Synthesizing Demonic Graph for Worst Case Performance

CSE 507 Final Project Report

Yihong Zhang, Mike He

December 2019

Abstract

Delicately designed algorithms play an important role in speeding up efficiency in various settings. For example, the Dijkstra algorithm in solving Single Source Shortest Path (SSSP) problem reduces the time complexity [2]. However, it's still very hard to tell whether some algorithms have significant efficiency faults. A specifically designed data input may cause to grow exponentially the time complexity of an algorithm that is actually fairly efficient on random data. This specifically applies to algorithms that heavily relies on heuristics. It's therefore important to figure out whether certain heuristics is general enough to work on all the input cases. In this project, we showcase how to use SMT solver as a powerful tool for evaluating the worse-case performance of algorithms and synthesizing data input that causes algorithm' concrete execution towards degeneration. As a case study and a proof of concept, we target at programs that take graph as input data and, more specifically, the Shortest Path Faster Algorithm (SPFA) and variants of it with heuristics. We choose to focus specifically on graph problems and programs because graph contains rich structural information, which will potentially cause more radical fluctuations of performance. To synthesize demonic graph for SPFA algorithm, we designed Sager, a project that uses symbolic evaluation and a scaling algorithm of graphs. Sager will first symbolically synthesize a small instance of graph that will causes SPFA and its variant to have worst performance, and then utilizes the scaling algorithm to scale the synthesized graph.

1 Motivation

Formally examining the efficiency of an algorithm is complicated. Traditionally, it requires tedious proof over complex procedures. Moreover, theoretical analysis often ignores constant factors and only focuses on asymptotic complexity, but constant factors play an important role in implementations, which can make implementation choice tricky. Therefore, one may find a formal algorithm analysis hard to be applied in industrial settings. Instead, people often use pressure tests for testing the performance of an algorithm in practice. However, randomly generated data usually are not strong enough to hack an algorithm and they mostly have a same shape that yield to certain random distributions (e.g. a randomly generated binary tree is likely to have depth $\Theta(\log n)$). This motivates us to develop an automated mechanism that is able to detect structural efficiency faults of various algorithms.

Moreover, many algorithms in practice will use one or another sort of heuristics, which could help them to be fast in practice. However, it's doubtful whether such heuristics fast enough, and it's even harder to construct data to examine worst case performance of these heuristics, as they usually are very dependent on the form of input data, and thus hard to predict what kind of data will expose such bottleneck performance.

Another real-world application of synthesizing demonic data is in the competitive programming. Competitive programming is a mind sport that allows people to come up with and implement algorithmic computer programs to solve problems related to graph theory, number theory, string algorithms, as well as advanced data structures. The problem-setters, hoping to test participant's knowledge on certain theory, usually don't want a fake, brute-force algorithm to pass all the test of a problems, but such fake algorithms usually involves some heuristics that make them super fast on randomly generated data, sometimes even faster than the standard solution, though they generally had a very bad theoretical time complexity. For instance, Knuth–Morris–Pratt (KMP) algorithm [3], an algorithm that tested frequently in competitive programming events, guarantee searching the occurrence of a word in a text will only take $O(n + m)$ time, which is the theoretical lower bound of time complexity. But on random data, where potential matches of words is scarce, the brute force algorithm can also have an approximately linear performance. Therefore, problem-setters usually want to hack such brute force algorithm with some specifically designed data in order to encourage more intelligent solutions.

For the purpose of comparing various constant factors, finding potential failures in heuristics, and hack brute-force algorithms with seemingly efficient performance, it would thus be natural to want to synthesize worse case performance data input. However, this is actually hard to achieve. In the competitive programming world, current state-of-the-art data fuzzer mainly relies on randomization, with insightful observations and empirical experiments, to generate data inputs. Such data generators are more or less *ad hoc*, depending on *a priori* algorithm knowledge. It'll be even more clueless on how worst a program can be if the program is equipped with some strategic heuristics.

To reduce the painfulness in such process, we propose the use of symbolic evaluation for the generation of demonic input data. Symbolic evaluation is a method to compile programs into logical constraints that can be solved by constraint

solvers [4]. It can therefore be used as an oracle for both angelic execution and demonic execution. With symbolic evaluation tools, we can encode the problem of finding demonic data into a set of logical constraints that wants to maximize the cost function. It therefore requires no previous knowledge of the program, and the entire process is made automated, as it requires no more efforts for users to manually construct the demonic data: even when users already know what a demonic data may look like, it's still hard to encode such properties into data.

We specifically focus on the cases where the programs accept graph inputs, i.e., programs that implement a graph algorithm. This is because graph contains rich structure information, and thus the "shape" of graph can radically affect the performance of the algorithm, while such shape is hard to be conceived. Moreover, in generating demonic data for performance evaluation, it's also critical to control the size of data generated: If the size of data can grow unwieldy, the running time for any algorithm that take more than constant time will tend to infinity. Thus, to state our problem more formally, we suppose there is an interpreter function $\mathit{eval}(P, D)$ that will yield the sequence of operations to be executed given the procedure and input. Now given a procedure P that accepts a graph G, a total function *cost* that maps operation to costs, and an integer n, our synthesizer aims to produce some graph G with size n that maximizes

$$\sum_{\mathit{instr} \in \mathit{eval}(P,D)} \mathit{cost}(\mathit{instr}).$$

2 Overview

We design Sager, an algorithm that can synthesize demonic graph for programs accepting graph input. Here, we define demonic graph to be a graph instance that could maximize the *cost* function. Sager contains two parts, both of which is modular and therefore can be customized for various settings. The first part of Sager is a solver-aided synthesizer that will evaluate the programs symbolically, encoding the whole program execution traces to calculate the symbolic cost of the program on the symbolic inputs. By solving the formula with a solver, Sager is able to obtain a relatively small instance of the graph that can cause efficiency issue of a specific algorithm, which is called a gadget in our terminology. In the second part, Sager will use a structural scaling algorithm to produce a much larger graph that will try to simulate the structural properties of the gadget as much as possible. Notice that since Sager have no prior knowledge of which specific property we want to learn from the gadget, we can't expect such a general scaling algorithm to fully represent various properties of the graph. For example, if the synthesized graph is a square, we don't know if we should synthesize a grid-like graph, or a graph that has many linear many path from source to target. Therefore, it's really about how much insight we have. If we already know the final shape of the demonic graph derived from the tiny gadget we have, then the only thing we need to do is to scale edge weights of the gadget to the corresponding edges at the final demonic graph.

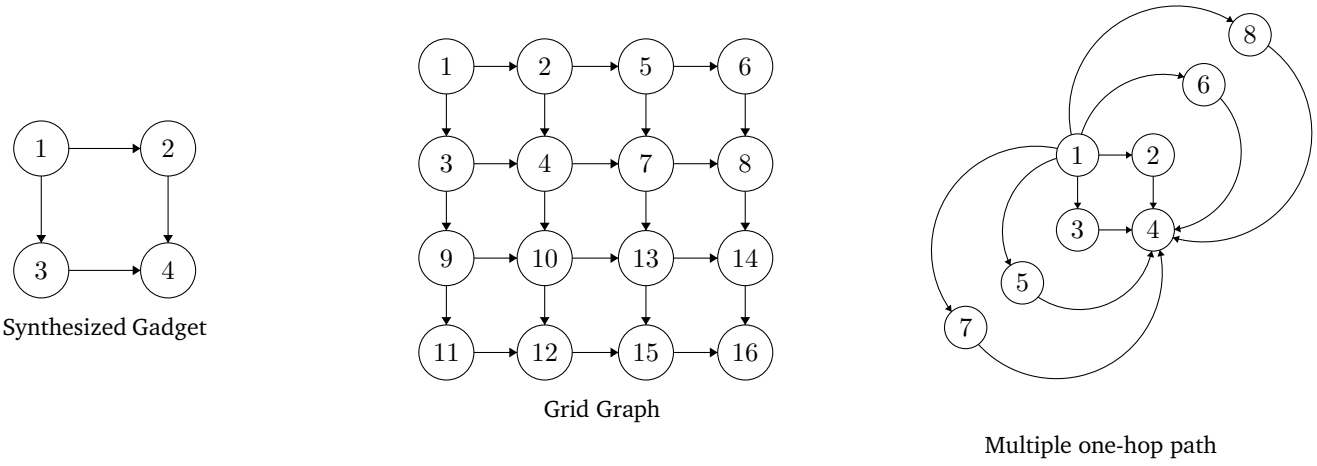

Figure 1: Possible desired scaled graphs

In the first part, we experiment with various graph encoding strategies as well as various solving strategies. It turns out the a moderately symmetry broken version of graph representation works slightly better on nontrivial symbolic graphs, while incremental solving strategy could gain a huge augmentation on solving speed. Moreover, we chose to use adjacent matrix as our target data structure to synthesis in order to avoid computations on symbolic unions which would make the synthesizer slow.

Due to limited time, we didn't spend too much time on the second part, which can be more refined given sufficient time. However, the current algorithm can achieve a linear growth of cost with respect to node size (i.e., the *cost*/*size* won't decrease when *size* goes up). We also provide two versions of the scaling algorithm that can model the structural property at different dimension. Finally, we perform extensive benchmarks on three variant of the target algorithm Shortest Path Fast Algorithm (SPFA). We found the graph generated by Sager, though not optimal, still demonstrate

bottleneck performances for many programs.

3 Algorithm

3.1 Logical Encoding

Besides a plain adjacent matrix encoding, we also explored sorts of symmetry breaking strategy. Ideally, symmetry breaking should only keep one instance of every group of isomorphic graphs. However, identifying whether two graphs are isomorphic is a very hard problem, so we can't expect a complete symmetry breaking algorithm. Also, solving the constraints imposed by symmetry breaking may also cause a huge overhead, so we should measure the trade-off between a powerful but more complicated strategy and a weak but more lightweight strategy.

we implemented two versions of symmetry breaking sb_l and sb_l^* described in [1] (Figure 2). sb_l is inspired by the idea that every isomorphic group of graphs must contains a canonical form G that satisfies $sb_l(G)$, i.e., the adjacent matrix representation of the graph is in lexicographical order. sb_l^* goes one step further, showing that in comparing the lexicographical order of two rows of an adjacent matrix, dropping the ith and jth column will still maintain the soundness of symmetry breaking. This, however, breaks the transitivity of $<$, because now $A[i] \leq_{i,i+1} A[i+1]$ and $A[i+1] \leq_{i+1,i+2} A[i+2]$ does not imply $A[i] \leq_{i,i+2} A[i+2]$. This requires us to impose $O(n^2)$ constraints to enforce the transitive relation that is implied in the $O(n)$ constraints by sb_l.

$$sb_l = \bigwedge_{i=1}^{n} A[i] \leq A[i+1]$$

$$sb_l^* = \bigwedge_{i<j \wedge j-i \neq 2} A[i] \leq_{i,j} A[j]$$

Figure 2: Two symmetry breaking strategy

Through our empirical experiments, we found that sb_l will gain us a slight efficiency promotion than the plain encoding, while sb_l^* will greatly slow down the solving time. We suspect this is due to much more complicated $O(n^2)$ constraints required by sb_l^*, which will cause the solver to spend a lot of time finding valid instances.

3.2 Solving Strategy

We also play around with different solving strategies: one using Z3-solver's builtin *maximize* functionality (Corresponding to ROSETTE's *maximize*), one using a binary search to search for possible maximal cost, and one using incremental solving. *maximize* turns out to work very slow, which may be due to the complexity of *MaxSAT* problems. Binary search assumes the cost is monotone and thus using the constraint $cost = x$ to find if such a cost exists and modify the search range accordingly. This will take $\log limit$ times of SMT solving, which we first thought would be efficient compared with $O(n)$ solving from $cost = 0$. However, this is not true, because to restart the solver is very expensive. We finally find incremental solving super efficient, and surprisingly, sometimes even more efficient than a single solving in binary search. We speculate this is because incremental solving can guide the solver to narrow down the search space heuristically, allowing them to fully exploit the constraints imposed early on *cost* during incremental solving.

3.3 SPFA algorithm

SPFA[5] is an variant of the well-known Bellman-Ford Algorithm that employs a queue to reduce unnecessary edge relaxation. Though the author originally claims that it outperforms Dijkstra algorithm with speed $O(k|E|)$ where k is a small constant, it's time complexity can be as bad as Bellman-Ford in worst case, which is $O(|V||E|)$. But in practice, SPFA runs very fast on random, sparse graph with near linear time, and there are various heuristics that makes SPFA robust to different graphs. Two of the mostly wide used are Small Label First (SLF) and Large Label Last (LLL).

SLF: When pushing an element into the queue, check if it's smaller than the distance of current front element. If so, push the element into the front of the queue.

LLL: When pushing an element into the queue, check if it's smaller than the average distance of vertices in the graph. If so, push the element into the front of the queue. [1]

[1] This is a slightly modified version of LLL, while the original LLL has a very expensive overhead.

```
dist ← MAKE-ARRAY(n, [∞]);
dist[s] ← 0;
for i ← 1 to |V| − 1 do
    for {u, v, weight} ∈ E do
        if dist[u] + weight < dist[v] then
            dist[v] ← dist[u] + weight;
        end
    end
end
```

Bellman-Ford Algorithm

```
Q ← MAKE-QUEUE();
dist ← MAKE-ARRAY(n, [∞]);
dist[s] ← 0;
Push s into Q;
while Q is not empty do
    u ← Q.POP();
    for {u, v, weight} ∈ E do
        if dist[u] + weight < dist[v] then
            dist[v] ← dist[u] + weight;
            Push v into Q if v is not in Q;
        end
    end
end
```

SPFA

Figure 3: SPFA and Bellman-Ford Algorithm

3.4 Self-similar Scaling

After synthesizing a small gadget, Sager will scale the demonic graph to be practical enough to time out the program in practice. In order to scaling up the data as well as maintaining core structural information, we implement an iterative graph unrolling algorithm. In each pass of unrolling, Sager will use depth-first search to substitute every node in the graph with a gadget. Sager will run multiple passes until the graph is large enough. Also, during the substitution, user need to specify which two nodes will play as *entry* node and *exit* node, so they can play as a connection node to connect the rest of the graph. In practice, such entry node and exit node can be found in the program's running trace (e.g., the first and the last node visited.), while different choice of entry node and exit node will cause the algorithm to simulate different structural information. For example, Figure 4 shows one scaling iteration starting by doing DFS on node 1 when n_{in} is node 1 and n_{out} is node 3, while Figure 5 shows the case when both n_{in} and n_{out} is node 1.

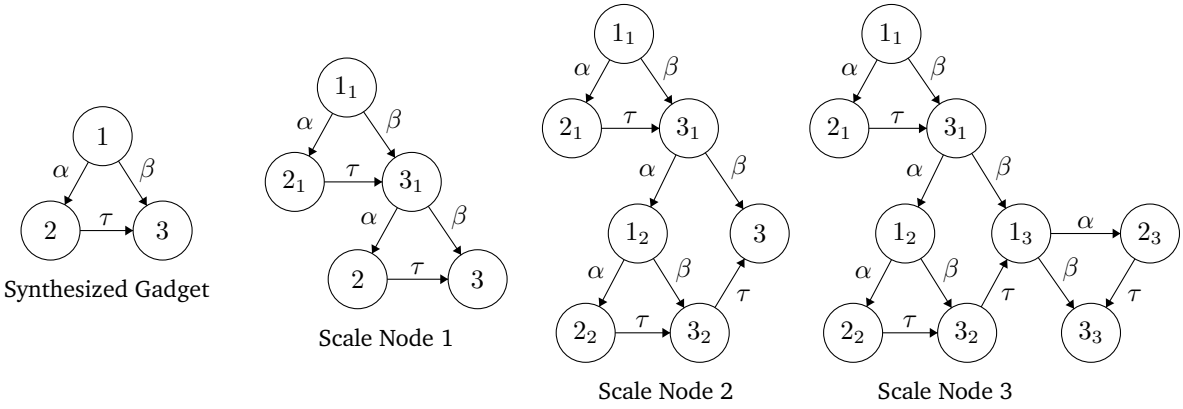

Figure 4: Example - Scale a small gadget on each node ($(n_{in}, n_{out}) = (1, 3)$)

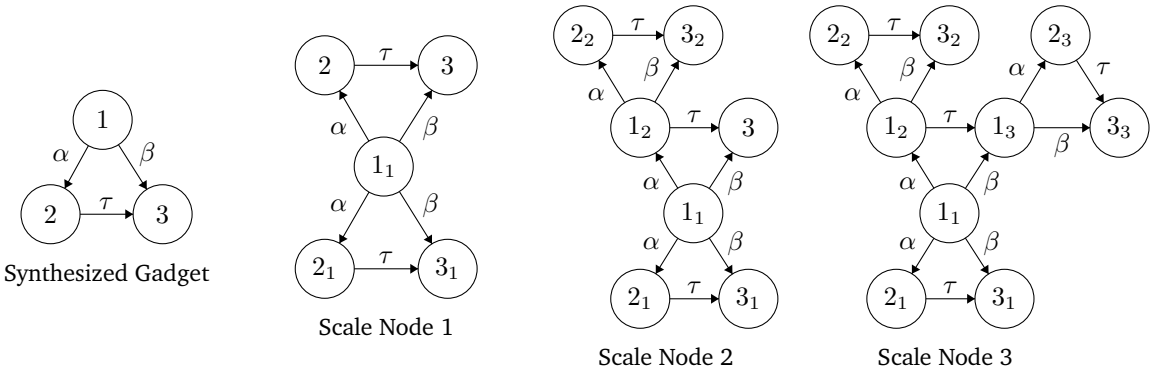

Figure 5: Example - Scale a small gadget on each node ($(n_{in}, n_{out}) = (1, 1)$)

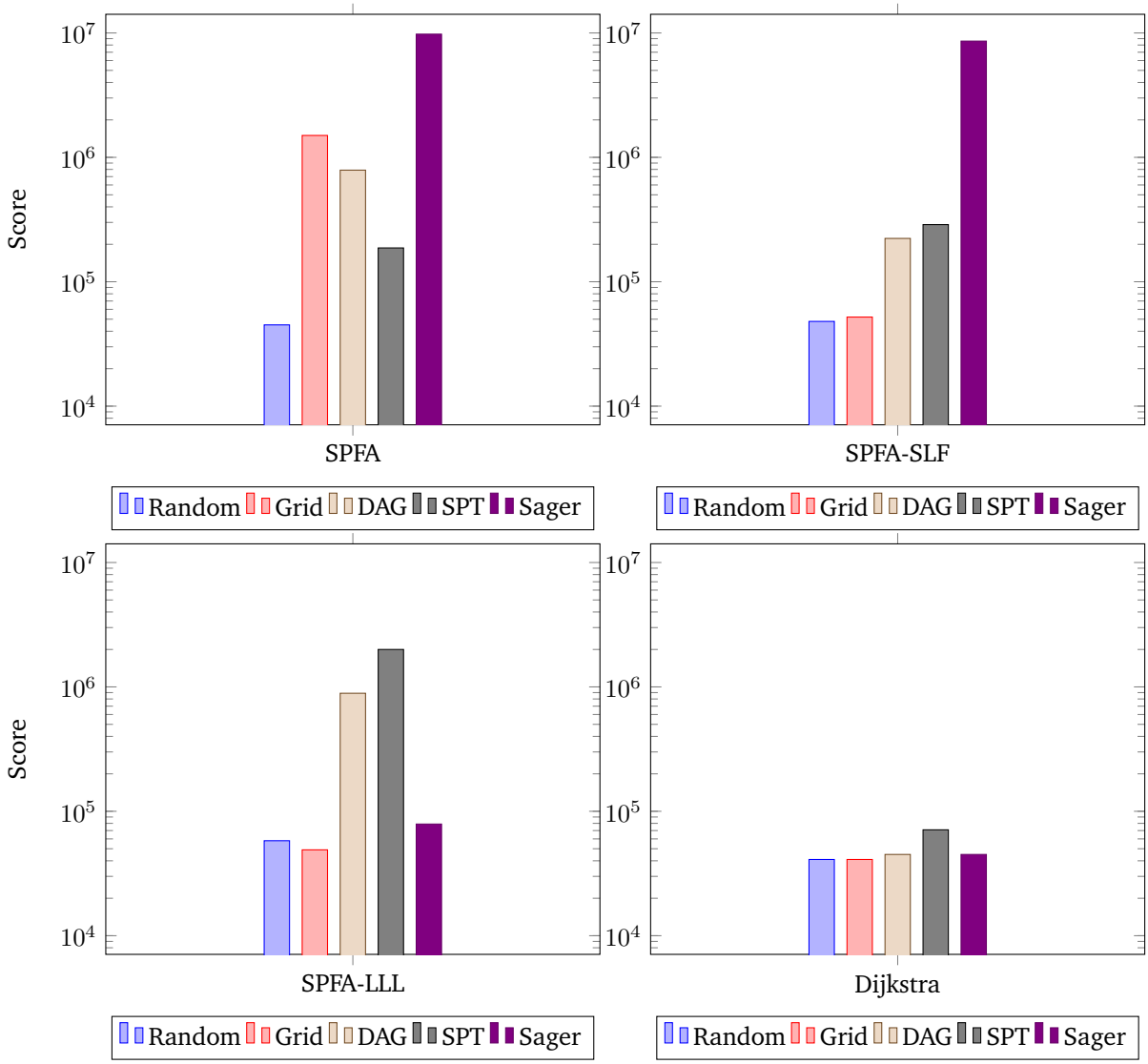

Figure 6: Comparison on 100,000 Nodes Graph

4 Results

We synthesized 4- and 5-vertex gadget (and in the case of original SPFA, also 6-vertex gadget) and scaled it up to graphs with 10,000 nodes and with 100,000 nodes for SPFA and SPFA with SLF and LLL optimizations. We compared our data with randomly generated connected graphs (Random), graphs constructed by using Shortest Path Tree (SPT), which, by empirical insight, can lead SPFA to search through suboptimal paths and causes SPFA slow and graphs generated by constructing a delicately designed Directed Acyclic Graph (DAG). We calculated the score based on how many times nodes are pushed into the queue and edges are being visited. Figure 6 and Figure 7 shows the performance of both SPFA and SPFA with heuristics on datasets. It can be seen Sager has a much better capability to synthesize data that cause a time complexity explode on original SPFA and SPFA with SLF heuristics. However, for LLL heuristics, data generated by Sager did not have much effect. This might be due to the complexity of LLL, which is a more far-sighted heuristics that depends on more global information (i.e., the average value of queue elements) rather than a local property.

5 Limitations

There are still some issues that discourage Sager from generating more powerful data. First, the speed of synthesizing the gadget is slow. Although we didn't have time to have a benchmark on solving time. It's roughtly within a minute for 4-node gadgets, 10 minutes for 5-node, and there is only one case that we successfully synthesize a 6-node gadget. Due to the growth of the complexity of graph synthesis, it nearly impossible for Sager to solve a graph with tens of nodes. Because of this limitation, we developed our scaling algorithm that can expand the small gadget to a larger graph while preserving local graph properties. However, there is no catch-all algorithm that is general enough to capture precisely the structural information that is essential to synthesize demonic graph (at least for now). According to our experiment with SPFA with LLL heuristics, it does not work well with algorithms whose complexity depends more on

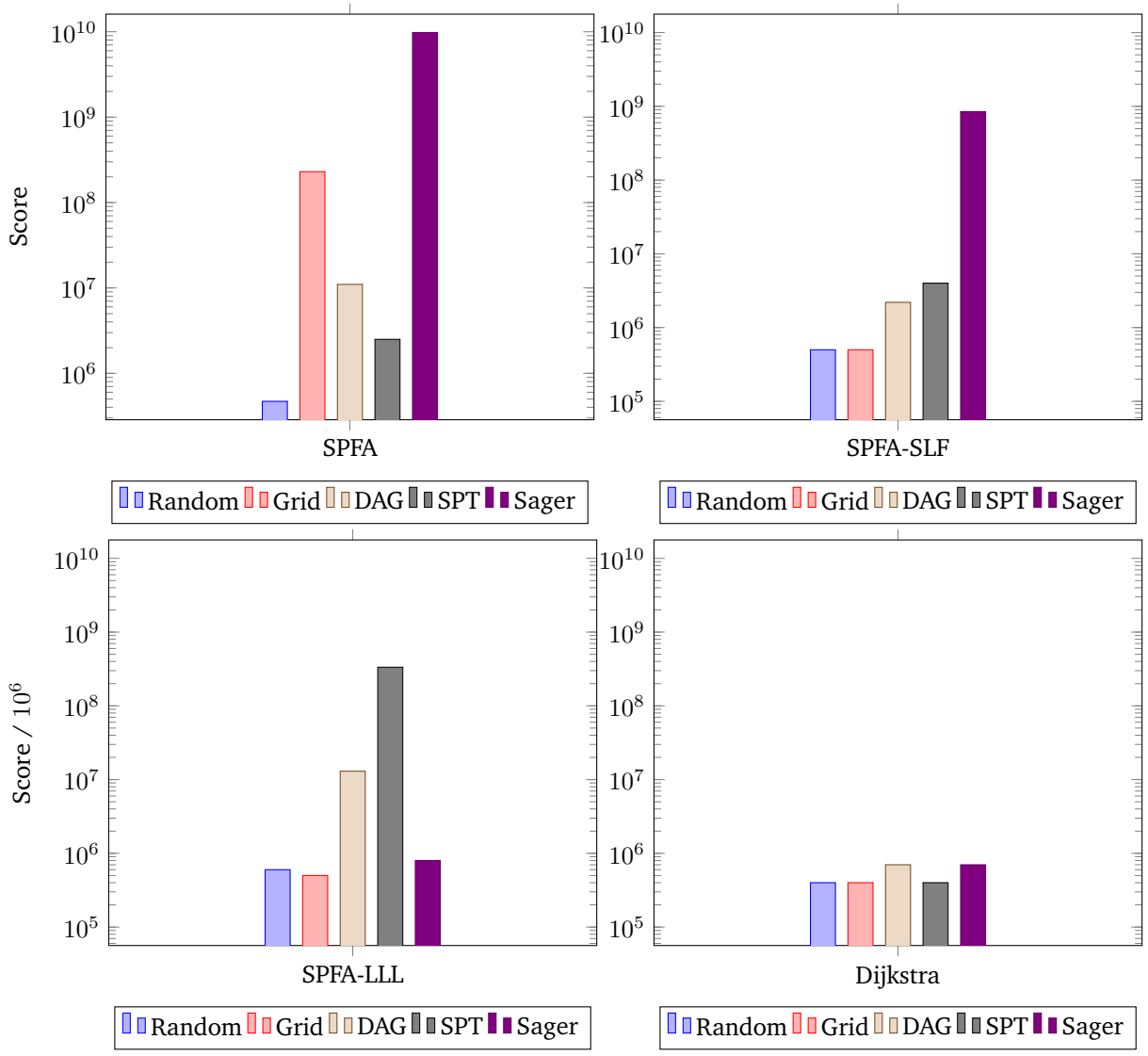

Figure 7: Comparison on 10,000 Nodes Graph

the overall structure of the graph, which means scaling up local properties cannot reach the bottleneck of the algorithm. Moreover, there are many shape of graphs that Sager can't synthesize due to its scaling strategy. For example, Sager can't synthesize a grid graph from a square with 4 nodes.

6 Collaborations

Mike He worked on writing implementations of SPFA and its variants for Sager synthesizer and for benchmark, addressing bugs encountered while implementing the solver for graph synthesis, implementing common strategies of hacking SPFA and its heuristics, doing the presentation and writing the report. Yihong works on formulating a general idea that both of us can work on, as well as on implementing different solving strategy and doing literature search on symmetry breaking of graph representation. He developed the first version of SPFA algorithm as well as the pipeline to automating the testing and benchmarking process. He also collects the final data.

7 Course Topics

There are multiple topics we covered while developing Sager. We applied optimizations for **symbolic evaluation** to make this procedure faster; we also used Rosette and z3-solver to synthesize the graph and used the **incremental solving** strategy to speed up the synthesizer. We use symbolic profiler to debug performance issues.

8 Conclusion

Sager makes it possible to generate graph structures in an automated way. With the help of SMT Solvers, Sager is able to synthesize "better" graphs than generators that relies on randomization. Sager also enables people without certain *a priori* insights of an algorithm to figure out whether there exists such an input that will cause the efficiency regression. Users only need to provide a concrete implementation of the algorithm and a cost function to obtain a data generator for free. Even though, there are still some limitations of this graph generating strategy, Sager can at least provide the user a "bottom line" about what kind of structure may cause the algorithm to run slow.

References

[1] Codish, M., Miller, A., Prosser, P., and Stuckey, P. J. Constraints for symmetry breaking in graph representation. *Constraints 24*, 1 (Jan 2019), 1–24.

[2] Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik 1*, 1 (Dec 1959), 269–271.

[3] Knuth, D. E., Morris, J. H., and Pratt, V. R. Fast Pattern Matching in Strings. *SIAM Journal on Computing 6*, 2 (March 1977), 323–350.

[4] Torlak, E., and Bodik, R. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, ACM, pp. 530–541.

[5] Wikipedia contributors. Shortest path faster algorithm — Wikipedia, the free encyclopedia, 2019. [Online; accessed 22-October-2019].