

A new combinatorial approach to parametric path analysis

Ernst Althaus, Sebastian Altmeyer, Rouven Naujoks

Angaben zur Veröffentlichung / Publication details:

Althaus, Ernst, Sebastian Altmeyer, and Rouven Naujoks. 2010. "A new combinatorial approach to parametric path analysis." AVACS - Automatic Verification and Analysis of Complex Systems.

Nutzungsbedingungen / Terms of use:

licgercopyright

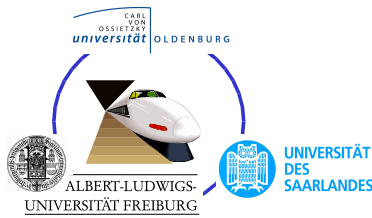
Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under the following conditions:

Deutsches Urheberrecht

Weitere Informationen finden Sie unter: / For more information see:

<https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publizieren>





AVACS – Automatic Verification and Analysis of
Complex Systems

REPORTS
of SFB/TR 14 AVACS

Editors: Board of SFB/TR 14 AVACS

A New Combinatorial Approach to Parametric
Path Analysis

by
Ernst Althaus¹ Sebastian Altmeyer²
Rouven Naujoks³

Publisher: Sonderforschungsbereich/Transregio 14 AVACS
(Automatic Verification and Analysis of Complex Systems)
Editors: Bernd Becker, Werner Damm, Martin Fränzle, Ernst-Rüdiger Olderog,
Andreas Podelski, Reinhard Wilhelm
ATRs (AVACS Technical Reports) are freely downloadable from www.avacs.org

Copyright © June 2010 by the author(s)
Author(s) contact: Ernst Althaus (ernst.althaus@uni-mainz.de).

A New Combinatorial Approach to Parametric Path Analysis

Ernst Althaus¹, Sebastian Altmeyer², and Rouven Naujoks³

¹ Johannes-Gutenberg-Universität Mainz ernst.althaus@uni-mainz.de

² Saarland University altmeyer@cs.uni-saarland.de

³ Max-Planck-Institut für Informatik naujoks@mpi-inf.mpg.de

Abstract. Hard real-time systems require tasks to finish in time. To guarantee the timeliness of such a system, static timing analyses derive upper bounds on the *worst-case execution time* of tasks. There are two types of timing analyses: numeric and parametric ones. A numeric analysis derives a numeric timing bound and, to this end, assumes all information such as loop bounds to be given a priori. If these bounds are unknown during analysis time, a parametric analysis can compute a timing formula parametric in these variables. A performance bottleneck of timing analyses, numeric and especially parametric, can be the so-called path analysis, which determines the path in the analyzed task with the longest execution time bound. In this paper, we present a new approach to the path analysis. This approach exploits the rather regular structure of software for hard real-time and safety-critical systems. As we show in the evaluation of this paper, we strongly improve upon former techniques in terms of precision and runtime in the parametric case. Even in the numeric case, our approach matches up to state-of-the-art techniques and may be an alternative to commercial tools employed for path analysis.

1 Introduction

Hard real-time systems require tasks to finish in time. To guarantee the timeliness of such a system, static timing analyses derive upper bounds on the worst-case execution time (WCET) of a task. To be useful in practice, timing analyses must be

- *sound*, to ensure the reliability of the guarantees,
- *precise*, to increase the chance to prove the satisfiability of the timing requirements, and
- *efficient*, to make them useful in industrial practice.

The high complexity of modern processors and modern embedded software hampers analyses to achieve all three properties at once. Exhaustive measurement, for instance, may be sound and precise but is infeasible for realistically sized programs. Simple end-to-end measurements are easy to derive, but are possibly unsound. Static timing analyses derive sound upper bounds on the execution

time by construction. In general, the analyzed programs are represented as control flow graphs (CFGs) with basic code blocks as nodes and edges representing the possible execution paths. A set of static analyses compute—amongst other information—upper bounds on the execution time of the basic blocks and upper bounds on the number of loop iterations. The execution time bound of the task is then given by a path P in the CFG for which the sum of occurrences of basic blocks of P times their execution time bound is maximal. The step of the timing analysis that computes this path is usually referred to as path analysis. The task of determining P is usually referred to as path analysis.

Timing analysis handling only numeric values as bounds on the maximal number of loop iterations are referred to as numeric or traditional analyses. The drawback of these analyses is that information such as bounds on the maximal number of loop iterations must be known statically, i.e. during design time. Some systems need guarantees for timely reactions which are not absolute, but dependent on a numerical parameter. In such cases, traditional timing analyses offer only two possibilities. Either one provides bounds for the unknown variables or one starts a new analysis each time the task is used with different values. The first option endangers precision, the second may unacceptably increase the analysis time. Parametric timing analyses circumvents this problem. Instead of computing numeric bounds valid for specific variable assignments only, parametric analyses derive symbolic formulas representing upper bounds on the task's execution times.

The path analysis can become the bottleneck in the timing analysis. In the case of parametric timing analysis this is true for both the running time and for the precision of the computed execution time bound, as we will explain later. In the numeric timing analysis, the path analysis can be the bottleneck of the running time, depending on the ILP-solver (see Section 4) that is internally used and on the target architecture, which determines the complexity of prior steps of the timing analysis. State-of-the-art techniques formulate the problem as the search for a longest path in the control flow graph that respects bounds on the number of times a loop can be traversed for each time, the loop is entered.

1.1 Timing Analysis

Static timing analyses represent programs to be analyzed as *control-flow graphs* $G = (V, E)$. A sequential list of instructions with a unique entry and a unique exit point are so-called basic blocks which constitute the nodes of the CFG. The edges of the CFG resemble the possible control flow between the basic blocks. For the sake of simplicity, we can assume a single, unique entry and even a single, unique exit node of the CFG. The first step of the timing analysis, the *CFG reconstruction* generates the CFG from the executable (see Figure 1 for the toolchain). Note that timing analysis has to resort to the level of the executable. Source code analysis can only deliver rough and possibly unsound estimates.

To be able to derive an upper bound on the execution time of the analyzed tasks, the timing analysis has to compute upper bounds on the execution times for each basic block and upper bounds on the number of iterations of each

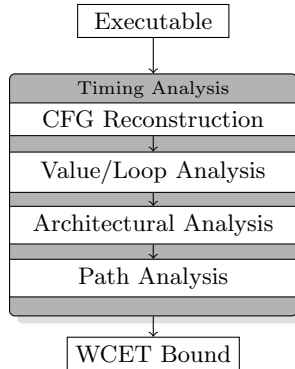


Fig. 1. Timing Analysis Toolchain

reachable loop within the CFG. The corresponding steps are the *value and loop-bound analysis* and the *architectural analysis*. In case of a numeric analysis, the loop-bounds are given as numeric values (in \mathbb{N}); in case of a parametric analysis, loop-bounds are represented by variables. Our new approach adapts the last step of a static timing analysis, the path analysis. Input to this step is only the CFG, the loop-bounds, and the execution time bounds for each basic block, which we assume to be given.

1.2 Numeric Path Analysis

Path analysis combines the timing information for each basic block and the loop-bounds and searches for the longest path within the executable's CFG. In this fashion, it computes an upper bound on a task's execution time. Searching the longest path is done using a technique called implicit path enumeration (IPET [1,2]): the control flow graph and the loop-bounds are transformed into *flow constraints*. The upper bounds for the execution times of the basic blocks as computed in the cache and pipeline analysis are used as weights. Figure 2 provides an example. The variables n_i , also called traversal counts, denote how often a specific edge is traversed. The first and the last basic block are left, resp. entered, exactly once ($n_1 = 1$; and $n_3 + n_6 = 1$;). For all other basic blocks, the sum of the traversal counts entering is equal to the sum of the leaving ones. The loop body (basic block 4, bounded by b_{loop}) is executed at most b_{loop} times as often as the loop is entered ($n_4 \leq b_{loop}n_2$;). The constant c_j denotes the cost of the basic block j . The maximum sum over the costs of a basic block times traversal counts entering it determines the final WCET bound.

The resulting ILP can be solved by any solver. In practice, CPLEX is often used as a commercial and lp_solve as a non-commercial solver.

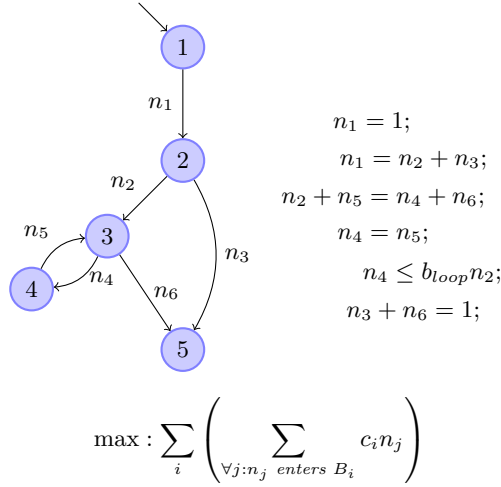


Fig. 2. Control flow graph and the corresponding flow constraints

1.3 Parametric Path Analysis

Parametric path analysis computes the WCET formula by symbolically searching the longest execution path in the program. As in numeric timing analysis, implicit path enumeration is used to generate flow constraints. These flow constraints, however, must be linear in order to be used in an ILP. The only difference with respect to numeric analysis is the type of the loop-bounds. Regarding the example in Figure 2, numeric analysis considers b_{loop} to be a numeric value and thus, computes a numeric WCET. The parametric analysis has to compute a WCET formula in the parameter b_{loop} . Although this seems to be a rather small difference, it has severe consequences. The expression

$$n_4 \leq b_{loop}n_2;$$

is a linear constraint, if b_{loop} is a numeric value. However, the same constraint is non-linear in case of a parametric loop-bound. Such non-linear constraints are caused by relative loop-bounds. As a solution, all relative loop-bounds have to be converted to absolute ones. In the example in Figure 2, the variable n_2 is bounded by 1. Thus, the relative loop-bound can be replaced by

$$n_4 \leq 1 * b_{loop};$$

Such a replacement is possible in general. To replace loop-bounds, each variable must be bounded. If we assume a variable to be unbounded, the whole ILP would be unbounded too. Note that there are only non-negative execution time bounds for basic blocks. Hence, we can disregard such cases and can assume each variable to be bounded. In case of nested parametric loop-bounds, however, a variable

may be bounded by another parametric loop-bound only. Assume $n_o \leq b_o$ to be the absolute loop-bound of an outer loop, and $n_i \leq b_i n_o$ the relative of the inner loop. After conversion, the absolute loop-bound of the inner loop is $n_i \leq b_i b_o$, which contains a non-linear term. The solution to this problem is to replace $b_i b_o$ by a new symbolic loop b' and to use $n_i \leq b'$ as the loop-bound constraint for the inner loop. Note that this step increases the number of parameters, and hence, the complexity of the symbolic ILP. Note that absolute loop-bounds lead to a lower precision than relative ones. Relative bounds still respect the relation to the loop entry edges. In case a loop is not part of the longest execution path through the CFG, a relative loop-bound will not increase the execution time bound. In contrast, absolute loop-bounds will contribute to the upper bound no matter if the corresponding loop is part of the worst-case path or not.

The constraint system after the conversion is linear and thus forms a valid symbolic integer linear problem. This ILP is then solved by a symbolic ILP-solver as proposed by Feautrier [3] using symbolic versions of the simplex [4] and cutting plane algorithm [5]. A free symbolic solver called *PIP* is also available.

1.4 Other approaches to Parametric Timing Analysis

Lisper proposed a parametric timing analysis [6] which has recently been implemented and extended by Bygde [7]. In contrast to our approach, they use a polyhedral abstract interpretation to compute the loop-bounds and to create an ILP for the path analysis in one step. Hence, the constraints used within their approach differ from the standard IPET model. In addition, their analysis—or at least the implementation of it—resorts to the level of the source code. Thus, a direct comparison between both approaches is not possible. As in [8], the parametric ILP was the bottleneck and has been replaced by a method called minimum propagation analysis [7] based on a tree-like representation of the parametric formula. The better performance of the new approach comes at the cost of a lower precision which is often less than one percent but may reach up to 30% in some benchmarks.

Other approaches to parametric timing analysis [9,10,11,12] have a completely different structure and resort completely to the source-code level. Hence, the new path analysis technique, which we explain in the next section, can not be applied to these approaches.

2 Longest Paths in Singleton-Loop-Graphs

In this section we will introduce a class of graphs, so-called singleton-loop graphs, to model CFGs of computer programs. Informally, a singleton-loop graph is a graph where every strongly connected component has a unique node with an incoming edge from outside the component, called the entry node, and where this property holds recursively within the component when the entry node is deleted. Exploiting the special structure of such graphs, we will show, how to compute longest execution paths with high efficiency. The proposed algorithm

will be able, not only to cope with numeric loop-bounds, but also to handle symbolic loop-bounds exactly. We will show, that our method can solve the problem in time polynomial in the input size and in the size of the output. After justifying the restriction to singleton-loop graphs, we formally define them. Then we will give an algorithm for solving the longest path problem in these graphs for which we will subsequently analyze its running time behavior – first in the numeric and then in the symbolic case – and show its correctness. Then we will show several properties of the produced output. Finally, we will show for a certain subclass of singleton-loop graphs, that our algorithm produces a solution set of minimal size in case of symbolic loop bounds.

2.1 Structure of a Control Flow Graph

Recall that we are interested in computing upper bounds on the WCETs of programs developed for embedded systems. These programs typically have a simple structure. Here we discuss the following constructs which can cause splits in the control flow graphs: for, repeat–until, and while–do loops, if–then–else constructs, function–calls and goto directives.

While for, repeat–until, while–do, if–then–else and similar constructs will be captured well by our approach as the resulting graphs will be singleton-loop graph as defined below, Function–calls can cause the following difficulty. If the program jumps from different points in the program to a function, you have to recall where you have jumped from in order to proceed to the correct position. There are two ways to tackle the problem in our approach. Either we analyze the running time of each function (in an inverted topological order) and replace the function call with the single edge weighted with the (parameterized) worst-case running time of that function (which will make the analysis less tight, as the context of the procedure call is unknown) or we assume that all procedures are inlined (which will make the CFG larger). The second way typically yields no problems, as, despite the increase of the CFG, we can solve the corresponding longest path problem very efficiently for programs of the size usually found in embedded systems. For both solutions we need that the call graph is acyclic.

For cyclic call graphs, our approach is not applicable. The same holds when goto’s are used. Notice that it is not even clear what a loop is in these cases. One possibility to define the corresponding longest path problem (the one that is used up to now) is to use the ILP-formulation as the definition of a valid solution. Note that for general graphs, solving the ILP is NP-hard as setting a loop-bound of one for each node leads to the longest simple path problem in general graphs.

2.2 Preliminaries

First, we will define, what kind of structure in the CFG is induced by a program loop.

Definition 1 (loop). *Given a directed graph $G = (V, E)$, we call a strongly connected component $S = (V_S, E_S)$ in G with $|E_S| > 0$, a loop of G .*

We denote by $\text{loops}(G)$, the set of all loops of G . Note that we don't want to call an isolated node a loop, thus, we demand E_S to be nonempty.

Definition 2 (entry node). *Given a directed graph $G = (V, E)$ and a loop $L = (V_L, E_L)$ of G , we call $\mathcal{E} \in V_L$ such that there exists an edge (u, e) in the cut-set*

$$\delta_G^+(V \setminus V_L) := \{(v', v) \in E \mid v' \in V \setminus V_L, v \in V_L\}$$

an entry node of G .

Definition 3 (singleton-loop). *A loop L in a graph G is called a singleton-loop if L has exactly one entry node e .*

For the unique entry node of a singleton loop L , we write $\mathcal{E}(L)$.

Definition 4 (sub-loops). *Given a loop $L = (V_L, E_L)$, we define*

$$\text{sloops}(L) := \bigcup_{v \text{ is entry node of } V_L} \text{loops}(G_v)$$

where G_v is the subgraph induced by $V \setminus \{v\}$.

Definition 5 (induced sub-loops). *Given a loop $L = (V, E)$, we call the recursively defined set*

$$\text{iloops}(L) := \{L\} \cup \left(\bigcup_{L_s \in \text{sloops}(L)} \text{iloops}(L_s) \right)$$

the set of induced sub-loops of L .

For a graph G , we extend the definition of iloops to graphs:

$$\text{iloops}(G) := \bigcup_{L_s \in \text{loops}(G)} \text{iloops}(L_s)$$

We call a graph G a *singleton-loop graph* if each induced sub-loop of G is a singleton-loop. For such a graph, we write $\mathcal{E}(G) := \{\mathcal{E}(L) \mid L \in \text{iloops}(G)\}$ to denote the set of entry nodes of all induced sub-loops in G .

Definition 6 (portal nodes, transit edges). *Given a directed graph $G = (V, E)$ and a loop $L = (V_L, E_L)$ in G , we call*

$$\mathcal{T}(L) := \delta_G^+(V_L)$$

the set of transit edges of L , i.e. the edges, leaving the loop L , and

$$\mathcal{P}(L) := \{p \in V_L \mid \exists (p, v) \in \mathcal{T}(L)\}$$

the set of portal nodes of L , i.e. the last nodes over which a path can leave the loop L .

Note that there is a one-to-one correspondence between singleton-loops and their entry nodes, which justifies the following definition.

Definition 7 (loop-bound function). *Given a singleton-loop graph $G = (V, E)$, we call a function*

$$b: \mathcal{E}(G) \rightarrow \mathbb{N} \cup \{+\infty\}$$

a loop-bound function for G .

Now we have to classify the valid paths, i.e. the paths that respect the loop-bound conditions. If for a loop L , a loop-bound of $b(\mathcal{E}(L))$ is given, we mean that an execution path is not allowed to enter L and iterate on L more than $b(\mathcal{E}(L))$ times, before the path leaves L again, or more formally:

Definition 8 (valid path). *Given a singleton-loop graph $G = (V, E)$, two nodes $s, t \in V$ and a loop-bound function b for G , we call a path $P := s \rightsquigarrow t$ a valid path if for all $L := (V_L, E_L) \in \text{iloops}(G)$ and for all sub-paths $(\mathcal{E}(L), v_0, v_1, \dots, v_k)$ of P with $v_i \in V_L$, the sub-path $(\mathcal{E}(L), v_0, v_1, \dots, v_{k-1})$ contains at most $b(\mathcal{E}(L))$ times the node $\mathcal{E}(L)$.*

The problem that we consider in the following, is to determine the longest valid paths from a single source node s to all other nodes in G with respect to a given edge weight function. At first glance, this might seem more complicated than finding only the longest path from a single source node to a single destination node, but as in the computation of shortest paths in graphs, these problems are most likely equally hard.

In the following, we will write $\text{lps}(G, s, t)$ for a longest valid path from a node s to a node t and $\overline{\text{lps}(G, s, t)}$ to denote the longest valid path from s to t , that contains t exactly once. Most of the times, we will limit the discussion to the task of computing just the path weights for sake of simplicity. Note, that this is not a real limitation, as the algorithm can easily be extended to also cope with the problem of reporting the paths as well.

In the following we will assume that for each $v \in V$ there is a path from s to t containing v . All other nodes can be removed by a preprocessing step in time $O(|V| + |E|)$. Note that the resulting graph has at least $|V| - 1$ edges.

2.3 The Algorithm

Let us recall that a problem instance is given by a a singleton-loop graph $G = (V, E)$, a source node $s \in V$, an edge weight function $w: E \mapsto \mathbb{N}$ and a loop-bound function $b: \mathcal{E}(G) \rightarrow \mathbb{N} \cup \{+\infty\}$. Since from now on, we will only talk about singleton-loop graphs, we will only write loops instead of singleton-loops.

$LPS(G, s) :=$

1. identify the loops $(L_j)_{j \in \{1, \dots, l\}}$ of G by computing the strongly connected components.

2. for each $L_j = (V_{L_j}, E_{V_j})$:
 - (a) modify L_j by replacing $\mathcal{E}(L)$ by two nodes \mathcal{E}_{out} and \mathcal{E}_{in} and by replacing all incoming edges $(v, \mathcal{E}(L))$ by edges $(v, \mathcal{E}_{\text{in}})$ and all outgoing edges $(\mathcal{E}(L), v)$ by edges $(\mathcal{E}_{\text{out}}, v)$
 - (b) call $LPS(L_j, \mathcal{E}_{\text{out}})$
 - (c) now we know the $\text{lps}(L_j, \mathcal{E}_{\text{out}}, v)$ for all $v \in V_{L_j}$, and thus we set

$$\begin{aligned} \text{lps}(G, \mathcal{E}(L_j), v) := \\ (\text{b}(\mathcal{E}(L_j)) - 1) \cdot \text{lps}(L_j, \mathcal{E}_{\text{out}}, \mathcal{E}_{\text{in}}) + \text{lps}(L_j, \mathcal{E}_{\text{out}}, v), \end{aligned}$$

that is the longest path weight from $\mathcal{E}(L_j)$ to v is to loop $\text{b}(\mathcal{E}(L_j)) - 1$ times through L and then to head for v .

- (d) replace L_j in G by a single node r_j and add an edge (r_j, x) for each $(p, x) \in \mathcal{T}(L_j)$ with appropriate weights, namely:

$$w(r_j, x) := \text{lps}(G, \mathcal{E}(L_j), p) + w(p, x)$$

Add an edge (v, r_j) for each $(v, \mathcal{E}(L_j)) \in E$ and set $w(v, r_j) := w(v, \mathcal{E}(L_j))$.

3. the altered graph is a DAG, thus we can easily determine the longest paths.
4. compute the longest path weights to nodes within the loops: Replace the nodes r_j again by the corresponding loops and set for each $L_j = (V_{L_j}, E_{V_j})$ and for all $v \in V_{L_j}$:

$$\text{lps}(G, s, v) := \overline{\text{lps}}(G, s, \mathcal{E}(L_j)) + \text{lps}(L_j, \mathcal{E}(L_j), v)$$

So far, we haven't discussed, how the $\overline{\text{lps}}(G, s, \mathcal{E}(L_j))$ in step 4 are computed. Note, that the entry node $\mathcal{E}(L_j)$ corresponds to a contraction node c in the condensed graph G' . When we compute the longest path weight from s to c , we set

$$\overline{\text{lps}}(G', s, c) := \max_{v \in \text{inc}(c)} \text{lps}(G', s, v) + w(v, c)$$

Running Time - Numeric Bounds Let us first analyze the algorithm's running time $T(|V|, |E|)$ for the case in which all loop-bounds are numeric values. In step 1), the strongly connected components of G are computed, which can be done in $O(|V| + |E|)$ time by depth-first search. Step 2a) can be computed in $O(\text{deg}(\mathcal{E}(L_j)))$ time. In step 2b), the algorithm is called recursively which takes $T(|V_{L_j}| + 1, |E_{L_j}|)$ time. The weight updates in 2c) can be performed in $O(|V_{L_j}|)$ and the updates in 2d) in $O(|\mathcal{T}(L_j)|)$ time. It is folklore, that the computation of longest path weights in a DAG, as done in step 3), takes no more than $O(|V| + |E|)$ time. Finally, step 4 can be done in $O(|V|)$. Hence without the recursive calls, we have a linear running time $O(|V| + |E|)$.

Note that the recursion depth of our algorithm is bounded by $|V|$, as a node is split at most once. Furthermore the edge set of the sub-loops are disjoint. Although nodes are split, we can argue that the total number of nodes in a certain recursion depth is bounded by $2|V|$ as follows: Let $V^{\text{out}} = \{v \in V \mid$

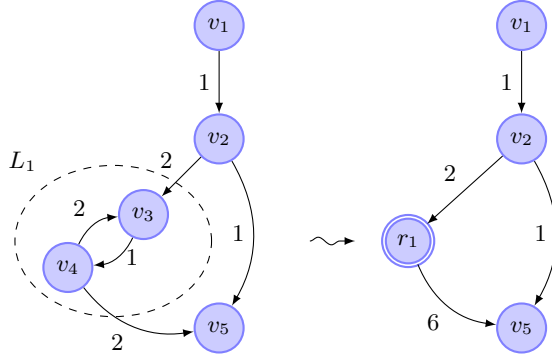


Fig. 3. We assume that $b(v_3) = 2$ and that v_1 is the source node s . The algorithm proceeds as follows: (a) the strongly connected component L_1 is identified (b) the longest path weight of 4 from v_3 to v_4 with respect to b is computed recursively (c) L_1 is replaced by r_1 (d) the longest path weights are computed in the resulting DAG.

v has at least 1 outgoing edge} and $V^{in} = \{v \in V \mid v \text{ has at least 1 incoming edge}\}$. Then $\sum_{L \in \text{sloops}(G)} |V_L^{out}| \leq |V^{out}|$ and $\sum_{L \in \text{sloops}(G)} |V_L^{in}| \leq |V^{in}|$, where V_L is the set of nodes of L after splitting the entry node. Thus, in total we have

$$\begin{aligned}
T(|V|, |E|) &= \sum_{L := (V_L, E_L) \in \text{sloops}(G)} [& (2) \\
& T(|V_L| + 1, |E_L|) + & (2b) \\
& O(\deg(\mathcal{E}(L))) + & (2a) \\
& O(|V_L|) + & (2c) \\
& O(|\mathcal{T}(L)|)] + & (2d) \\
& O(|V| + |E|) + & (1), (3) \\
& O(|V|) & (4) \\
& = O(|V| \cdot (|V^{out}| + |V^{in}| + |E|)) \\
& = O(|V| \cdot |E|)
\end{aligned}$$

Running Time - Symbolic Bounds So far, we have treated loop-bounds as numeric values. In the presence of symbolic loop-bounds we have to change our algorithm slightly. Instead of a unique longest path we now have to consider for each target node a set of longest paths to that node (see Figure 4 for an example). When concatenating two paths we now have to concatenate all pairs of paths. Since the operations on the path weights include multiplications and additions, they can be represented as polynomials over the symbolic loop-bounds. Clearly, we would aim at getting all possible path weights that are maximal for at least one choice of the symbolic loop-bound parameters. On the down side, testing whether a path weight is maximum for some choice (or instantiation) of the parameters seems to be non trivial. In our experiments it has turned out, that keeping all paths with weights that are not dominated by another weight (i.e., all coefficients in the weight polynomial are at least as big as the coefficient in the other weight polynomial) keeps the solution set sparse in practice and can be

implemented very efficiently. For a problem instance $I = (G, s, t)$, consisting of a graph, a source node s and a destination node t , we denote by $\mathcal{D}(I)$ (or short $\mathcal{D}(s, t)$, if G can be deduced from the context) the set of longest path weights from s to t computed by our algorithm. The property of $\mathcal{D}(I)$, that its elements are pairwise non-dominating can be achieved by eliminating dominated elements after the execution of step 2c. We write $\text{slbs}(I)$ for the number of symbolic loop-bounds of a problem instance $I = (G, s, t)$ and we write $\text{lbs}(I) := \text{lbs}(G) := |\text{loops}(G)|$ for the number of induced loops of G .

Let us analyze the running time of the algorithm in presence of symbolic loop-bounds.

Theorem 1. *The algorithm's running time is polynomial in the input size and in the size of the output.*

Proof. First, note that the running time only changes for the parts of the algorithm in which calculations on path weights are performed, namely the parts 2c), 2d) and 4). We will restrict this proof to the operations involved in step 2c), since the number of operations involved in 2c) is certainly not smaller than the ones in 2d) and 4).

Let us first count the number of operations on weight polynomials. Consider a longest path P from the source node s to the destination node t . Let $\text{lps}(u, v)$ denote the longest path weights, computed by the algorithm for the longest paths from node u to node v , then for each loop $L = (V_L, E_L) \in \text{loops}(G)$ that is traversed by P , we have $|\text{lps}(\mathcal{E}(L), p_L)| \leq |\text{lps}(s, p_L)|$ for $p_L \in \mathcal{P}(L)$ over which P leaves L again. Furthermore, for each $p_L \in \text{portals}(L)$ we have $O(|\text{lps}(\mathcal{E}(L), \mathcal{E}(L))| \cdot |\text{lps}(\mathcal{E}(L), p_L)|)$ operations, since the addition involves the addition of all pairs of weights in $\text{lps}(\mathcal{E}(L), \mathcal{E}(L))$ and in $\text{lps}(\mathcal{E}(L), p_L)$. Since L is strongly connected, $|\text{lps}(\mathcal{E}(L), \mathcal{E}(L))| \leq |\text{lps}(\mathcal{E}(L), p_L)|$. Thus the number of operations is bounded by $|\text{lps}(\mathcal{E}(L), p_L)|^2 \leq |\text{lps}(s, p_L)|^2$. Since each node in V_L can be a portal node of L , the total number of operations on polynomials occurring on the first recursion level is bounded by $\sum_{v \in V} |\text{lps}(s, v)|^2 \leq (\sum_{v \in V} |\text{lps}(s, v)|)^2$. But, since $\sum_{v \in V} |\text{lps}(s, v)|$ is just the number of path weights, reported by the algorithm, the number of operations on polynomials is polynomial in the number of reported path weights. Note that each weight has a unique representation and that all operations on the weight polynomials can be carried out in time polynomial in the size of these polynomials.

What is left to show is, that the weight polynomials computed for the nodes in the input graph have a size polynomially bounded by the size of the weight polynomials that are reported by our algorithm, that is the weight polynomials of the longest paths from the source node s to the sink node t . We will use a structural induction over the input graph G to prove so. If G contains no loops, the claim is true since G must be a DAG and therefore, all computed longest path weights are just constants. So, let us assume that G contains loops. By induction hypothesis, the claim holds now for each problem instance $(L, \mathcal{E}_{\text{out}}, p)$ where L is a loop of G , where the entry node of L is split into the nodes \mathcal{E}_{out} and \mathcal{E}_{in} and where p is an arbitrary portal node of L . But then the claim is also true

for $(L, \mathcal{E}(L), p)$ what can be seen as follows: Recall that a longest path weight from $\mathcal{E}(L)$ to p is given by the equation

$$\text{lpw}(G, \mathcal{E}(L), p) = (b(\mathcal{E}(L)) - 1) \cdot \text{lpw}(L, \mathcal{E}_{\text{out}}, \mathcal{E}_{\text{in}}) + \text{lpw}(L, \mathcal{E}_{\text{out}}, p)$$

for some path weights $\text{lpw}(L, \mathcal{E}_{\text{out}}, \mathcal{E}_{\text{in}})$ and $\text{lpw}(L, \mathcal{E}_{\text{out}}, p)$. But then, $\text{lpw}(G, \mathcal{E}(L), p)$ is as least as large as the maximum of the sizes of $\text{lpw}(L, \mathcal{E}_{\text{out}}, \mathcal{E}_{\text{in}})$ and of $\text{lpw}(L, \mathcal{E}_{\text{out}}, p)$ as each term in $\text{lpw}(L, \mathcal{E}_{\text{out}}, \mathcal{E}_{\text{in}})$ appears with a multiple of $b(\mathcal{E}(L))$, $\text{lpw}(L, \mathcal{E}_{\text{out}}, p)$ does not contain $b(\mathcal{E}(L))$ and each term in $\text{lpw}(L, \mathcal{E}_{\text{out}}, p)$ can eliminate only terms that are not multiplied with $b(\mathcal{E}(L))$. The last thing we have to show now is, that the claim holds for (G', s, t) , where again G' denotes the condensed graph. We compute the longest path weights in the directed acyclic graph G' by the recurrence

$$\text{lps}(G, s, u) = \max_{v \in \text{inc}(u)} (\text{lps}(G, s, v) + w(v, u))$$

starting with $u := t$. Consider now a weight polynomial $P = \text{lpw}(G, s, v) + w(v, u)$. Since we consider the condensed graph G' , $w(v, u)$ is a polynomial containing only variables associated with the loop that in turn is associated with the node v (in the case that v is not a condensed node, $w(v, u)$ is just a constant). Thus, except for the constant terms, P contains at least as many terms as there are in $\text{lps}(G, s, v)$ or in $w(v, u)$, which completes the proof. \square

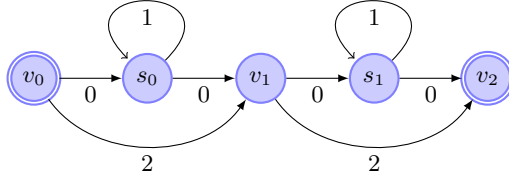


Fig. 4. The different weights for the longest paths from v_0 to v_2 are 4 , $2 + b(s_0)$, $2 + b(s_1)$ and $b(s_0) + b(s_1)$

Correctness Now, we will show, that our algorithm indeed computes the weight of a longest valid path $\text{lps}(G, s, t)$ from a source vertex s to a destination vertex t . In the following, when talking about paths we always mean valid paths. Again we will assume that G is a singleton-loop graph with weight function $w: E \mapsto \mathbb{N}$ and that we are given a loop-bound function $b: \mathcal{E}(G) \rightarrow \mathbb{N} \cup \{+\infty\}$. We will show the claim by induction over the recursion-level of the algorithm. If we assume that G contains no loops, G must be a directed acyclic graph and thus, our algorithm is correct. So, now assume that G contains loops. The induction hypothesis tells us now that for all recursive calls of our algorithm, we obtain correct results. Let

$p := s \rightsquigarrow t$ be a longest path in G . Let us assume w.l.o.g. that p shares at least one node with a sub-loop of G , i.e. for some $L := (V_L, E_L) \in \text{sloops}(G) : \mathcal{E}(L) \in V_L$. Thus p can be written as $p = s \rightsquigarrow p' \rightsquigarrow t$ with $p' = (\mathcal{E}(L) =: v_0, v_1, \dots, v_k)$ such that $v_i \in V_L$ and k is maximal. Since any sub-path of a longest path must be again a longest path between its starting- and end-node (with respect to the validity), we have that $w(p') = \text{lps}(G, \mathcal{E}(L), v_k)$. Consider now the condensed graph \bar{G} obtained by replacing loop L by a node r as described in the algorithm. Then the path $s \rightsquigarrow r \rightarrow v_k \rightsquigarrow t$ is valid and has weight $w(p)$. Therefore, $w(\text{lps}(G, s, t)) \leq w(\text{lps}(\bar{G}, s, t))$. On the other hand, $w(\text{lps}(G, s, t))$ cannot be strictly less than $w(\text{lps}(\bar{G}, s, t))$, because otherwise there would be a path in \bar{G} with weight strictly greater than a longest path in G , which also would not traverse L , since the weights of these paths are unequal. But this would mean, that there is also a path in G – just bypassing L – with the weight $w(\text{lps}(\bar{G}, s, t))$, which leads to a contradiction.

What is left to show is, that our algorithm computes correct values for $\text{lps}(G, \mathcal{E}(L), v_k)$. Let $p = (\mathcal{E}(L) =: v_0, v_1, \dots, v_k)$ with $v_i \in V_L$ be a longest path. We can assume that p contains exactly $b(\mathcal{E}(L))$ (respectively $b(\mathcal{E}(L)) + 1$ if $v_k = \mathcal{E}(L)$) times the node $\mathcal{E}(L)$, otherwise we could extend the path by the path $v_k \rightsquigarrow \mathcal{E}(L) \rightsquigarrow v_k$ without violating validity. Now each sub-path p' of p with $p' = \mathcal{E}(L) \rightsquigarrow \mathcal{E}(L)$ must have the same weight, since otherwise, by replacing the lower weight sub-path by the corresponding higher weight sub-path, we could obtain a path with higher weights. Thus, we can assume that there exists a longest path $\mathcal{E}(L) \rightsquigarrow^{p'} \mathcal{E}(L) \rightsquigarrow^{p''} \dots \rightsquigarrow^{p'} \mathcal{E}(L) \rightsquigarrow^{p''} v_k$ with weight $(b(\mathcal{E}(L)) - 1) \cdot w(p') + w(p'')$. Since p' and p'' must be longest paths, we are left to show that our algorithm computes the weights $\text{lps}(G, \mathcal{E}(L), \mathcal{E}(L))$ and $\text{lps}(G, \mathcal{E}(L), v_k)$ correctly. But this follows directly by the way we alter the loop L , i.e. by splitting the entry node of L into the two nodes \mathcal{E}_{out} and \mathcal{E}_{in} . Since L was a loop, every node in L is reachable from \mathcal{E}_{out} . by induction hypothesis the algorithm now computes recursively the right values, where obviously $w(\text{lps}(\bar{L}, \mathcal{E}_{\text{out}}, \mathcal{E}_{\text{in}})) = w(\text{lps}(G, e, e))$. Which finishes this proof.

The Output Size In the previous section, we have shown, that in the presence of symbolic loop-bounds, the running time of our algorithm is polynomial in the input size and in the size of the output. In this section, we will give some insights about how big the output can actually get. In the following lemma, we will derive an upper bound on the output size $\mathcal{D}(I)$.

Lemma 1. *For any problem instance $I = (G, s, t)$ $|\mathcal{D}(I)| \leq 2^{2^{\text{lps}(G)}}$.*

Proof. Let P be a path reported by our algorithm. We recursively define the loop-pattern $\mathcal{L}(P, G)$ of a path P in G as follows. If G is a DAG, the loop-pattern of any path is empty. To define a loop-pattern in the general case, we shortly recall some facts of the algorithm. We compute P as a longest path in the condensed graph (for which we have contracted each loop of G into a single node), which is known to be an acyclic graph, since the loops are the strongly

connected components of G . Let L_1, \dots, L_k be the subloops of G that are entered by P . Each loop L_i is entered exactly once (as the condensed graph is a DAG). Within the loop, P traverses a unique subpath P'_i for $b_{L_i} - 1$ times and then a path P''_i to a portal node. The loop pattern of P is then defined as the sequence

$$\mathcal{L}(P) := (L_1(\mathcal{L}(P'_1, L'_1), \mathcal{L}(P''_1, L'_1)), L_2(\mathcal{L}(P'_2, L'_2), \mathcal{L}(P''_2, L'_2)), \dots, L_k(\mathcal{L}(P'_k, L'_k), \mathcal{L}(P''_k, L'_k))),$$

where L'_i is the graph obtained by splitting $\mathcal{E}(L_i)$.

We prove the following claims:

1. Any two paths with the same loop-pattern (not necessarily with same source and target node) computed by the algorithm have the same cost up to a constant term.
2. Any two $s - t$ -paths reported by the algorithm have different loop-patterns.
3. Let $T(\text{lbs}(G))$ be the maximum possible number of loop patterns of a graph G , then T can be bounded by the recurrence $T(0) = 1, T(\text{lbs}(G)) \leq T(\text{lbs}(G) - 1)^2 + 1$.

We will prove the points in turn. We prove the first point by structural induction over G . If G is a DAG, all path weights are constants (as there are no symbolic loop-bounds). Hence, there is nothing to show. Now consider two paths P and Q in a graph G constructed by our algorithm with $\mathcal{L}(P, G) = \mathcal{L}(Q, G)$. Let L_1, \dots, L_k be the subloops of G contained in $\mathcal{L}(P, G)$. Decompose P into $(\bar{P}_0, (P'_1)^{b_{L_1}-1}, P''_1, \bar{P}_1, \dots, (P'_k)^{b_{L_k}-1}, P''_k, \bar{P}_k)$, where \bar{P}_i is the path from the portal of L_i used by P (respectively from the source of P if $i = 0$) to $\mathcal{E}(L_{i+1})$ (respectively to the target of P for $i = k$), P'_i is the path used for cycling within L_i and P''_i is the path from the entering node of L_i to the portal-node used by P . Analogously decompose Q . The cost of P can then be written as

$$w(\bar{P}_0) + \sum_{i=1}^k [(b_{L_i} - 1)w(P'_i) + w(P''_i) + w(\bar{P}_i)],$$

the cost of Q as

$$w(\bar{Q}_0) + \sum_{i=1}^k [(b_{L_i} - 1)w(Q'_i) + w(Q''_i) + w(\bar{Q}_i)].$$

The costs of \bar{P}_i and \bar{Q}_i are some constants. By induction hypothesis, the costs of P'_i and Q'_i only differ by a constant. The same holds for the cost of P''_i and Q''_i . Hence the difference of $w(P)$ and $w(Q)$ is a sum of constants and thus constant.

The second point immediately follows from the first, as our algorithm won't report two paths whose weights only differ in a constant.

For the third point, we argue as follows: Let L_1, \dots, L_k be the subloops of G given in topological order. Each loop pattern can be constructed by choosing for each subloop L_i either that it is not entered, or we use some loop pattern of L_i for

the cycling path and one loop pattern for the path to the portal. Hence, we get $T(\text{lbs}(G)) \leq \prod_i T(\text{lbs}(L'_i))^2 + 1$. Notice that $\sum_i \text{lbs}(L_i) = \text{lbs}(G)$ and $\text{lbs}(L'_i) = \text{lbs}(L_i) - 1$ and hence $\sum_i \text{lbs}(L'_i) = \text{lbs}(G) - k$. A simple calculation shows that $T(\text{lbs}(G))$ is maximized, if each induced subloop of G contains exactly one subloop. Hence we get $T(\text{lbs}(G)) \leq T(\text{lbs}(G) - 1)^2 + 1$.

Note that if L_i as a numeric loop-bound, the length of a path that does not enter L_i and the length of a path that enters L_i but no subloop of L_i differ only in a constant. Hence, in this case we can drop the addition of one in the recursive formula. Finally for $T(\text{lbs}(G)) = T(\text{lbs}(G) - 1)^2 + 1$, $T(0) = 1$ holds $T(\text{lbs}(G)) \leq 2^{2^{\text{lbs}(G)}}$ and for $T'(\text{lbs}(G)) = T'(\text{lbs}(G) - 1)^2$, $T'(1) = 2$ holds $T'(\text{lbs}(G)) \geq 2^{2^{\text{lbs}(G)-1}}$. \square

The next lemma shows, that the analysis of Lemma 1 is almost tight.

Lemma 2. *There exists a problem instance $I = (G, s, t)$, such that $|\mathcal{D}(I)| = 2^{2^{\text{slbs}(G)-1}}$ and $\text{slbs}(G) = 1$.*

Proof. We will first prove a slightly different claim, namely we proof the bound using symbolic loop-bounds for all loops. More precisely, we show that there exists a problem instance I such that

- a) $\mathcal{D}(I) = 2^{2^{\text{slbs}(G)-1}}$
- b) there is an element in $\mathcal{D}(I)$ containing a positive constant
- c) there is an element in $\mathcal{D}(I)$ containing the constant 0

where x is the number of symbolic loop-bounds.

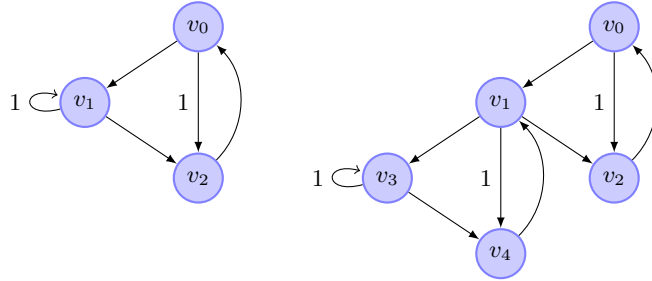


Fig. 5. Suppose the edge weights are 0 if not stated otherwise, then the longest path weights from v_0 to v_2 constructed by our algorithm for the left graph are $b(v_0)$, $b(v_0)b(v_1)$, $-1 + b(v_0) + b(v_1)$ and $1 - b(v_1) + b(v_0)b(v_1)$. The right graph was constructed from the left one by replacing the selfloop $\{v_1\}$ by the loop $\{v_0, v_1, v_2\}$. For the right graph, the algorithm computes 16 non-dominant longest path weights. In general, graphs obtained by repeatedly replacing the selfloop by the loop of the left graph, yield $2^{2^{\text{slbs}(G)-1}}$ non-dominant longest path weights and furthermore, one can show that there is a longest path weight that consists of $2^{\text{slbs}(G)} - 1$ terms where $\text{slbs}(G)$ is the number of symbolic loop-bounds.

We construct a graph G in the following way: we start with the left graph G_l in Figure 5 and repeatedly replace the selfloop of G by the loop of G_l . Note, that after k iterations, the resulting graph consists of $k + 2$ induced subloops. We will show the claim by induction over the recursive structure of G , i.e. over the number $\text{slbs}(G)$ of induced subloops of G .

For the base case ($\text{slbs}(G) = 2$), in which G corresponds to the graph G_l , it is easy to verify that

$$\mathcal{D}(v_0, v_2) = \{b(v_0), b(v_0)b(v_1), -1 + b(v_0) + b(v_1), 1 - b(v_1) + b(v_0)b(v_1)\}$$

and that one of these weight polynomials contains a positive constant.

Now let us consider the induction step. Recall that we want to compute $\mathcal{D}(v_0, v_2)$. In the first step of our algorithm, the node v_0 is split into two nodes v_{in} and v_{out} and recursively the sets $\mathcal{D}(v_1, v_1)$, $\mathcal{D}(v_{out}, v_{in})$ and $\mathcal{D}(v_{out}, v_2)$ are computed. Then, the set $\text{lpw}(v_0, v_2)$ is computed as the set of polynomials given by $(b(v_0) - 1) \cdot l_{ee} + l_{ep}$ for all $l_{ee} \in \mathcal{D}(v_{out}, v_{in})$ and for all $l_{ep} \in \mathcal{D}(v_{out}, v_2)$.

By the way, we have chosen the edge weights, we have that $\mathcal{D}(v_{out}, v_{in}) = \mathcal{D}(v_{out}, v_2) = \mathcal{D}(G, v_1, v_1)$. In particular, note that the constant polynomial 1 is not contained in $\mathcal{D}(v_{out}, v_{in})$, since by induction hypothesis, $\mathcal{D}(v_1, v_1)$ contains a polynomial with a positive constant (which must be greater or equal to 1), dominating 1.

This in turn, means that $|\text{lpw}(v_0, v_2)| = |\mathcal{D}(v_1, v_1)|^2$. We now have to show, that the elements in $\text{lpw}(v_0, v_2)$ are pairwise non-dominating. Let $l_1, l_2 \in \text{lpw}(v_0, v_2)$, given as $l_1 = (b(v_0) - 1) \cdot l_{ee} + l_{ep}$ and $l_2 = (b(v_0) - 1) \cdot l'_{ee} + l'_{ep}$ for weights $l_{ee}, l'_{ee} \in \mathcal{D}(v_{out}, v_{in})$ and $l_{ep}, l'_{ep} \in \mathcal{D}(v_{out}, v_2)$, l_1 . We distinguish two cases. In the first one, we assume that $l_{ee} \neq l'_{ee}$. Since by induction hypothesis, l_{ee} and l'_{ee} are pairwise non-dominant, the terms in l_1 and l_2 containing variable $b(v_0)$, namely $b(v_0) \cdot l_{ee}$ and $b(v_0) \cdot l'_{ee}$ - and thus also l_1 and l_2 - must be pairwise non-dominant. Now we assume that $l_{ee} = l'_{ee}$. In this case, the terms, not containing the variable $b(v_0)$, are $l_{ep} - l_{ee}$ and $l'_{ep} - l_{ee}$. Since by induction hypothesis l_{ee} and l'_{ee} are pairwise non-dominant, l_1 and l_2 must also be pairwise non-dominant.

Thus, $|\mathcal{D}(v_0, v_2)| = |\mathcal{D}(v_1, v_1)|^2$. Since by induction hypothesis $|\mathcal{D}(v_1, v_1)| = 2^{2^{x-2}}$, $|\mathcal{D}(v_0, v_2)| = 2^{2^{x-1}}$, which establishes the first part of the claim. For the second part, note that by induction hypothesis, there is a weight in $l_1 \in \mathcal{D}(v_1, v_1)$ with a positive constant and there is a weight in $l_2 \in \mathcal{D}(v_1, v_1)$ with zero constant. Thus the weight $l \in \mathcal{D}(v_0, v_2)$ with $l = (b(v_0) - 1) \cdot l_2 + l_1 = b(v_0) \cdot l_2 + l_1 - l_2$ has again a positive constant as a term. On the other hand, the weight $l \in \mathcal{D}(v_0, v_2)$ with $l = (b(v_0) - 1) \cdot l_2 + l_2 = b(v_0) - 1 \cdot l_2$ has a zero as constant term.

So far, we have assumed that all loop-bounds are symbolic. Now we will examine an instance in which all but one loop-bounds are numeric. For this, we use the same graph G as constructed above, but we assume that only the bound for the selfloop is symbolic. Then we can show by the very same induction as above, that by choosing the right numeric values for the loop-bounds, $|\mathcal{D}(v_0, v_2)| = 2^{2^{\text{iloops}(G)-1}}$. For the base case, we choose $b(v_0) := 3$ and obtain $\mathcal{D}(v_0, v_2) = \{3, 2 + b(v_1), 1 + 2b(v_1), 3b(v_1)\}$ which clearly satisfies the induction properties.

The only difference occurs now in the induction step, when arguing, that no two path weights l_1, l_2 exist, such that the first one dominates the other. Since there is just one symbolic loop-bound, all path weight polynomials consist of only two terms, i.e. $\mathcal{D}(v_1, v_1)$ consists of polynomials $c_i + c'_i \cdot a$ where a is the symbolic loop-bound in G . It is easy to see that these polynomials are non-dominated if and only if they can be ordered in such a way, that the constants c_i appear in increasing order while the c'_i appear in decreasing order.

The idea of proof is now to show, that the numeric bound $b(v_0)$ can recursively be chosen in such a way, that the constructed weights in $\mathcal{D}(v_0, v_2)$ can be ordered in the same way. To see this, suppose we are given two weights in $\mathcal{D}(v_0, v_2)$, namely $p_1 := (b(v_0) - 1) \cdot d_1 + d'_1$ and $p_2 := (b(v_0) - 1) \cdot d_2 + d'_2$ for $d_1, d_2, d'_1, d'_2 \in \mathcal{D}(v_0, v_2)$. Clearly by choosing $b(v_0)$ big enough, it is possible to put p_1 and p_2 into the same relative order as d_1 and d_2 . We will now show, that there is such a value for $b(v_0)$ for all pairs of weights in $\mathcal{D}(v_0, v_2)$ that is not too big.

Consider again the pair p_1, p_2 . Let us denote by $c(p)$ the constant term of such a weight p and by $v(p)$ the coefficient of the variable term. Without loss of generality, let us restrict our discussion to the case $c(d_1) \geq c(d_2)$. It is easy to verify that for

$$b(v_0) := \left\lceil \max \left\{ \frac{c(d'_2) - c(d'_1)}{c(d_1) - c(d_2)}, \frac{v(d'_1) - v(d'_2)}{v(d_2) - v(d_1)} \right\} \right\rceil + 1$$

we have $c(p_1) \geq c(p_2)$ and $v(p_1) \leq v(p_2)$.

Choosing $b(v_0)$ as the maximum of all possible choices of $p_1, p_2 \in \mathcal{D}(v_0, v_2)$ we can establish the same relative order as for the d_i , completing the proof. \square

So far, we have discussed, how many path weights are reported by our algorithm. Now, we will show, that there exist instances, for which at least exponential many path weights have to be reported from any correct algorithm.

Lemma 3. *There exists a problem instance I , such that any correct algorithm must report $2^{\text{slbs}(G)}$ longest paths.*

Proof. Consider the graph G_f in Figure 4. By repeated concatenation of the subgraph of G_f induced by the nodes $\{v_0, s_1, v_1\}$, we obtain a weighted graph $G = (V, E, w)$ that consists of nodes $V = \{v_0, \dots, v_{\text{slbs}(G)}, s_0, \dots, s_{\text{slbs}(G)-1}\}$, edges

$$E = \{(v_i, s_i), (s_i, v_i), (s_i, s_i), (v_i, v_{i+1}) \mid i \in \{0, \dots, \text{slbs}(G) - 1\} \cup \{(s_{\text{slbs}(G)-1}, v_{\text{slbs}(G)})\}\}$$

and of edge weights as given in the graph G_f .

Then there are exactly $2^{\text{slbs}(G)}$ different paths from v_0 to $v_{\text{slbs}(G)}$, namely one for each choice of bypassing a selfloop $\{s_i\}$ via the edge (v_i, v_{i+1}) or not. For these paths we have the set of corresponding weights

$$\left\{ \sum_{p_i \in p} p_i \cdot b(s_i) + 2 \cdot \sum_{p_i \in p} (1 - p_i) \mid p \in \{0, 1\}^{\text{slbs}(G)} \right\}$$

The claim now is, that for each weight in this set, there is an instantiation I of its symbolic loop-bounds, such that this weight dominates all other weights. To see this, consider a weight w . For each loop-bound variable s_i contained in w , set $I(b(s_i)) := 2\text{slbs}(G) + 1$ and for all other bounds to 0. Now consider any other weight w' and let n (n') be the number of variables in w (in w') and k be the number of variables that w and w' share. Then $w[I] = n(2\text{slbs}(G) + 1) + 2(\text{slbs}(G) - n)$ and $w'[I] = k(2\text{slbs}(G) + 1) + 2(\text{slbs}(G) - n')$. If now $k = n$, then there must be a variable in w' which is not in w , since $w \neq w'$. Thus, $n' > n$ and therefore $w[I] = n(2\text{slbs}(G) + 1) + 2(\text{slbs}(G) - n) > n(2\text{slbs}(G) + 1) + 2(\text{slbs}(G) - n') = w'[I]$. Since $k \leq n$, let us now assume that $k < n$. Then $w[I] - w'[I] = (n - k)(2\text{slbs}(G) + 1) + 2(n' - n)$ can only be negative if $n' < n$, but on the other hand $n' - n \geq -\text{slbs}(G)$, which implies that in this case $w[I] - w'[I] \geq (n - k)(2\text{slbs}(G) + 1) - 2\text{slbs}(G) = 2\text{slbs}(G)(n - k - 1) + n - k > 0$. Hence any algorithm has to report w .

Thus any correct algorithm must report all the $2^{\text{slbs}(G)}$ paths. \square

But not only the number of indistinguishable longest path weights is exponential in the worst case, even the weight polynomials can become large as we will show in the following lemma.

Lemma 4. *There exists a problem instance I , such that $\mathcal{D}(I)$ contains a weight with $2^{\text{slbs}(G)} - 1$ terms with non-zero coefficients.*

Proof. Consider the same graph construction as in Lemma 2. We again prove a slightly stricter claim, namely that there is

- a weight polynomial in $\mathcal{D}(G, v_0, v_2)$ such that all terms have non-zero coefficients except the term consisting of all loop-bound variables
- a weight polynomial in $\mathcal{D}(G, v_0, v_2)$ such that all terms have zero coefficients except the term consisting of all loop-bound variables

We again proof the claim by induction. As stated in the proof of this lemma, in the base case ($\text{slbs}(G) = 2$), the set of computed path weights is $\mathcal{D}(v_0, v_2) = \{b(v_0), b(v_0)b(v_1), -1 + b(v_0) + b(v_1), 1 - b(v_1) + b(v_0)b(v_1)\}$. Clearly the claim holds in this case. For the induction step, consider the weights $l, l' \in \mathcal{D}(v_1, v_1)$ such that l corresponds to the weight in the first part of the claim and l' corresponds to the weight in the second part of the claim. Recall from the discussion in the proof of Lemma 2, that $\mathcal{D}(v_0, v_2)$ consists of weights build by evaluating the expression $(b(v_0) - 1) \cdot l_1 + l_2$ for $l_1, l_2 \in \mathcal{D}(v_1, v_1)$. Since $l \in \mathcal{D}(v_1, v_1)$, $(b(v_0) - 1) \cdot l + l = b(v_0) \cdot l$ is in $\mathcal{D}(v_0, v_2)$ and thus the first part of the claim also holds for G . But also the weight $(b(v_0) - 1) \cdot l + l' = b(v_0) \cdot l + (l' - l)$ is in $\mathcal{D}(v_0, v_2)$. Since l and l' have distinct terms with non-zero coefficients and by induction hypothesis, l has $2^{\text{slbs}(G)-1} - 1$ terms with non-zero coefficients, the number of terms in $(b(v_0) - 1) \cdot l + l'$ with non-zero coefficients must be $2(2^{\text{slbs}(G)-1} - 1) + 1 = 2^{\text{slbs}(G)} - 1$, which finishes the proof. \square

In the next lemma, we will show that there can be a huge discrepancy between the output size and the size of paths, that have to be reported from any correct algorithm.

Lemma 5. *There is a problem instance $I = (G, s, t)$, such that the minimal output size is 2 but $|\mathcal{D}(I)| = 2^{2^{\text{lbs}(G)-1}}$.*

Proof. Reconsider again the example given in Figure 5 and the graph G as constructed in the proof of Lemma 2. We have argued in this proof, that the set $\text{lpw}(v_0, v_2)$ of path weights consists of all weights $(b(v_0) - 1) \cdot l_{ee} + l_{ep}$ for $l_{ee}, l_{ep} \in \mathcal{D}(v_1, v_1)$, since $\mathcal{D}(v_{out}, v_{in}) = \mathcal{D}(v_{out}, v_2) = \mathcal{D}(G, v_1, v_1)$. But one could do better, since it can be assumed wlog. that a longest path from v_0 to v_2 just corresponds to $b(v_0)$ times a path in $\mathcal{D}(v_1, v_1)$. Thus, instead creating a set $\text{lpw}(v_0, v_2)$ of cardinality $|\mathcal{D}(v_1, v_1)|^2$, we create one of cardinality $|\mathcal{D}(v_1, v_1)|$. But this leads by an inductive argument to a set of weights of the base case, namely 4 path weights. You could also apply this idea to the base case and get just the two path weights $b(v_0)$ and $b(v_0)b(v_1)$, but then the induction step in the proof of Lemma 2 would not work out in the same way, as there would be no path weight in $\mathcal{D}(v_1, v_1)$ with a positive constant (even though, it is possible to modify G slightly to make it work). So, we end up with four (respectively 2) longest path weights which still is in stark contrast to the set of $2^{2^{\text{lbs}(G)-1}}$ weights, constructed by our algorithm. \square

Lemma 5 states, that – at least in theory – our algorithm can produce outputs that are unnecessarily large. In general, one could try to eliminate all path weights that are convex combinations of other weights, which could be done in polynomial time by solving linear programs. Unfortunately solving LPs is too slow in practice for our purposes and furthermore, there are examples, in which even eliminating convex combinations does not reduce the output size significantly. Thus, we will discuss in the next section, a certain subclass of graphs, for which we are able to give an algorithm, that produces an output of minimal size.

2.4 While-Programs

The subclass of graphs that we consider in this section, are singleton-loop graphs with the additional property that each induced sub-loop has exactly one portal which coincides with the entry node of the loop. Note, that each computer program can be transformed into an equivalent program which loops are only while-loops. Since a while-loop has exactly one entry node and one portal node which coincides with the entry node, we can assume wlog. that our input graph exhibits this special property. The reason why we haven't considered this case before is of practical nature. Converting a program into a while-program changes its running time behavior. On the other hand, most of the CFGs that we have considered in our experiments, consisted mostly of such loops. Thus, the ideas presented in this section, can lead to a significant reduction in the output size in practice.

In this section, the input graph G , has always the additional property that $\forall L \in \text{iloops}(G): \mathcal{P}(L) = \{\mathcal{E}(L)\}$. In this setting, we can modify our algorithm as follows. Since the entry node of a loop always coincides with its only portal

node, we have in step 2c): $\text{lps}(G, \mathcal{E}(L_j), \mathcal{E}(L_j)) = b(\mathcal{E}(L_j)) \cdot \text{lps}(L_j, \mathcal{E}_{\text{out}}, \mathcal{E}_{\text{in}})$ and can thus avoid the addition operation. Furthermore, we eliminate after each update step, dominated polynomials.

First, we will show, that for this class of graphs, the reported path weight polynomials are small and second, that also the number of reported path weights is exponentially smaller in the worst case. Finally, we show that the modified algorithms reports the minimal number of path weights in case of symbolic loop bounds.

Lemma 6. *For all problem instances I , $\mathcal{D}(I)$ contains only weights with at most $\text{slbs}(G) + 1$ terms with non-zero coefficients.*

Proof. First observe that by the construction of our algorithm, all weights are indeed polynomials over the symbolic loop-bounds. We will now show, that any such polynomial consists of at most $\text{slbs} + 1$ terms, by first proving several claims about the possible structure of such terms and by concluding from that, that there cannot be more than $\text{slbs} + 1$ such terms in the polynomial. The claims to show are:

1. Any path in G enters the loops of G in a unique order, i.e. there don't exist two paths P_1, P_2 in G , such that there exist loops L_1, L_2 of G such that the paths P_1 and P_2 can be written as $P_1 = (\dots, \mathcal{E}(L_1), \dots, \mathcal{E}(L_2), \dots)$ and $P_2 = (\dots, \mathcal{E}(L_2), \dots, \mathcal{E}(L_1), \dots)$.
2. Any term in the weight polynomial contains only variables associated with induced subloops contained in $\text{iloops}(L)$ for some $L \in \text{loops}(G)$.
3. Let $L \in \text{loops}(G)$ and $L' \in \text{iloops}(L)$. Furthermore, let $(L_i)_{i=1\dots k}$ be the sequence of induced subloops of L such that $L = L_1, L' = L_k$ and such that L_{i+1} is a subloop of L_i for all $i \in \{1, \dots, k-1\}$. If a term T in the weight polynomial contains the variable associated with L' , then T also contains all variables associated with the loops L_i .

We now prove the first claim, by assuming that such two paths P_1 and P_2 exist. This implies that there are two paths (subpaths of P_1 and P_2) from $\mathcal{E}(L_1)$ to $\mathcal{E}(L_2)$ and vice versa. Since L_1 and L_2 are loops of G , L_1 and L_2 are strongly connected. But we have just seen that the two nodes $\mathcal{E}(L_1), \mathcal{E}(L_2)$ are also connected, which directly implies that L_1 and L_2 can't be strongly connected components of G , which leads to a contradiction.

For the second claim, suppose there exists a term, containing variables associated with induced subloops of two different loops L_1 and L_2 of G . By construction of our algorithm, multiplication (and thus, addition of a variable to a term) only occurs, if $L_1 \in \text{iloops}(L_2)$ or if $L_2 \in \text{iloops}(L_1)$. Thus, L_1 must be a subgraph of L_2 or vice versa and hence, L_1 and L_2 can only be both loops of G , if $L_1 = L_2$.

For the last claim, let us assume otherwise, and let there be a $j \in \{1, \dots, k-1\}$ such that the associated variable of L_j is not contained in T but also such that L_{j+1} contributes its variable to T . By the construction of our algorithm, the variable associated with L_{j+1} was added to T , after splitting $\mathcal{E}(L_j)$ and

identifying L_{j+1} as a strongly connected component of this altered graph. But splitting the node $\mathcal{E}(L_j)$ also involves the multiplication of the weight polynomial with the variable associated with L_j , which contradicts the fact, that this variable is not contained in T .

Now let us count the number of possible terms in a longest path weight, that contain at least one loop-bound variable. For each loop L of G we have a possible set of terms. For each term in such a set, we know that only variables associated with induced subloops of the corresponding loop of G are contained in it. Since a variable must be contained in such a term whenever the variable of a subloop is contained in the term, there are exactly $\text{slbs}(L)$ possible terms per set. Thus in total, the number of possible terms, that contain at least one variable is bounded by $\sum_{L \in \text{loops}(G)} \text{slbs}(L) = \text{slbs}(G)$. Together with the fact, that there is only one term, containing no variables, this completes the proof. \square

Lemma 7. *For our modified algorithm any problem instance I has $\mathcal{D}(I) \leq 2^{\text{slbs}}$.*

Proof. The claim follows from the proof of Lemma 1: In this case, the longest paths within the loop and to the portal node use the same loop pattern and we only have to combine paths using the same loop pattern. Hence the recursion for the number of longest path weights can be expressed as $T(\text{lbs}(G)) \leq \prod_i (T(\text{lbs}(L'_i)) + 1)$. Notice that the plus one disappears if the loop-bound of L' is numeric, as we know that one of the path weights of L_i is numeric and hence can be compared to the case where the subloop is not entered. Hence, we can express it purely in the number of symbolic loop-bounds $\text{slbs}(G)$ and get $T(\text{slbs}(G)) \leq \prod_i T(\text{slbs}(L'_i)) + 1$. This recursion is maximized if G has $\text{slbs}(G)$ subloops with one symbolic loop-bound, where we get $T(\text{slbs}(G)) = 2^{\text{slbs}(G)}$. \square

The next theorem states the main result of this section. Not only, that in the worst case, the output size is much smaller than in the general case of singleton-loop graphs, but also we are able to modify our algorithm, such that it computes an output of minimal size. in case of symbolic loop bounds

Theorem 2. *The modified algorithm reports a minimal number of longest paths in case of symbolic loop bounds.*

Proof. Let us assume without loss of generality, that G contains only induced sub-loops L such that the longest path weights from $\mathcal{E}(L)$ back to $\mathcal{E}(L)$ are unequal to 0. Now consider the set $\mathcal{D}(G, s, t)$ of path weights computed by our algorithm and let P be a path with weight $w_P \in \mathcal{D}(G, s, t)$. We want to show, that for w_P , there exists an instantiation I such that under I , all weights not equal to w_P in $\mathcal{D}(G, s, t)$ are smaller than w_P under I (written: $w_P[I]$). We will prove the claim by structural induction over G . If G contains no loops, G is a DAG and thus, $\mathcal{D}(G, s, t)$ just consists of a single constant, which establishes the base case. For the induction step let us assume otherwise, namely that $\text{loops}(G) \neq \emptyset$. We consider the condensed graph G' by replacing in G all loops $(L_i)_{1 \leq i \leq |\text{loops}(G)|}$ by nodes $(c_i)_{1 \leq i \leq |\text{loops}(G)|}$ and we denote by $C := \{c_i \mid 1 \leq i \leq |\text{loops}(G)|\}$ the set of all contraction nodes. We will now show that we can construct an instantiation I under which $w_P[I] > w'_P[I]$ for any other longest path with weight

$w'_P \in \mathcal{D}(G, s, t)$. For a path P' , we define $C_{P'} := \{c_i \in C \mid P' \text{ traverses } L_i\}$. We define now I such that

- $\forall c_i \in C \setminus C_P: \forall L \in \text{iloops}(L_i): I(\mathbf{b}(L)) = 0$
- For $c_i \in C_P$, let I_i be a maximizing instantiations, as given by the induction hypothesis, then $\forall c_i \in C_P: \forall L \in \text{iloops}(L_i)$ s.t. $L \neq L_i: I(\mathbf{b}(L)) = I_i(L)$.
- Chose the $\mathbf{b}(L_i)$ large enough such that for all paths $P' \neq P$

$$\sum_{c_i \in C_P \setminus C_{P'}} \mathbf{b}(L_i)w_{c_i}[I_i] + \sum_{c_i \in C_P \cap C_{P'}} \mathbf{b}(L_i)(w_{c_i} - w'_{c_i})[I_i] > c_{P'} - c_P,$$

where w_{c_i} (and w'_{c_i}) are the weight polynomials corresponding to the loop L_i for the paths P (and P') and where c_P (and $c'_{P'}$) are the constants in the weights w_P (and $w'_{P'}$).

At this point, it is important to note, that it is always possible to choose loop-bounds, such that the last inequality can be established. This can be seen as follows: if $w_{c_i} = w'_{c_i}$ for all $c_i \in C_P \cap C'_{P'}$ and $C_P \setminus C'_{P'} = \emptyset$, then each non constant term in w_P also exists in $w'_{P'}$. But since both weights are reported by the algorithm, in particular $w'_{P'}$ does not dominate w_P , which implies that c_P must be larger than $c'_{P'}$. Thus, setting all loop-bounds to zero, establishes the inequality. Now assume otherwise. In the case that $C_P \setminus C'_{P'} \neq \emptyset$ we can set for one $c_i \in C_P \setminus C'_{P'}: \mathbf{b}(L_i) := c'_{P'} - c_P + 1$ to establish the inequality, since $w_{c_i}[I_i] \geq 1$. Otherwise there must be an $c_i \in C_P \cap C'_{P'}$ such that $(w_{c_i} - w'_{c_i})[I_i] > 0$, which also means that $(w_{c_i} - w'_{c_i})[I_i] \geq 1$, since the weights are integral. Thus, setting $\mathbf{b}(L_i) := c'_{P'} - c_P + 1$ establishes again the last inequality, which yields:

$$\begin{aligned} w_P[I] &= c_P + \sum_{c_i \in C_P} \mathbf{b}(L_i)w_{c_i}[I] \\ &> c'_{P'} + \sum_{c_i \in C_P} \mathbf{b}(L_i)w_{c_i}[I] - \sum_{c_i \in C_P \setminus C'_{P'}} \mathbf{b}(L_i)w_{c_i}[I] - \\ &\quad \sum_{c_i \in C_P \cap C'_{P'}} \mathbf{b}(L_i)(w_{c_i} - w'_{c_i})[I_i] \\ &> c'_{P'} + \sum_{c_i \in C'_{P'}} \mathbf{b}(L_i)w'_{c_i}[I] - \sum_{c_i \in C'_{P'} \setminus C_P} \mathbf{b}(L_i)w'_{c_i}[I] \\ &= c'_{P'} + \sum_{c_i \in C'_{P'}} \mathbf{b}(L_i)w'_{c_i}[I] = w'_{P'}[I] \end{aligned}$$

which finishes the proof. \square

2.5 The Implementation

The algorithm as described above, suffers from a high constant factor cost due to the following points, for which we will show in this section, how they can be avoided.

- After calling the algorithm on the loops of G , the graph G is altered by removing all vertices and edges, associated with its loops. Even worse this modification has to be undone to determine the longest paths to the nodes within the loops.
- Calling the algorithm recursively on its altered loops involves copying parts of the graph structure.
- As discussed above, in the presence of symbolic loop-bounds, a longest path is not necessarily unique anymore, which involves maintaining sets of polynomials over path weights for each node. This significantly increases the work that has to be done in each computation that involves path weights.

The graph operations can involve a high number of memory allocations which typically leads to an avoidable worsening of the running time. While the second point can be addressed by using a subgraph filter and by bookkeeping and undoing the changes needed for the recursion steps on the loops, the first point has to be addressed in a more complicated way, by changing the algorithm slightly: Instead of altering parts of the graph, we store the longest path weights that have been computed recursively on the loops of the graph in a hash table. Instead of deleting the loops, we bypass them in the following way: Whenever we encounter a node in the computation of the longest paths which is a portal node of a loop L , we jump directly to the corresponding entry node $\mathcal{E}(L)$ of L and use the path informations stored in this table. Since a hash table access can be done in constant time, the modifications of the underlying graph can be avoided at very low costs.

The last point can be addressed in the following way: as mentioned above, the symbolic weights can be represented as polynomials over the symbolic loop-bounds. The first observation that we can make is that we expect only a small number of coefficients to be non-zero. So instead of storing all possible terms, we only store the ones with such coefficients.

The terms in turn are stored as bit-vectors, such that each bit indicates whether a variable is present in the term or not. This enables us, to store the terms (along with their coefficients) in an ordered way, by simply interpreting these bit-vectors as numbers. Addition can then simply be done in linear time. But one question remains, namely how a variable that occurs with a power greater than one can be represented by such a bit-vector. The answer is, that this never occurs. To see that, note that the multiplication of two symbolic weights only occurs in the context of loops, when combining paths from s to some entry node and paths from this entry node to some other node in its loop. But then, the corresponding weights do not share any symbolic weight by definition of a loop. This allows to implement a multiplication as a simple and fast bitwise-and-operation on the bit-vectors of the terms.

2.6 Transformation of non Singleton-Graphs

This far, our new approach is applicable to singleton graphs only. We assume that each loop has a unique entry node. To analyze non-singleton graphs, we

must obtain this property. To this end, we duplicate the loop-body and redirect loop entry edges, such that each loop body has a unique entry node; see Figure 6 for an example.

This transformation comes at the cost of an increased runtime: Although the complexity of the algorithm itself remains unchanged, the input-size increases. In how far this influences the actual performance of the algorithm strongly depends on the analyzed control-flow graph. We will discuss this point in the Section 4.

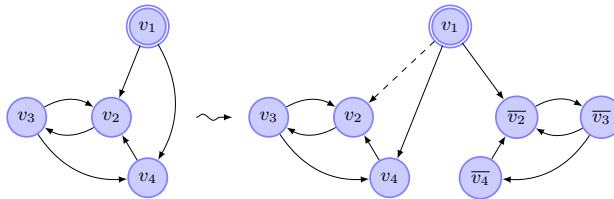


Fig. 6. Transformation of non-singleton loops into singleton loops.

3 Handling Control Flow Constrains

An advantage of the implicit path enumeration technique is given by the possibility to define further constraints on the control flow. A typical example is the exclusion of infeasible path; different conditions of the program to be analyzed might depend on each other, such that certain control flow paths can be excluded. These cases can be coded using control flow constraints

One way to handle such constraints in our approach is to assign boolean literals to the edges of the control-flow graph that indicate whether an edge is part of the graph or not. The goal is now to determine a longest path over all possible truth assignments of the boolean variables occurring in the deadpath constraints. A naive way of doing so, would be to solve the problem from scratch for each possible assignment and to compare in retrospect the obtained longest paths. The obvious drawback of this approach is that in any case an exponential number of problems have to be solved. One possible improvement rests upon the observation that in different calls of our algorithm, the computation of longest subpaths are potentially repeated —independent of the given truth assignment. Thus, instead of computing the longest paths for each truth assignment, we just perform a single call in which we bookkeep for each computed subpath the literals that occurred on that path. At the point in our algorithm, when two paths are concatenated, we first check, whether the literals occurring in both paths are compatible, i.e. whether there can be a truth assignment that allows a path to consist of these two subpaths, which is clearly not the case if in the first subpath there is an edge with a variable that occurs in its negation in the second subpath. Once our algorithm has finished, we are left with a set of longest paths

together with a set of boolean literals. In the final step, we examine these paths for all possible truth assignments. Since this set should be significantly smaller in practice than the union of all sets computed in the naive way, this should lead to significant speed up.

4 Evaluation

Regarding the precision, the new method does not contain any inherent over-approximation and thus, computes exact results even in the parametric case. The loop-bound conversion from relative to absolute loop-bounds as in [8] is not needed.

In this section, we evaluate the runtime improvement on several benchmark suites for real-time applications. The most eminent benchmark suite in the area of timing analysis for hard real-time systems is the Mälardalen WCET benchmark suite⁴. In the following, we refer to our new method as the *Silent (Single-Loop ENTry)* method.

All programs have been compiled via gcc to the ARM7⁵ target architecture, one of the most eminent processors for embedded systems. The binary-files then defined the control-flow graph which is the input to both, our new method and implicit path enumeration (IPET). Note that timing analysis has to resort to the binary level. Compiler transformation may influence the control flow extracted from high-level source files. All tests have been performed on an Intel Core2Duo, 2GHz with 2 GB Ram under Ubuntu operating system. The execution times were measured using the unix-command *time* and thus, include in all cases the time for the algorithm and all input/output operations. For the small execution time (time < 5 seconds), we repeated the tests 10 times and derived the average runtime.

The testcases within Mälardalen Benchmark Suite are rather small. In fact, 30 of the 33 benchmarks are solved in less than one second by IPET and Silent. Unfortunately, other free benchmarks suites (Mibench, Papabench) for real-time systems contain academic examples of limited size, too. To create realistically sized examples, we artificially increased the size of the five largest benchmarks by code duplication and different combination of the different files. Benchmarks *nsichneu-X* are created by duplicating the largest benchmark file *nsichneu x* times. The other files (*all5-X*) are different combination of all five benchmarks.

Analyzing executables instead of source-code files has several consequences. Calls to external library functions may be implicit in the source-code, but contribute to the complexity of the executable's control-flow graph. Hence, the analyzed control-flow graph and control-flow of the source-code may differ. Even if source code contains only singleton loops, the corresponding executable may exhibit non-singleton loops. In addition, complex compiler optimizations—although not common for safety-critical systems—transform the control-flow graph. An overview of the actual structure of the benchmarks is given in Table 1. Beneath

⁴ <http://www.mrtc.mdh.se/projects/wcet/benchmarks.html>

⁵ <http://www.arm.com/products/CPUs/families/ARM7Family.html>

Name	Size (in Byte)		Singleton Graph	# duplicated loops
	C-File	Exec		
adpcm	26582	156759	no	5
compress	13411	149804	no	65
jfdctint	16028	146858	yes	-
nsichneu	118351	176240	yes	-
statemate	52618	162879	no	7
nsichneu-2	208273	235222	yes	-
nsichneu-4	468944	292305	yes	-
nsichneu-6	702670	386961	yes	-
nsichneu-8	936396	481609	yes	-
nsichneu-10	670452	284593	yes	-
all5-a	90274	215433	no	77
all5-b	315562	247443	no	77
all5-c	766144	426427	no	77
all5-d	990502	520579	no	5
all5-e	979908	518338	no	9
all5-f	942084	502580	no	74

Table 1. Benchmarks; Sizes (of source code and of executables) and structure

the sizes of the source-code and of the executable of the benchmarks, it shows which benchmark fits into the singleton-graph model and, if not, how often loops need to be duplicated. Note that this number does not refer to the number of non-singleton loops, but to the number of duplicated loops in the transformed control-flow graph. If a loop has n loop entries, it needs to be duplicated n -times.

Note in addition that code duplication to increase the benchmarks does not necessarily increase the number of non-singleton loops in the graph. If, for instance, a library routines exhibits such a loop, code duplication just leads to more call sites, but not to more such loops.

The evaluation shows that the Silent method strongly outperforms the prior path analysis technique using `lp_solve` and can even compete with CPLEX. While `lp_solve` needs up to over 100 seconds for the larger benchmarks, Silent and CPLEX solve each test-case in about one second or less. Hence, the new method may save up to 99% of the runtime compared to the free ILP solver. Only for smaller test cases, the difference between both method is still very obvious but less dramatic. The evaluation also shows a dependency between non-singleton loops and performance. The more loops need to be duplicated, the worse is the performance of the analysis. Here, the analysis of *compress* with 65 loop duplications, for instance, takes longer than the analysis of other benchmarks of comparable size. However, if this number is low, as for instance for *statemate* or *adpcm*, the performance is only slightly degraded.

4.1 Parametric Path Analysis – Performance

In the parametric case, precision and runtime are of interest. The Silent method can only be compared against PIP directly. PIP, however, exhibits an extremely poor performance. None of the 5 largest benchmarks from Mälardalen bench-

Name	Size (in Byte)		Runtime (s)		
	C-File	Exec	Silent	lp_solve	CPLEX
adpcm	26582	156759	0.04	0.07	0.02
compress	13411	149804	0.3	0.03	0.03
jfdctint	16028	146858	0.01	0.1	0.02
nsichneu	118351	176240	0.02	0.86	0.05
statemate	52618	162879	0.05	0.3	0.04
nsichneu-2	208273	235222	0.03	3.46	0.08
nsichneu-4	468944	292305	0.05	13.69	0.08
nsichneu-6	702670	386961	0.08	30.85	0.11
nsichneu-8	936396	481609	0.11	57.31	0.18
nsichneu-10	670452	284593	0.11	108.8	0.13
all5-a	90274	215433	0.97	4.5	0.04
all5-b	315562	247443	0.95	14.58	0.05
all5-c	766144	426427	1.01	48.13	0.12
all5-d	990502	520579	0.14	92.1	0.11
all5-e	979908	518338	0.16	113.3	0.12
all5-f	942084	502580	0.64	65.9	0.17

Table 2. Performance evaluation: Silent / lp_solve / CPLEX

mark suite can be solved via PIP, since out-of-memory errors are reported. These instances are still trivial for our new approach. Therefore, we compare the runtime of the parametric path analysis with the numeric one. Note that the number of parameters is usually quite limited in practice. Bygde et al. considers cases with at most 2 parameters. Table 3 shows that the parametric case exhibits an

Name	Runtime	Runtime # of parameters						# of parametric formulas					
	Numeric	1	2	3	4	6	8	1	2	3	4	6	8
nsichneu-2	0.03	0.03	0.03	0.03	0.03	-	-	2	2	2	4	-	-
nsichneu-4	0.05	0.05	0.05	0.05	0.05	0.05	0.06	2	4	8	16	16	16
nsichneu-6	0.08	0.08	0.08	0.08	0.08	0.08	0.11	2	4	8	16	16	32
nsichneu-8	0.11	0.11	0.11	0.11	0.12	0.12	0.12	2	4	8	16	32	64
nsichneu-10	0.11	0.12	0.12	0.12	0.12	0.12	0.13	2	4	8	16	32	64
all5-a	0.97	1	1.73	1.9	2.02	2.11	2.11	1	23	33	33	33	33
all5-b	0.95	2.03	2.03	2.04	2.36	2.35	2.38	13	13	13	17	17	17
all5-c	1.01	1.01	1.01	1.01	1.24	3.42	3.44	1	1	1	4	15	15
all5-d	0.14	0.22	0.28	0.3	0.33	0.45	0.45	12	21	21	21	25	25
all5-e	0.16	0.28	0.33	0.62	0.67	1.11	1.11	12	12	33	33	56	56
all5-f	0.64	0.64	0.64	0.66	0.71	1.19	1.19	1	1	1	2	8	8

Table 3. Performance evaluation: Silent method, parametric test cases.

increased runtime compared to the numeric case. This increase ranges from a factor of ≈ 1 (*nsichneu*) to a factor of < 4 (*all5-5*). Considering the small execution times of the Silent method, even such an increase is acceptable. Note that Bygde et al. also fail to compute parametric formulas in case of larger examples. Benchmark *nsichneu*, for instance, can not be solved using the method presented in [7] due to the high complexity of their method. The runtime in the paramet-

ric case is output-polynomial. The right part of Table 3 shows the number of parametric formulas reported by the Silent method. Although we can see that the output-size influences the performance, the number of duplicated loops still has a stronger impact in practice. The maximal number of reported formulas is 64 (*nsichneu-8* and *nsichneu-10*, 8 parameters, 0 duplicated loops)—runtime of the analysis in both cases, however is only about 10% of the runtime of *all5-f*, with 8 parametric formulas, but 74 duplicated loops. Note that our algorithm does not perform any further simplifications on the reported formulas. Hence, some formulas may be dominated by others and can be neglected.

4.2 Parametric Path Analysis – Precision

Rather simple control-flow structures cause the imprecision of the path analysis via PIP. Figure 7 shows two parallel loops. If we assume for instance that L_1 has a parametric loop-bound, the worst-case path as derived by PIP will always contain loop L_1 . Loop nesting leads in the same manner to an overapproximation.

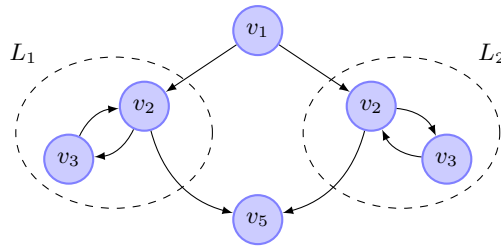


Fig. 7. Parallel Loops

In the following, we present two simple benchmarks from Mälardalen benchmark suite, for which PIP was able to derive a parametric timing formula: *insertion-sort* and *matrix-multiplication*.

Insertion Sort This benchmark implements insertion sort. It contains one loop to initialize an array of size n and then sorts this array using a nested loop of depth 2.

$$Time_{PIP}(n) = 156n^2 + 674n + 1186$$

$$Time_{Simple}(n) = 131n^2 + 71n + 1185$$

Matrix Multiplication This benchmark first initializes two $n \times n$ matrices and then multiplies them in $O(n^3)$ using nested parametric loops of depth 3.

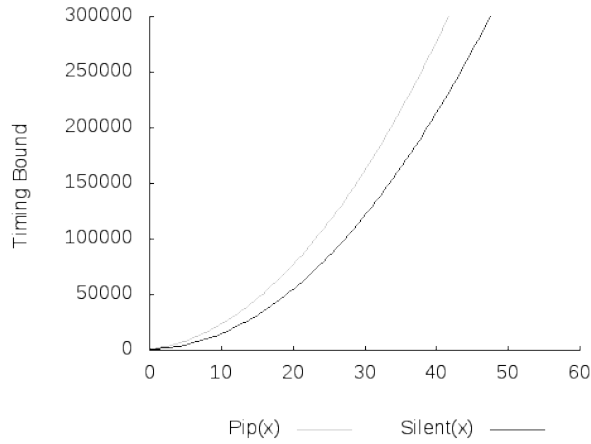


Fig. 8. PIP vs. Silent—Insertionsort

$$Time_{PIP}(n) = \begin{cases} 386n^3 + 782n^2 \\ \quad + 790n + 643 & \text{if } n > 1 \\ 2992 & \text{if } n \leq 1 \end{cases}$$

$$Time_{Simple}(n) = 111n^3 + 164n^2 + 845n + 793$$

In both benchmarks, the coefficients in $Time_{PIP}$ are higher than in $Time_{Simple}$, such that the overapproximation grows as the parameters increase. The overapproximation of the PIP method is caused by loop nesting causes and the missing relation between execution counts of inner and outer loops. Note that the parametric bounds computed by Silent method are precise, i.e., instantiating the formula delivers the same results as the numeric analysis with annotated loop-bounds.

5 Conclusions

We have given a new algorithm to compute the path analysis in the state-of-the-art method for computing worst-case execution times of programs. Our algorithm outperforms the existing methods in the case of a parametric timing analysis and is competitive with current methods based on commercial integer linear programming solvers for the special case of non-parametric loop-bounds. For the first time, it is now possible to use parametric timing analysis for whole programs.

In the special case of while-programs with symbolic loop bounds, our algorithm reports the minimal possible number of distinct paths. We want to investigate on algorithms that achieve this also in the general case of singleton-loop graphs.

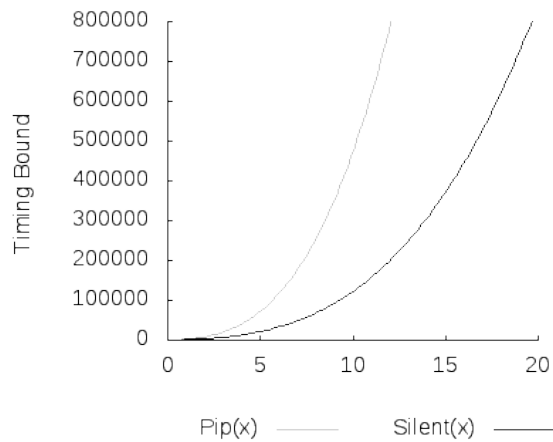


Fig. 9. PIP vs. Silent—Matrix Multiplication

Given the efficiency of the integer linear programming solvers in the previous approach to the path analysis, we want to investigate whether the integer linear programming formulation is integral in the case of singleton-loop graphs.

References

1. Li, Y.T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: DAC '95: Proceedings of the 32nd annual ACM/IEEE Design Automation Conference, New York, NY, USA, ACM (1995) 456–461
2. Theiling, H.: ILP-based Interprocedural Path Analysis. In: Proceedings of the Workshop on Embedded Software, Grenoble, France (2002)
3. Feautrier, P.: (The parametric integer programming's home <http://www.piplib.org>)
4. Dantzig, G.B.: Linear Programming and Extensions. Princeton University Press, Princeton, NJ (1963)
5. Gomory, R.E.: An algorithm for integer solutions to linear programming. In Graves, R.L., Wolfe, P., eds.: Recent Advances in Mathematical Programming, New York, McGraw-Hill (1969) 269–302
6. Lisper, B.: Fully automatic, parametric worst-case execution time analysis. Third International Workshop on Worst-Case Execution Time Analysis (2003) 77–80
7. Bygde, S., Ermedahl, A., Lisper, B.: An efficient algorithm for parametric wcet calculation. In: RTCSA '09: Proceedings of the 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Washington, DC, USA, IEEE Computer Society (2009) 13–21
8. Altmeyer, S., Hümbert, C., Lisper, B., Wilhelm, R.: Parametric timing analysis for complex architectures. In: Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'08), Kaohsiung, Taiwan, IEEE Computer Society (2008) 367–376
9. Bernat, G., Burns, A.: An approach to symbolic worst-case execution time analysis. In: 25th IFAC Workshop on Real-Time Programming. Palma (Spain). (2000)

10. Chapman, R., Burns, A., Wellings, A.: Combining static worst-case timing analysis and program proof. *Real-Time Syst.* **11**(2) (1996) 145–171
11. Coffman, J., Healy, C.A., Mueller, F., Whalley, D.B.: Generalizing parametric timing analysis. In: LCTES. (2007) 152–154
12. Vivancos, E., Healy, C., Mueller, F., Whalley, D.: Parametric timing analysis. In: LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems, New York, NY, USA, ACM Press (2001) 88–93