The Bounded Pathwidth of Control-flow Graphs

Giovanna Kobus Conrado, Amir Goharshady, Chun Kit Lam

The Bounded Pathwidth of Control-flow Graphs

GIOVANNA KOBUS CONRADO, AMIR KAFSHDAR GOHARSHADY, and CHUN KIT LAM,
Hong Kong University of Science and Technology, Hong Kong

Pathwidth and treewidth are standard and well-studied graph sparsity parameters which intuitively model the degree to which a given graph resembles a path or a tree, respectively. It is well-known that the control-flow graphs of structured goto-free programs have a tree-like shape and bounded treewidth. This fact has been exploited to design considerably more efficient algorithms for a wide variety of static analysis and compiler optimization problems, such as register allocation, μ-calculus model-checking and parity games, data-flow analysis, cache management, and liftetime-optimal redundancy elimination. However, there is no bound in the literature for the *pathwidth* of programs, except the general inequality that the pathwidth of a graph is at most $O(\lg n)$ times its treewidth, where n is the number of vertices of the graph.

In this work, we prove that control-flow graphs of structured programs have bounded pathwidth and provide a linear-time algorithm to obtain a path decomposition of small width. Specifically, we establish a bound of $2 \cdot d$ on the pathwidth of programs with nesting depth d. Since real-world programs have small nesting depth, they also have bounded pathwidth. This is significant for a number of reasons: (i) pathwidth is a strictly stronger parameter than treewidth, i.e. any graph family with bounded pathwidth has bounded treewidth, but the converse does not hold; (ii) any algorithm that is designed with treewidth in mind can be applied to bounded-pathwidth graphs with no change; (iii) there are problems that are fixed-parameter tractable with respect to pathwidth but not treewidth; (iv) verification algorithms that are designed based on treewidth would become significantly faster when using pathwidth as the parameter; and (v) it is easier to design algorithms based on bounded pathwidth since one does not have to consider the often-challenging case of merge nodes in treewidth-based dynamic programming. Thus, we invite the static analysis and compiler optimization communities to adopt pathwidth as their parameter of choice instead of, or in addition to, treewidth. Intuitively, control-flow graphs are not only tree-like, but also path-like and one can obtain simpler and more scalable algorithms by relying on path-likeness instead of tree-likeness.

As a motivating example, we provide a simpler and more efficient algorithm for spill-free register allocation using bounded pathwidth instead of treewidth. Our algorithm reduces the runtime from $O(n \cdot r^{2 \cdot \mathtt{tw} \cdot r+2 \cdot r})$ to $O(n \cdot \mathtt{pw} \cdot r^{\mathtt{pw} \cdot r+r+1})$, where n is the number of lines of code, r is the number of registers, $\mathtt{pw}$ is the pathwidth of the control-flow graph and $\mathtt{tw}$ is its treewidth. We provide extensive experimental results showing that our approach is applicable to a wide variety of real-world embedded benchmarks from SDCC and obtains runtime improvements of 2-3 orders of magnitude. This is because the pathwidth is equal to the treewidth, or one more, in the overwhelming majority of real-world CFGs and thus our algorithm provides an exponential runtime improvement. As such, the benefits of using pathwidth are not limited to the theoretical side and simplicity in algorithm design, but are also apparent in practice.

CCS Concepts: • **Theory of computation → Fixed parameter tractability**; **Graph algorithms analysis**; **Discrete optimization**; • **Software and its engineering → Formal software verification**.

1 INTRODUCTION

Control-flow Graphs [Allen 1970]. A *Control-flow Graph* (CFG) G of a program P is a graph representation of all execution paths in P. The graph G has one vertex for every statement or line of code in P and a directed edge between two vertices if they can potentially be executed consecutively in a run of the program. CFGs are highly ubiquitous and almost all flow-sensitive static analysis and compiler optimization tasks are modeled as graph problems over the CFGs. This includes data-flow analyses [Khedker et al. 2017; Myers 1981; Reps et al. 1995a; Sharir et al. 1978], alias and pointer analyses [Hind et al. 1999; Smaragdakis et al. 2015], shape analysis [Reps

Authors' address: Giovanna Kobus Conrado, gkc@connect.ust.hk; Amir Kafshdar Goharshady, goharshady@cse.ust.hk; Chun Kit Lam, cklamaq@connect.ust.hk, Department of Computer Science and Engineering, Department of Mathematics, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong.

1995; Wilhelm et al. 2000], model-checking of LTL and μ-calculus [Ferrara et al. 2005; Obdrzálek 2003], register allocation [Callahan and Koblenz 1991; Chaitin et al. 1981], instruction selection and scheduling [Bernstein and Rodeh 1991; Blindell et al. 2015], invariant generation [Monniaux and Gawlitza 2012], termination analysis [Zhu and Kincaid 2021], and program synthesis [Srivastava et al. 2013]. In this work, we consider CFGs that have one vertex per statement/line of code. Some of the analyses mentioned above use coarser notions of CFGs, e.g. having each vertex correspond to a basic block of code. Our results trivially extend to such coarser CFGs, too.

Sparsity of CFGs. While many static analysis and compiler optimization tasks are reduced to graph problems over the CFGs, it is often the case that the resulting graph problem is either NP-hard, e.g. for register allocation [Chaitin et al. 1981], or has an inefficient polynomial-time algorithm, e.g. for interprocedural data-flow analysis [Reps et al. 1995a]. However, it is well-known that control-flow graphs are sparse, since every node of a CFG has bounded outdegree. Hence, applying algorithms designed for general graphs is extremely wasteful in these scenarios. Instead, there has been ample research focused on formalizing the sparsity of CFGs, e.g. Thorup [1998], and then exploiting it to design faster algorithms [Bodlaender et al. 1998; Chatterjee et al. 2020; Obdrzálek 2003].

Graph Width Parameters. Width parameters are a family of parameters that are often used to formalize the sparsity and structural complexity of a graph. They are ubiquitous in parameterized complexity [Bodlaender 1988; Cygan et al. 2015; Harvey and Wood 2017]. The two most well-studied width parameters are pathwidth [Robertson and Seymour 1983] and treewidth [Robertson and Seymour 1984, 1986, 1990].

Treewidth [Robertson and Seymour 1984]. The *treewidth* of a graph G is a measure of its sparsity and tree-likeness. Only trees and forests have treewidth 1 and any graph that has treewidth $\mathtt{tw}$ can be decomposed into small subgraphs, each containing at most $\mathtt{tw}+1$ vertices, that are connected to each other in a tree-like manner. This is called a *tree decomposition*. See Section 3 for a formal definition and Figure 3 for an intuitive example. The importance of treewidth stems from the fact that many NP-hard graph problems are solvable in polynomial time when the treewidth is bounded [Bodlaender 1997; Cygan et al. 2015; Goharshady and Mohammadi 2020; Robertson and Seymour 1986]. Moreover, even problems that have polynomial-time solutions on general graphs can often be solved faster on graphs of bounded treewidth [Fomin et al. 2018; Izumi et al. 2022]. This is usually due to the fact that one can perform dynamic programming on a bounded-treewidth graph in virtually the same manner as a tree [Bodlaender 1988]. On the other hand, many well-studied families of graphs that are often encountered in practice have bounded treewidth [Bodlaender 1998]. Examples include cacti, series-parallel graphs, outerplanar graphs, and, most important in our context, control-flow graphs of structured programs [Thorup 1998].

Pathwidth [Robertson and Seymour 1983]. The *pathwidth* of a graph G is a measure of its path-likeness. A graph that has pathwidth $\mathtt{pw}$ can be decomposed into small subgraphs, each with at most $\mathtt{pw}+1$ vertices, that are connected to each other as a path. This is called a *path decomposition*. See Section 3 for details and Figure 4 for an example path decomposition of the graph in Figure 3. Path decompositions are a special case of tree decompositions, in the same manner that paths are special cases of trees. More specifically, every path decomposition is also a tree decomposition, but the converse is not true. This signifies that pathwidth is a stronger parameter than treewidth and all algorithms that use a tree decomposition of a graph can also use a path decomposition instead with no need for any modification in the algorithm itself.

Relationship between Treewidth and Pathwidth. Given that every path decomposition is also a tree decomposition, it is easy to see that the treewidth of a graph is less than or equal to its pathwidth.

Intuitively, if one can decompose a graph into a *path* of small subsets, the same construction is also a decomposition into a *tree* of small subsets. On the other hand, the pathwidth of any graph G with n vertices is at most $O(\lg n)$ times its treewidth [Korach and Solel 1993]. As such, any problem that is fixed-parameter tractable (FPT) with respect to treewidth* is also FPT for pathwidth. However, the converse does not hold. There are problems that are FPT when the pathwidth is bounded, but not when only the treewidth is bounded. While such problems are rare and often handcrafted, the first natural example was recently found in Belmonte et al. [2022]. More importantly, even when a problem is FPT with respect to both pathwidth and treewidth, the pathwidth-based algorithms are often much more efficient, e.g. Obdrzálek [2003] and Meybodi et al. [2022]. Intuitively, dynamic programming algorithms using pathwidth [Arnborg 1985] and treewidth [Bodlaender 1988] are similar to dynamic programming over paths and trees, respectively. So, it is no surprise that the former is more efficient as it does not have to consider any tree branching.

Treewidth of Control-flow Graphs. A fundamental result regarding the sparsity of control-flow graphs was obtained in Thorup [1998], in which it was shown that all structured, i.e. goto-free, programs have control-flow graphs with bounded treewidth. This work proved that the treewidth is at most 3 for Pascal and Algol programs, 5 for Modula, and 7 for C programs†. Similar results were then obtained for Java [Chatterjee et al. 2017; Gustedt et al. 2002], Solidity [Chatterjee et al. 2019a] and Ada [Burgstaller et al. 2004]. In case of Java, it is possible to construct unrealistic and pathological programs with arbitrary treewidth, but this requires an unreasonably large nesting depth, e.g. n nested `for` loops to force a treewidth of $\Omega(n)$. In practice, the treewidth is rarely more than 5 [Gustedt et al. 2002]. Finally, the work [Krause et al. 2020] showed that the treewidth increases by at most g if the program has up to g goto statements. Based on these bounded treewidth results, many works designed more efficient algorithms for various compiler optimization and static analysis problems with the assumption that the treewidth is bounded.

Treewidth-based Compiler Optimization. The bounded treewidth of CFGs led to efficient algorithms for otherwise NP-hard tasks in compiler optimization. For example, Thorup [1998] provided a practical linear-time 4-approximation for the classical problem of register allocation, i.e. finding the optimal number of registers for a program P such that no spilling occurs. Without the bounded-treewidth assumption, this problem is equivalent to graph coloring and hard-to-approximate within any constant factor unless P=NP [Chaitin et al. 1981; Krause 2014]. In Bodlaender et al. [1998], it was shown that optimal register allocation can be performed in linear time when the number of registers is bounded. This work provided an algorithm with the runtime $O(n \cdot r^{2 \cdot \mathtt{tw} \cdot r + 2 \cdot r})$ that, given a program with n statements whose treewidth is $\mathtt{tw}$, decides whether it is possible to map each variable of the program to one of r registers so that there is no spilling. Note that this algorithm is linear when both the treewidth $\mathtt{tw}$ and the number of registers r are bounded. The work Krause [2013b] provided a linear-time algorithm for a different formulation of the register allocation problem, in which the number of registers and the treewidth are bounded, but the goal is to minimize a certain cost function instead of entirely avoiding spills. Using treewidth for compiler optimizations is already the default behavior in the SDCC compiler which is used for compiling embedded C programs [Krause 2013b]. Treewidth has also been used for lifetime-optimal speculative partial redundancy elimination (LOSPRE) [Krause 2021], optimal bank selection [Krause 2013a], instruction

*A problem is Fixed-parameter Tractable (FPT) with respect to the parameter w if it can be solved in $O(n^c \cdot f(w))$, where n is the size of the input, c is a constant, and f is any computable function [Cygan et al. 2015]. This definition captures the idea that on instances in which the parameter is bounded, the problem can be solved in polynomial time $O(n^c)$. However, the runtime's dependence on the parameter can be arbitrarily bad.

†The work Thorup [1998] claimed a bound of 6 for the treewidth of C programs, but this was later corrected to 7 by Krause et al. [2020].

selection [Koes 2009], compiler optimizations based on data-flow [Nordgaard and Meyer 2020] and reducing cache misses [Ahmadi et al. 2022; Chatterjee et al. 2019c].

Treewidth-based Verification. Treewidth has been used extensively as a sparsity parameter for program verification, especially in static analysis. In model-checking, the well-known theorem of Courcelle [1990] shows that any graph property written as a formula in the monadic second order logic can be decided by a finite-state tree automaton, in linear-time, on any input graph of bounded treewidth. Moreover, a similarly linear-time algorithm for μ-calculus model-checking and solving parity games was provided by Obdrzálek [2003]. Treewidth also helps reduce the runtime complexity in local, but not global, LTL model checking [Ferrara et al. 2005]. Moreover, it is a central ingredient in iterative-free and catamorphic program analysis [Ogawa et al. 2003a,b]. The work Chatterjee et al. [2020] shows the bounded-treewidth assumption can reduce the runtime of on-demand interprocedural data-flow analysis from quadratic to linear. Similar techniques have been used for computing algebraic path properties in concurrent settings [Chatterjee et al. 2016]. The work Sankaranarayanan [2020] performs reachability analysis using treewidth-based message-passing. See Aiswarya [2022] for a more comprehensive discussion of how treewidth is exploited in program verification.

Practical Impacts of Sparsity. The sparsity of control-flow graphs, as formalized by their small treewidth, is significant in two categories of program analyses:

- *NP-hard problems:* Certain classical tasks, such as optimal register allocation [Chaitin et al. 1981] or cache management [Calder et al. 1998; Lavaee 2016; Petrank and Rawitz 2002], are equivalent to graph problems, such as optimal coloring, which are known to be NP-hard or hard-to-approximate within a constant multiplicative factor unless P=NP. For these problems, the community mostly gave up on optimal solutions and instead focused on designing efficient heuristics that work well in practice, but do not provide any optimality guarantees. Using sparsity parameters, specifically treewidth, one can design optimal algorithms whose runtimes depend polynomially on the size of the program but exponentially on the width. Examples of these approaches are Thorup [1998], Chatterjee et al. [2019c] and Ahmadi et al. [2022]. Simply put, it is possible to push most of the complexity on the treewidth or pathwidth, finding algorithms that work in polynomial time on all real-world sparse instances. Such algorithms are the only practical solutions that are guaranteed to output optimal results or approximations within a constant factor.
- *Scalable Lightweight Formal Methods:* On the other side of the spectrum, there are lightweight verification tasks which are known to be solvable in polynomial time, such as interprocedural data-flow analysis [Reps et al. 1995b] or algebraic program analysis [Kincaid et al. 2021]. Despite being polynomial-time, classical algorithms for these problems are often not scalable enough for modern huge codebases with millions of lines of code. In these cases, exploiting sparsity has led to algorithms with polynomial runtimes of a lower degree [Chatterjee et al. 2019b, 2016, 2018; Conrado et al. 2023; Goharshady and Zaher 2023], often linear, which are much more practical than their classical counterparts that ignore sparsity. Other examples of problems in this category are μ-calculus model-checking [Obdrzálek 2003] and classical algorithms for the analysis of Markov chains and MDPs [Asadi et al. 2020; Chatterjee and Lacki 2013].

Our Pathwidth Bound. In this work, we prove that control-flow graphs of structured programs have a *pathwidth* of at most $2 \cdot d$, where d is the nesting depth of the program (Section 4). We also provide a simple linear-time algorithm that produces a path decomposition of width at most $2 \cdot d$. Since real-world programs have bounded nesting depth, this shows that they also have a

bounded pathwidth. To the best of our knowledge, this is the first constant bound on the pathwidth of control-flow graphs. Previously, it was known that CFGs have a pathwidth of $O(\lg n)$, based on the general bound of Korach and Solel [1993], but this was not useful for compiler optimization and static analysis algorithms whose runtimes have (super-)exponential dependence on the width.

Intuition. We remark that it is quite intuitive to expect real-world well-written programs to have control-flow graphs with bounded pathwidth, as they tend to follow an almost-linear structure in their execution. Moreover, most coding standards discourage or prohibit having many nested levels of if statements or loops with huge bodies.

Significance of Bounded Pathwidth. Our bounded pathwidth result is significant for a number of reasons:

- Pathwidth is a strictly stronger parameter than treewidth, i.e. any graph family with bounded pathwidth has bounded treewidth by definition, but the converse does not hold. Hence, any compiler optimization or static analysis algorithm that was designed based on treewidth can be applied using pathwidth with no change in the algorithm.
- As mentioned, there are problems that are FPT with respect to pathwidth, but not treewidth [Belmonte et al. 2022]. Hence, the bounded-pathwidth assumption allows us to theoretically solve a strictly larger family of problems over CFGs.
- Several verification algorithms that are designed based on treewidth would become significantly faster when using pathwidth as the parameter, i.e. if they are given a path decomposition as part of their input instead of a tree decomposition. This is a free runtime improvement that does not even require any change in the algorithm and is due to the fact that these approaches' runtimes are dominated by the so-called merge nodes, i.e. nodes of the tree decomposition that have two or more children. Path decompositions have no merge nodes by definition. As a concrete example, the runtime of the μ-calculus model-checking algorithm of Obdrzálek [2003] decreases from $O(n \cdot \mathtt{tw}^2 \cdot m^2 \cdot \delta^{2\cdot((\mathtt{tw}+1)\cdot m)^2})$ to $O(n \cdot \mathtt{pw}^2 \cdot m^2 \cdot \delta^{((\mathtt{pw}+1)\cdot m)^2})$ by simply using pathwidth instead of treewidth. Here, $\mathtt{tw}$ and $\mathtt{pw}$ are the treewidth and pathwidth of the program, respectively, δ is the alternation depth of the formula, m is the formula size and n is the number of lines in the program. Although pathwidth can in general be much larger than treewidth, this is not the case for control-flow graphs since we prove they have bounded pathwidth. On the experimental side, as we will see in Section 6, most programs have a pathwidth of at most 3 and the width usually remains the same no matter we use a tree decomposition or a path decomposition. Note that both runtimes above are linear in n, but using a path decomposition provides a significant exponential improvement in the dependence on δ and m and thus the overall runtime.
- Informally speaking, it is much easier to design algorithms based on pathwidth rather than treewidth. This is because path decompositions have no branching and dynamic programming over them is similar to paths, whereas in tree decompositions, one has to handle branching (merge) nodes as well.

Our Contributions. In this work, our contributions are as follows:

- We prove that the control-flow graph of any structured program P with nesting depth d has a pathwidth of at most $2 \cdot d$.
- Our result is constructive. We provide an efficient linear-time algorithm that, given the program P, produces a path decomposition of width at most $2 \cdot d$.
- As a motivating example, we consider the problem of register allocation as formalized in Chaitin et al. [1981]. Using pathwidth, we provide an efficient FPT algorithm with runtime $O(n \cdot \mathtt{pw} \cdot r^{\mathtt{pw}\cdot r+r+1})$, where $\mathtt{pw}$ is the pathwidth and r is the number of registers. In contrast,

the treewidth-based algorithm in [Bodlaender et al. 1998] takes $O(n \cdot r^{2 \cdot \mathtt{tw} \cdot r + 2 \cdot r})$ time for the same problem.
- We provide extensive experimental results over the SDCC benchmarks [Dutta 2000; Dutta et al. 2003], which are real-world embedded programs written in C. Our experiments show that our pathwidth-based algorithm significantly outperforms the treewidth-based algorithm of Bodlaender et al. [1998] and that our approach is the first to achieve enough scalability to handle real-world instances of the register allocation problem in embedded environments with up to 8 registers. Note that register allocation is a notoriously hard problem and is NP-hard even for 3 registers [Chaitin et al. 1981]. Thus, we are providing the first optimal and non-heuristic algorithm for register allocation that is also applicable to the real-world instances in embedded systems. However, finding algorithms that can handle larger numbers of registers remains an elusive and unsolved problem.

In summary, we show that control-flow graphs of structured programs have bounded pathwidth and that pathwidth-based algorithms are at least as efficient as treewidth-based algorithms and often significantly more efficient. As such, we invite the compiler optimization and static analysis communities to adopt pathwidth as their primary parameter instead of, or in addition to, treewidth when dealing with control-flow graphs.

Limitation. Our bounded pathwidth result applies to intra-procedural control-flow graphs, i.e. flow graphs of a single function. Thus, its main limitation is that it does not hold for inter-procedural control-flow graphs (ICFGs). This is a shared limitation with the classical results on treewidth such as Thorup [1998]. In the inter-procedural setting, one can obtain dense call graphs, and thus dense ICFGs, e.g. by creating a program in which every function calls every other function. The call graph of such an adversarial program would be the complete graph K_n, which has a treewidth/pathwidth of $n - 1$.

Organization. In Section 2, we fix a syntax for our programs. In Section 3, we formally define the concepts of treewidth and pathwidth. In Section 4, we show that the control-flow graph of structured programs has pathwidth of at most $2 \cdot d$, where d is the nesting depth of the program. In Section 5, we present our pathwidth-based algorithm for spill-free register allocation. Finally, Section 6 contains our experimental results and a comparison with the previous treewidth-based algorithm of Bodlaender et al. [1998].

2 STRUCTURED PROGRAMS AND CONTROL-FLOW GRAPHS

In this section, we define an abstract programming language called STRUCTURED, and use it to formalize the concept of a control-flow graph and present our results in a general language-independent fashion. Our syntax closely follows Thorup [1998]. The techniques and algorithms in the sequel are applicable to any structured program written in common high-level languages, such as C, C++, Java and Pascal. As is standard, we do not allow our programs to have arbitrary `goto` statements. Similar to the case of treewidth, if a program has g `goto` statements, then the pathwidth increases by at most g. See Krause et al. [2020] for a more detailed treatment of this point.

Syntax. Figure 1 provides the syntax of the STRUCTURED language. Our language contains all the typical constructs of a structured programming language, such as sequential composition of statements, conditional branching, nested while loops, and `break` and `continue` statements. Note that our `break`/`continue` statements are labeled and allow jumping to the end or next iteration of any loop in which the statement is contained. We also allow for atomic statements α that do not affect the control-flow graph of the program, e.g. a variable assignment. Finally, it is trivial to

$$
\begin{array}{lr}
P ::= \texttt{program}\ S\ \texttt{margorp} & \\
S ::= & \textit{statement} \\
\quad S; S & \textit{sequential composition} \\
\quad \texttt{if}_l\ B\ \texttt{then}\ S\ \texttt{else}\ S\ \texttt{fi}_l & \textit{conditional branching} \\
\quad \texttt{while}_l\ B\ \texttt{do}\ S\ \texttt{done}_l & \textit{while loop} \\
\quad \texttt{return} & \textit{termination} \\
\quad \texttt{break}_l & \textit{breaking loop } l \\
\quad \texttt{continue}_l & \textit{jumping to the next iteration of loop } l \\
\quad \alpha & \textit{atomic statement}
\end{array}
$$

Fig. 1. Syntax of the STRUCTURED language. B is a boolean expression. We drop the label l when it is clear from context. The statements $\texttt{break}_l$ and $\texttt{continue}_l$ can only appear in the body of the loop $\texttt{while}_l$.

define other common structured constructs such as `switch` or a `for` loop as syntactic sugar. We drop these for brevity.

Semantics. Our programs follow the usual semantics. We do not define comprehensive semantics for our language. Instead, we simply define the control-flow graph, which is the only semantic concept we need to reference in the rest of the paper.

Entry and Exit Nodes. Let σ be a statement. We define $\lceil\sigma\rceil$, the entry node of σ, as follows:

$$
\begin{aligned}
\lceil \texttt{program}\ S\ \texttt{margorp} \rceil &= \texttt{program} & \\
\lceil S_1; S_2 \rceil &= \lceil S_1 \rceil & \textit{sequential composition} \\
\lceil \texttt{if}_l\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi}_l \rceil &= \texttt{if}_l & \textit{conditional branching} \\
\lceil \texttt{while}_l\ B\ \texttt{do}\ S\ \texttt{done}_l \rceil &= \texttt{while}_l & \textit{while loop} \\
\lceil \sigma \rceil &= \sigma & \textit{other cases}
\end{aligned}
\tag{1}
$$

We define $\lfloor\sigma\rfloor$, the exit node of σ as follows:

$$
\begin{aligned}
\lfloor \texttt{program}\ S\ \texttt{margorp} \rfloor &= \texttt{margorp} & \\
\lfloor S_1; S_2 \rfloor &= \lfloor S_2 \rfloor & \textit{sequential composition} \\
\lfloor \texttt{if}_l\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi}_l \rfloor &= \texttt{fi}_l & \textit{conditional branching} \\
\lfloor \texttt{while}_l\ B\ \texttt{do}\ S\ \texttt{done}_l \rfloor &= \texttt{done}_l & \textit{while loop} \\
\lfloor \sigma \rfloor &= \sigma & \textit{other cases}
\end{aligned}
\tag{2}
$$

Note that we distinguish between the different words in our program, e.g. two `if` statements that appear in different points of the program are not considered the same. Thus, the entry and exit nodes are well-defined and unique. Whenever needed, we add labels to our `if` and `while` statements to explicitly distinguish them, but we drop the labels if there is no ambiguity.

Control-flow Graph. The control-flow graph of a program P is a directed graph $G = (V, E)$ in which V contains one vertex for every `return`, $\texttt{break}_l$, $\texttt{continue}_l$ and α statement in P, as well as dedicated vertices for the entry and exit nodes of the program, conditional statements, and loop statements. The edges in E are defined as follows:

- If $P = \texttt{program}\ S\ \texttt{margorp}$, then there is an edge from `program` to $\lceil S \rceil$ and an edge from $\lfloor S \rfloor$ to `margorp`. There is also an edge from every `return` node to `margorp`.

```
program
  while1 B do
    if1 B then
      break1
    else
      α1
    fi1;
    while2 B do
      if2 B then
        break2
      else
        α2;
        continue2
      fi2
    done2
  done1;
  if3 B then
    α3
  else
    α4
  fi3
margorp
```

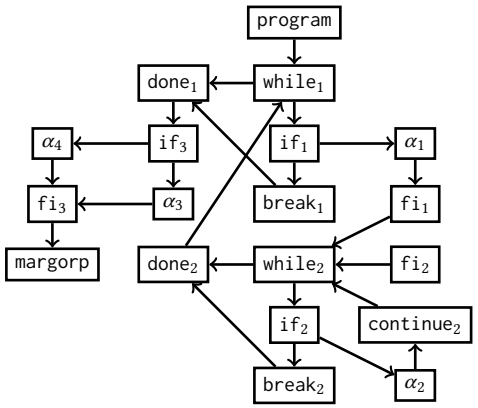

Fig. 2. A STRUCTURED program (left) and its control-flow graph (right). In this example, $\texttt{fi}_2$ is dead code and will never be reached.

- For every "$S_1; S_2$" that appears in P, there is an edge from the exit node $\lfloor S_1 \rfloor$ to the entry node $\lceil S_2 \rceil$.
- For every "`if` B `then` S_1 `else` S_2 `fi`" that appears in P, we have an edge from this `if` node to each of the entry nodes $\lceil S_1 \rceil$ and $\lceil S_2 \rceil$. We also have edges from $\lfloor S_1 \rfloor$ and $\lfloor S_2 \rfloor$ to `fi`.
- For every "`while` B `do` S `done`" that appears in P, we have an edge from this `while` node to the entry node $\lceil S \rceil$ and an edge from the exit node $\lfloor S \rfloor$ back to this `while` node. Moreover, we have an edge from `while` to `done`.
- If the rules above create an outgoing edge from a $\texttt{break}_l$ or $\texttt{continue}_l$ statement, we ignore that edge and do not add it to the CFG. Instead, we add an edge from every $\texttt{continue}_l$ statement to the corresponding loop node $\texttt{while}_l$. Similarly, we add an edge from every $\texttt{break}_l$ statement to the `done` node corresponding to $\texttt{while}_l$.

Example. Figure 2 shows an example program in our STRUCTURED language (left) and its control-flow graph (right).

Nesting Depth. We define the *depth* $d(S)$ of a statement S in a natural way, as the maximum number of nested statements within S. Formally,

$$\begin{aligned} d(\texttt{program}\ S\ \texttt{margorp}) &= 1 + d(S) \\ d(S_1; S_2) &= \max\{d(S_1), d(S_2)\} \\ d(\texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi}) &= 1 + \max\{d(S_1), d(S_2)\} \\ d(\texttt{while}\ B\ \texttt{do}\ S\ \texttt{done}) &= 1 + d(S) \\ d(\alpha) = d(\texttt{break}) = d(\texttt{continue}) &= d(\texttt{return}) = 0 \end{aligned} \tag{3}$$

The *nesting depth* d of the whole program P is simply $d(P)$.

Nesting Depth of Real-world Programs. Real-world programs often have a nesting depth of two or three. A depth of five or more is strictly discouraged or sometimes even prohibited by style guides and coding standards [Ausnit-Hood et al. 1997; Barr 2009; King et al. 1999; McConnell 2004; Sutter and Alexandrescu 2004]. This is because a large nesting depth makes the program much less human-readable and is often the cause of many bugs. Understanding more than three levels of nesting is generally hard for humans [Marca 1981; Yourdon 1985]. As such, it is reasonable to assume that the nesting depth d is a small constant. This assumption is also confirmed by our experimental results in Section 6.

3 PATHWIDTH AND TREEWIDTH

In this section, we briefly define the notions of pathwidth and treewidth. We refer to Cygan et al. [2015] for a more detailed treatment.

Tree decompositions [Robertson and Seymour 1984, 1986, 1990]. A *tree decomposition* of the graph $G = (V, E)$ is a tree $T = (\mathcal{B}, E_T)$, in which:

- Each node $b \in \mathcal{B}$ has a corresponding subset $V_b \subseteq V$ of vertices of G. To avoid confusion, we use the word *bag* to refer to the nodes of T and reserve the word *vertex* for vertices of G. Also, with a slight misuse of notation, we sometimes do not distinguish between b and V_b and use the word bag to refer to both of them.
- Every vertex appears in at least one bag, i.e. $\bigcup_{b\in\mathcal{B}} V_b = V$.
- For every edge $e = (u, v) \in E$, there is a bag b that contains both of its endpoints, i.e. $\{u, v\} \subseteq V_b$. We define $E_b \subseteq E$ as the set of edges whose both endpoints appear in V_b. So, we must have $\bigcup_{b\in\mathcal{B}} E_b = E$.
- Every vertex appears in a connected subtree of T. In other words, if the bag b_3 is on the unique path between bags b_1 and b_2 in T then $V_{b_1} \cap V_{b_2} \subseteq V_{b_3}$, i.e. every vertex that appears in both b_1 and b_2 has to appear in every bag that is on the path from b_1 to b_2.

Width [Robertson and Seymour 1986]. The *width* $w(T)$ of the tree decomposition T is simply the size of the largest bag minus 1[‡]. More formally,

$$w(T) := \max_{b\in\mathcal{B}} |V_b| - 1.$$

Treewidth [Robertson and Seymour 1986]. The *treewidth* of the graph G is the smallest width among all tree decompositions of G.

Intuition. Informally, a tree decomposition of width w is a covering of the graph G with small subsets of vertices, i.e. bags, of size at most $w + 1$, such that the bags themselves are connected in a tree-like manner. The smaller these bags are, the more tree-like our graph G looks. Hence, a graph with bounded treewidth is considered to be tree-like.

Example. Figure 3 shows a graph G (left) and a tree decomposition T of G (right). It also shows how the bags cover G. Note that the width of this tree decomposition is 2. There are no tree decompositions of G of smaller width, therefore the treewidth of G is also 2.

Path Decomposition [Robertson and Seymour 1983]. A tree decomposition $T = (\mathcal{B}, E_T)$ is called a *path decomposition* if the tree T is a path. As such, path decompositions are a very special case of tree decompositions.

[‡]The minus 1 is for historical reasons and does not majorly affect any of the results.

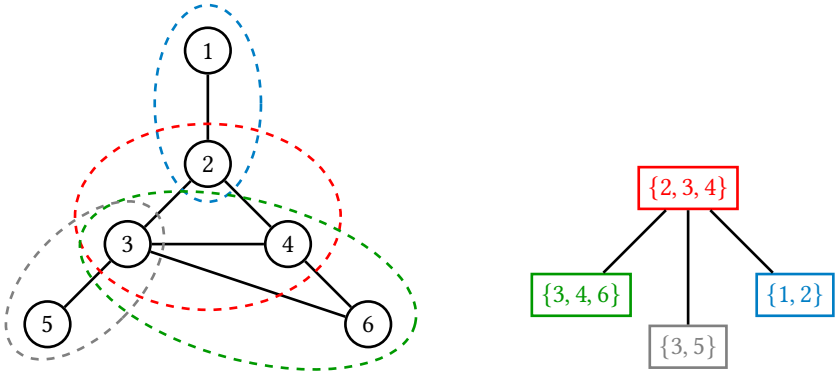

Fig. 3. A graph G (left) and a tree decomposition of G (right).

Fig. 4. A path decomposition for the graph of Figure 3.

Pathwidth [Robertson and Seymour 1983]. The *pathwidth* of G is the smallest width among its path decompositions.

Intuition. The intuition behind pathwidth is similar to treewidth, except that the decomposition is now a path. In other words, if G has pathwidth w, then we can cover it by bags of size at most $w + 1$, such that the bags themselves are connected in a path. Hence, a graph with small pathwidth w looks like a path with thickness $w + 1$.

Example. Figure 4 shows a path decomposition for the same graph G as in Figure 3. The bags cover the graph in the same way as in Figure 3. Note that the width of this path decomposition is 2. There are no path decompositions of G of smaller width, therefore the pathwidth of G is also 2.

Dynamic Programming. As mentioned above, the importance of tree and path decompositions comes from the fact that one can perform dynamic programming on them in essentially the same manner as dynamic programming on trees and paths, respectively. See Bodlaender [1988] and Arnborg [1985] for more details.

Merge Nodes. In a tree decomposition T, we say that a bag $b \in B$ is a *merge* bag, if it has more than one child or, equivalently, if its degree is larger than two. By definition, there are no merge bags in a path decomposition. As such, dynamic programming on path decompositions is usually simpler and more efficient than tree decompositions. This is because the runtime of dynamic programming algorithms over tree decompositions is often dominated by the the steps that process merge nodes. As an example, the extra complexity of handling merge nodes is the reason why the algorithm for μ-calculus model-checking in Obdrzálek [2003] is more efficient on a path decomposition than a tree decomposition. This also applies to Meybodi et al. [2022].

4 PATHWIDTH OF CONTROL-FLOW GRAPHS

In this section, we present a simple formula for obtaining a path decomposition of width $2 \cdot d$ for a given STRUCTURED program P of nesting depth d. In contrast to the complexity of the known algorithms for bounding treewidth, such as Thorup [1998] and Krause et al. [2020], our construction

for bounded pathwidth is surprisingly simple. This is despite the fact that pathwidth is a strictly stronger parameter than treewidth.

We remark that STRUCTURED does not contain function calls, and thus only represents single procedures. Our results will also only be applicable to the intra-procedural case. This is a shared limitation with the classical results on treewidth such as Thorup [1998]. Nevertheless, many standard analyses are modeled as intraprocedural problems, such as register allocation or LTL model-checking [Ferrara et al. 2005]. Even for inter-procedural analysis, methods for solving problems within a single procedure can help make the solutions more scalable. An example of this is data-flow analysis [Chatterjee et al. 2019c].

Notation. Let $\langle S \rangle$ be a path decomposition constructed for program fragment S and be of the form $\langle S \rangle = \langle b_1, b_2, \ldots, b_k \rangle$, where each b_i is a bag, i.e. a subset of vertices. Given a vertex $v \in V$, we define $\langle S \rangle + v$ as the path decomposition that is obtained from $\langle S \rangle$ by adding v to every bag. Formally,

$$\langle S \rangle + v = \langle b_1 \cup \{v\}, b_2 \cup \{v\}, \ldots, b_k \cup \{v\} \rangle.$$

Given two path decompositions $\langle S_1 \rangle$ and $\langle S_2 \rangle$, we define $\langle S_1 \rangle \cdot \langle S_2 \rangle$ to be their concatenation.

Path Decomposition of a Program. Given a program P, we construct a path decomposition for its control-flow graph using the rules below:

$$\begin{aligned}
\langle \texttt{program}\ S\ \texttt{margorp} \rangle &= \langle \{\texttt{program}, \lceil S \rceil\} \rangle \cdot (\langle S \rangle + \texttt{margorp}) \\
\langle S_1; S_2 \rangle &= \langle S_1 \rangle \cdot \langle \{\lfloor S_1 \rfloor, \lceil S_2 \rceil\} \rangle \cdot \langle S_2 \rangle && \textit{sequential composition} \\
\langle \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi} \rangle &= (\langle S_1 \rangle + \texttt{if} + \texttt{fi}) \cdot (\langle S_2 \rangle + \texttt{if} + \texttt{fi}) && \textit{conditional branching} \\
\langle \texttt{while}\ B\ \texttt{do}\ S\ \texttt{done} \rangle &= \langle S \rangle + \texttt{while} + \texttt{done} && \textit{while loop} \\
\langle \sigma \rangle &= \langle \{\sigma\} \rangle && \textit{other cases}
\end{aligned} \tag{4}$$

Intuition. Before we formally prove the correctness of our construction, let us first consider the intuition behind it. Our goal is to obtain a valid path decomposition of the program's CFG by a structurally recursive algorithm. Informally speaking, a path decomposition should (i) cover all vertices, (ii) cover all edges, and (iii) have every vertex in a connected interval. The second property turns out to be the trickiest here. The idea behind our construction is as follows: Consider a program of the form `while` B `do` S `done`. If we take a valid path decomposition $\langle S \rangle$ of the subprogram S, it already covers all the CFG edges that have both endpoints in S, as well as all vertices in S. Thus, we should modify the decomposition $\langle S \rangle$ by (a) adding the two new vertices corresponding to `while` and `done`, and (b) ensuring that every edge adjacent to these new vertices is covered. We achieve both (a) and (b) by simply adding the two new vertices to every bag of $\langle S \rangle$. This way, (a) is trivially achieved, and (b) holds because $\langle S \rangle$ already covered all the vertices of S. In other words, if there is an edge with one endpoint u in S and the other endpoint v in the new vertices, then u has already appeared in some bag in $\langle S \rangle$ and we are now adding v to every bag, including this one. So, the edge from u to v is covered. The case of `if` statements is similar.

Example. Figure 5 shows the path decomposition obtained by our algorithm for the program and control-flow graph of Figure 2. Each of the vertical segments represents a bag in the decomposition. For each vertex v of the CFG, there is a horizontal segment that shows which bags contain v. This path decomposition has a width of 8.

We now provide a formal proof of correctness and bounded width:

Theorem 4.1. *The construction above produces a valid path decomposition $\langle P \rangle$ of width $2 \cdot d$ for any given program P with nesting depth d.*

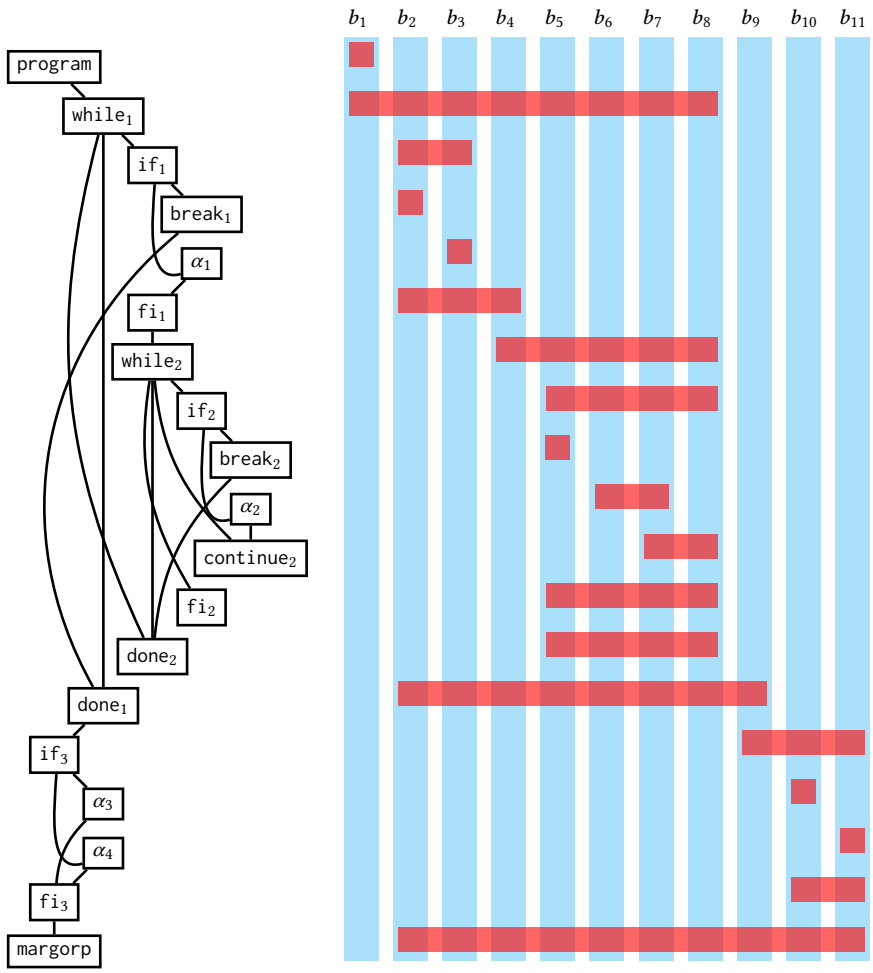

Fig. 5. A program and its CFG (left) and the path decomposition generated for this CFG (right). Edge directions have been omitted.

Proof. First, note that for every fragment S of our program P, the path decomposition $\langle S \rangle$ contains the entry point $\lceil S \rceil$ in its first bag and the exit point $\lfloor S \rfloor$ in its last bag. This property is preserved inductively in the construction above. Further, it is clear by construction that every vertex $v \in V$ of the control-flow graph G appears in some bag. More specifically, every vertex that corresponds to a statement in a fragment S appears in some bag of $\langle S \rangle$. This is also preserved inductively. We argue that for every edge $(u, v) \in E$ of the control-flow graph G there is a bag that contains both of its endpoints. We can do this by case-work:

- In $\langle$`program` S `margorp`$\rangle$, the edge from `program` to $\lceil S \rceil$ is covered in the first bag, which only contains these two vertices. Moreover, the edges from $\lfloor S \rfloor$ and `return` nodes to `margorp` are covered in $\langle S \rangle$ + `margorp` since $\langle S \rangle$ is a path decomposition that contains all vertices appearing in S and `margorp` is added to every one of its bags.
- In $\langle S_1; S_2 \rangle$ there is a dedicated bag $\{\lfloor S_1 \rfloor, \lceil S_2 \rceil\}$ that covers the edge from the exit node $\lfloor S_1 \rfloor$ to the entry node $\lceil S_2 \rceil$.

- In $\langle$`if` B `then` S_1 `else` S_2 `fi`$\rangle$, the edge from `if` to the entry node $\lceil S_1 \rceil$ is covered in $\langle S_1 \rangle$ + `if` + `fi`. So is the edge from the exit node $\lfloor S_1 \rfloor$ to `fi`. The case for S_2 is symmetric.
- In $\langle$`while` B `do` S `done`$\rangle$, we add the `while` and `done` vertices to every bag. So, any edge that connects any vertex in S to either `while` or `done` is covered. This includes all the edges originating from `break` and `continue` statements.

It is straightforward to check that every vertex appears in a connected segment (subpath) of our path decomposition. Finally, our construction starts with a path decomposition of width 0 for `return, continue, break` and atomic statements and its width increases by 2 when handling conditionals and loops and by 1 when handling a program. Note that the construction for sequential composition always has a width of at least 1. Hence, the final width of $\langle P \rangle$ is at most $2 \cdot d$ where d is the nesting depth of P. □

The theorem above proves that the control-flow graph of a program P with nesting depth d has a pathwidth of at most $2 \cdot d$. Moreover, simply following our recursive definition immediately leads to an algorithm for constructing such a path decomposition in $O(n \cdot d)$ where n is the size of the program, i.e. number of lines in the program. Hence, we have provided a linear-time algorithm that finds a path decomposition of constant width for any real-world program P that has bounded nesting depth. Finally, note that our decomposition has $O(n)$ bags.

Labeled* `break` *and* `continue` *statements. Our construction above handles labeled `break` and `continue` statements, such as those in Java. This is in contrast to the classical tree decomposition constructions in which labeled `break/continue` statements are as costly as arbitrary `goto`'s and break the bounded-width guarantee. Hence, our results are applicable to Java programs, too, and produce low-width path decompositions not only in practice, but also with a theoretical guarantee.

`elif`. We remark that our approach can easily be extended to handle multi-part conditionals, such as `elif` and `switch` statements without affecting the bound in Theorem 4.1. Specifically, we let

$$\langle \texttt{if } B_1 \texttt{ then } S_1 \texttt{ elif } B_2 \texttt{ then } S_2 \cdots \texttt{ elif } B_{n-1} \texttt{ then } S_{n-1} \texttt{ else } S_n \texttt{ fi } \rangle$$
$$= (\langle S_1 \rangle + \texttt{if} + \texttt{fi}) \cdot (\langle S_2 \rangle + \texttt{if} + \texttt{fi}) \cdots (\langle S_n \rangle + \texttt{if} + \texttt{fi}).$$

It is easy to verify that the construction above produces a valid path decomposition since there is no edge in the control-flow graph going from S_i to S_j when $i \neq j$. There are only edges from `if` to $\lceil S_i \rceil$ and from $\lfloor S_i \rfloor$ to `fi`. `switch` statements are a special case of `elif` and handled similarly.

***Handling* `goto`.** If we are given an unstructured program that has g `goto` statements in it, then we can simply add the destination of every `goto` to every bag of the path decomposition. Hence, a program with g `goto` statements is guaranteed to have a pathwidth of at most $2 \cdot d + g$. This is similar to the case of treewidth [Krause et al. 2020].

Coarser CFGs. If we contract an edge (u, v) in the graph G and merge its two endpoints into a new vertex w, the pathwidth cannot increase. More specifically, we can replace every occurrence of either u or v in the path decomposition with w and this can only decrease the size of the bags. Thus, our bounded-pathwidth result is also applicable to coarser CFGs in which every basic block is contracted into a single vertex.

Further Optimizations. While the construction in (4) is guaranteed to produce a path decomposition of constant width $2 \cdot d$, the proof of Theorem 4.1 shows that we can make some of the bags in the path decomposition smaller without violating any of the requirements. Even when the overall width is unaffected, it might be useful in practice to have smaller bags since many classical dynamic programming algorithms process each bag separately and the time they spend on it depends on the bag's size. Specifically, we can have the following optimizations:

- We currently compute $\langle \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi} \rangle$ as $(\langle S_1 \rangle + \texttt{if} + \texttt{fi}) \cdot (\langle S_2 \rangle + \texttt{if} + \texttt{fi})$. We can optimize this by adding `fi` not to every bag in $\langle S_1 \rangle$, but only to the last bag of $\langle S_1 \rangle$. This is because the only vertex in S_1 that can potentially have an edge to `fi` is $\lfloor S_1 \rfloor$ which appears in the last bag of $\langle S_1 \rangle$. Similarly, we do not need to add `if` to every bag in $\langle S_2 \rangle$ and it suffices to add it in the first bag. Finally, we can reorder the two parts and put $\langle S_2 \rangle$ before $\langle S_1 \rangle$ if it improves the bag sizes.
- We let $\langle \texttt{while}_l\ B\ \texttt{do}\ S\ \texttt{done}_l \rangle = \langle S \rangle + \texttt{while}_l + \texttt{done}_l$. As argued above, this is to ensure we cover every potential edge between any vertex in $\langle S \rangle$ and $\{\texttt{while}_l, \texttt{done}_l\}$. However, if there is no $\texttt{break}_l$ statement in $\langle S \rangle$, then $\texttt{done}_l$ can only have an edge from $\texttt{while}_l$. So, we do not need to add $\texttt{done}_l$ to every bag and can instead let $\langle \texttt{while}_l\ B\ \texttt{do}\ S\ \texttt{done}_l \rangle = \langle \{\texttt{while}_l, \texttt{done}_l\} \rangle \cdot (\langle S \rangle + \texttt{while}_l)$, i.e. we add a single bag that only contains the two new vertices and then add only one of them to every other bag.

5 AN EFFICIENT PATHWIDTH-BASED ALGORITHM FOR REGISTER ALLOCATION

In this section, we consider the classical problem of register allocation as formalized in Chaitin et al. [1981] as a motivating example for using pathwidth as a parameter. We provide a linear-time algorithm based on the bounded-pathwidth assumption and show that our algorithm is much more efficient than the treewidth-based algorithm of Bodlaender et al. [1998]. Note that our main contribution is the pathwidth bound itself. We use register allocation as a proof-of-concept to show that the pathwidth bound leads to huge practical improvements. This was a natural choice since it is also the main motivating problem for the treewidth bound [Thorup 1998].

Register Allocation [Chaitin et al. 1981]. In modern programming languages, the programmer can define as many variables as they wish. However, the compiler has to decide which variables to store in the registers of the processor and which to store on the main memory (RAM). Accessing registers is significantly faster than the main memory, but the number of registers is often very limited. Given a program P, and an integer r, the register allocation problem asks whether it is possible to map the variables in P to r registers such that there is no need to access the main memory (no spilling). More specifically, if two variables x_1 and x_2 might be alive at the same time, then they interfere with each other and should be put in different registers. A variable is alive at a certain point of the program if it has been assigned a value and this value might be used in the future.

Graph Formulation. Register allocation is classically formulated as a graph coloring problem [Chaitin et al. 1981]. Consider the control-flow graph G of the program P. The lifetime of every variable x is a connected subset of vertices of G that can be computed in linear time using a traditional gen-kill data-flow analysis [Kildall 1973]. Variables whose lifetimes intersect must be put in different registers. Hence, we can reformulate the register allocation problem as the graph coloring problem on the so-called *interference graph* H. The graph H has one vertex corresponding to each variable in P and an edge between x_1 and x_2 if their lifetimes have a non-empty intersection. It is now clear that a valid register allocation map is basically a valid coloring of H with r colors, such that the endpoints of each edge have different colors. Thus, we have a reduction from register allocation to graph coloring. It turns out this reduction works in the other direction, too, i.e. for any given graph H, one can find a G and a family of connected subgraphs of G whose interference graph is H. Thus, register allocation inherits all the classical hardness results known for graph coloring [Chaitin et al. 1981]. Specifically, the problem is NP-hard even for 3 registers/colors.

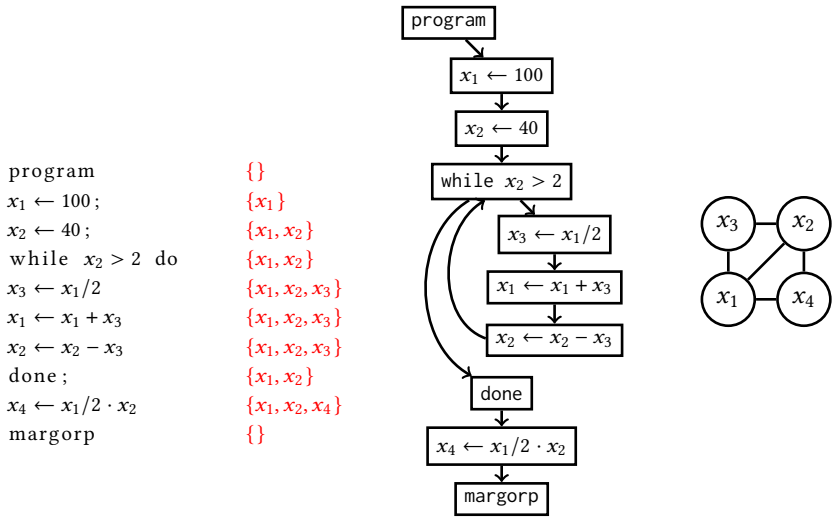

Fig. 6. A STRUCTURED program (left), its control-flow graph G (middle) and its interference graph H (right).

Connected Lifetimes. In the standard formulations of register allocation, the lifetime of every variable is assumed to be a *connected* subgraph of G. If a variable's lifetime has several connected components, we consider each component as a separate variable. This is a standard technique [Chaitin et al. 1981] which is semantics-preserving. Additionally, there is no downside to assigning different components of a variable's lifetime, which are independent of each other, to different registers. This cannot increase, but might decrease, the number of registers required.

Example. Figure 6 shows a structured program, its control-flow graph and the interference graph of its four variables. The set of live variables at each line is shown in red. Note that x_3 is only used within the loop, whereas x_4 is only used at the end of the program, so the lifetime of x_3 is disjoint from x_4 and there is no edge between them in the interference graph.

Pathwidth of the Interference Graph. Based on the formulation above and Theorem 4.1, we can assume that we are given a CFG G with bounded pathwidth $w \leq 2 \cdot d$, and a graph H whose every vertex corresponds to a *connected* subset of vertices of G, and there is an edge between two vertices in H iff their corresponding subgraphs intersect in G. Our goal is to decide whether H can be colored with r colors. Our first step is the following lemma:

Lemma 5.1. *If H can be colored with r colors, then the pathwidth of H is at most $w^* := (w+1) \cdot r - 1$, where w is the pathwidth of G.*

Proof. Consider a vertex v of G. At most r subgraphs corresponding to vertices in H can include v. Otherwise, there would be clique of size more than r in H and H would not be r-colorable. Take an optimal path decomposition of width w of G and replace every vertex v in every bag by the set of vertices of H whose subgraph contains v. It is straightforward to check that this yields a valid path decomposition of H. There are at most $(w+1) \cdot r$ vertices in each bag of this path decomposition, so its width is at most $(w+1) \cdot r - 1$. By definition, this is also an upper-bound on the pathwidth of H. □

Note that the lemma above is also constructive and combining it with our algorithm of Section 4 for obtaining a path decomposition of G, we can get a path decomposition of H of width $w^* \leq$

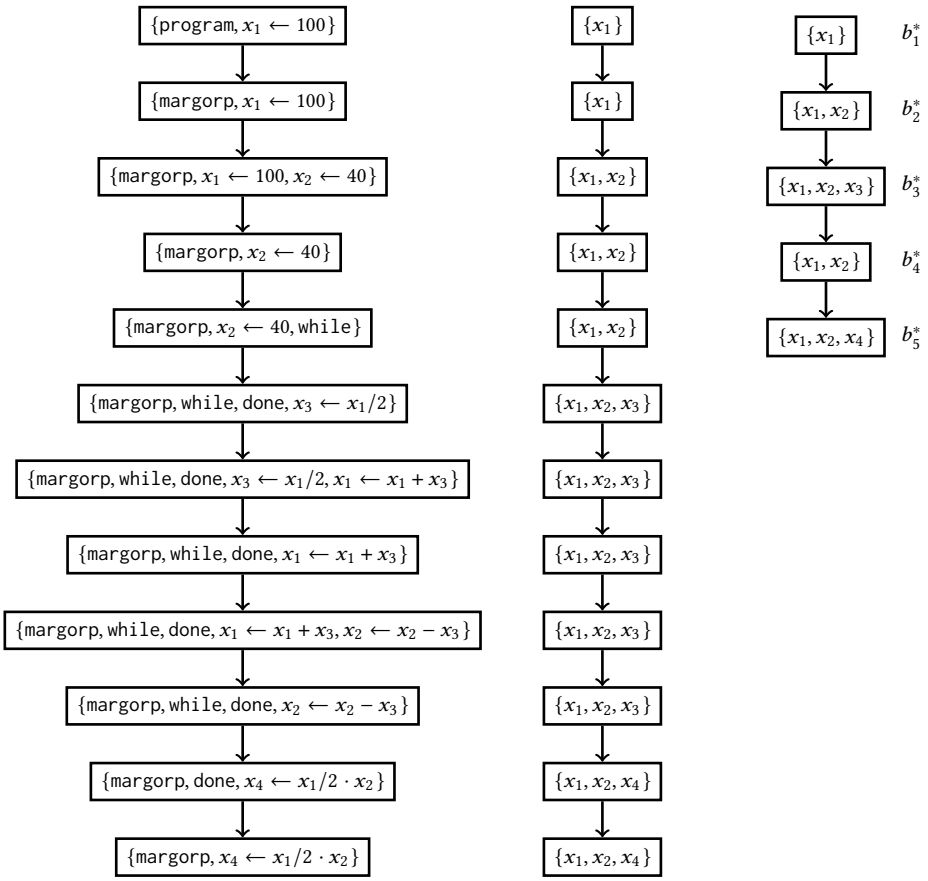

Fig. 7. A path decomposition $\langle P \rangle$ of the control-flow graph G of the program P in Figure 6 (left), a path decomposition $\langle H \rangle$ of the interference graph H, produced by substituting each line with the variables that are live at that line (middle), and the simplified path decomposition $\langle H \rangle^*$ (right).

$(w + 1) \cdot r - 1 = (2 \cdot d + 1) \cdot r - 1$ in linear time with respect to the size of the program. More specifically, we first apply the construction in the proof above to obtain a path decomposition of width at most w^* for H. If this construction fails, then the lemma guarantees that r registers are not enough and thus the answer to the register allocation instance is negative. Otherwise, the construction provides us a path decomposition of H with small width w^*, which we can use for dynamic programming (as shown further below).

Example. Figure 7 shows how a path decomposition of the interference graph H (middle) is obtained from a path decomposition of the CFG G (left). We then further simplify this decomposition to obtain the decomposition $\langle H \rangle^*$ (right).

Graph Coloring in Bounded Pathwidth. Now that we know the graph H has bounded pathwidth w^*, our problem is reduced to coloring a graph of bounded pathwidth with a constant number r of colors. We apply a standard dynamic programming technique to obtain an algorithm with runtime

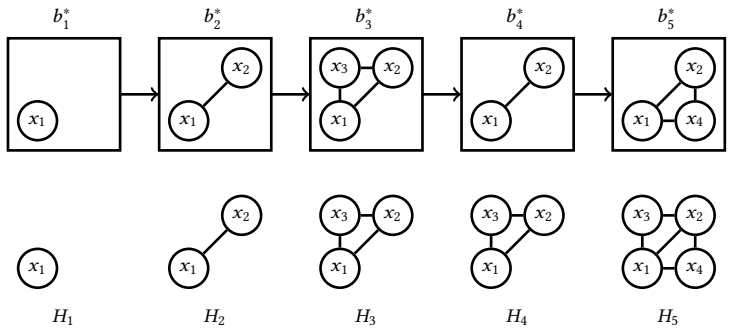

Fig. 8. The vertices and edges appearing in every bag of the path decomposition $\langle H \rangle^*$ of Figure 7 (top) and the canonical subgraphs (bottom).

$O(n \cdot w^* \cdot r^{w^*+1})$. Let the obtained path decomposition of H be $\langle H \rangle = \langle b_1, b_2, \ldots, b_n \rangle$[§]. We first preprocess and simplify $\langle H \rangle$ to remove repeated bags and make sure that for every index i, we have $V_{b_i} \Delta V_{b_{i+1}} = 1$. In other words, we would like to ensure that every bag differs from its neighbors in at most one vertex. See Figure 7 (right) as an example. This can be done in time $O(n \cdot w^*)$ by adding intermediate bags between any pair of neighboring bags that differ by two or more vertices. As a result, we get a new tree decomposition $\langle H \rangle^* = \langle b_1^*, b_2^*, \ldots, b_t^* \rangle$ where $t \in O(n)$. We note that $t \in O(n)$ since every bag differs with each of its neighbors in at most one vertex. In other words, every b_{i+1}^* either introduces a new vertex that was not present in b_i^* or forgets a vertex that existed in b_i^*. On the other hand, every vertex appears in a connected subpath and is thus introduced and forgotten exactly once. Hence, the number of bags is at most twice the number of vertices.

Canonical Subgraphs. We define the i-th *canonical subgraph* H_i of H as the part of H that corresponds to $\langle b_1^*, \ldots, b_i^* \rangle$. More formally, we have $H_i := (V_i, E_i)$ where $V_i := \bigcup_{j=1}^{i} V_{b_j^*}$ is the set of vertices that appear in the first i bags and $E_i := \bigcup_{j=1}^{i} E_{b_j^*}$ is the set of edges appearing in these bags.

Example. Figure 8 shows the path decomposition $\langle H \rangle^*$ of Figure 7 with some extra information. The top portion depicts the vertices and edges appearing in every bag b_i^*. The bottom portion depicts the canonical subgraphs H_i.

Dynamic Programming. For every index $1 \leq i \leq t$ and every partial coloring map $c : V_{b_i^*} \to \{1, 2, \ldots, r\}$ that assigns a color to every vertex in the bag b_i^*, we define a boolean dynamic programming variable $\mathsf{dp}[i, c]$. At the end of its computation, $\mathsf{dp}[i, c]$ should satisfy the following invariant:

$$\mathsf{dp}[i, c] = \begin{cases} 1 & \text{if there is a valid coloring of } H_i \text{ with } r \text{ colors in which } V_{b_i^*} \text{ is colored according to } c \\ 0 & \text{otherwise} \end{cases}$$

Computing the* dp *Values. We compute our dp values from left to right, i.e. in the order of increasing i.

[§]We are using n both for the number of vertices of G and H. This is justified by the fact that a program with n statements can have at most n variable declarations and hence at most n variables. So, there is no need to distinguish between the number of variables and lines of code. Similarly, our algorithm in Section 4 produces a path decomposition with linearly many bags, so we can assume the number of bags is also n.

- At $i = 1$, the only vertices in H_1 are those that appear in the bag b_1^*, i.e. $V_{b_1^*}$. Hence, $\mathsf{dp}[i, c] = 1$ iff c is a valid coloring of $V_{b_1^*}$. This can be checked by simply taking any edge in $E_{b_1^*}$ and verifying that c assigns a different color to its two endpoints.
- If $V_{b_i^*}$ has a vertex u that is not in $V_{b_{i-1}^*}$, i.e. $V_{b_i^*} = V_{b_{i-1}^*} \sqcup \{u\}$, then the only vertices that can potentially be neighbors of u in H_i are the other vertices in $V_{b_i^*}$. This is because every vertex appears in a contiguous subpath of the path decomposition. Hence, to compute $\mathsf{dp}[i, c]$, we first check the edges between u and its neighbors in $V_{b_i^*}$ to ensure that c assigns different colors to their endpoints. If not, we set $\mathsf{dp}[i, c] = 0$. If all the checks pass, we set $\mathsf{dp}[i, c] = \mathsf{dp}[i, c_{|V_{b_{i-1}^*}}]$. This is because c is already fixing the colors in $V_{b_i^*}$ and $V_{b_{i-1}^*} \subseteq V_{b_i^*}$, so all the colors in $V_{b_{i-1}^*}$ are also fixed by c.

  Example. The bag b_3^* in Figure 8 contains x_3 which was not included in b_2^*. If c is a coloring that assigns the same color to both x_3 and x_2 or both x_3 and x_1, then we have $\mathsf{dp}[3, c] = 0$ since c is an invalid coloring. Otherwise, we simply use the same colors as in c and set $\mathsf{dp}[3, c] = \mathsf{dp}[2, \{(x_1, c(x_1)), (x_2, c(x_2))\}]$.

- Finally, if $V_{b_i^*}$ has one vertex fewer than $V_{b_{i-1}^*}$, i.e. if $V_{b_i^*} = V_{b_{i-1}^*} \setminus \{u\}$, then we again have $H_i = H_{i-1}$. The only problem in computing $\mathsf{dp}[i, c]$ is that c is only assigning colors to $V_{b_i^*}$ and not to u. Hence, we have to try every possible color for u. In other words, we set

$$\mathsf{dp}[i, c] = \bigvee_{q=1}^{r} \mathsf{dp}\left[i - 1, c \cup \{(u, q)\}\right].$$

  Example. The bag b_4^* in Figure 8 has one vertex fewer than b_3^*, i.e. it lacks the vertex x_3. However, note that $H_4 = H_3$. Let $c = \{(x_1, 1), (x_2, 2)\}$ be the coloring that assigns color 1 to x_1 and color 2 to x_2. We simply iterate over every possible color q for x_3 and take the disjunction of the values of $\mathsf{dp}[3, c \cup \{(3, q)\}]$.

Final Answer. Our final answer to the coloring problem is

$$\bigvee_{c: V_{b_t^*} \to \{1,2,\ldots,r\}} \mathsf{dp}[t, c].$$

This is because we have $H_t = H$ and any coloring of the entire graph H has to conform with a partial coloring c at bag b_t^*.

Runtime Analysis. In the dynamic programming algorithm above, we defined a total of $O(t \cdot r^{w^*})$ dp variables. The time we spent for computing each variable is $O(w^* + r)$, except that the variables in the first bag take $O(w^{*2})$ each. The final answer is computed in $O(r^{w^*})$, trying every possible partial coloring c. So, our overall runtime is $O(t \cdot w^* \cdot r^{w^*+1})$. Recall that $t \in O(n)$, i.e. a nice path decomposition has a linear number of bags, regardless of the pathwidth. So, our overall runtime for the coloring algorithm is $O(n \cdot w^* \cdot r^{w^*+1})$.

Theorem 5.2. *Given a program P with nesting depth d and pathwidth $w \leq 2 \cdot d$, the algorithm above decides whether r registers are sufficient in $O(n \cdot w \cdot r^{w \cdot r + r + 1})$.*

Proof. Based on Lemma 5.1 and Theorem 4.1, we have $w^* \leq (w + 1) \cdot r - 1$. □

Note that, in comparison, the treewidth-based algorithm of Bodlaender et al. [1998] for the same problem takes $O(n \cdot r^{2 \cdot \mathsf{tw} \cdot r + 2 \cdot r})$ time. While this is still linear in terms of the program size n, our dependence on the pathwidth (instead of treewidth) and r is exponentially better than the treewidth-based algorithm. This has significant practical ramifications as shown in Section 6 below.

6 EXPERIMENTAL RESULTS

Implementation. We implemented (i) our pathwidth-based register allocation algorithm of Section 5 and (ii) the treewidth-based algorithm of Bodlaender et al. [1998] for the same problem[¶], and integrated them within the Small Device C Compiler (SDCC) [Dutta 2000; Dutta et al. 2003]. Our implementations are in C++. We used SDCC to obtain the control-flow graphs, lifetimes of the variables, and also tree decompositions of control-flow graphs. SDCC uses a modified and highly-optimized variant of the algorithm of Thorup [1998] to obtain the tree decompositions. See Krause et al. [2020] for details. The choice of SDCC is due to the fact that both our algorithm and Bodlaender et al. [1998]'s have an exponential dependence on the number of registers. Thus, the main use-case is when the number of registers is small, which is naturally the case in embedded systems. To the best of our knowledge, SDCC is one of the most commonly-used C compilers for such programs. Solving register allocation with a large number of registers remains an open problem

Computing Path Decompositions. To obtain path decompositions, we used our construction from Section 4 with minor heuristic optimizations. Specifically, if a vertex v appears in a bag b, but removing it from b keeps the decomposition valid, we will remove this vertex and simplify our decomposition, potentially also reducing its width. Similarly, if a bag is a subset of one of its neighboring bags, then it does not contribute anything to the decomposition and can be safely discarded. This is especially helpful for removing bags from the beginning and end of the path decomposition. These heuristics can be applied in linear time to reduce the number of bags in the decomposition. We first apply these heuristics and then generate the nice path decomposition. Although the process of making a nice tree decomposition might add back some of the bags that were removed by the heuristics, it guarantees that the final decomposition is nice so that we can apply the dynamic programming algorithm of Section 5 and that the path decomposition has $O(n)$ bags so that the desired time complexity is achieved. We apply the exact same heuristics in our implementation of Bodlaender et al. [1998], as well.

Machine. Our experiments were performed on an Intel i9-12900HK (3.8 GHz) machine running NixOS Linux with 32 GB of RAM.

Benchmarks and Experimental Setup. We used the programs in the SDCC regression test suite as our benchmarks. These benchmarks are real-world embedded programs. In embedded environments, register allocation has a significant effect on the efficiency and avoiding spilling is a natural and important goal. Moreover, as these programs are meant to be executed in small embedded devices with few registers, they are the perfect real-world candidates for comparing pathwidth-based and treewidth-based algorithms for register allocation. For each function in our benchmark set, we computed a path decomposition and a tree decomposition as explained above, and found the least number of registers required to allocate all variables without spilling, using our algorithm of Section 5 and the treewidth-based algorithm of Bodlaender et al. [1998]. We limited the number of registers to 8 and the maximal memory usage per instance to 4 GB. We also set a time limit of 10 minutes per instance.

Width Distributions. For every value w, Figure 9 (left) shows the number of instances in which the width of the tree decomposition, resp. the width of the path decomposition, was w. The plots in Figure 9 (middle and right) show the distribution of widths based on instance size. As expected,

[¶]We had to implement Bodlaender et al. [1998]'s algorithm on our own, since there is no available implementation or tool support for it, presumably due to the fact that the treewidth-based algorithm is not practical and our pathwidth-based approach is the first to be able to handle real-world instances (in embedded environments).

tree decompositions generally have the same or smaller width than path decompositions. However, the difference is not significant and the vast majority of the benchmarks have a pathwidth of 4 or smaller. Moreover, in 4, 143 benchmarks, the tree decomposition and the path decomposition had the same width. There are only 10 benchmarks for which our algorithm obtains a path decomposition with a width of 7 or higher. These 10 benchmarks are not structured programs and have `goto` statements in their code.

Width	Instances	
	TD	PD
1	3276	3276
2	1228	849
3	55	238
4	0	119
5	0	42
6	1	26
7	0	3
8	0	3
11	0	1
12	0	1
14	0	1
17	0	1

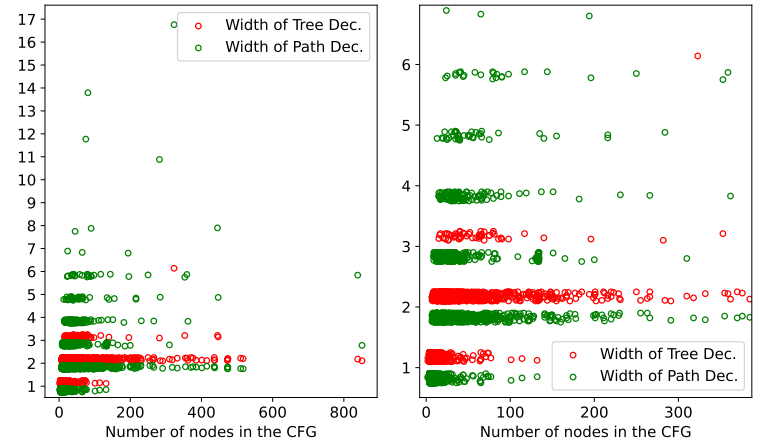

Fig. 9. Distribution of the widths of obtained tree decompositions and path decompositions. Each dot corresponds to one instance. The right plot is a zoomed-in version of the middle plot. We have added small perturbations to increase readability.

Number of Registers. Our benchmark set contained 4, 560 programs. Of these, 564 required more than 8 registers. On each benchmark, we only tried $1 \leq r \leq 8$, until we found the smallest r that sufficed or concluded that more than 8 registers are necessary. This is because both our algorithm and the treewidth-based algorithm of Bodlaender et al. [1998] have an exponential dependence on the number of registers and are only applicable to embedded systems with few registers. Figure 10 shows the distribution of the number of required registers based on instance size. Our benchmarks are spread-out well and cover all possible values of r. As expected, larger benchmarks generally tend to need more registers.

Failures. In our experiments, the algorithm of Bodlaender et al. [1998] ran out of memory on 53 benchmarks and timed out on 300 other benchmarks. In contrast, our algorithm ran out of memory on only 6 benchmarks and never timed out. There was no benchmark on which Bodlaender et al. [1998] succeeded and our algorithm failed.

Runtimes. The average runtime of Bodlaender et al. [1998] was 41.395 seconds per instance, whereas our average runtime was only 0.073 seconds. Hence, our algorithm is more than 500 times faster than its treewidth-based counterpart. Figure 11 shows the runtimes for each instance. Note that the y axis is in logarithmic scale. Hence, it is clear that our approach is several orders of magnitude faster and that Bodlaender et al. [1998] times out regularly even on small benchmarks with less than 100 lines of code.

Register Allocation Heuristics. We remark that SDCC has its own implementation of register allocation for a different formulation of the problem in which a cost is assigned to each spill. However, it often falls back on heuristics and is neither guaranteed to use the optimal number

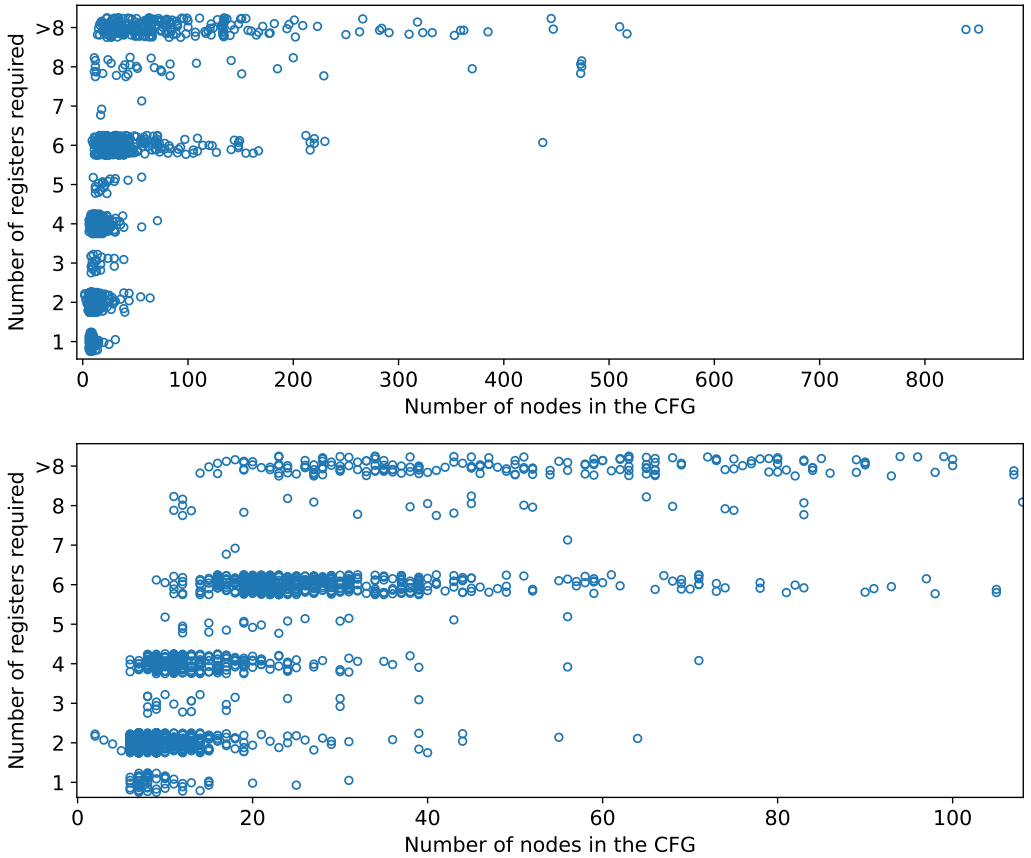

Fig. 10. Number of registers required for spill-free register allocation over each benchmark. The bottom figure is a zoomed-in version of the top figure. We have added small perturbations to increase readability.

of registers nor to avoid spills, and therefore has not been considered in our experiments. It is not straightforward to experimentally compare our register allocation algorithm with standard heuristics mainly because we solve the problem optimally, i.e. find the minimum number of registers for spill-free allocation, whereas the heuristics are (i) not guaranteed to use the minimum number of registers, and (ii) not guaranteed to be spill-free but instead try to minimize the number of spills, usually on a best-effort basis. The idea behind optimal register allocation is to spend more time in compilation to ensure the program itself runs as fast as possible. This is not achievable with heuristics and the previous treewidth-based algorithm was also too slow to be applied in practice, as evidenced by the many timeouts in Figure 11. Thus, we provide the first algorithm for optimal spill-free register allocation that can handle embedded benchmarks. We note that, if applied to the same input, the program compiled using heuristics will run less efficiently and need more registers. It is not possible to provide concrete numbers for the speedup since it inherently depends on the underlying architecture and the relative cost of spills in comparison with cache hits. However, there are results in the literature showing that optimal register allocation significantly reduces the compiled code's size, by almost 10% compared to the heuristics [Krause 2013b]. This is of particular importance in embedded systems with limited storage.

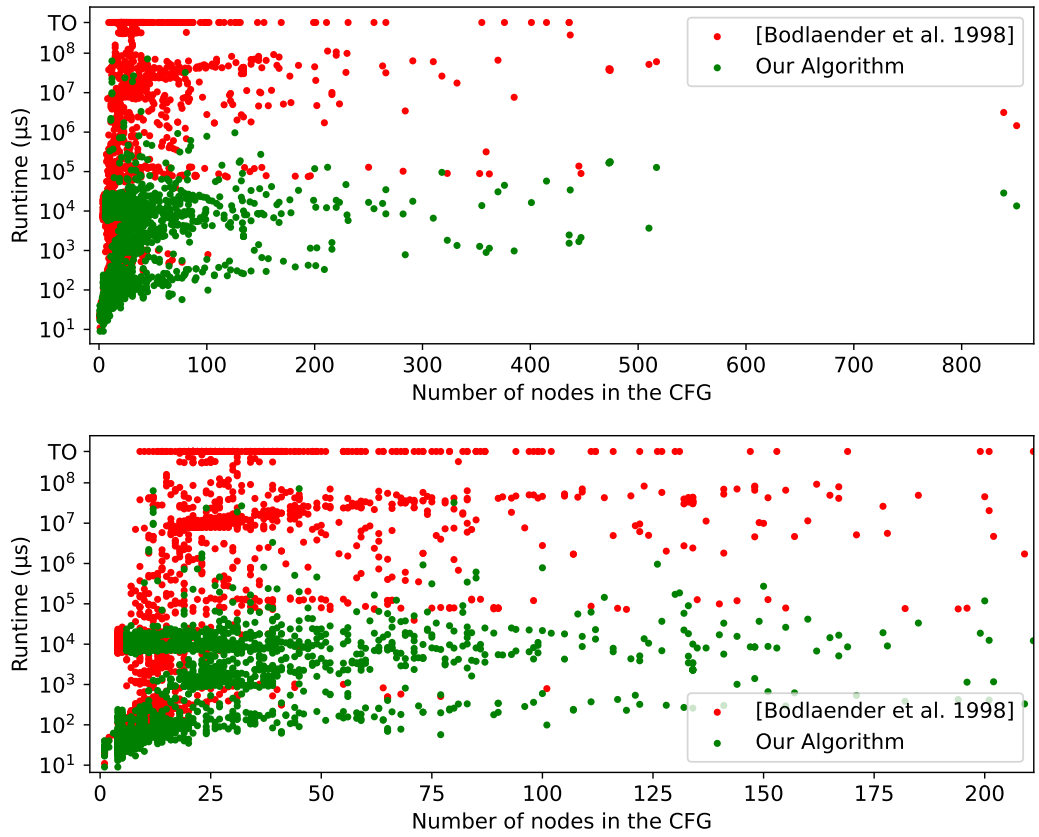

Fig. 11. Comparison of the runtimes of our algorithm and that of Bodlaender et al. [1998]. Note that the y axis is in logarithmic scale. The bottom figure is a zoomed-in version of the top figure.

Discussion. Our experimental results illustrate that our pathwidth-based algorithm for the problem of register allocation is significantly faster than the previous treewidth-based method. This is despite the fact that the pathwidth is always greater than or equal to the treewidth (by definition). Our algorithm has a runtime of $O(n \cdot \mathsf{pw} \cdot r^{\mathsf{pw} \cdot r + r + 1})$ whereas Bodlaender et al. [1998] takes $O(n \cdot r^{2 \cdot \mathsf{tw} \cdot r + 2 \cdot r})$ time. Hence, although our widths can be larger, this is more than compensated for by the exponentially better dependence on r and the widths. Moreover, in the vast majority of the benchmarks, the path decomposition and the tree decomposition have the same width. In this case, our algorithm is faster than the treewidth-based algorithm by a factor of $O(w^{-1} \cdot r^{w \cdot r + r - 1})$, where w is the width of the decompositions. Finally, since pathwidth-based algorithms are much simpler, the constant factor hidden by the O notation is expected to be smaller for our algorithm. These factors all contribute to the gained efficiency. As a bonus, our algorithm also uses much less space.

7 CONCLUSION

In this work, we proved that control-flow graphs (CFGs) of structured programs have a bounded pathwidth of at most $2 \cdot d$, where d is the nesting depth of the program. We argued that pathwidth should be adopted as the parameter of choice for compiler optimization and static analysis tasks that are reduced to graph problems over the CFGs. From a theoretical standpoint, pathwidth is a

stronger parameter and there are problems that are FPT with respect to pathwidth but not treewidth. Apart from being a stronger parameter, adopting pathwidth is beneficial in the sense that it often reduces the runtime complexity of the problems, e.g. in the case of μ-calculus model-checking, and also leads to simpler and more elegant algorithms. On the other hand, using the classical problem of register allocation as a motivating example, we showed that the benefits of using pathwidth are not limited to theory and are also apparent in practice. Thus, we invite the compiler optimization and static analysis communities to consider the bounded-pathwidth property when they design algorithms that exploit the sparsity of a CFG.

ACKNOWLEDGMENTS AND NOTES

The authors are grateful to the anonymous reviewers for detailed comments which significantly improved this work. The research was partially supported by Hong Kong Research Grants Council ECS Project 26208122. G.K. Conrado and C.K. Lam were supported by the Hong Kong PhD Fellowship Scheme (HKPFS). Following the norms of theoretical computer science, authors are listed in alphabetical order.

REFERENCES

Ali Ahmadi, Majid Daliri, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. 2022. Efficient approximations for cache-conscious data placement. In *PLDI*. 857–871.

C. Aiswarya. 2022. How treewidth helps in verification. *SIGLOG News* 9, 1 (2022), 6–21.

Frances E Allen. 1970. Control flow analysis. *SIGPLAN Notices* 5, 7 (1970), 1–19.

Stefan Arnborg. 1985. Efficient algorithms for combinatorial problems on graphs with bounded decomposability—a survey. *Numerical Mathematics* 25, 1 (1985), 1–23.

Ali Asadi, Krishnendu Chatterjee, Amir Kafshdar Goharshady, Kiarash Mohammadi, and Andreas Pavlogiannis. 2020. Faster Algorithms for Quantitative Analysis of MCs and MDPs with Small Treewidth. In *ATVA*, Vol. 12302. 253–270.

Christine Ausnit-Hood, Kent A Johnson, Robert G Pettit IV, and Steven B Opdahl. 1997. *Ada 95 Quality and Style*. Springer.

Michael Barr. 2009. *Embedded C Coding Standard*. Netrino.

Rémy Belmonte, Eun Jung Kim, Michael Lampis, Valia Mitsou, and Yota Otachi. 2022. Grundy distinguishes treewidth from pathwidth. *SIAM Journal on Discrete Mathematics* 36, 3 (2022), 1761–1787.

David Bernstein and Michael Rodeh. 1991. Global instruction scheduling for superscalar machines. In *PLDI*. 241–255.

Gabriel Hjort Blindell, Roberto Castañeda Lozano, Mats Carlsson, and Christian Schulte. 2015. Modeling Universal Instruction Selection. In *CP*, Vol. 9255. 609–626.

Hans L Bodlaender. 1988. Dynamic programming on graphs with bounded treewidth. In *ICALP*. 105–118.

Hans L Bodlaender. 1997. Treewidth: Algorithmic techniques and results. In *MFCS*. 19–36.

Hans L Bodlaender. 1998. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science* 209, 1-2 (1998), 1–45.

Hans L. Bodlaender, Jens Gustedt, and Jan Arne Telle. 1998. Linear-Time Register Allocation for a Fixed Number of Registers. In *SODA*. 574–583.

Bernd Burgstaller, Johann Blieberger, and Bernhard Scholz. 2004. On the tree width of Ada programs. In *Ada-Europe*. 78–90.

Brad Calder, Chandra Krintz, Simmi John, and Todd M. Austin. 1998. Cache-Conscious Data Placement. In *ASPLOS*. 139–149.

David Callahan and Brian D. Koblenz. 1991. Register Allocation via Hierarchical Graph Coloring. In *PLDI*. 192–203.

Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. Register Allocation Via Coloring. *Comput. Lang.* 6, 1 (1981), 47–57.

Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. 2019a. The treewidth of smart contracts. In *SAC*. 400–408.

Krishnendu Chatterjee, Amir Kafshdar Goharshady, Prateesh Goyal, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2019b. Faster Algorithms for Dynamic Algebraic Queries in Basic RSMs with Constant Treewidth. *ACM Trans. Program. Lang. Syst.* 41, 4 (2019), 23:1–23:46.

Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2016. Algorithms for algebraic path properties in concurrent systems of constant treewidth components. In *POPL*. 733–747.

Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2020. Optimal and Perfectly Parallel Algorithms for On-demand Data-Flow Analysis. In *ESOP*. 112–140.

Krishnendu Chatterjee, Amir Kafshdar Goharshady, Nastaran Okati, and Andreas Pavlogiannis. 2019c. Efficient parameterized algorithms for data packing. In *POPL*. 53:1–53:28.

Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. 2017. JTDec: A Tool for Tree Decompositions in Soot. In *ATVA*, Vol. 10482. 59–66.

Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. 2018. Algorithms for Algebraic Path Properties in Concurrent Systems of Constant Treewidth Components. *ACM Trans. Program. Lang. Syst.* 40, 3 (2018), 9:1–9:43.

Krishnendu Chatterjee and Jakub Lacki. 2013. Faster Algorithms for Markov Decision Processes with Low Treewidth. In *CAV*, Vol. 8044. 543–558.

Giovanna Kobus Conrado, Amir Kafshdar Goharshady, Kerim Kochekov, Yun-Chen Tsai, and Ahmed Khaled Zaher. 2023. Exploiting the Sparseness of Control-flow and Call Graphs for Efficient and On-demand Algebraic Program Analysis. In *OOPSLA*.

Bruno Courcelle. 1990. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and computation* 85, 1 (1990), 12–75.

Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. 2015. *Parameterized algorithms*. Springer.

Sandeep Dutta. 2000. Anatomy of a Compiler: A Retargetable ANSI-C Compiler. *Circuit Cellar* 121 (2000), 5.

Sandeep Dutta, Daniel Drotos, Kevin Vigor, Johan Knol, Scott Dattalo, Karl Bongers, Bernhard Held, Frieder Ferlemann, Jesus Calvino-Fraga, Borut Razem, et al. 2003. Small device C compiler. http://sdcc.sourceforge.net

Andrea Ferrara, Guoqiang Pan, and Moshe Y Vardi. 2005. Treewidth in verification: Local vs. global. In *LPAR*. 489–503.

Fedor V Fomin, Daniel Lokshtanov, Saket Saurabh, Michał Pilipczuk, and Marcin Wrochna. 2018. Fully polynomial-time parameterized computations for graphs and matrices of low treewidth. *TALG* 14, 3 (2018), 1–45.

Amir Kafshdar Goharshady and Fatemeh Mohammadi. 2020. An efficient algorithm for computing network reliability in small treewidth. *Reliab. Eng. Syst. Saf.* 193 (2020), 106665.

Amir Kafshdar Goharshady and Ahmed Khaled Zaher. 2023. Efficient Interprocedural Data-Flow Analysis Using Treedepth and Treewidth. In *VMCAI*. 177–202.

Jens Gustedt, Ole A Mæhle, and Jan Arne Telle. 2002. The treewidth of Java programs. In *ALENEX*. 86–97.

Daniel J Harvey and David R Wood. 2017. Parameters tied to treewidth. *Journal of Graph Theory* 84, 4 (2017), 364–385.

Michael Hind, Michael G. Burke, Paul R. Carini, and Jong-Deok Choi. 1999. Interprocedural pointer alias analysis. *TOPLAS* 21, 4 (1999), 848–894.

Taisuke Izumi, Naoki Kitamura, Takamasa Naruse, and Gregory Schwartzman. 2022. Fully Polynomial-Time Distributed Computation in Low-Treewidth Graphs. In *SPAA*. 11–22.

Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. 2017. *Data flow analysis: theory and practice*. CRC.

Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *POPL*. 194–206.

Zachary Kincaid, Thomas Reps, and John Cyphert. 2021. Algebraic Program Analysis. In *CAV*. 46–83.

Peter King, Patrick Naughton, Mike DeMoney, Jonni Kanerva, Kathy Walrath, and Scott Hommel. 1999. Code Conventions for the Java Programming Language.

David Ryan Koes. 2009. *Towards a more principled compiler: Register allocation and instruction selection revisited*. Ph.D. Dissertation. Carnegie Mellon University.

Ephraim Korach and Nir Solel. 1993. Tree-width, path-width, and cutwidth. *Discrete Applied Mathematics* 43, 1 (1993), 97–101.

Philipp Klaus Krause. 2013a. Optimal placement of bank selection instructions in polynomial time. In *M-SCOPES*. 23–30.

Philipp Klaus Krause. 2013b. Optimal register allocation in polynomial time. In *CC*. 1–20.

Philipp Klaus Krause. 2014. The complexity of register allocation. *Discret. Appl. Math.* 168 (2014), 51–59.

Philipp Klaus Krause. 2021. LOSPRE in linear time. In *SCOPES*. 35–41.

Philipp Klaus Krause, Lukas Larisch, and Felix Salfelder. 2020. The tree-width of C. *Discrete Applied Mathematics* 278 (2020), 136–152.

Rahman Lavaee. 2016. The hardness of data packing. In *POPL*. 232–242.

David Marca. 1981. Some Pascal style guidelines. *SIGPLAN Notices* 16, 4 (1981), 70–80.

Steve McConnell. 2004. *Code complete*. Pearson.

Mohsen Alambardar Meybodi, Amir Kafshdar Goharshady, Mohammad Reza Hooshmandasl, and Ali Shakiba. 2022. Optimal Mining: Maximizing Bitcoin Miners' Revenues from Transaction Fees. In *Blockchain*. 266–273.

David Monniaux and Thomas Martin Gawlitza. 2012. Invariant generation through strategy iteration in succinctly represented control flow graphs. *LMCS* 8 (2012).

Eugene M Myers. 1981. A precise inter-procedural data flow algorithm. In *POPL*. 219–230.

Sigve Nordgaard and Jan Christian Meyer. 2020. Feasibility of Optimizations Requiring Bounded Treewidth in a Data Flow Centric Intermediate Representation. (2020).

Jan Obdrzálek. 2003. Fast Mu-Calculus Model Checking when Tree-Width Is Bounded. In *CAV*, Vol. 2725. 80–92.

Mizuhito Ogawa, Zhenjiang Hu, and Isao Sasano. 2003a. Iterative-free program analysis. In *ICFP*. 111–123.

Mizuhito Ogawa, Zhenjiang Hu, Isao Sasano, and Masato Takeichi. 2003b. *Catamorphic Approach to Program Analysis*.

Erez Petrank and Dror Rawitz. 2002. The hardness of cache conscious data placement. In *POPL*. 101–112.

Thomas W. Reps. 1995. Shape Analysis as a Generalized Path Problem. In *PEPM*. 1–11.

Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995a. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*. 49–61.

Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995b. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*. 49–61.

Neil Robertson and Paul D Seymour. 1983. Graph minors. I. Excluding a forest. *Journal of Combinatorial Theory, Series B* 35, 1 (1983), 39–61.

Neil Robertson and Paul D Seymour. 1984. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B* 36, 1 (1984), 49–64.

Neil Robertson and Paul D. Seymour. 1986. Graph minors. II. Algorithmic aspects of tree-width. *Journal of algorithms* 7, 3 (1986), 309–322.

Neil Robertson and Paul D Seymour. 1990. Graph minors. IV. Tree-width and well-quasi-ordering. *Journal of Combinatorial Theory, Series B* 48, 2 (1990), 227–254.

Sriram Sankaranarayanan. 2020. Reachability analysis using message passing over tree decompositions. In *CAV*. 604–628.

Micha Sharir, Amir Pnueli, et al. 1978. *Two approaches to interprocedural data flow analysis*. Courant Institute of Mathematical Sciences.

Yannis Smaragdakis, George Balatsouras, et al. 2015. Pointer analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69.

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S Foster. 2013. Template-based program verification and program synthesis. *Software Tools for Technology Transfer* 15, 5 (2013), 497–518.

Herb Sutter and Andrei Alexandrescu. 2004. *C++ coding standards: 101 rules, guidelines, and best practices.* Pearson.

Mikkel Thorup. 1998. All structured programs have small tree width and good register allocation. *Information and Computation* 142, 2 (1998), 159–181.

Reinhard Wilhelm, Mooly Sagiv, and Thomas Reps. 2000. Shape analysis. In *CC*. 1–17.

Edward Yourdon. 1985. *Managing the Structured Techniques: Strategies for Software Development.* Prentice Hall.

Shaowei Zhu and Zachary Kincaid. 2021. Termination analysis without the tears. In *PLDI*. 1296–1311.