EGG-sploring the search space.

Using different algorithms to explore the search space with EGG.

John Taggart, CSEP590D, Spring Quarter 2022

Problem Definition

An EGraph is a tool which used to efficiently rewrite and optimize languages during compilation via a process known as "equality saturation". Through the "equality saturation" process, possible rewrites of the language are explored to find a more optimal compiled output. The "equality saturation" process can be applied in many contexts involving compilation, including database query optimization.

The EGraph is a data structure consisting of multiple "EClasses". Each EClass contains sets of equivalent "ENodes". Each ENode is a function symbol of the language being optimized. EGraphs can be used to iteratively complete the equality saturation process via computing and merging EClasses.

EGG (short for "EGraphs Good") is a tool which implements this EGraph design and can be used to achieve equality saturation [5]. For my project, I modified the algorithms used by EGG to explore the search space when computing these EGraphs. I then benchmarked the performance of each of these algorithms.

Implementing the different search algorithms.

By default, EGG is written in a manner that explores the rewrite search space via BFS-style traversals. EGG first explores the possible ways the EGraph could be rewritten in its current state and then rebuilds the EGraph to incorporate those rewrites. This process occurs in EGG's Runner class' run_one() method, which gets the EGraph's possible rewrites using RewriteScheduler objects.

BFS

I implemented default BFS by just creating a class which uses the RewriteScheduler trait and using its default methods which implement a basic search algorithm. In my project, the class which does this is called BFSScheduler.

BackoffScheduler

The BackoffScheduler prevents specific rules from being applied too often. This helps reduce EGraph growth in the presence of "explosive" rules such as associativity and commutativity [2]. This is the default implementation pre-provided with EGG and is implemented in a class called BackoffScheduler.

Beam Search

Beam Search operates like BFS, but it only returns beam_width number of results. To use Beam Search to explore the search space, I implemented a simple version in a class called BeamScheduler. To implement Beam Search, BeamScheduler just runs the RewriteScheduler's search method *but* only returns the first `beam_width` values in the matches. The beams have an arbitrary width, so it decreases accuracy in order to improve algorithm performance.

Unlike the other algorithms evaluated in this project, Beam Search is an incomplete algorithm which is *not* guaranteed to give the optimal results. This means that it does not necessarily provide the same output as the other algorithms since it will not find the optimal solution if the beam_width is not large enough.

When testing the performance of the algorithms, I benchmarked BeamScheduler's performance with beam widths of 20, 40, and 100.

DFS

To implement DFS, I had to adjust how the Runner class works. The Runner class in EGG is built to collectively execute the rules in one iteration BFS-style [4]. This meant that I could not just plug-in a different RewriteScheduler object to implement my algorithm; I actually had to modify how EGG explores the search space.

To make the Runner class executes a DFS-esque search pattern instead of a BFS one, I adjusted the Runner class to apply the rules to the EGraph as soon as EGG encounters them. This DFS algorithm is written in a method called "run_one_dfs()", which can be swapped in for the BFS-based "run_one()" method. To try to introduce some small optimizations on the RewriteScheduler-level, I made my DFS code use the slightly-optimized BackoffScheduler to obtain possible rewrites instead of the default RewriteScheduler.

Benchmarking methodology.

To measure the performance of the search algorithms, I ran tests against EGG using the above algorithms. The tests used were sourced from the EGG GitHub repository [5] and adapted from Remy Wang's UDP EGG repository [3]. I benchmarked the performance of EGG while running these test suites.

- transitive – tests based around evaluating transitive boolean operations. [5]
- lambda – tests based around lambda calculus. [5]
- math – tests using a language of various mathematical operations. [5]
- prop – tests using some propositional language. [5]
- simple – tests using a language with commutative addition and multiplication. [5]
- udp – tests which evaluate UDP query equivalence using EGG. [3]

These tests were run on an ARM-based 2021 10-core M1 Pro Mac with 16GB RAM running macOS 12.2.1.

To prevent the tools used to measure the below benchmarks from accidentally interacting with each other and impacting the obtained metrics, I ran each benchmark separately. To keep data comparisons between the results obtained algorithms simple, this project aggregates the sum of the iterations in each test suite and compares those instead of comparing each individual test.

To ensure that my benchmark measurements accurately represented the algorithms they were evaluating, I ran them five times each and then recorded the lowest value. I always made sure that the metrics were similar across the five test

runs--if there were any outlier results, I reran the tests until I had five consistent values in-a-row.

Runtime

The Rust-based testing tool which EGG uses by default does not provide millisecond-level accuracy, so I had to use this Unix "time" command instead. I ran the "time" command with a custom configuration to measure the runtime of each test suite in milliseconds.

The runtime recorded by the "time" command does include the overhead of spinning up and running the test suite; however, this should not impact our ability to use these metrics benchmarking since this overhead should be identical across all algorithms.

Peak memory usage.

To measure peak memory usage, I used a script to obtain the peak memory usage in KB of the test suite during execution. This script was sourced from GitHub [1].

Number of search algorithm executions required to achieve equality saturation.

To measure the number of search calls required before EGG would find the optimal solution, I added a search iteration counter variable and print statement to the Runner class. This allowed for me to obtain quick-and-simple counts of the number of search iterations completed.

Best EGraph "variant" computed.

The tests used for this project operated via using EGG to find the best "variant" of the EGraph for the provided data. It then compared the variant output by EGG with the expected "best variant" hardcoded in the test. If any test failed, this meant that the algorithm was unable to compute the best possible variant.

Results.

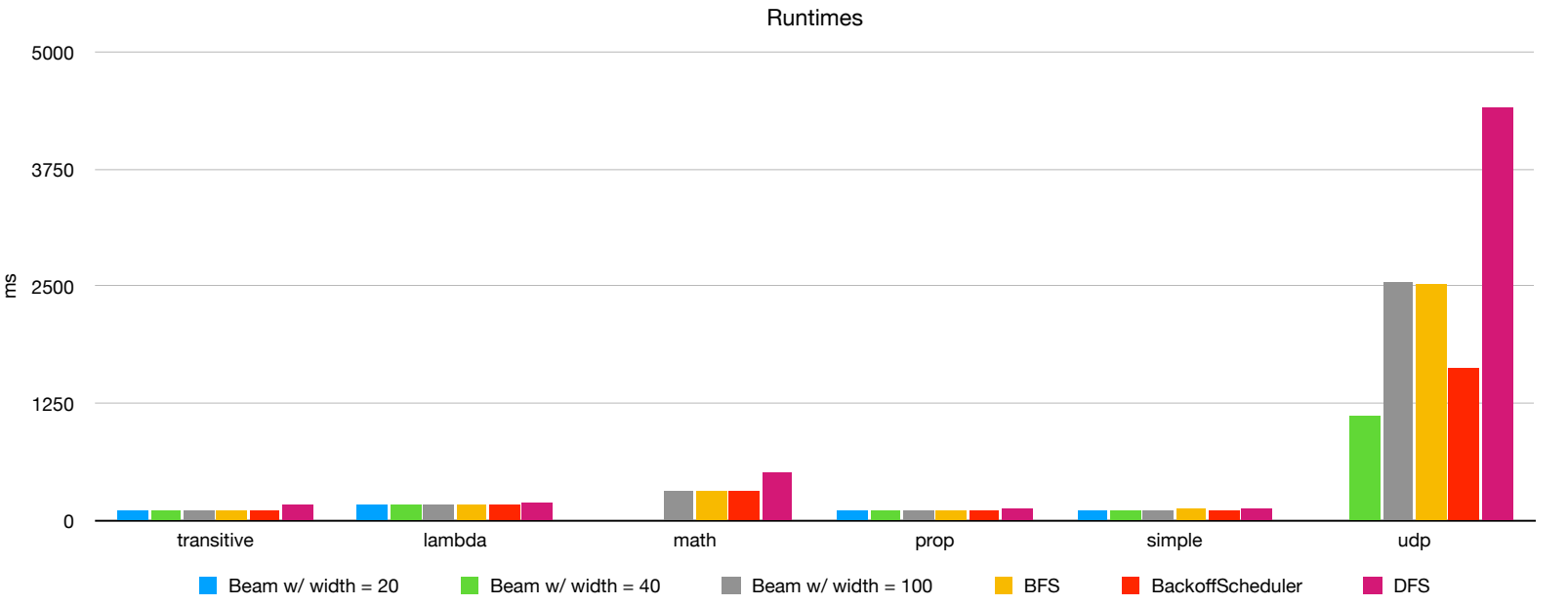
If an EGG test suite had any test fail, the metrics measured were severely worsened due to how the Rust testing library handles failures. For this reason, I am excluding all metrics obtained from test failure cases.

The only tests cases which ever failed were some of the Beam Search cases. This is because Beam Search did not return the optimal EGraph result for certain beam widths.

Runtimes (ms)

	transitive	lambda	math	prop	simple	udp
Beam w/ width = 20	108	165	X	105	106	X
Beam w/ width = 40	105	176	X	108	106	1118
Beam w/ width = 100	110	174	314	106	100	2538
BFS	105	169	316	113	121	2514
BackoffScheduler	112	169	318	103	103	1630
DFS	177	198	523	132	120	4403

"X" = Tests failed.



Runtime benchmark results.

Across all test suites, the runtime measurements were the worst for the DFS algorithm. This was especially pronounced for the UDP test case, where DFS was 73% slower than the second-slowest test case.

For the "simple" and "prop" test cases, the default BFS approach was the second-slowest. In all other test cases, it performed very similarly to the other BFS-based algorithms.

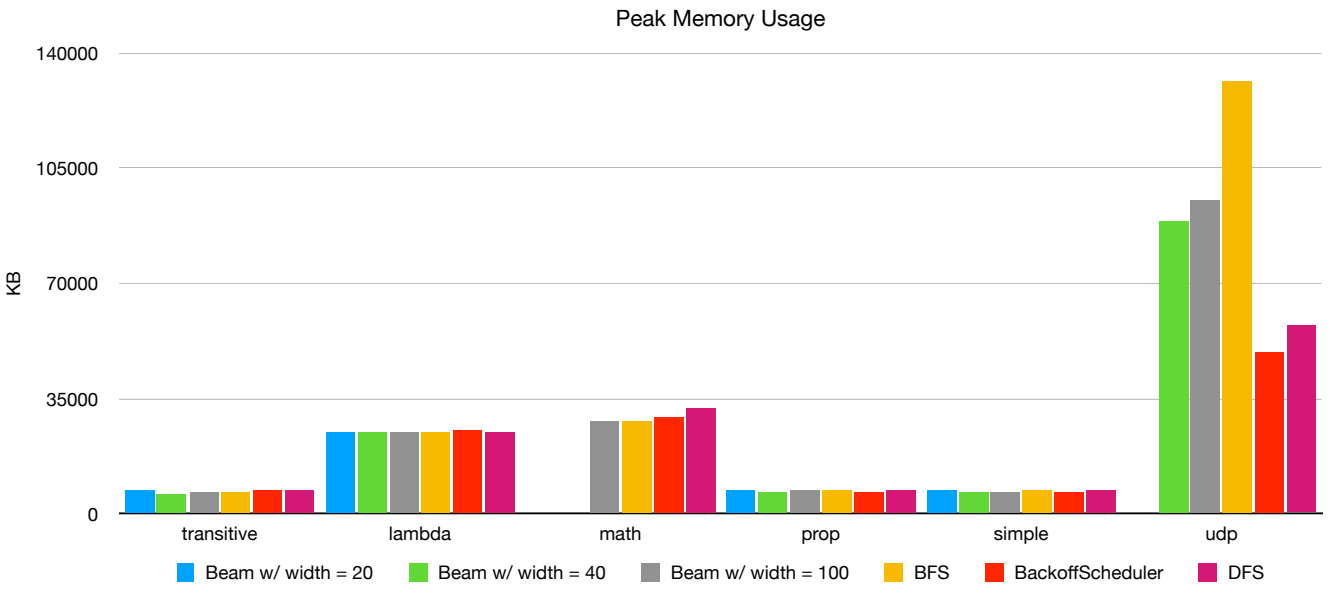
For the cases where they succeeded, the Beam Search algorithms with beam widths of 20 and 40 outperformed the rest of the algorithms. This is because they had to explore a smaller search space.

All other algorithms all performed very similarly across all test cases.

Peak Memory Usage (KB).

	transitive	lambda	math	prop	simple	udp
Beam w/ width = 20	7072	24848	X	7280	7136	X
Beam w/ width = 40	6272	24768	X	6990	6992	88800
Beam w/ width = 100	6992	25008	28400	7088	6960	95264
BFS	7008	24944	28416	7072	7072	131472
BackoffScheduler	7392	25344	29248	6992	7024	49424
DFS	7168	25072	32528	7056	7072	57104

"X" = Tests failed.



Peak Memory Usage

The results of my peak memory usage tests show no clear "best" or "worst" algorithm across all test suites.

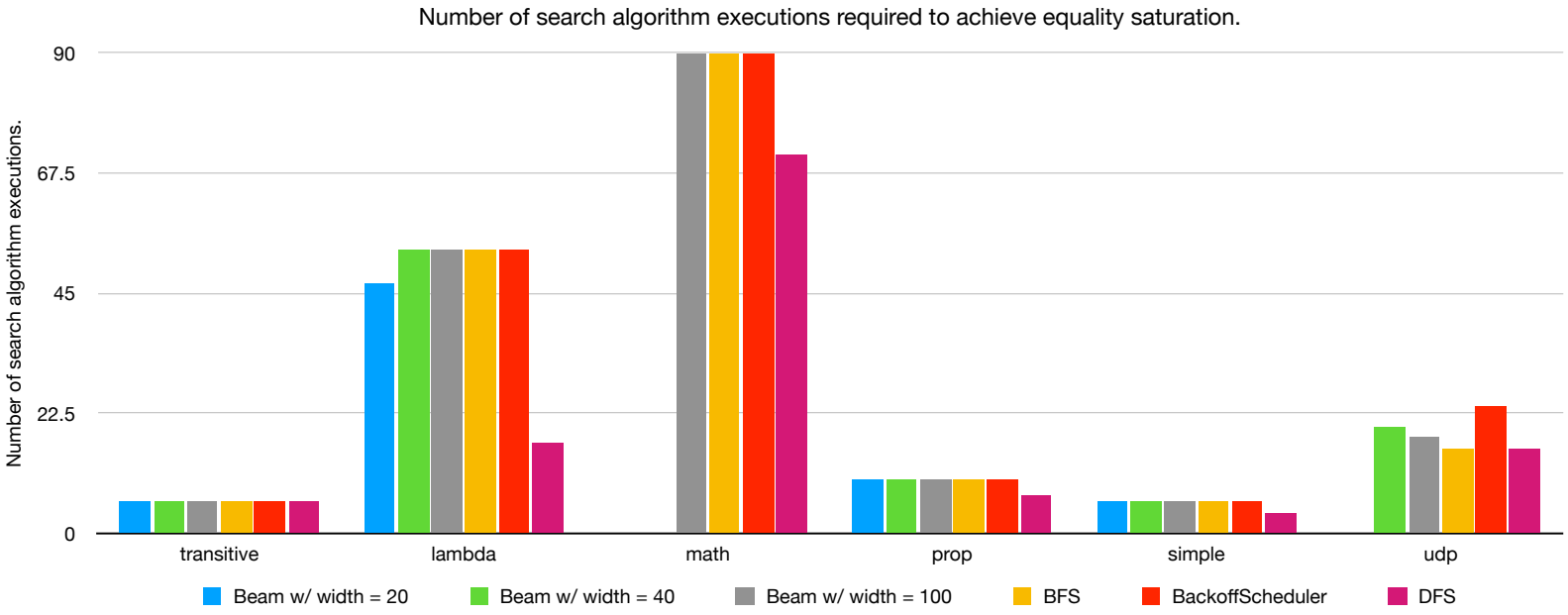
In the udp suite, BFS consumed by-far the most memory of the algorithms, while BackoffScheduler appeared to have consumed the least memory.

In the transitive suite, my beam search with a beam width of 40 used the least memory of all algorithms. This is not surprising: beam search operates via reducing the search space, which would lead to less resource usage.

Number of search algorithm executions required to achieve equality saturation.

	transitive	lambda	math	prop	simple	udp
Beam w/ width = 20	6	47	X	10	6	X
Beam w/ width = 40	6	53	X	10	6	20
Beam w/ width = 100	6	53	90	10	6	18
BFS	6	53	90	10	6	16
BackoffScheduler	6	53	90	10	6	24
DFS	6	17	71	7	4	16

"X" = Tests failed.



Number of search algorithm executions required to achieve equality saturation.

Across all test cases, DFS achieved equality saturation with the fewest number of search algorithm executions. By going depth-first, DFS has more thorough computations with each execution of its search algorithm than the other breadth-first-based algorithms. As seen earlier though, this comes at a significant runtime cost.

All other algorithms performed roughly the same across all test suites. This is because they're all breadth-first algorithms.

The Beam Search algorithm with a beam width of 20 slightly outperformed all other breadth-first algorithms on the lambda test suite: this could be because some less optimal execution paths were pruned.

When obtaining this metric, the number of iterations executed by an algorithm for any test class stayed consistent across multiple executions. This is due to the deterministic nature of this metric and these test suites.

Best EGraph "variant" computed.

BFS, DFS, BackoffScheduler, and Beam Search with a beam width of 100 all passed all the test suites, indicating that they all generated the same optimal EGraph variants for each test case.
The Beam Search cases with beam widths of 40 and 20 failed the "math" test suite, and the Beam Search cases with beam width of 20 failed the "udp" test suite. In both cases, the failure messages were because they did not compute the optimal EGraph variant.

This is not too surprising: Beam Search functions via forcefully pruning the search space to reduce memory usage. The reduction in the search space prevented these test cases from finding the optimal solution.

The Beam Search case with a beam width of 100 only succeeded because a beam width of 100 was sufficient for searching these test cases for the optimal variant. There certainly exists use-cases for EGG where a beam width of 100 would not be sufficient for find the optimal variant.

Conclusion

Beam search can lead to improvements in runtime memory usage, but at the cost of accuracy. In situations where this tradeoff is worth doing and we are sure that the search space will not ever become too big for the beam width, Beam Search should be used.

Overall, my metrics show that EGG's BFS-derived Runner code is the optimal approach for "equality saturation." Of the BFS-based approaches, BackoffScheduler is the best approach: it adds a couple optimizations that slightly further improve performance without impacting the generated EGraph output.

Bibliography

[1]
262588213843476. 2010. memusg -- Measure memory usage of processes. *GitHub*. Retrieved May 8, 2022 from https://gist.github.com/netj/526585
[2]
Thomas Koehler, Phil Trinder, and Michel Steuwer. 2021. Sketch-Guided Equality Saturation: Scaling Equality Saturation to Complex Optimizations in Languages with Bindings. *arXiv:2111.13040 [cs]* (November 2021). Retrieved May 8, 2022 from https://arxiv.org/abs/2111.13040
[3]
Remy Wang. udp/src at main · remysucre/udp. *GitHub*. Retrieved May 8, 2022 from https://github.com/remysucre/udp/tree/main/src
[4]
Max Willsey. 2022. CSEP590, Week 3. *www.youtube.com*. Retrieved May 8, 2022 from https://www.youtube.com/watch?v=ebZzL-6UBPc
[5]
2022. egg: egraphs good. *GitHub*. Retrieved May 8, 2022 from https://github.com/egraphs-good/egg/