AVERAGE CASE ANALYSIS OF SOME ELIMINATION-BASED DATA-FLOW ANALYSIS ALGORITHMS

JOHANN BLIEBERGER

Department for Computer-Aided Automation, TU Vienna

Abstract. The average case of some elimination-based data-flow analysis algorithms is analyzed in a mathematical way. Besides this allows for comparing the timing behavior of the algorithms, it also provides insights into how relevant the underlying statistics are when compared to practical settings.

1. Introduction

Elimination-based approaches [22] are used for data flow analysis problems [20, 21, 18, 10, 3, 7] that cannot be solved with iterative approaches [16, 12]. There exist other applications for elimination methods, which go beyond the area of program analysis [27]. For solving data flow analysis problems there are two families of elimination-based approaches: algebraic methods and methods using path expressions.

Algebraic elimination methods [1, 13, 11, 25] consist of three steps: (1) reducing the flowgraph to a single node, (2) eliminating variables in the data flow equations by substitution, and (3) back-propagating the solution to other nodes. Algebraic elimination methods require two algebraic operations for a set of equations: *substitution* and *loop-breaking*. The substitution transformation is the replacement of the occurrence of a variable by its term whereas loop-breaking eliminates the occurrence of a variable on the right-hand side. Though not very efficient, Gaussian elimination is a generic algebraic elimination method to solve data flow equations in cubic time [19].

Path expressions were introduced in [27] to solve data flow equations. The flowgraph is seen as a deterministic finite state automaton [14] whose language consists of all paths emanating from the start node to a node. The language is represented as a regular expression whose alphabet is the edge-set of the flowgraph. To find the data flow solution of a node, a path homomorphism is applied to the path expression. The operators $\cdot$, $\cup$, and $*$ of the regular expressions are reinterpreted. An elimination method using path expressions comprises two steps: (1) the computation of path expressions for all nodes in the flowgraph, and (2) the application of the path homomorphism. An inefficient algorithm for converting flowgraph to path expressions is described in [14] and runs in $\mathcal{O}(n^3)$.

As an example Figure 2 shows the CFG of the program fragment given in Figure 1. Node 3 is the if-statement. The edge to Node 4 is the then-branch and is followed only if $c1$ is *true*. The edge $3 \to 2$ has assigned condition $\neg c1 \wedge \neg c3$ and the edge $3 \to 6$ has assigned $\neg c1 \wedge c3$. In a similar way edges $5 \to 4$, $5 \to 2$, and $5 \to 6$

Dagstuhl Seminar Proceedings 08161
Scalable Program Analysis
http://drops.dagstuhl.de/opus/volltexte/2008/1572

```
begin                          -- Node 1
   repeat                      -- Node 2
      if c1 then               -- Node 3
         repeat                -- Node 4
            ...                -- Node 4
         until c2              -- Node 5
      endif
   until c3                    -- Node 5
end                            -- Node 6
```

FIGURE 1. Example: Program Source Code

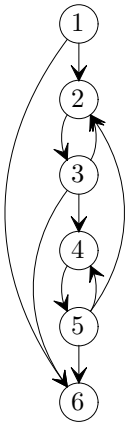


FIGURE 2. Example: Control Flow Graph

have assigned conditions $c2$, $c2 \wedge \neg c3$, and $c2 \wedge c3$, respectively. Edge $1 \rightarrow 6$ is only present to facilitate algorithms performed on the CFG and has assigned *false*. All the other edges have assigned *true*.

In this paper we present a survey on how the average case behavior of some elimination algorithms can be determined. In detail we present the main ideas of [4] in Section 2, which shows that Sreedhar's algorithm ([26]) behaves linearly on the average. In Sections 3 to 5 we show how the average case timing behavior of an algorithm presented in [23] can be determined. It turns out that the the difference between the average case and the worst-case behavior is only a small constant. In contrast, employing different statistics an average case behavior of the same algorithm can be found which fits practical settings much better.

2. Sreedhar's Algorithm

Sreedhar et al. [26] have presented an efficient and easy to implement elimination-based algorithm to solve data flow problems. The algorithm starts with a general (reducible) directed CFG G. The union of G and the dominator tree of G is called a *DJ graph*. The data flow problem is solved by redirecting and removing edges in the DJ graph until the remaining graph is the dominator tree of G. Because the

dominator tree is part of the DJ graph, each node can be assigned a certain level (equal to its distance from the root).

Edges being part of the dominator tree and not being part of G are called *d-edges*. Edges being part of the dominator tree and of G are called *dj-edges*. The remaining edges are *j-edges*[1].

Three different operations are performed in a bottom-up fashion on the graph by the algorithm: *Eager1*, *Eager2a*, and *Eager2b*.

Sreedhar et al. [26, 24] give a thorough worst-case performance analysis of the algorithm showing that the number of Eager (Eager1 + Eager2a + Eager2b) operations is at most $O(e \cdot n)$ where n denotes the number of nodes and e denotes the number of edges in G. A more detailed description and analysis of Sreedhar's algorithm and how the DJ graph can be used to solve the underlying system of equations, can be found in [26].

In contrast, Sreedhar reports a linear, i.e., $O(e)$, time behavior based on some practical application programs.

In [4] we have proved that for goto-free programs, the average case behavior is indeed linear. By "goto-free" programs we mean programs written in programming languages without a goto statement like *Modula-2* [30] and *Java* [2] or programs not using goto statements or statements with similar effects (cf. [9]). Some programming languages allow to exit loop statements not only at the beginning (while-loops) and at the end (repeat-loops) of loop-statements, but also at certain points within the loop body. Exit-statements are a form of "tamed" goto-statements, which while retaining structured programs, give more freedom to the programmer and often result in more readable and understandable program code. The analysis in [4] covers such *exit-statements*, too. As a byproduct the results of [4] also apply to the average size of the so-called *dominance frontier* [8]. The *dominance frontier* $DF(u)$ of a CFG node u is defined as the set of all CFG nodes v such that u dominates a predecessor of v but does not strictly dominate v.[2]

In the following we present the main ideas of the proof given in [4].

DJ graphs for goto-free programs can be derived by a graph grammar. One single production of this grammar is shown in Figure 3.

By assigning a probability to each production of the graph grammar, we are able to determine how probable a DJ graph with n nodes is. Using multivariate generating functions and well-known methods from singularity analysis we can prove the following theorem, where the p_i refer to probabilities assigned to some specific productions of the graph grammar given in [4] and the *branch predicate* is true if the grammar allows for branching language features such as if-statements.

Theorem 1. *Let G be a DJ graph with n nodes which can be derived by the graph grammar given in* [4]*. Then the average number of Eager2b operations performed by Sreedhar's algorithm $\mathcal{E}_n$ can be determined as follows.*

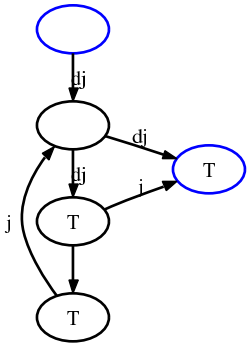
If the branch predicate is true, then

(a) *if further* $p_4 = p_6 = p_8 = p_9 = p_{10} = p_{11} = p_{12} = p_{13} = p_{14} = p_{15} = p_{16} = p_{17} = p_{18} = p_{19} = p_{20} = p_{21} = p_{22} = p_{23} = 0$,

$$\mathcal{E}_n = c_1 \cdot n^{3/2} + O\left(n^{1/2}\right),$$

[1] Sreedhar et al. [26] only introduced d- and j-edges; we have defined dj-edges in order to facilitate the description of our graph grammar

[2] Node x strictly dominates y if x dominates y but $x \neq y$.



$T \to T_0$ loop T exit when c T endloop T

FIGURE 3. Production of Graph Grammar

(b) *otherwise*

$$\mathcal{E}_n = c_2 \cdot n + O(1).$$

If the branch predicate is false and

(c) *if further* $p_{10} = p_{11} = 0$, *we obtain*

$$\mathcal{E}_n = c_3 \cdot n^2 + O(n),$$

(d) *otherwise*

$$\mathcal{E}_n = c_4 \cdot n + O(1).$$

□

Taking a closer look at Theorem 1 we find that programming languages of case (c) consist of repeat-until-loops only. Programming languages of case (a) support only restricted forms of if-statements and repeat-until-loops. Except for such very uncommon languages, the following corollary holds.

Corollary 1. The average number of Eager2b operations performed by Sreedhar's algorithm for goto-free programs is linear in the size of the program. □

Remark 1. Note that a program P, consisting only of straight-line code and repeat-until-loops written in a certain programming language L that also supports other language features like if-statements and while-loops, implies quadratic running time of Sreedhar's algorithm for this specific program P.

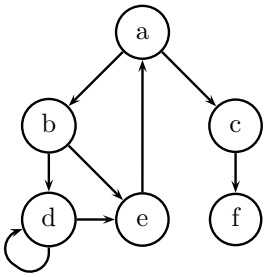
Since, however, the probabilities for if-statements and while-loops are non-zero, according to Theorem 1 the *average* case performance of Sreedhar's algorithm for programs written in L is linear.

It is shown in [26] that the number of Eager2b operations corresponds to the size of the dominance frontier [8] (which is needed for SSA[3] analysis, a method common in compiler construction).

Thus, we have also proved the following corollary as a byproduct.

Corollary 2. Under the same assumptions as in Corollary 1, the average size of the dominance frontier is linear in the program size. □

[3]Static Single Assignment



(a) Flowgraph

$$
\begin{aligned}
P(a,a) &= (a \to b \cdot (b \to d \cdot d \to d^* \cdot d \to e \cup \\
&\qquad b \to e) \cdot e \to a)^* \\
P(a,b) &= P(a,a) \cdot a \to b \\
P(a,c) &= P(a,a) \cdot a \to c \\
P(a,d) &= P(a,b) \cdot b \to d \cdot d \to d^* \\
P(a,e) &= P(a,b) \cdot b \to e \cup P(a,d) \cdot d \to e \\
P(a,f) &= P(a,c) \cdot c \to f
\end{aligned}
$$

(b) Regular Expressions

FIGURE 4. Flowgraph and Regular Path Expressions

3. Annotated Decomposition Trees

For elimination frameworks a new data structure called Annotated Decomposition Tree (ADT) that recursively splits the reducible flowgraph into intervals has been introduced in [23]. An interval is a subgraph of the flowgraph and has the following properties: (1) every interval has a single entry node, and (2) the single-entry node of the interval dominates all nodes of the interval.

The ADT is a binary leaf tree. An inner node in the ADT represents a composition operation that composes two disjoint intervals G_1 and G_2. The leaves of the tree represent trivial intervals consisting of a single node in the flowgraph[4]. The composition operation is a generalization of work published in [28, 29, 15].

Definition 1. Let $G_1(V_1, E_1, r_1)$ and $G_2(V_2, E_2, r_2)$ be flowgraphs such that V_1 and V_2 are disjoint sets. The composition $G_1 \oplus_{(F,B)} G_2$ is defined as

$$(V_1 \cup V_2, E_1 \cup E_2 \cup (F \times \{r_2\}) \cup (B \times \{r_1\}), r_1)$$

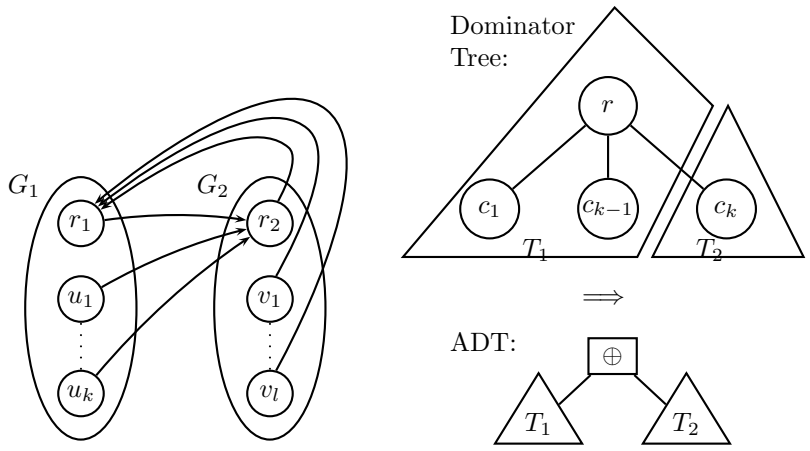
where $F \subseteq V_1$ and $B \subseteq V_2$ denote the sources of the forward and backward edges. Node r_1 becomes the new single-entry node of the composed interval.

The composition of two intervals G_1 and G_2 is depicted in Figure 5(a). The single-entry nodes of the intervals are denoted by r_1 and r_2. The edge set $F \times \{r_2\}$ connects a subset of nodes in G_1 to r_2. The edge set $B \times \{r_1\}$ connects a subset of nodes in G_2 to r_1.

By Definition 1, root node r_1 dominates all nodes of G_1 and G_2 because every node in the composed interval can only be reached via r_1. The same holds for r_2, i.e., r_2 dominates all nodes in G_2. This implies that the nodes of G_1 form a sub-tree in the dominator tree with r_1 as a root-vertex of the sub-tree, and single-entry node r_2 is immediately dominated by r_1.

The forward edges of a reducible flow graph form a directed acyclic graph imposing a topological order $<$ such that for all edges $(u, v) \in E_F$, $u < v$ holds. Since the single-entry node of an interval dominates all nodes in the interval, the single-entry node of the interval is smaller than the nodes in the interval with respect to the topological order. The composition implies that $r_1 < r_2$. Given a composition

[4]Because ADTs are binary leaf trees, there are $n - 1$ inner nodes where n is the number of leaves.



(a) Composition (b) Decomposition

FIGURE 5. Composition and Decomposition of Reducible Flowgraphs.

$G_1 \oplus G_2$, the inequality

$$\forall u \in V_1 : \forall v \in V_2 : r_1 \leq u < r_2 \leq v \tag{1}$$

holds. Assume a total order R of nodes in the flowgraph $[u_1, \ldots, u_n]$ such that for $(u_i, u_j) \in E_F$, $i < j$. An interval decomposition of the flowgraph partitions the ordered nodes into two parts. Vertex r_1 has index 1 and all the nodes between 1 and $r_2 - 1$ belong to the interval G_1. The nodes from r_2 to n belong to G_2. By recursively applying the decomposition for ordered nodes, we have a range representation of the tree. For example a possible total order for the flowgraph in Figure 4(a) is $[a, b, d, e, c, f]$. The first composition of the ADT splits the ordered nodes in two halves, i.e., $[[a, b, d, e], [c, f]]$. By recursively splitting intervals, we obtain $[[[a], [[b, d], e]], [c, f]]$ representing the intervals of the flowgraph.

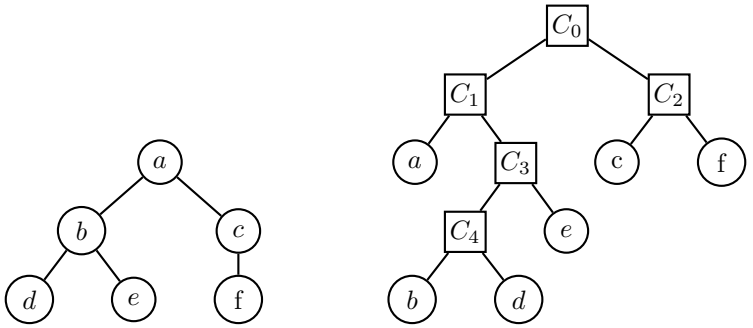
Without proof we state that each reducible flowgraph can be decomposed into two smaller reducible flowgraphs (w.r.t. the composition operation $\oplus$) until only trivial flowgraphs (comprising one single node) are left. Keeping track of the decomposition process by a tree-like structure is straight-forward. By decorating the tree structure with information on the edges we get *Annotated Decomposition Trees* (ADTs).

The ordered dominator tree and the decomposition tree of Figure 4 are displayed in Figure 6.

Assuming that all ADTs (i.e., reducible CFGs) are equally probable we are able to analyze the average case behavior of the ADT-based algorithm presented in [23]. In addition, as pointed out in [5], we need to

(1) determine the number of ADTs with n nodes and
(2) find the average path length of ADTs that corresponds to the number of update operations of the algorithm [23].

The next sections will be concerned with these tasks.



(a) Ordered Dominator Tree (b) Decomposition Tree

FIGURE 6. Dominator and Decomposition Tree of Example.

4. Enumerating Reducible Flow Graphs

Using the decomposition method illustrated above, we get the following recurrence relation for the number of reducible flow graphs:

$$r_0 = 1, \qquad r_n = 2^n \sum_{k=0}^{n-1} r_k r_{n-k-1}$$

The first few values of r_n are displayed in Table 1.

Pulling out the major factor we obtain

$$r_n = 2^{\frac{n^2+3n}{2}} p_n$$

and a second recurrence relation

$$p_0 = 1, \qquad p_n = \frac{1}{2} \sum_{k=0}^{n-1} \frac{p_k p_{n-k-1}}{2^{k(n-k-1)}}$$

Now we can prove the following theorem (cf. [6]).

Theorem 2. *The limit* $\lim_{n\to\infty} p_n$ *exists and*

$$\beta = \lim_{n\to\infty} p_n = 0.71533743361486974094407547448471 15\ldots. \quad \square$$

In fact one can even provide a complete asymptotic expansion of p_n (details can be found in [6]) as follows.

Theorem 3. *Let the sequence* (α_j) *be defined by*

$$\alpha_0 = 1,$$

$$\alpha_j = -\frac{1}{2^j - 1} \sum_{i=1}^{j} 2^{j(i+1)} p_i \alpha_{j-i}$$

for $j \geq 1$. *The first few values of* α_j *are* $\alpha_1 = -2$, $\alpha_2 = -\frac{16}{3}$, *and* $\alpha_3 = -\frac{19 \cdot 2^8}{3 \cdot 7}$.

Then we have for each $s \geq 0$

$$p_n = \beta \cdot \left(\sum_{r=0}^{s} \frac{\alpha_r}{2^{rn}} + O_{s+1}\left(\frac{n}{2^{(s+1)n}}\right) \right), \ n \to \infty. \quad \square$$

n	r_n
1	1
2	2
3	16
4	288
5	10240
6	700416
7	92864512
8	24184487936
9	12484798840832
10	12835745584644096
11	26339606633209921536
12	107993030830149951553536
13	885112171099428768672907264
14	14505223494706550858367937544192
15	475365227058478388903633481696804864
16	31155437679322378551183102532362203824128
17	4083730172099442706977088626238641029051842560
18	1070541698651870103092428850642121888501001793568768
19	561276448465663221666495855250671194867830902552354881536
20	588543258383518726413960717220681231298059528311109827745021952

TABLE 1. The first few values of r_n.

Thus we can derive a complete asymptotic expansion of the number of reducible flow graphs.

Theorem 4. *The number of reducible flow graphs with n nodes fulfills for each s*

$$\beta \cdot 2^{\frac{n^2+n-2}{2}} \cdot \left(\sum_{r=0}^{s} \frac{\alpha_r}{2^{rn}} + O_{s+1}\left(\frac{n}{2^{(s+1)n}}\right)\right), (n \to \infty)$$

where

$$\beta = 0.71533743361486974094407547448471158 99\ldots$$

and the constants α_r are defined in Theorem 3. □

These results will be used in the following section to analyze the algorithm introduced in [23].

5. The average path length of ADTs

The number of update operations done by the algorithm presented in [23] is equal to the path length of the underlying ADT (cf. [5]).

Now, as shown in [5] the path length of ADTs can be computed by the recurrence relation:

$$s_n = (n+1)r_n + 2^{n+1} \sum_{k=0}^{n-1} r_k s_{n-k+1}$$

$$s_0 = 1,$$

where r_n is defined above.

n	s_n
1	1
2	8
3	128
4	3712
5	191488
6	17866752
7	3098476544
8	1022435000320
9	652025545097216
10	811180990981472256
11	1980240222899616612352
12	952198425277129733387059 2
13	904295244352414187653127208 96
14	169949888545332023189873690148 8640
15	633011289315344399669701362436464 96768
16	467837378813403404836595328639280423423 1808
17	686727923890534485219201697104950065817221857280
18	200364399252851344109979601356275212294786926916927488
19	116274912098600979308050056333407230201725474764183409000448

TABLE 2. The first few values of s_n.

The first few values of s_n are displayed in Table 2.

Recall that in the standard case, i.e., the path length of binary trees, the worst-case appears if the tree degenerates to a list, giving a path length of $\frac{n(n+1)}{2}$, in the best case we have a balanced tree with a path length of $n \log_2 n$ and on the average we get $n\sqrt{\pi n} - 3n + O(\sqrt{n})$ (cf. [17]).

Continuing our above calculations and again pulling out the major factor, we get a second recurrence relation

$$\begin{aligned} t_0 &= 0 \\ t_n &= (n+1)p_n + \sum_{k=0}^{n-1} \frac{p_k t_{n-k-1}}{2^{k(n-k-1)}} \end{aligned}$$

where p_n is defined above.

Pulling out another factor, we obtain

$$t_n = (n+1)(n+2)u_n$$

and the recurrence relation

$$\begin{aligned} u_0 &= \frac{1}{2} \\ u_n &= \frac{p_n}{n+2} + \sum_{k=0}^{n-1} \frac{(k+1)(k+2)}{(n+1)(n+2)} \frac{u_k p_{n-k-1}}{2^{k(n-k-1)}} \end{aligned}$$

Now, we are able to prove the following theorem (details can be found in [5]).

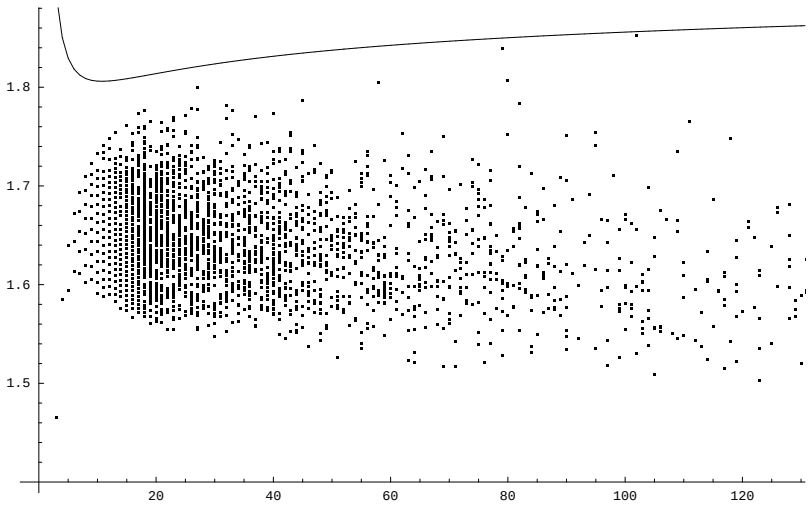


FIGURE 7. Numbers from SPEC2K

Theorem 5. *For $n \to \infty$ we have*

$$\frac{u_n}{p_n} = \frac{1}{2} + \frac{1}{n+2} + \frac{\psi}{(n+1)(n+2)} + O\left(\frac{n}{2^n}\right)$$

where

$$\psi = -1.838564071338257129893046559 48\ldots. \quad \square$$

Hence we obtain for the average number of update operations performed by the algorithm of [23].

Theorem 6. *For $n \to \infty$ we have*

$$\frac{t_n}{p_n} = (n+1)(n+2)\left(\frac{1}{2} + \frac{1}{n+2} + \frac{\psi}{(n+1)(n+2)} + O\left(\frac{n}{2^n}\right)\right). \quad \square$$

The path length of ADTs has its worst-case if the tree degenerates to a list. The worst-case is

$$\frac{(n+1)(n+2)}{2} + n = (n+1)(n+2)\left(\frac{1}{2} + \frac{1}{n+2} - \frac{1}{(n+1)(n+2)}\right).$$

Hence the difference between worst-case and average case is approximately

$$0.838564071338257129893046559 48\ldots$$

update operations.

We have applied the algorithm [23] to the reducible flow graphs in the SPEC2K benchmark suite. The results are displayed in Figure 7, where the line above denotes the worst-case. Assuming that the timing behavior is $c \cdot n^\alpha$, Figure 7 shows the value of α for all reducible flowgraphs in the SPEC2K. On the average we get $\alpha \approx 1.61053$.

Our analysis above is based on the assumption that all CFGs are equally probable. If this were true in practical settings, the SPEC2K benchmarks should have produced values being much closer to the worst-case than they actually are. Thus we have to conclude that practical settings do not conform to these statistics.

However, performing an analysis similar to that done for Sreedhar's algorithm, we get the following theorem.

Theorem 7. *Let U_n denote the number of update operations performed by the algorithm in* [23].

Then we have for goto-free programs

$$U_n = c \cdot n^{1.5} + O(n), \ (n \to \infty)$$

where constant c depends on the probability distribution of the statements only.

In this case $\alpha \approx 1.61053$ from SPEC2K is much closer to the predicted value 1.5.

6. Conclusions

We have surveyed average case analyzes of some elimination-based data-flow analysis algorithms.

It turned out, that analyzing such algorithms on a strict mathematical basis is possible. However, it is very important to choose statistics which conform to practical settings. In particular, we have pointed out that statistics assuming that all CFGs are equally probable do not conform to practical settings, while assuming that the programs are "goto-free" fits practical settings much better.

References

[1] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Comm. ACM*, 19(3):137–147, 1976.

[2] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Reading, MA, 3rd edition, 2000.

[3] Johann Blieberger. Data-flow frameworks for worst-case execution time analysis. *Real-Time Syst.*, 22(3):183–227, 2002.

[4] Johann Blieberger. Average case analysis of DJ graphs. *Journal of Discrete Algorithms*, 4:649–675, 2006.

[5] Johann Blieberger. On the average path length of Annotated Decomposition Trees. (manuscript in preparation), 2008.

[6] Johann Blieberger and Peter Kirschenhofer. On the number of reducible flowgraphs. (manuscript in preparation), 2008.

[7] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant computations. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, 1998.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[9] Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS)*, Orlando, Florida, USA, Dec. 2000.

[10] Thomas Fahringer and Bernhard Scholz. A Unified Symbolic Evaluation Framework for Parallelizing Compilers. *IEEE Transactions on Parallel and Distributed Systems*, 11(11), November 2000.

[11] Susan L. Graham and Mark Wegman. Fast and usually linear algorithm for global flow analysis. *J. ACM*, 23(1):172–202, 1976.

[12] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, New York, 1 edition, 1977.

[13] Matthew S. Hecht and Jeffrey D. Ullman. A simple algorithm for global data flow analysis problems. *SIAM J. Comput.*, 4(4):519–532, 1977.

[14] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1):60–65, 2001.

[15] Rahul Joshi, Uday Khedker, Vinay Kakade, and Medha Trivedi. Some interesting results about applications of graphs in compilers. *CSI Journal*, 31(4), 2002.

[16] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206. ACM SIGACT and SIGPLAN, ACM Press, 1973.

[17] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., second edition, 1973.

[18] E. Mehofer and B. Scholz. A Novel Probabilistic Data Flow Framework. In *International Conference on Compiler Construction (CC 2001)*, Lecture Notes in Computer Science (LNCS), Vol. 2027, pages 37 – 51, Genova, Italy, April 2001. Springer.

[19] Marvin C. Paull. *Algorithm design: a recursion transformation framework*. Wiley-Interscience, New York, NY, USA, 1988.

[20] G. Ramalingam. Data flow frequency analysis. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 267–277, New York, NY, USA, 1996. ACM Press.

[21] Torsten Robschink and Gregor Snelting. Efficient path conditions in dependence graphs. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 478–488, New York, NY, USA, 2002. ACM Press.

[22] B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–315, September 1986.

[23] Bernhard Scholz and Johann Blieberger. A new Elimination-Based Data Flow Analysis Framework Using Annotated Decomposition Trees. In *CC'2007, LNCS 4420*, pages 202–217, Braga, Portugal, March 2007.

[24] Vugranam C. Sreedhar. *Efficient Program Analysis Using DJ Graphs*. PhD thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, 1995.

[25] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. A new framework for elimination-based data flow analysis using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 20(2):388–435, 1998.

[26] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. A new framework for elimination-based data flow analysis using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 20(2):388–435, 1998.

[27] R E Tarjan. A unified approach to path programs. *J. ACM.*, 28(3):577–593, 1981.

[28] O. Vernet and L. Markenzon. Maximal reducible flowgraphs. Technical Report RT029/DE9, Departamento de Engenharia de Sistemas, Instituto Militar de Engenharia, Rio de Janeiro, RJ, Brasil, 1998.

[29] O. Vernet and L. Markenzon. Solving problems for maximal reducible flowgraphs. *Disc. Appl. Math.*, 136:341–348, 2004.

[30] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, Heidelberg, Germany, 2nd edition, 1983.