consumption of the baseline. We noticed that the base of the exponential is very close to 1, so we also tried to fit the data with a linear model, which turned out to have a much lower $R^2$ score (around 0.87, compared to the current model's 0.9569).

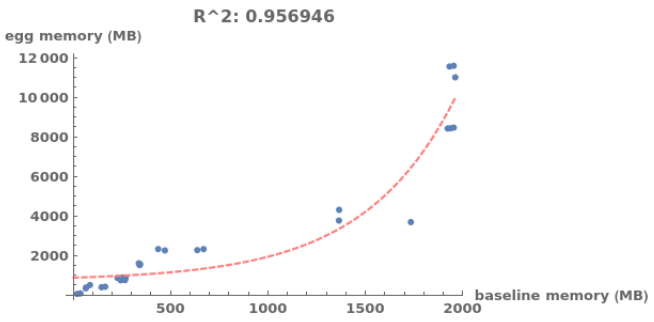


Fig. 1. Memory consumption (resident set size) of EgSolver and baseline

## 4.2 Time

Figure 2 compares the search and extraction time of EgSolver to the entire duration of the baseline.

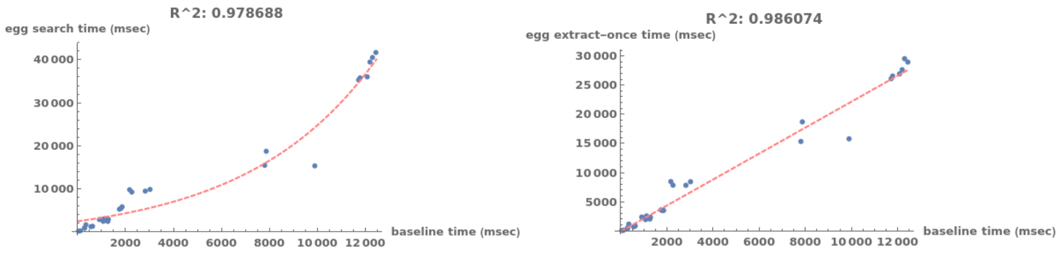


Fig. 2. Search and extraction time of EgSolver compared to baseline

The plots suggest different patterns:

(1) The search time for EgSolver follows a exponential profile: $-1628.19 + 4142.5 \cdot 1.00019^x$, where $x$ is entire duration of baseline. The $R^2$ metric is 0.9787. Note that the base of the exponential term (1.00019) is very close to 1, so we tried to fit the data using a linear function, which resulted in a much lower $R^2$ score around 0.91. This result is similar to the memory benchmark.

(2) The extract time for EgSolver follows a linear pattern, which is around 2.23 times the entire duration of baseline. The $R^2$ metric is 0.9860.

## 4.3 Exponential or linear?

In the last two subsections, we have presented 3 models, two being exponential, and one being linear. But we notice the exponential models have bases very close to 1, so at the moment we're unsure whether they are indeed linear or exponential. Should we get more data points, we may answer this question with more confidence. That said, the small base still suggests that egg provides a compact structure to store all the additional programs.

## 5 CONCLUSION

In this paper we presented a method to incorporate egg into bottom-up enumeration with observational equivalence. As opposed to the baseline which only stores one program for each equivalent class, EgSolver stores all programs which give the user multiple benefits of solution extraction. Through experiments, we found that although the number of programs grows exponentially with size, egg provides a compact and efficient structure to store those programs and maintain an e-graph.

## 5.1 Limitations and future work

Our tool currently hard-codes the grammar in the SyGuS specification files. One of the improvements is to let egg understand the parsed grammar in the SyGuS file at runtime. This way, we can also get more data points for the benchmark.

Currently, the cost functions that we use are all calculated as the cost of the operator plus the costs of its children. There may be better cost functions that utilizes the context of a program.

We have to reconstruct the extractor each time we extract a program, which traverses the entire e-graph and as a result takes a long time. This is because egg's Extractor API contains multiple private fields that we cannot access. If we modify the egg library, or write our custom extractor, we can avoid reconstructing the extractor and the extraction phase will run faster.

Lastly, we wish to run larger benchmarks, which require more memory. If we can get access to machines with 128GB memory, more data points can be collected and our answers to the research questions can be more convincing.

## REFERENCES

[1] Mukund Raghothaman and Abhishek Udupa. 2014. Language to Specify Syntax-Guided Synthesis Problems. *CoRR* abs/1405.5590 (2014). arXiv:1405.5590 http://arxiv.org/abs/1405.5590

[2] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434304