An evaluation of skewed abulation and staged solving on the topic of Context-Free Language Reachability optimization

ZIYU ZHOU, Fudan University, China

Context-free language reachability (CFL-reachability) is a fundamental framework in the field of program analysis. The optimization of the CFL-reachability solving strategy has been extensively studied, yet it remains an open problem in the field of program analysis. In this paper, we evaluate two recent works on the topic of CFL-reachability, namely *skewed tabulation* and *staged soving*. We reproduce the experiments in the original papers to evaluate the efficiency of the two methods. We also conduct a comparative study to compare the performance of the two methods on the SPEC CPU 2017 benchmark. The results show that the staged solving method is more efficient than the skewed tabulation method in most cases. However, the skewed tabulation method is more memory-efficient in some cases. This work provides a reference for the choice of CFL-reachability solving methods in practice. The datasets and evaluation scripts are available at https://github.com/megrezbunny/An-evaluation-of-skewed-tabulation-and-staged-solving.

CCS Concepts: • **Theory of computation → Grammars and context-free languages**.

Additional Key Words and Phrases: Context-Free Language, Reachability, Program Analysis

1 Introduction

Context-Free language reachability (CFL-reachability) is a fundamental framework in the field of program analysis[8]. A wide range of program analysis problems can be formulated as CFL-reachability problems, including pointer analysis[14], value-flow analysis[13], interprocedural data-flow analysis[9], taint analysis[2], and type-based flow analysis[7].

The setting of a typical CFL-reachability problem includes two basic elements: An edge-labeled bidirectional graph $G = (V, E)$ representing the program being analyzed, and a context-free Language (CFL) $L = (N, \Sigma, P, S)$ modeling the property of our interest. N and Σ are two disjoint finite sets of symbols, called the nonterminal set and the terminals set respectively; The set P consists of production rules, each of the form $a \rightarrow b$, where $a \in N$ is a nonterminal and $b \in (N \cup \Sigma)^*$ is a string of symbols; The start symbol is denoted by $S \in N$. Each edge in E is labeled with a symbol $X \in (N \cup \Sigma)$, and is referred to as an X-edge $u \xrightarrow{X} v$. Two nodes u and v are said to be CFL-reachable iff there exists a path p from u to v such that the string concatenation of edge labels along the path is accepted by L. The objective is to determine whether pair of nodes (u, v) is CFL-reachable for every $(u, v) \in V \times V$.

There is a standard algorithm[6] which can solve the CFL-reachability problem with a time complexity of $O(n^3)$, where n is the number of nodes. The idea of the standard algorithm is to iteratively generate *summary edges* until a fix point is reached. Before applying the standard algorithm, the CFG needs to be transformd into a normalized form, which ensures that there are at most two symbols on the right side of any production rule. Then, the algorithm facilitates the production rules to generate summary edges using the present edges. For a production rule $A \rightarrow BC$, the algorithm generates a summary edge $u \xrightarrow{A} v$ for each pair of nodes (u, v) such that there is a node w with $u \xrightarrow{B} w$ and $w \xrightarrow{C} v$.

The optimization of the CFL-reachability solving strategy has been extensively studied, yet it remains an open problem in the field of program analysis. [1] has achieved a slightly subcubic time complexity, which is the best known result so far in terms of theoretical complexity. In practice, a variety of works have been proposed to improve the efficiency of CFL-reachability solving, including [4] and [10]. [4] introduces a new tabulation technique called *skewed tabulation* to speed up the

Author's Contact Information: Ziyu Zhou, 21302010028@m.fudan.edu.cn, Fudan University, Shanghai, China.

CFL-reachability by eliminitaing the wasted and unnecessary summary edges; [10] proposes an alternative approach called *staged solving*, which decomposes the CFG into a smaller CFG $\mathcal{L}$ and a regular language $\mathcal{R}$, and solves the $\mathcal{L}$-reachability and $\mathcal{R}$-reachability in two distinct stages. The $\mathcal{R}$-reachability can be computed with near quadratic complexity because of the regularity of $\mathcal{R}$, thus significantly improving the overall efficiency.

In this paper, we focus on the two methods above and perform an evaluation of their efficiency, both respectively and comparatively. The results show that the staged solving method is more efficient than the skewed tabulation method in most cases. However, the skewed tabulation method is more memory-efficient in some cases.

The rest of the paper is organized as follows. In Section 2 and 3, we introduce the two methods and evaluate them by reproducing the experiments in the original papers respectively. In Section 4, we conduct a comparative study to compare the performance of the two methods on the SPEC CPU 2017 benchmark. Finally, Section 5 discusses the limitations and conclude the paper.

2 Skewed Tabulation Evaluation

This section gives a brief description of the skewed tabulation technique proposed in [4], and evaluates the efficiency of the method by reproducing the experiments in the original paper.

2.1 Skewed Tabulation

the skewed tabulation consists of two parts: static skewing and dynamic skewing.

The static skewing is performed during the preprocessing stage, where the CFG is restructured to another form, which promotes recursive behavior from the nonterminals more close to the start symbol on the parse tree, and eliminates the recursive behavior from the nonterminals more far from the start symbol. For instance, suppose the CFG is $S \longrightarrow AB$, $B \longrightarrow Bb$. The static skewing will transform the CFG into $S \longrightarrow AB$, $S \longrightarrow Sb$. In this way, the recursive behavior of B is eliminated and the recursive behavior of S is promoted.

The dynamic skewing is performed during the runtime. The authors show that some summary edges generated are unnecessary to be inserted into the graph. These edges, called *propagating edges*, are identified by the dynamic skewing algorithm and are not inserted into the graph, thus reducing the number of edges and improving the efficiency of the algorithm.

2.2 Evaluation Setting

In the original paper, the authors implement the skewed tabulation method on top of a previous CFL-reachability solver POCR [5]. They evaluate their artifact `CFL-Skewed` on the SPEC CPU 2017 benchmark, excluding 3 unlinkable programs. The graphs are constructed by an open-source tool SVF [12], and preprocessed with some graph simplification techniques. POCR is referred to as `CFL-RHS` in the paper as the baseline. The evaluation includes 3 subjects: alias analysis[14], value-flow analysis[13], and taint analysis[2]. They set a time limit of 48 hours and a memory limit of 128G for each run.

With their published artifact and graphs, we conduct the evaluation on the same benchmark and subjects, except for the taint analysis. The experiments are performed on a machine with an 18-core 3.00GHz Intel i9-10980XE CPU and 64GB RAM, running Ubuntu 20.04. We set a time limit of 600 seconds and a memory limit of 64G for each run. The results are measured in terms of runtime and memory usage.

2.3 Result Analysis

The results are shown in Table 1 and Table 2. It can be seen that in terms of time consumption, CFL-Skewed outperforms CFL-RHS in all cases. On average, CFL-Skewed achieves a 2.62x speedup

Program	Alias Analysis		Value-Flow Analysis	
	CFL-RHS	CFL-Skewed	CFL-RHS	CFL-Skewed
cactus	-	-	11.25	10.72
deepsjeng	41.66	23.06	6.32	5.16
imagick	-	-	-	-
lbm	0.09	0.04	0.02	0.02
leela	161.69	72.51	8.27	7.74
mcf	4.25	2.08	0.09	0.07
nab	4.01	1.4	10.08	10.02
omnetpp	-	-	480.2	308.13
parest	-	-	-	-
perlbench	-	-	-	-
povray	-	-	-	-
x264	337.74	65.08	172.37	160.36
xz	2.96	1.5	2.65	2.52

Table 1. Skewed Tabulation: Running time (in seconds) results on SPEC2017 benchmarks. The - mark indicates out-of-time.

Program	Alias Analysis		Value-Flow Analysis	
	CFL-RHS	CFL-Skewed	CFL-RHS	CFL-Skewed
cactus	-	-	266.50	244.27
deepsjeng	161.44	196.56	28.17	21.46
imagick	-	-	-	-
lbm	4.68	2.50	1.78	1.72
leela	262.99	86.15	78.90	63.36
mcf	26.20	13.83	3.58	3.07
nab	31.27	13.20	49.05	43.87
omnetpp	-	-	1835.87	979.96
parest	-	-	-	-
perlbench	-	-	-	-
povray	-	-	-	-
x264	609.39	227.04	534.74	340.98
xz	27.29	8.41	38.41	31.84

Table 2. Skewed Tabulation: Memory consumption (in MB) results on SPEC2017 benchmarks.

for alias analysis and a 1.15x speedup for value-flow analysis. In the original paper, the authors claim that CFL-Skewed achieves a 3.34x speedup for alias analysis, and a 1.13x speedup for value-flow analysis. The results are consistent considering the difference in the hardware environment.

Regarding memory consumption, the results are almost exactly the same as the Table 3 in the original paper, with only one exception. Our results show that CFL-RHS comsumes 161.44MB and 196.56MB memory respectively in the alias analysis of `deepsjeng`, while these two numbers are 196.43MB and 161.20MB in the original paper. **We suspect that the authors may mave made a mistake to swap those two numbers accidentally**, as the other numbers are consistent with the original paper. Nevertheless, the results show that on average, CFL-Skewed reduces 47.00% memory

consumption for alias analysis and 20.00% memory consumption for value-flow analysis, compared to CFL-RHS. This result not too far from the original claim of 60.05% and 20.38% respectively.

3 Staged Solving Evaluation

This section gives a brief description of the staged solving technique proposed in [10], and also, evaluates the efficiency of the method by reproducing the experiments in the original paper.

3.1 Staged Solving

The staged solving strategy tries to decompose the original CFG into a smaller CFG $\mathcal{L}$ and a regular language $\mathcal{R}$. In practice, the subproblem of $\mathcal{L}$-reachability only involves a small partition of the original edges; Meanwhile, the $\mathcal{R}$-reachability can be computed with near quadratic complexity because of the regularity of $\mathcal{R}$. The challange is to find such a decomposition that $\mathcal{L}$ does not depend on $\mathcal{R}$, and then solve the two subproblems sequentially.

The authors propose two essential concepts: the *context-free patterns(CFPs)* and the *CFP-based grammar decomposition*. the CFPs are utilized to express the context-free aspect of a program analysis problem formulated as CFL-reachability. The authors discuss the CFPs they identify across a variety of program analysis problems in the paper, including alias analysis and value-flow analysis.

The CFP-based grammar decomposition is a technique to decompose the original CFG into a smaller CFG $\mathcal{L}$ and a regular language $\mathcal{R}$, tackling the core problem of mutual dependencies between the nonterminals in the original CFG. The authors elaborate on the decomposition process and the solution to the two subproblems in the paper.

3.2 Evaluation Setting

The authors implement the staged solver `STG` on top of SVF[12] and LLVM[3], and compare it with a baseline solver `Subcubic` implementing the subcubic algorithm[1]. Two other state-of-the-art solvers `POCR`[5] and `PEARL`[11] are also included. The authors further combine the staged solving strategy with `POCR` and `PEARL`, resulting in `POCR-STG` and `PEARL-STG` respectively. The benchmark consists of 10 programs they chosen from the SPEC CPU 2017 benchmark and the graphs extracted by SVF[12]. Their evaluation includes alias analysis and value-flow analysis, with a time limit of 6 hours and a memory limit of 512GB for each run.

We employ their artifact and graphs to conduct the evaluation on the same benchmark and subjects for all 6 solvers. The hardware environment, time limit, and memory limit are the same as the skewed tabulation evaluation in Section 2.2. The results are measured in terms of runtime and memory usage.

3.3 Result Analysis

The results are shown in Table 3 and Table 4. It can be seen that In value-flow analysis, only `STG` can solve all the programs within the time limit and memory limit. The other solvers, including `SUBCUBIC`, `POCR`, `PEARL`, `POCR-STG`, and `PEARL-STG`, all exceed memory limit on some programs. The average speedup of `STG` over `SUBCUBIC` is 128.27x, which is different from the original claim of 861.59x, but acceptable considering the difference in experiment configurations. In alias analysis, the speedup is 3.49x, which is close to the original claim of 4.1x.

In terms of memory consumption, the results we obtained are nearly the same as the original paper. A slight problem is that, due to the performance difference of the hardware environment, most experiments take about 6x longer time than the original paper, but the memory should not be the bottleneck in most cases. **So we suspect that the evalutation script provided by the authors seems to mistake the time-out as memory-out most of the time, which may lead to some confusion in the results.**

Program	SUBCUBIC		STG		POCR		POCR-STG		PEARL		PEARL-STG	
	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem
cactus	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	482.46	2.68
imagick	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	-	-
leela	64.76	0.2	12.3	0.24	20.46	0.36	10.55	0.5	27.01	0.12	6.25	0.25
nab	11.61	0.11	4.15	0.13	9.39	0.11	4.48	0..15	5.86	0.06	2.44	0.14
omnetpp	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM
parest	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	339	2.72
perlbench	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM
povray	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM	367.43	2.01
x264	183.28	0.58	73.64	1.02	92.18	0.84	66.17	1.4	89.15	0.46	56.32	1.07
xz	7.2	0.06	2.12	0.09	3.22	0.06	1.74	0.11	2.47	0.05	1.86	0.1

Table 3. Staged Solving: Running time (in seconds) and memory consumption (in GB) results on SPEC2017 benchmarks of alias analysis. The - mark indicates time-out.

Program	SUBCUBIC		STG		POCR		POCR-STG		PEARL		PEARL-STG	
	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem
cactus	OoM	OoM	58.92	15.49	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM
imagick	OoM	OoM	36.65	14.93	OoM	OoM	324.16	59.16	OoM	OoM	235.71	18.61
leela	OoM	OoM	2.9	0.39	OoM	OoM	4.57	0.74	90.27	1.06	3.89	0.48
nab	506.49	0.7	3.3	0.26	OoM	OoM	87.62	7.95	59.37	0.71	4.81	0.37
omnetpp	OoM	OoM	73.29	20.94	OoM	OoM	182.81	34.28	OoM	OoM	213.09	22.63
parest	304.08	3.32	9.53	1.8	204.26	6.34	12.21	2.09	32.96	4.33	12	2.12
perlbench	OoM	OoM	106.48	26.46	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM
povray	OoM	OoM	86.11	13.9	OoM	OoM	OoM	OoM	OoM	OoM	OoM	OoM
x264	OoM	OoM	13.49	3.64	OoM	OoM	213.64	25.61	OoM	OoM	50.79	4.73
xz	309.13	0.7	1.55	0.22	194.72	7.93	2.21	0.4	23.13	0.59	1.82	0.28

Table 4. Staged Solving: Results of value-flow analysis.

4 Comparative Study

4.1 Evaluation Setting

Despite the fact that the two methods are both evaluated on the SPEC CPU 2017 benchmark, the programs chosen are different, and the graphs generated by SVF[12] are also different in the two papers. To make a fair comparison between the two methods,we collect the graphs generated by SVF[12] in the staged solving evaluation benchmark, and run the skewed tabulation artifact on the graphs. All experiment configurations are the same as the previous two sections.

4.2 Result Analysis

The results are shown in Table 5 and Table 6. It can be seen that when running on the same graphs, the performance of the two methods is quite different.

In alias analysis, the staged solving method `STG` outperforms the skewed tabulation method `CFL-Skewed` in both time consumption and memory consumption, achieving a 15.9x speedup and a 1.23x memory consumption decrease on average; While in value-flow analysis, though `STG` is still faster than `CFL-Skewed` with an average speedup of 12.1x, `CFL-Skewed` outperforms `STG` in terms of memory consumption, with an average memory decrease of 3.94x.

Program	SUBCUBIC		STG		CFL-RHS		CFL-Skewed	
	Time	Mem	Time	Mem	Time	Mem	Time	Mem
cactus	OoM	OoM	OoM	OoM	-	-	-	-
imagick	OoM	OoM	OoM	OoM	-	-	-	-
leela	64.76	0.2	12.3	0.24	-	-	-	-
nab	11.61	0.11	4.15	0.13	175.104	0.4	63.53	0.19
omnetpp	OoM	OoM	OoM	OoM	-	-	-	-
parest	OoM	OoM	OoM	OoM	-	-	-	-
perlbench	OoM	OoM	OoM	OoM	-	-	-	-
povray	OoM	OoM	OoM	OoM	-	-	-	-
x264	183.28	0.58	73.64	1.02	-	-	-	-
xz	7.2	0.06	2.12	0.09	70.8	0.21	34.86	0.09

Table 5. alias analysis results of both methods on SPEC2017 benchmarks. The - mark indicates out-of-time.

Program	SUBCUBIC		STG		CFL-RHS		CFL-Skewed	
	Time	Mem	Time	Mem	Time	Mem	Time	Mem
cactus	OoM	OoM	58.92	15.49	-	-	-	-
imagick	OoM	OoM	36.65	14.93	-	-	-	-
leela	OoM	OoM	2.9	0.39	10.67	0.1	10.26	0.08
nab	506.49	0.7	3.3	0.26	114.84	0.18	114.16	0.17
omnetpp	OoM	OoM	73.29	20.94	-	-	-	-
parest	304.08	3.32	9.53	1.8	12.52	0.32	12.14	0.29
perlbench	OoM	OoM	106.48	26.46	-	-	-	-
povray	OoM	OoM	86.11	13.9	-	-	-	-
x264	OoM	OoM	13.49	3.64	-	-	-	-
xz	309.13	0.7	1.55	0.22	14.43	0.08	14.37	0.07

Table 6. value-flow analysis results of both methods on SPEC2017 benchmarks.

In conclusion, the staged solving method is more efficient than the skewed tabulation method on the SPEC2017 benchmark, especially in alias analysis. However, the skewed tabulation method is more memory-efficient in some cases.

5 Limitation and Conclusion

Limitation. (1) Although the two artifacts are evaluated on the same benchmark programs and the same graphs, the comparability of the results is limited as we are unable to verify the two artifacts compute the same set of edges, which is not outputed by the artifacts. (2) The results may be influenced by the different hardware environments.

Conclusion. In this paper, we evaluate the skewed tabulation method and the staged solving method on the SPEC2017 benchmark, and conduct a comparative study between the two methods. The results show that the staged solving method is more efficient than the skewed tabulation method in most cases. However, the skewed tabulation method is more memory-efficient in some cases. This work provides a reference for the choice of CFL-reachability solving methods in practice.

References

[1] Swarat Chaudhuri. 2008. Subcubic algorithms for recursive state machines. *SIGPLAN Not.* 43, 1 (Jan. 2008), 159–169. doi:10.1145/1328897.1328460

[2] John Kodumal and Alex Aiken. 2004. The set constraint/CFL reachability connection in practice. *SIGPLAN Not.* 39, 6 (June 2004), 207–218. doi:10.1145/996893.996867

[3] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. doi:10.1109/CGO.2004.1281665

[4] Yuxiang Lei, Camille Bossut, Yulei Sui, and Qirun Zhang. 2024. Context-Free Language Reachability via Skewed Tabulation. *Proc. ACM Program. Lang.* 8, PLDI, Article 221 (June 2024), 24 pages. doi:10.1145/3656451

[5] Yuxiang Lei, Yulei Sui, Shuo Ding, and Qirun Zhang. 2022. Taming transitive redundancy for context-free language reachability. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 180 (Oct. 2022), 27 pages. doi:10.1145/3563343

[6] David Melski and Thomas Reps. 2000. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science* 248, 1 (2000), 29–98. doi:10.1016/S0304-3975(00)00049-9 PEPM'97.

[7] Nomair A. Naeem and Ondrej Lhotak. 2008. Typestate-like analysis of multiple interacting objects. *SIGPLAN Not.* 43, 10 (Oct. 2008), 347–366. doi:10.1145/1449955.1449792

[8] Thomas Reps. 1998. Program analysis via graph reachability1An abbreviated version of this paper appeared as an invited paper in the Proceedings of the 1997 International Symposium on Logic Programming [84].1. *Information and Software Technology* 40, 11 (1998), 701–726. doi:10.1016/S0950-5849(98)00093-7

[9] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 49–61. doi:10.1145/199448.199462

[10] Chenghang Shi, Haofeng Li, Lu Jie, and Lian Li. 2024. Better Not Together: Staged Solving for Context-Free Language Reachability. 1112–1123. doi:10.1145/3650212.3680346

[11] Chenghang Shi, Haofeng Li, Yulei Sui, Jie Lu, Lian Li, and Jingling Xue. 2023. Two Birds with One Stone: Multi-Derivation for Fast Context-Free Language Reachability Analysis. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 624–636. doi:10.1109/ASE56229.2023.00118

[12] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC '16)*. Association for Computing Machinery, New York, NY, USA, 265–266. doi:10.1145/2892208.2892235

[13] Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Transactions on Software Engineering* 40, 2 (2014), 107–122. doi:10.1109/TSE.2014.2302311

[14] Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. *SIGPLAN Not.* 431, 1 (Jan. 2008), 197–208. doi:10.1145/1328897.1328464