The Fine-Grained and Parallel Complexity of Andersen's Pointer Analysis

34

ANDERS ALNOR MATHIASEN, Aarhus University, Denmark

ANDREAS PAVLOGIANNIS, Aarhus University, Denmark

Pointer analysis is one of the fundamental problems in static program analysis. Given a set of pointers, the task is to produce a useful over-approximation of the memory locations that each pointer may point-to at runtime. The most common formulation is Andersen's Pointer Analysis (APA), defined as an inclusion-based set of m pointer constraints over a set of n pointers. Scalability is extremely important, as points-to information is a prerequisite to many other components in the static-analysis pipeline. Existing algorithms solve APA in $O(n^2 \cdot m)$ time, while it has been conjectured that the problem has no truly sub-cubic algorithm, with a proof so far having remained elusive. It is also well-known that APA can be solved in $O(n^2)$ time under certain sparsity conditions that hold naturally in some settings. Besides these simple bounds, the complexity of the problem has remained poorly understood.

In this work we draw a rich fine-grained and parallel complexity landscape of APA, and present upper and lower bounds. First, we establish an $O(n^3)$ upper-bound for general APA, improving over $O(n^2 \cdot m)$ as $n = O(m)$. Second, we show that even *on-demand* APA ("may a *specific* pointer a point to a *specific* location b?") has an $\Omega(n^3)$ (combinatorial) lower bound under standard complexity-theoretic hypotheses. This formally establishes the long-conjectured "cubic bottleneck" of APA, and shows that our $O(n^3)$-time algorithm is optimal. Third, we show that under mild restrictions, APA is solvable in $\tilde{O}(n^\omega)$ time, where $\omega < 2.373$ is the matrix-multiplication exponent. It is believed that $\omega = 2 + o(1)$, in which case this bound becomes quadratic. Fourth, we show that even under such restrictions, even the on-demand problem has an $\Omega(n^2)$ lower bound under standard complexity-theoretic hypotheses, and hence our algorithm is optimal when $\omega = 2 + o(1)$. Fifth, we study the parallelizability of APA and establish lower and upper bounds: (i) in general, the problem is P-complete and hence unlikely parallelizable, whereas (ii) under mild restrictions, the problem is parallelizable. Our theoretical treatment formalizes several insights that can lead to practical improvements in the future.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**; • **Theory of computation** → *Theory and algorithms for application domains*; *Program analysis.*

Additional Key Words and Phrases: static pointer analysis, inclusion-based pointer analysis, fine-grained complexity, Dyck reachability

ACM Reference Format:

Anders Alnor Mathiasen and Andreas Pavlogiannis. 2021. The Fine-Grained and Parallel Complexity of Andersen's Pointer Analysis. *Proc. ACM Program. Lang.* 5, POPL, Article 34 (January 2021), 29 pages. https://doi.org/10.1145/3434315

Authors' addresses: Anders Alnor Mathiasen, Aarhus University, Aabogade 34, Aarhus, 8200, Denmark, au611509@uni.au.dk; Andreas Pavlogiannis, Aarhus University, Aabogade 34, Aarhus, 8200, Denmark, pavlogiannis@cs.au.dk.

2475-1421/2021/1-ART34
https://doi.org/10.1145/3434315

```
1   ...
2   *a=42;
3   *b=84;
4   c = *a;
5   //is c 42 or 84?
```

Type	Statement
1	$a = b$
2	$a = \&b$
3	$a = *b$
4	$*a = b$

Fig. 1. A program where analysis depends on aliasing (left) and the four types of statements in APA (right).

1 INTRODUCTION

Programs execute by allocating memory for storing data and manipulating pointers to that memory. Pointer analysis takes a static view of a program's heap and asks the question "given a pointer a, what are the memory locations that a may point-to at program runtime?" Such information is vital to almost all questions addressed by static analyses in general [Ghiya et al. 2001; Hind 2001], hence many static analyzers begin with some form of pointer analysis. In particular, for an analysis to be useful, it needs to be able to determine *aliasing*, i.e., whether two pointers may be pointing to the same memory location. For example, in the program of Figure 1, the value of c depends on whether a and b are aliases. Naturally, aliasing is decided by (implicitly or explicitly) computing whether the intersection of the points-to space of the two pointers is empty.

As usual in static analyses, points-to information can be modeled at various degrees of precision, which has consequences on the decidability and complexity of the problem. Flow-sensitive formulations, which take into account the order of execution of pointer-manipulation statements, are typically intractable, with results ranging from undecidability [Ramalingam 1994], to PSPACE-completeness [Chakaravarthy 2003] and NP-/co-NP-hardness [Landi and Ryder 1991]. In contrast, flow-insensitive formulations can be viewed as relational approaches that ignore the order of execution, and typically result in more tractable algorithmic problems. Another feature that affects complexity is the level of indirection (i.e., how many nested dereferences can occur in a single statement) [Horwitz 1997], and thus is typically kept small. Flow-insensitive analyses are faster and achieve remarkable precision in practice [Blackshear et al. 2011; Das et al. 2001; Shapiro and Horwitz 1997]. This sweet spot between efficiency and precision has made flow-insensitive analyses dominant over alternatives. Popular approaches in this domain are inclusion-based [Andersen 1994], equality-based [Steensgaard 1996] and unification-based [Das 2000]. We refer to [Smaragdakis and Balatsouras 2015] for an excellent exposition.

Andersen's pointer analysis. The most commonly used and actively studied formulation is Andersen's Pointer Analysis (APA) [Andersen 1994]. The input is a set of n pointers and m statements of the four types shown in Figure 1. The solution to the analysis is the least fixpoint of a set of inclusion constraints between the points-to sets of the pointers (see Section 2.1 for details). APA has been the subject of a truly huge body of work, ranging from adoptions to diverse programming languages [Jang and Choe 2009; Lyde et al. 2015; Sridharan and Fink 2009], extensions to incorporate various features (e.g., context/flow/field-sensitivity) [Hardekopf and Lin 2011; Hirzel et al. 2004; Pearce et al. 2004; Whaley and Lam 2002] and implementations in various frameworks [Wal 2003; Lhoták and Hendren 2003; Vallée-Rai et al. 1999], to name a few.

Complexity. The standard statement in the literature with regards to the complexity of APA is that it is cubic. However, the parameter (n or m, for n pointers and m statements) on which this cubic bound is expressed is often left unspecified, leading to a variety of statements. The standard expression is an $O(m^3)$ bound [Melski and Reps 2000; Møller and Schwartzbach 2018], by reducing

the problem to m inclusion set constraints [McAllester 1999]. Other works give a more refined bound of $O(n^2 \cdot m)$ [Kodumal and Aiken 2004; Pearce et al. 2004] which is an improvement over $O(m^3)$ as $n = O(m)$. Note that, in general, m can be as large as $\Theta(n^2)$, hence both types of statements result in worst-case dependency on n that is at least quartic, as already observed in [Kodumal and Aiken 2004]. Finally, in most literature, the core algorithm constructs incrementally the closure of a flow graph by introducing edges dynamically. However, the complexity analysis often ignores the cost for inserting edges. As already noted by others [Heintze and McAllester 1997; Sridharan and Fink 2009], the cost of edge insertion needs to be accounted for when analyzing the complexity. Under this consideration, the complexity of all these approaches is at least quartic in n.

The need for a faster algorithm is apparent from the long literature of heuristics [Aiken et al. 1997; Berndl et al. 2003; Dietrich et al. 2015; Fähndrich et al. 1998; Hardekopf and Lin 2007; Heintze and Tardieu 2001b; Pearce et al. 2004; Pek and Madhusudan 2014; Rountev and Chandra 2000; Su et al. 2000; Vedurada and Nandivada 2019; Xu et al. 2009]. Despite all efforts, no algorithmic breakthrough below the cubic bound has been made for over 25 years. In some cases, the complexity of APA can be reduced to quadratic [Sridharan and Fink 2009]. This reduction holds when the instances adhere to certain sparsity conditions, which hold naturally in some settings.

Exhaustive vs on-demand. One popular approach to reducing running time lies on the observation that we are typically interested in *on-demand* variants of the problem. That is, we want to decide whether a may point to b for a *given* pointer a and memory location b, rather than the *exhaustive* case that computes the points-to set of every pointer. Under this restriction, many techniques devise analysis algorithms that aim to solve on-demand APA faster [Chatterjee et al. 2018; Heintze and Tardieu 2001a; Lu et al. 2013; Sridharan et al. 2005; Sui and Xue 2016; Vedurada and Nandivada 2019; Zhang et al. 2013; Zheng and Rugina 2008]. On the theoretical side, it is an open question whether on-demand analysis has lower complexity than exhaustive analysis.

Lower bounds and cubic bottlenecks. Despite the complete lack of algorithmic improvements for APA for over 25 years, no lower bounds are known. The two basic observations for exhaustive APA are that (i) the output has size $\Theta(n^2)$, which leads to a trivial similar lower bound for running time, and (ii) the problem is at least as hard as computing the transitive closure of a graph [Sridharan and Fink 2009; Zhang 2020]. For the on-demand case, no lower bound is known. APA is often reduced to a specific framework of set constraints [Heintze 1992; Su et al. 2000], which is computationally equivalent to CFL-Reachability [Melski and Reps 2000]. Set constraints and CFL-Reachability are known to have cubic lower bounds [Heintze and McAllester 1997]. Unfortunately, these lower bounds do not imply any lower bound for APA as the reduction is only one way (i.e., *from* APA *to* set constraints). The recurrent encounter of cubic complexity is frequently referred to as the "cubic bottleneck in static analysis", though the bottleneck is only conjectured for APA, with a proof so far having remained elusive.

Parallelization. The demand for high-performance static analyses has lead various researchers to implement parallel solvers of APA [Blaß and Philippsen 2019; Liu et al. 2019; Mendez-Lojo et al. 2012; Méndez-Lojo et al. 2010; Su et al. 2014; Wang et al. 2017]. Despite their practical performance, the parallelizability of APA has remained open on the theoretical level. In contrast, the richer problem of set constraints is known to be non-parallelizable, via its reduction to CFL-Reachability [Melski and Reps 2000] which is known to be P-complete [Reps 1996].

Main contributions. In this work, we draw a rich fine-grained complexity landscape of Andersen's Pointer Analysis, by resolving open questions and improving existing bounds. We refer to Section 3

for a formal presentation of our main results as well as a discussion on their implications to the theory and practice of pointer analysis.

Consider as input an APA instance (A, S) of $n = |A|$ pointers and $m = |S|$ statements. Our main contributions are as follows.

(1) We show that Exhaustive APA is solvable in $O(n^3)$ time, regardless of m. To our knowledge, this gives the sharpest cubic bound on the analysis, as it holds even when $m = \Theta(n^2)$.
(2) We show that even On-demand APA does not have a (combinatorial) sub-cubic algorithm (i.e., with complexity $O(n^{3-\epsilon}$ for some fixed $\epsilon > 0$) under the combinatorial BMM hypothesis. This formally proves the long-conjectured cubic bottleneck for APA.
(3) We consider a bounded version of Exhaustive APA where points-to information is witnessed by bounding the execution of type-4 (Figure 1) statements by a poly-logarithmic bound. We show that bounded Exhaustive APA is solvable in $\tilde{O}(n^\omega)$ time, where $\tilde{O}$ hides poly-logarithmic factors and ω is the matrix-multiplication exponent. It is known that $\omega < 2.373$ [Le Gall 2014], hence our algorithm is sub-cubic.
(4) It is believed that $\omega = 2 + o(1)$, in which case our previous bound becomes nearly quadratic. We complement this result by showing that even On-demand APA with witnesses that are logarithmically bounded (i.e., a simpler problem than that in Item 3) does not have a sub-quadratic algorithm (i.e., with complexity $O(n^{2-\epsilon})$ for some fixed $\epsilon > 0$) under the Orthogonal Vectors hypothesis [Williams 2019]. Hence, our algorithm for Item 3 is optimal when $\omega = 2+o(1)$.
(5) We show that APA is P-complete, and hence unlikely parallelizable. On the other hand, we show that bounding APA as in Item 4 is in NC, and hence highly parallelizable.

Technical contributions. Our main theoretical results rely on a number of technical novelties that might be of independent interest.

(1) Virtually all existing algorithms for APA represent the analysis as a *flow-graph* that captures inclusion constraints between pointers. In contrast, we develop a *Dyck-graph* representation over the Dyck language of 1 parenthesis type $\mathcal{D}_1$, which allows us to develop new insights for the problem, and establish upper and lower bounds.
(2) We show that Dyck-Reachability over $\mathcal{D}_1$ can be solved in time $\tilde{O}(n^\omega)$, where ω is the matrix-multiplication exponent, by a purely combinatorial reduction of the problem to $O(\log^2 n)$ matrix-multiplications.
(3) Our lower bounds are based on *fine-grained complexity*, an emerging field in complexity theory that establishes relationships between problems in P. We believe that this field can have an important role in understanding and optimizing static program analyses. Our work makes some of the first steps in this direction.

2 PRELIMINARIES

In this section we give a formal presentation of Andersen's pointer analysis and develop some general notation. We also define Dyck graphs and show how reachability relationships in such graphs can be used to represent points-to relationships between pointers. Finally, we present the main theorems of this paper. Given a number $n \in \mathbb{N}$, we denote by $[n]$ the set $\{1, 2, \ldots, n\}$.

2.1 Andersen's Pointer Analysis

We begin with giving the formal definition of Andersen's pointer analysis, as well as a bounded version of the problem.

Table 1. The four types of statements of APA, the inclusion constraints they generate and the associated operational semantics.

Type	Statement	Inclusion Constraint	Operational Semantics
1	$a = b$	$[\![b]\!] \subseteq [\![a]\!]$	$[a] \leftarrow [a] \cup [b]$
2	$a = \&b$	$b \in [\![a]\!]$	$[a] \leftarrow [a] \cup \{b\}$
3	$a = *b$	$\forall c \in [\![b]\!] : [\![c]\!] \subseteq [\![a]\!]$	$[a] \leftarrow [a] \cup \left(\bigcup_{c \in [b]} [c]\right)$
4	$*a = b$	$\forall c \in [\![a]\!] : [\![b]\!] \subseteq [\![c]\!]$	$\forall c \in [a] : [c] \leftarrow [c] \cup [b]$

Andersen's pointer analysis (APA**).** An instance of APA is a pair (A, S), where A is a set of n *pointers*[1] and S is a set of m *statements*. Each statement has one of the four types shown in Table 1. Conceptually, the pointers may reference memory locations during the runtime of a program which uses the statements to manipulate the pointers.

(1) A type 1 statement $a = b$ represents pointer assignment.
(2) A type 2 statement $a = \&b$ represents making a point to the location of b.
(3) A type 3 statement $a = *b$ represents an indirect assignment $a = c$, where c is pointed by b.
(4) A type 4 statement $*a = b$ represents an indirect assignment $c = b$, where c is pointed by a.

As standard practice, more complex statements such as $*a = *b$ have been normalized by introducing slack pointers [Andersen 1994; Horwitz 1997; Møller and Schwartzbach 2018]. We assume wlog that S does not contain the same statement twice, and hence $m = O(n^2)$ [2]. Given some $i \in [4]$, we denote by S_i the statements of S of type i. Given a pointer a, we let $[\![a]\!] \subseteq A$ be the *points-to* set of a. Typically in pointer analysis, $[\![a]\!]$ is an over-approximation of the locations that a can point-to during the lifetime of a program. In Andersen's inclusion-based pointer analysis, the sets $[\![a]\!]$ are defined as follows. Each statement generates an inclusion constraint between various points-to sets, as shown in Table 1. The solution to APA is the smallest assignment $\{[\![a]\!] \to 2^A\}_{a \in A}$ that satisfies all constraints.

Exhaustive vs on-demand. As standard in the literature, we distinguish between the *exhaustive* and *on-demand* versions of the problem. In each case, the input is an instance of APA. The Exhaustive APA problem asks to compute the points-to set of every pointer $a \in A$. The On-demand APA problem asks to compute whether $b \in [\![a]\!]$ for a given pair of pointers $a, b \in A$. Hence, On-demand APA is a simplification of Exhaustive APA where the size of the output is a single bit, as opposed to $\Theta(n^2)$ bits required to output the points-to set of every pointer. The two variants can be viewed as analogues to the all-pairs and single-pair formulations of graph problems (e.g., reachability).

Operational semantics. Since the statements in APA come out of programs, it is convenient to consider them as executable instructions and assign simple operational semantics to them. The semantics are over a global store $[] : A \to 2^A$ that maps every pointer to its points-to set, which is initially empty. Executing one statement corresponds to updating the store as shown in Table 1. This operational view already hints an (albeit inefficient) algorithm for solving APA, namely, by iteratively executing some statement until no execution modifies the store.

[1] Although in practice not all variables are pointers, we will use this term liberally for simplicity of presentation.
[2] Note that the analyzed program might indeed contain the same statement twice. Generating the APA instance (A, S) from the program is performed in time linear in the size of the program, after which the input is in the assumed form.

$b = \&a,$ $d = \&a$	1. $b = \&a$	5. $d = *e$	1. $[b] \leftarrow \{a\}$	5. $[d] \leftarrow \{c, a\}$
$c = \&b,$ $d = \&c$	2. $c = \&b$	6. $*d = c$	2. $[c] \leftarrow \{b\}$	6. $[a] \leftarrow \{b\}$
$*d = c,$ $d = *e$	3. $d = \&c$	7. $d = a$	3. $[d] \leftarrow \{c\}$	7. $[d] \leftarrow \{c, a, b\}$
$e = *d,$ $e = \&f$	4. $e = *d$	8. $*d = c$	4. $[e] \leftarrow \{b\}$	8. $[b] \leftarrow \{a, b\}$

Fig. 2. An instance of APA (left), a witness program for $b \in [\![b]\!]$ (middle), and the updates to the store while executing the witness (right). Note that the statement $e = \&f$ is not used, while $*d = c$ is used twice.

Witnesses. The operational semantics allow us to define witnesses of points-to relations. Given two pointers $a, b \in A$, a *witness program* (or simply, *witness*) for $b \in [\![a]\!]$ is a sequence of statements from S that results in $b \in [a]$. See Figure 2 for an illustration.

Bounded APA**.** Motivated by practical applications, we introduce a bounded version of APA that restricts the length of witnesses. Consider two pointers $a, b \in A$ and a witness program $\mathcal{P}$ for $b \in [\![a]\!]$. Given some $i \in [4]$ and $j \in \mathbb{N}$, we say that $\mathcal{P}$ is (i, j)*-bounded* if $\mathcal{P}$ executes at most j statements of type i. For example, the witness for $b \in [\![b]\!]$ in Figure 2 is $(3, 2)$-bounded but not $(2, 2)$-bounded. The problem of (i, j)-bounded APA asks for a solution to the inclusion constraints of APA such that for any two pointers a, b with $b \notin [\![a]\!]$, any witness program that results in $b \in [a]$ executes more than j statements of type i.

Remark 1. *The boundedness of* (i, j)*-bounded* APA *is only one-way, i.e., for relationships of the form* $b \notin [\![a]\!]$ *and not of the form* $b \in [\![a]\!]$*. In particular, it is allowed to have* $b \in [\![a]\!]$ *even if this is witnessed only by programs that execute statements of type* i *more than* j *times.*

Bounded versions of APA do not necessarily have a unique solution, e.g., if the shortest witness for a pointing to b exceeds the bound, we can have $b \in [\![a]\!]$ or $b \notin [\![a]\!]$. However, any solution suffices as long as (i) every points-to relationship $b \in [\![a]\!]$ reported has a witness, and (ii) all points-to relationships that have a bounded witness are reported (wrt the given bound). Similar techniques for witness bounding are used widely in practice in order to speed up static analyses.

2.2 Dyck Reachability and Representation of Andersen's Pointer Analysis

Here we develop some notation on Dyck languages and Dyck reachability, and use it to represent instances of APA as Dyck graphs.

Dyck languages. Given a non-negative integer $k \in \mathbb{N}$, we denote by $\Sigma_k = \{\epsilon\} \cup \{\alpha_i, \overline{\alpha}_i\}_{i=1}^{k}$ a finite *alphabet* of k parenthesis types, together with a null element ϵ. We denote by $\mathcal{D}_k$ the Dyck language over Σ_k, defined as the language of strings generated by the following context-free grammar $\mathcal{G}_k$:

$$\mathcal{S} \to \mathcal{S}\,\mathcal{S} \mid \alpha_1\,\mathcal{S}\overline{\alpha}_1 \mid \cdots \mid \alpha_k\,\mathcal{S}\overline{\alpha}_k \mid \epsilon$$

In words, $\mathcal{D}_k$ contains all strings where parentheses are properly balanced. In this work we focus on the special case where $k = 1$, i.e., we have only one parenthesis type. To capture the relationship between $\mathcal{D}_1$ and APA, we will let α_1 be $\&$ and $\overline{\alpha}_1$ be $*$. This relationship will become clearer later in this section.

Dyck graphs. A Dyck graph $G = (V, E)$ is a digraph where edges are labeled with elements of Σ_1, i.e., $E \subseteq V \times V \times \Sigma_1$ and edges have the form $\tau = (a, b, \lambda)$. Often we will only be interested in the endpoints a, b of an edge, in which case we represent $\tau = (a, b)$, and we will denote by $\lambda(\tau)$ the label of τ. The label $\lambda(P)$ of a path in G is the concatenation of the labels along the edges of P.

We often represent P from a to b graphically as $a \overset{\lambda(P)}{\rightsquigarrow} b$, and, given some $i \in \mathbb{N}$, we write $a \overset{i\&}{\rightsquigarrow} b$ (resp.,$a \overset{i*}{\rightsquigarrow} b$) to denote a path $P\colon a \rightsquigarrow b$ with label i consecutive symbols $\&$ (resp., $*$), possibly interleaved with ϵ symbols. We say that b is *Dyck-reachable* (or *D-reachable*) from a if there exists a path $P\colon x \rightsquigarrow y$ with $\lambda(P) \in \mathcal{D}_1$. We say that b *flows into* a via a node c if (i) we have $b \xrightarrow{\&} c$, and (ii) a is D-reachable from c. The D_1-Reachability problem takes as input a Dyck graph and asks to return all pairs of nodes (b, a) such that a is D-reachable from b.

Graph representation of APA**.** For convenience, we frequently represent instances of APA using Dyck graphs. In particular, given an instance (A, S) of APA, we use a Dyck graph $G = (A, E)$ where E represents all statements in $S \setminus S_4$. In particular, we have the following edges.

(1) For every type 1 statement $a = b$, we have $b \xrightarrow{\epsilon} a$ in E.
(2) For every type 2 statement $a = \&b$, we have $b \xrightarrow{\&} a$ in E.
(3) For every type 3 statement $a = *b$, we have $b \xrightarrow{*} a$ in E.

The instance (A, S) is represented as a pair (G, S_4) where $G = (A, E)$ is the Dyck graph and S_4 is the type-4 statements of S. See Figure 3 for an illustration. The motivation behind this representation comes from the following lemma, which establishes a correspondence between paths in Dyck graphs and APA without statements of type 4.

Lemma 2.1. *Consider the Dyck graph representation* $(G = (E, V), S_4)$ *of* (A, S)*, and the modified* APA *instance* $(A, S \setminus S_4)$*. For every two pointers* $a, b \in A$*, we have* $b \in [\![a]\!]$ *in* $(A, S \setminus S_4)$ *iff* b *flows into* a *in* G*.*

Resolved Dyck graphs. Consider an APA instance (A, S) and the corresponding Dyck-graph representation $(G = (A, E), S_4)$. The *resolved Dyck graph* $\overline{G} = (A, \overline{E})$ is a Dyck graph where $\overline{E}$ is the smallest set that satisfies the following conditions.

(1) $E \subseteq \overline{E}$.
(2) For every statement $*a = b$, for every node c that flows into a in $\overline{G}$, we have $b \xrightarrow{\epsilon} c$.

Intuitively, all type 4 statements have been resolved as ϵ-edges in $\overline{G}$. See Figure 3 for an illustration. The following lemma follows directly from Lemma 2.1.

Lemma 2.2. *For every two pointers* $a, b \in A$*, we have* $b \in [\![a]\!]$ *iff* b *flows into* a *in* $\overline{G}$*.*

Intuition behind the Dyck-graph representation. Virtually all algorithms for APA in the literature use a flow graph for representing inclusion relationships between pointers. Pointer inclusion occurs when the analysis discovers that, for two pointers a, b we have $[\![b]\!] \subseteq [\![a]\!]$, represented via an edge $b \to a$ in the flow graph.

Our Dyck graph G is a richer structure compared to the standard flow graph. In fact, we obtain the (initial) flow graph if we remove from G edges representing pointer references (labeled with $\&$) and dereferences of type 3 (labeled with $*$). The only information missing from G is statements of type 4. The analysis can be seen as iteratively discovering type 4 statements $*a = b$ and inserting an edge $b \xrightarrow{\epsilon} c$ in G. This process terminates with the resolved graph $\overline{G}$. Lemma 2.2 implies that, at that point, all points-to information can be expressed as flows-into relationships in $\overline{G}$.

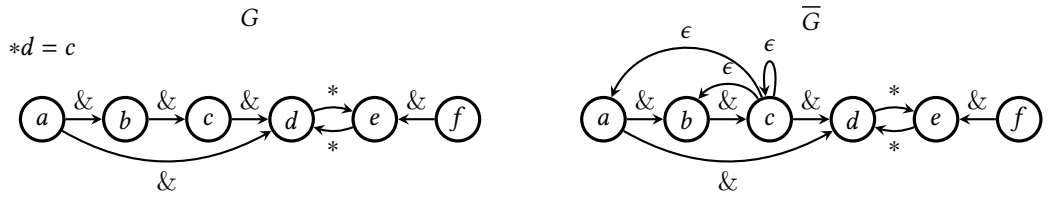

Fig. 3. The Dyck graph for the APA instance of Figure 2 (left), and the resolved Dyck graph $\overline{G}$ (right).

3 SUMMARY OF MAIN RESULTS

We are now ready to present our main theorems, followed by a discussion on their implications to the theory and practice of pointer analysis. In later sections we develop the proofs.

Cubic upper-bound of APA**.** It is well stated that APA can be solved in cubic time. However, "cubic" refers to the size of the input, and typically has the form $O(m^3)$ [Melski and Reps 2000; Møller and Schwartzbach 2018] or $O(n^2 \cdot m)$ [Kodumal and Aiken 2004; Pearce et al. 2004], for n pointers and m statements. Note that m can be as large as $\Theta(n^2)$, which yields the bound $O(n^4)$, as already observed in [Kodumal and Aiken 2004]. Our first theorem shows that in fact, the problem is solvable in $O(n^3)$ time regardless of m. Although we do not consider this our major result, we are not aware of a proven $O(n^3)$ bound with n being the number of pointers. We also hope that the theorem will provide a future reference for a formal $O(n^3)$ complexity statement for APA.

Theorem 3.1. Exhaustive APA *is solvable in $O(n^3)$ time, for any m, where n is the number of pointers and m is the number of statements.*

Cubic hardness of APA**.** Given the cubic upper-bound of Theorem 3.1, it is natural to ask whether sub-cubic algorithms exist for the problem, i.e., algorithms with running time $O(n^{3-\epsilon})$, for some fixed $\epsilon > 0$. Indeed, the rich literature of heuristics (e.g., [Aiken et al. 1997; Berndl et al. 2003; Dietrich et al. 2015; Fähndrich et al. 1998; Hardekopf and Lin 2007; Heintze and Tardieu 2001b; Pearce et al. 2004; Pek and Madhusudan 2014; Rountev and Chandra 2000; Su et al. 2000; Vedurada and Nandivada 2019; Xu et al. 2009]) is indicative of the need for such an improvement. On the other hand, no lower-bound has been known. In fine-grained complexity, there is a widespread distinction between *combinatorial* and *algebraic* algorithms. The most famous combinatorial lower bound is for Boolean Matrix Multiplication (BMM). The respective hypothesis states that there is no combinatorial $O(n^{3-\epsilon})$ algorithm for multiplying two $n \times n$ Boolean matrices, for any fixed $\epsilon > 0$. The BMM hypothesis has formed the basis for many lower bounds in graph algorithms, verification, and static analysis [Abboud and Vassilevska Williams 2014; Bansal and Williams 2009; Chatterjee et al. 2018, 2016; Williams and Williams 2018].

Given the combinatorial nature of APA, we examine whether combinatorial sub-cubic improvements are possible. First, note that the edge set $\{(x_i, x_j)\}$ of a digraph can be represented as a set of pointer assignments $\{(x_j = x_i)\}$. This observation leads us to the following remark.

Remark 2. *Even without statements of type 3 and type 4 (i.e., without pointer dereferences),* Exhaustive APA *is at least as hard as computing all-pairs reachability in a digraph. Since, under the BMM hypothesis, all-pairs reachability does not have a combinatorial sub-cubic algorithm, the same lower-bound follows for* Exhaustive APA*.*

On the other hand, single-pair reachability is solvable in linear-time in the size of the graph, and is thus considerably easier than its all-pairs version. Hence, the relevant question is whether On-demand APA (i.e., given pointers a, b, is it the case that $b \in [\![a]\!]$?) has sub-cubic complexity. We answer this question in negative.

Theorem 3.2. On-demand APA *has no sub-cubic combinatorial algorithm under the BMM hypothesis.*

Note that Theorem 3.2 indeed relates a problem with output size $\Theta(1)$ (On-demand APA) to a problem with output size $\Theta(n^2)$ (BMM). The theorem has four implications. First, it establishes formally the long-conjectured "cubic bottleneck" for APA. Second, it shows that the algorithm of Theorem 3.1 is optimal even for On-demand APA, as far as combinatorial algorithms are concerned. Third, it indicates that the hardness of APA does not come from the requirement to produce large-sized outputs (i.e., of size $\Theta(n^2)$ for the points-to set of each pointer), as sub-cubic complexity is also unlikely for constant-size outputs. Fourth, it shows that all on-demand analyses (e.g., [Chatterjee et al. 2018; Heintze and Tardieu 2001a; Lu et al. 2013; Sridharan et al. 2005; Vedurada and Nandivada 2019; Zhang et al. 2013; Zheng and Rugina 2008]), which attempt to reduce complexity by avoiding the exhaustive computation of all points-to sets, can only provide heuristic improvements without any guarantees.

Bounded APA**.** Given the hardness of APA under Theorem 3.2, we next seek mild restrictions that allow for algorithmic improvements below the cubic bound. Perhaps surprisingly, we show that bounding the number of times statements of type 4 are executed suffices.

Theorem 3.3. *For all $j \in \mathbb{N}^+$, $(4, j)$-bounded* Exhaustive APA *is solvable in $\tilde{O}(n^\omega \cdot j)$ time, where n is the number of pointers and ω is the matrix-multiplication exponent. In particular, $(4, \tilde{O}(1))$-bounded* Exhaustive APA *is solvable in $\tilde{O}(n^\omega)$ time.*

Here $\tilde{O}$ hides poly-logarithmic factors (i.e., factors of the form $\log^c n$, for some constant c). It is known that $\omega < 2.373$ [Le Gall 2014], hence the bound is sub-cubic. Besides its theoretical interest, Theorem 3.3 also has practical relevance, as it reduces the problem to a small number of matrix multiplications. First, some sub-cubic algorithms for matrix multiplication, like Strassen's [Strassen 1969], often lead to observable practical speedups over simpler, cubic algorithms [Huang et al. 2016; Huss-Lederman et al. 1996]. Second, this reduction can take advantage of both highly optimized practical implementations for matrix multiplication [Kaporin 1999; Laderman et al. 1992] and specialized hardware [Dave et al. 2007]. Third, the algorithm behind Theorem 3.3 is an "anytime algorithm", in the spirit of [Boddy 1991; Chatterjee et al. 2015]. The algorithm computes $(4, j)$-bounded Exhaustive APA iteratively for increasing values of j. It can be terminated in any iteration j according to the runtime requirements of the analysis. At that point, the algorithm has executed for at most $\tilde{O}(n^\omega \cdot j)$ time, and is guaranteed to have computed all points-to relationships as witnessed by $(4, j)$-bounded programs. Hence, (i) a timeout does not waste analysis time, and (ii) the obtained results provide measurable completeness guarantees.

It is believed that $\omega = 2 + o(1)$, in which case Theorem 3.3 yields a quadratic bound. Given such an improvement, a natural question is whether sub-quadratic algorithms are possible when we restrict our attention to witnesses that are poly-logarithmically bounded. Clearly this is not possible for Exhaustive APA, as the size of the output can be $\Theta(n^2)$, but the question becomes interesting in the case of On-demand APA that has output size $\Theta(1)$. We answer this question in negative.

Theorem 3.4. $(\text{All}, \tilde{O}(1))$*-bounded* On-demand APA *has no sub-quadratic algorithm under the Orthogonal Vectors hypothesis.*

Recall that in the $(\text{All}, \tilde{O}(1))$-bounded version of the problem, we restrict our attention to witness programs of poly-logarithmic length (i.e., on all statement types). The above bound holds even when $m = O(n)$. Orthogonal Vectors is a well-studied problem with a long-standing quadratic worst-case upper bound. The corresponding hypothesis states that there is no sub-quadratic algorithm for the problem [Williams 2019]. It is also known that the strong exponential time hypothesis (SETH) implies the Orthogonal Vectors hypothesis [Williams 2005]. Under Theorem 3.4, our algorithm from Theorem 3.3 is *optimal* when $\omega = 2 + o(1)$.

Finally, to establish Theorem 3.3, we solve D_1-Reachability in nearly matrix-multiplication time.

Theorem 3.5. *All-pairs* D_1-Reachability *is solvable in* $\tilde{O}(n^\omega)$ *time, where* n *is the number of nodes and* ω *is the matrix-multiplication exponent.*

Observe the different regimes of D_k-Reachability for various values of k. When $k = 0$, the problem becomes standard graph reachability, which is solvable in $O(n^\omega)$ time [Munro 1971]. When $k \geq 2$, the problem is solvable in $O(n^3)$ time, and this bound is believed to be tight (wrt polynomial improvements) [Heintze and McAllester 1997]. The case of $k = 1$ was recently solved independently in [Bradford 2018]. However, our algorithm behind Theorem 3.5 is more straightforward: it establishes a purely combinatorial reduction of the problem to $O(\log^2 n)$ many transitive-closure operations. From there, it relies on algebraic, fast-matrix multiplication for performing each transitive closure in $O(n^\omega)$ time. In contrast, the algorithm in [Bradford 2018] is considerably longer and relies on intricate algebraic transformations. In addition, our algorithm is a $\log n$-factor faster. We refer to Section 6.1 for a more detailed comparison.

Parallelizability of APA**.** Parallelization is an important aspect of the complexity of a problem. In the case of APA, this is further motivated by a multitude of parallel implementations [Blaß and Philippsen 2019; Liu et al. 2019; Mendez-Lojo et al. 2012; Méndez-Lojo et al. 2010; Su et al. 2014; Wang et al. 2017]. The question is thus whether APA is effectively parallelizable. We answer this question in negative.

Theorem 3.6. On-demand APA *is P-complete.*

Theorem 3.6 implies that all efforts on complete parallelization must stay at the current, heuristic level, while their improvements vanish on hard instances. On the other hand, in spirit similar to Theorem 3.3, we can seek for mild restrictions to the problem that make it parallelizable. Recall that, given some $i \in \mathbb{N}$, the complexity class NC^i contains the problems that are solvable in parallel time $O(\log^i n)$ with polynomially many processors. We show the following theorem.

Theorem 3.7. $(4, \log^i n)$*-bounded* Exhaustive APA *is in* NC^{i+2}.

Thus, APA with poly-logarithmically many applications of type 4 statements is highly parallelizable. Together, Theorem 3.6 and Theorem 3.7 expose the core hardness in the parallelization of APA as stemming from handling statements of type 4.

In the following sections we present details of the above theorems. To improve readability, in the main paper we present algorithms, examples, and proofs of all theorems. To highlight the main

steps of the proofs, we also present all intermediate lemmas; many lemma proofs, however, are relegated to a technical report [Mathiasen and Pavlogiannis 2020].

4 A BASELINE ALGORITHM FOR ANDERSEN'S POINTER ANALYSIS

In this section, we present a baseline algorithm for APA. It has similar flavor as other algorithms in the literature, but allows us to more easily establish Theorem 3.1.

Algorithm 1: AndersenAlgo

Input: An instance (A, S) of APA.
Output: A set (a, b) of all points-to relationships $b \in [\![a]\!]$.

```
   // Initialization
1  Let W, Done ← ∅
2  foreach a = &b do
3  |  Insert (b, a, &) in W
4  |  Insert (b, a, &) in Done
5  end
6  foreach a = b do
7  |  Insert (b, a, ϵ) in W
8  |  Insert (b, a, ϵ) in Done
9  end
   // Computation
10 while W ≠ ∅ do
11 |  Extract (a, b, t) from W
12 |  if t = ϵ then
13 |  |  ProcessEps(a, b)
14 |  else
15 |  |  ProcessRef(a, b)
16 |  end
17 end
18 return {(a, b) : (b, a, &) ∈ Done}

   // Process inclusion edges
19 Function ProcessEps(a, b):
20 |  foreach (c, a, &) ∈ Done do
21 |  |  Establish(c, b, &)
22 |  end

   // Establish new inclusion and
      reference edges
23 Function Establish(a, b, t):
24 |  if (a, b, t) ∉ Done then
25 |  |  Insert (a, b, t) in W
26 |  |  Insert (a, b, t) in Done

   // Process reference edges
27 Function ProcessRef(a, b):
28 |  foreach (b, c, ϵ) ∈ Done do
29 |  |  Establish(a, c, &)
30 |  end
31 |  foreach c = *b in S do
32 |  |  Establish(a, c, ϵ)
33 |  end
34 |  foreach *b = c in S do
35 |  |  Establish(c, a, ϵ)
36 |  end
```

Algorithm AndersenAlgo**.** The algorithm performs a form of dynamic transitive closure of a Dyck graph, in similar spirit to existing algorithms in the literature. The key difference is that instead of just maintaining inclusion relationships $a \xrightarrow{\epsilon} b$, the flow graph also explicitly captures points-to relationships $a \xrightarrow{\&} b$ in its edges. In each iteration, the algorithm processes a newly inserted edge $a \xrightarrow{t} b$, where $t \in \{\epsilon, \&\}$, and inserts new edges that represent flows-into and inclusion relationships that are implied by $a \xrightarrow{t} b$, the current state of the flow graph, and the statements in S. The algorithm can be seen as an on-the-fly version of the difference-propagation technique [Pearce et al. 2004; Sridharan and Fink 2009], without the need to store difference-sets explicitly. See Algorithm 1 for a detailed description.

Proof of Theorem 3.1. The correctness of the algorithm follows by a straightforward induction. Here we argue about the complexity. The initialization clearly runs in time $O(m) = O(n^2)$. For every pair of pointers (a, b), the worklist $\mathcal{W}$ can have at most one element of the form (a, b, ϵ) and at most one element of the form $(a, b, \&)$, due to the set Done. Hence, the main loop in Line 10 is executed at most twice for every pair of pointers (a, b), and thus $O(n^2)$ times in total. For every

such pair, the loops in Lines 20 and 28 are executed once for each pointer c. Hence, these loops are executed in $O(n^3)$ time in total. Finally, each of the loops in Lines 31 and 34 is executed once for every pointer a. Summing over all statements of type 3 and type 4, which are $O(m)$ many, we obtain that these loops are executed $O(n \cdot m)$ times in total. Since $m = O(n^2)$, we obtain the desired bound $O(n^3)$. □

5 THE CUBIC HARDNESS OF ANDERSEN'S POINTER ANALYSIS

In this section we prove Theorem 3.2, i.e., that even On-demand APA (given pointers a, b, is it that $b \in [\![a]\!]$?) does not have a combinatorial sub-cubic algorithm under the BMM hypothesis.

Exhaustive vs On-Demand APA**.** Recall the difference between Exhaustive APA and On-demand APA. The former problem asks for the points-to set $[\![a]\!]$ of *every* pointer a, while the latter focuses on a *specific* pair of pointers a, b, and asks whether $b \in [\![a]\!]$. Thus, On-demand APA is a special version of Exhaustive APA. The (combinatorial) cubic hardness of Exhaustive APA follows straightforwardly via a reduction from the graph transitive closure, which has the same combinatorial cubic lower bound. Indeed, given a directed graph $H = (V, E)$, we simply create an APA instance (A, S), where $A = \{a_1, a_2 : a \in V\}$ contains two copies of each node in V, and $S = S_1 \cup S_2$, where $S_1 = \{a_1 = b_1 : (b, a) \in E\}$ and $S_2 = \{a_1 = \&a_2 : a \in V\}$. That is, the type-1 statements directly correspond to the edges in H, and the type-2 statements are dummy statements that initialize the points-to sets. It follows immediately that after solving the APA instance (A, S), for every node $a \in V$, the points-to set $[\![a_1]\!]$ contains all b_2 such that a is reachable by b in H. Notably, the APA instance does not even make use of type-3 and type-4 statements.

We remark that the reduction does not apply for the on-demand version of APA. In the following we establish the proof of Theorem 3.2. In this direction, we establish a fine-grained reduction [Williams 2019] from the problem of deciding whether a graph contains a triangle to On-demand APA.

Reduction from finding triangles. Consider an undirected graph $H = (V, E)$, of n' nodes, where the task is to determine if H contains a triangle. For notational convenience, we take the node set of H to be the set of integers $[n']$. Hence, the task is to determine if there exist distinct $i, j, k \in [n']$ such that $(i, j), (j, k), (k, i) \in E$. Our reduction constructs four pointers a_i, b_i, c_i, d_i for every node $i' \in [n']$, and uses one additional pointer s such that $s \in [\![c_1]\!]$ iff H has a triangle.

Intuition. The search for a triangle (i, j, k) of H can be seen as a search for two nodes i and k such that k is both a distance-1 and distance-2 neighbor of i. In our reduction, the two pointers c_k and d_k are such that $c_k \xrightarrow{\epsilon} d_k$ in the resolved Dyck graph $\overline{G}$ iff k is both distance-1 and distance-2 neighbor of some node i. This is achieved in two steps.

(1) In the initial Dyck graph G of the APA instance, c_k flows into a_i, by introducing two statements $b_j = \&c_k$ and $a_i = b_j$, where j is a neighbor of both i and k.
(2) We have a statement $*a_i = d_k$.

We also introduce some additional statements between all d_i and between all c_i such that c_1 is D-reachable from d_1 in $\overline{G}$ iff there exists some k such that $c_k \xrightarrow{\epsilon} d_k$ in $\overline{G}$ (i.e., by the above, k is a distance-1 and distance-2 neighbor of some node i). Finally, we have a statement $d_1 = \&s$, so that $s \in [\![c_1]\!]$ iff the above condition holds. Figure 4 provides an illustration.

Formal construction. We now proceed with the formal construction. We construct an instance of On-demand APA as follows.

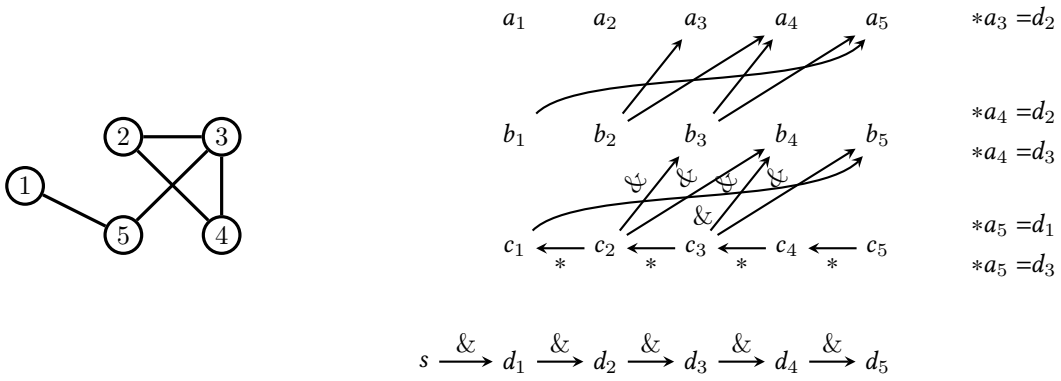

Fig. 4. An undirected graph (left) and the corresponding On-demand APA instance (right).

(1) We introduce a distinguished pointer s.
(2) For every node $i \in [n']$, we introduce four pointers a_i, b_i, c_i, d_i.
(3) For every $(i, j) \in E$ with $j < i$, we have (i) $a_i = b_j$, and (ii) $b_i = \&c_j$.
(4) For every $(i, j) \in E$ with $j > i$, we have $*a_i = d_j$.
(5) Finally, we have the following sets of assignments.

$$d_1 = \&s \quad d_2 = \&d_1 \quad \cdots \quad d_n = \&d_{n-1} \qquad \text{and} \qquad c_1 = *c_2 \quad c_2 = *c_3 \quad \cdots \quad c_{n-1} = *c_n \ .$$

The on-demand question is whether $s \in [\![c_1]\!]$. Observe that the number of pointers of our APA instance is $O(n')$, and the above construction can be easily carried out in time proportional to the size of H.

Correctness. We now establish the correctness of the above construction. The key idea is as follows. Recall our definition of the resolved Dyck graph $\overline{G}$ from Section 2.2. An edge $d_k \xrightarrow{\epsilon} c_k$ is inserted in $\overline{G}$ iff k is both a distance1- and distance-2 neighbor of some node i. In turn, this implies the existence of a triangle in H that contains i and k. We have the following lemma.

Lemma 5.1. *We have that s flows into c_1 in $\overline{G}$ iff H has a triangle.*

Proof. We prove each direction separately.

($\Rightarrow$). Assume that H has a triangle (i, j, k), with $i > j > k$. Then we have $a_i = b_j$ and $b_j = \&c_k$ and thus $c_k \overset{\&}{\rightsquigarrow} a_i$ in $\overline{G}$. In addition, we have $*a_i = d_k$, and thus $d_k \xrightarrow{\epsilon} c_k$ in $\overline{G}$. Observe that this creates a path $s \overset{k\&}{\rightsquigarrow} d_k \overset{\epsilon}{\rightsquigarrow} c_k \overset{(k-1)*}{\rightsquigarrow} c_1$, which witnesses that s flows into c_1 in $\overline{G}$.

($\Leftarrow$). Assume that s flows into c_1. Observe that for all a_i, if some node x flows into a_i then x is a c node. It follows that $\overline{G}$ is identical to G with some additional edges from d nodes to c nodes. Hence, since s flows into c_1, there exists some $k \in [n']$ such that $\overline{G}$ has an edge $d_k \xrightarrow{\epsilon} c_k$. This means that (i) there exists an i such that c_k flows into a_i (thus k is a distance-2 neighbor of i), and (ii) there is a statement $*a_i = d_k$ (thus k is a distance-1 neighbor of i). Hence H has a triangle containing i and k. The desired result follows. □

We conclude this section with the proof of Theorem 3.2.

Proof of Theorem 3.2. Due to Lemma 5.1, we have that s flows into c_1 iff H contains a triangle. By Lemma 2.2 we have that $s \in [\![c_1]\!]$ iff H contains a triangle. By [Williams and Williams 2018], triangle detection has no sub-cubic combinatorial algorithm under the combinatorial BMM-hypothesis.

The desired result follows. □

6 A SUB-CUBIC ALGORITHM FOR BOUNDED ANDERSEN'S POINTER ANALYSIS

In this section, we first show Theorem 3.3, i.e., that computing points-to relationships when bounding the number of applications of type 4 statements admits a sub-cubic algorithm. To this end, we first prove in Section 6.1 Theorem 3.5, i.e., that D_1-Reachability can be solved in nearly matrix-multiplication time. Afterwards, we use this result to prove Theorem 3.3 in Section 6.2.

6.1 A Sub-cubic Algorithm for D_1-Reachability

In this section we establish a combinatorial reduction of all-pairs D_1-Reachability to $O(\log^2 n)$ matrix multiplications, establishing that the problem is solvable in nearly matrix-multiplication time. We first set up some helpful notation, and then present the main algorithm. Consider a Dyck graph $G = (V, E)$.

Path indexing. Consider a path $P = x_1, \ldots, x_l$. Given some $i \in [l]$, we denote by $P[i] = x_i$. Given $i, j \in [l]$, with $i \leq j$, we denote by $P[i : j] = x_i, \ldots, x_j$. For simplicity, we let $P[: j] = P[1 : j]$ and $P[i :] = P[i : l]$.

Stack heights. Consider a path P of length l. We denote by $\#_{\&}(P)$ (resp., $\#_*(P)$) the number of $\&$ (resp., $*$) symbols that appear in the label $\lambda(P)$. The *stack height* of P is defined as $\mathrm{SH}(P) = \#_{\&}(P) - \#_*(P)$ (note that we can have $\mathrm{SH}(P) < 0$). The *maximum stack height* of a path is defined as $\mathrm{MSH}(P) = \max_{P'} \mathrm{SH}(P')$, where P' ranges over prefixes of P.

Monotonicity and local maxima. Consider a path P of length l. We say that P is *monotonically increasing* (resp., *monotonically decreasing*) if for all i with $1 \leq i < l$, we have $\mathrm{SH}(P[: i]) \leq \mathrm{SH}(P[: i + 1])$ (resp., $\mathrm{SH}(P[: i]) \geq \mathrm{SH}(P[: i + 1])$). Given some i with $1 \leq i \leq l$, we say that P has a *local maxima* in i if the following conditions hold.

(1) Either $i = 1$ or $\mathrm{SH}(P[: i - 1]) < \mathrm{SH}(P[: i])$.
(2) For every $j > i$ such that $\mathrm{SH}(P[: i]) < \mathrm{SH}(P[: j])$, there exists some l with $i < l < j$ such that $\mathrm{SH}(P[: l]) < \mathrm{SH}(P[: i])$.

Bell-shape-reachability. We call a path P *bell-shaped* if it has exactly one local maxima. If P is bell-shaped, it can be decomposed as $P\colon P_1 \circ P_2$ where P_1 (resp., P_2) is a monotonically increasing (resp., monotonically decreasing) path. Consider two nodes x, y. We say that y is *$i\&$-reachable* (resp., *$i*$-reachable*) from x, for some $i \in \mathbb{N}$, if there is a path $x \overset{i\&}{\rightsquigarrow} y$ (resp., $x \overset{i*}{\rightsquigarrow} y$). We say that y is *bell-shape-reachable* from x if there exists a bell-shaped path $x \rightsquigarrow y$.

Node distances. Given two nodes x, y, we define the *distance* $\delta(x, y)$ from x to y as the length of the shortest path $P\colon x \rightsquigarrow y$ with $\lambda(P) \in \mathcal{D}_1$, if such a path exists, otherwise $\delta(x, y) = \infty$. The *maxima-distance* $\gamma(x, y)$ is the smallest number of local maxima among all shortest paths $x \rightsquigarrow y$. The following known lemma states that the distances between two reachable nodes is at most quadratic. Note that $\gamma(x, y) \leq \delta(x, y)$, hence the same bound holds for the maxima-distance.

Lemma 6.1 ([Deleage and Pierre 1986]). *For every $x, y \in V$, if $\delta(x, y) < \infty$ then $\delta(x, y) = O(n^2)$.*

Routine BellReachAlgo**.** The main component of our algorithm for D_1-Reachability is a routine BellReachAlgo that computes bell-shape-reachability. Given an input Dyck graph $G = (V, E)$, BellReachAlgo computes all pairs of nodes (x, y) such that y is bell-shape-reachable from x. The algorithm constructs a sequence of $O(\log L)$ plain (i.e., not Dyck) digraphs $(G_i = (K, R_i))_i$, where L is an upper bound on the distance $\delta(x, y)$ of every pair of nodes $x, y \in V$, given by Lemma 6.1. The node set K is common to all G_i and consists of three copies x_1, x_2, x_3 for every node $x \in V$.

Intuitively, the algorithm performs a form of successive doubling on the length of the bell-shaped paths that witness reachability. In iteration i, the algorithm performs all-pairs reachability in G_i, and using this reachability information, constructs the edge set R_{i+1}. In high level, G_i consists of three copies of the graph G, where bell-shaped paths of maximum stack height at most $2^i - 1$ are summarized as ϵ-labeled edges in the first and second copy. Paths between the nodes in the first and third copy are used to summarize monotonically increasing and (resp., decreasing) paths in G with labels of the form $2^i\&$ (resp., 2^i*). We refer to Algorithm 2 for a detailed description and to Figure 5 for an illustration.

Algorithm 2: BellReachAlgo

Input: A Dyck graph $G = (V, E)$
Output: A set $\{(x, y)\}_{x,y \in V}$ such that y is bell-shape-reachable from x.

```
// Initialization
Construct a node set K = {x_1, x_2, x_3 : x ∈ V}
Construct an edge set R_1, initially R_1 ← ∅
foreach j ∈ [2] do
    Insert (x_j, y_j) ∈ R_1 iff (x, y, ε) ∈ E
    Insert (x_j, y_{j+1}) ∈ R_1 iff (x, y, &) ∈ E
    Insert (y_{j+1}, x_j) ∈ R_1 iff (x, y, *) ∈ E
end
Construct the graph G_1 = (K, R_1)
Let L ← an upper bound on δ(x, y) for all x, y ∈ V
// Computation
foreach i ∈ [⌈log L⌉] do
    Compute all-pairs reachability in G_i
    Construct an edge set R_{i+1}, initially R_{i+1} ← ∅
    foreach j ∈ [2] do
        Insert (x_j, y_i) ∈ R_{i+1} iff x_1 ⇝ y_1 in G_i
        Insert (x_j, y_{j+1}) ∈ R_{i+1} iff x_1 ⇝ y_3 in G_i
        Insert (y_{j+1}, x_j) ∈ R_{i+1} iff y_3 ⇝ x_1 in G_i
    end
    Construct the graph G_{i+1} = (K, R_{i+1})
end
return R_{⌈log L⌉+1}
```

Correctness of BellReachAlgo. It is straightforward that for each iteration i, if $x_1 \rightsquigarrow y_1$ in G_i, then y is D-reachable from x in G. The following lemma captures the inverse direction restricted to bell-shaped paths, i.e., if $x \overset{\epsilon}{\rightsquigarrow} y$ via a bell-shaped path P in G with $\mathrm{MSH}(P) \leq 2^i - 1$, then $x_1 \rightsquigarrow y_1$ in G_i. The key invariants are stated in the following lemma.

Lemma 6.2. *Consider an execution of the routine* BellReachAlgo. *For each* $i \in [\lceil \log L \rceil]$, *the following assertions hold.*

(1) If y *is D-reachable from* x *via a bell-shaped path* P *in* G *with* $\mathrm{MSH}(P) \leq 2^i - 1$, *then* $x_1 \rightsquigarrow y_1$ *in* G_i.
(2) If $x \overset{2^i\&}{\rightsquigarrow} y$ *via a monotonically increasing path in* G *where the last edge is* $\&$*-labeled, then* $x_1 \rightsquigarrow y_3$ *in* G_i.
(3) If $y \overset{2^i*}{\rightsquigarrow} x$ *via a monotonically decreasing path in* G *where the first edge is* $*$*-labeled, then* $y_3 \rightsquigarrow x_1$ *in* G_i.

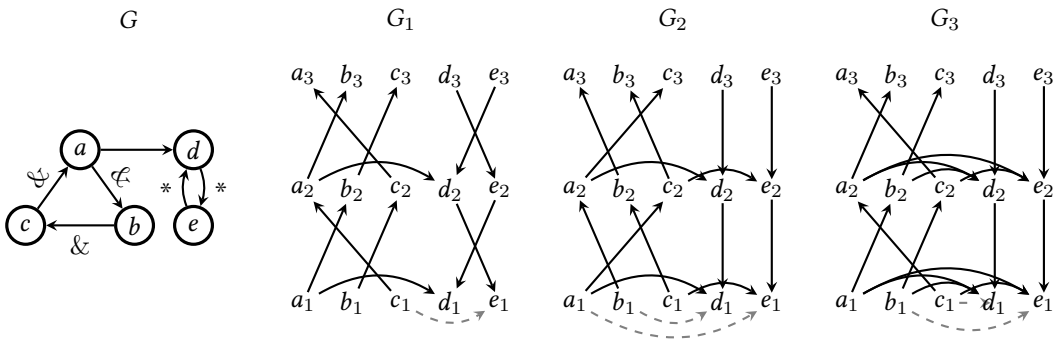

Fig. 5. Illustration of BellReachAlgo on the Dyck graph G (left). Bell-shape-reachability in G as witnessed by paths $P\colon x \rightsquigarrow y$ with $\mathrm{MSH}(P) \leq 2^i - 1$ is captured in graph G_i (right) by the path $x_1 \rightsquigarrow y_1$. Dashed edges in G_i represent the summarization of the path, which is carried over to G_{i+1} as a single edge.

Algorithm D_1-ReachAlgo**.** We are now ready to describe our algorithm D_1-ReachAlgo for D_1-Reachability. The algorithm performs $\lceil \log L \rceil$ iterations of BellReachAlgo, where L is an upper bound on the distance between any two reachable nodes in G. See Algorithm 3 for a detailed description.

Algorithm 3: D_1-ReachAlgo

Input: A Dyck graph $G = (V, E)$
Output: A set $\{(x, y)\}_{x,y \in V}$ such that y is D-reachable from x.

```
// Initialization
Let G_1 = (V, E_1) be a Dyck graph with E_1 = E
// Computation
foreach i ∈ [⌈log L⌉ + 1] do
    Let X = BellReachAlgo on input G_i
    Let E_{i+1} = E_i ∪ {(x, y, ϵ) : (x, y) ∈ X}
    Construct the graph G_{i+1} = (V, E_{i+1})
end
return E_{⌈log L⌉+1}
```

Correctness of D_1-ReachAlgo**.** We now establish the correctness of D_1-ReachAlgo. We start with an intuitive description of the correctness, and afterwards we make the argument formal (see Figure 6 for an illustration).

Intuitive argument of correctness. Consider an iteration i, and let $x, y \in V$ such that y is D-reachable from x via a path $P\colon x \rightsquigarrow y$. At the end of the iteration, due to the execution of routine BellReachAlgo, all bell-shaped paths $u \rightsquigarrow v$ in G_i are summarized as ϵ-labeled edges $u \xrightarrow{\epsilon} v$ in G_{i+1}. Hence, P is summarized by a path P' in G_{i+1}, where the bell-shaped sub-paths of P are replaced by ϵ-edges in P'. How many times do we need to perform this iteration until the whole of P is summarized by a single ϵ-labeled edge? The key insight is that the number of local maxima in P' is *at most half* of that in P. Hence, it suffices to compute bell-shape-reachability a number of times that is logarithmic in the maxima-distance $\gamma(x, y)$. Since $\gamma(x, y) \leq \delta(x, y)$ and $\delta(x, y) \leq L = O(n^2)$ (by Lemma 6.1), $\lceil \log L \rceil = O(\log n)$ iterations suffice.

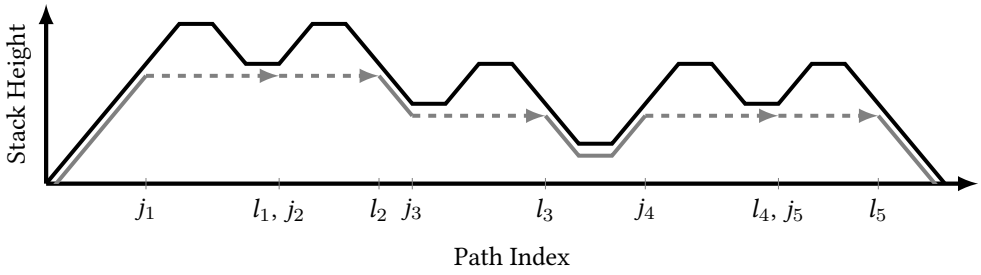

Fig. 6. Illustration of a path P in graph G_i (black) and its summarization path P' in graph G_{i+1} (gray). The number of local maxima in P' is at most half of that in P.

Formal correctness. We now proceed to make the above argument formal. Given some iteration i of D_1-ReachAlgo, consider the graphs G_i and G_{i+1}. Let $P\colon x \rightsquigarrow y$ be any path that witnesses D-reachability of y from x in G_i. Let $(j_\ell, l_\ell)_\ell$ be the index pairs that mark bell-shaped sub-paths in P. We require that each $(j_\ell, l_\ell)_\ell$ is maximal, in the following way.

(1) None of $P[j_1 - 1, l_1]$, $P[j_1, l_1 + 1]$ and $P[j_1 - 1, l_1 + 1]$ is bell-shaped.
(2) If $\ell > 1$, then $P[j_\ell, l_\ell + 1]$ is not bell-shaped, and if $P[j_\ell - 1, l_\ell]$ or $P[j_\ell - 1, l_\ell + 1]$ is bell-shaped, then $j_\ell - 1 \leq l_{\ell-1}$.

Intuitively, the first bell-shaped sub-path of P is as long as possible, and every following bell-shaped sub-path is as long as possible provided that it does not overlap with the previous bell-shaped sub-path. We decompose P as

$$P = P_1^{\downarrow} \circ P_1^{\uparrow} \circ P[j_1 : l_1] \circ P_2^{\downarrow} \circ P_2^{\uparrow} \circ P[j_2 : l_2] \circ \cdots \circ P[j_k, l_k] \circ P_{k+1}^{\downarrow} ,$$

where each $P_\ell^{\downarrow}$ (resp., $P_\ell^{\uparrow}$) is a monotonically decreasing (resp., monotonically increasing) path. Note that $P_1^{\downarrow} = \epsilon$. Observe that P has k local maxima, one in each bell-shaped sub-path $P[j_\ell : l_\ell]$. In G_{i+1}, the path P is summarized by a path P' identical to P, but with all the bell-shaped sub-paths $P[j_\ell : l_\ell]$ replaced by edges $x_{j_\ell} \xrightarrow{\epsilon} y_{j_\ell}$ (see Figure 6 for an illustration). Given some index $1 \leq h \leq |P|$ with $h \notin [j_\ell + 1, l_\ell - 1]$ for each $\ell \in [k]$, we denote by $f(h)$ the corresponding index in P'.

Remark 3. *For every index h of P', we have* $\mathrm{SH}(P'[:h]) = \mathrm{SH}(P[: f^{-1}(h)])$.

We first have two technical lemmas. The first lemma states that all local maxima in P' appear on the first node of bell-shaped sub-paths of P.

Lemma 6.3. *Assume that P' has a local maxima at some h. Then $f^{-1}(h) = j_\ell$ for some $\ell \in [k]$.*

The following lemma formalizes the following observation: if the beginning of a bell-shaped sub-path of P marks a local maxima for P', then the beginning of the next bell-shaped sub-path of P cannot mark a local maxima for P'. This is shown by arguing that the two bell-shaped sub-paths of P are next to each other, i.e., there are no monotonically decreasing and increasing paths separating them.

Lemma 6.4. *Assume that P' has a local maxima at some h. Then $P_{j_\ell+1}^{\downarrow} = P_{j_\ell+1}^{\uparrow} = \epsilon$, where $j_\ell = f^{-1}(h)$.*

With Lemma 6.3 and Lemma 6.4, we can now formalize the insight that the maxima-distance between any two nodes halves in each iteration of D_1-ReachAlgo. Given some iteration i of the algorithm, we denote by $\gamma_i(x, y)$ the maxima-distance from x to y in the graph G_i. We have the following lemma.

Lemma 6.5. *For each $i \in [\lceil \log L \rceil]$, for any two nodes $x, y \in V$ such that y is reachable from x in G, we have that $\gamma_{i+1}(x, y) \leq \gamma_i(x, y)/2$.*

Finally, we prove Theorem 3.5, i.e., that all-pairs D_1-Reachability is solvable in $\tilde{O}(n^\omega)$ time.

Proof of Theorem 3.5. We first argue about the correctness of D_1-ReachAlgo. It follows immediately from the correctness of the routine BellReachAlgo that if D_1-ReachAlgo returns that y is D-reachable from x then there is a path y is D-reachable from x in G. Here we focus on the inverse direction, i.e., assume that there is a path $P\colon x \rightsquigarrow y$ in G with $\lambda(P) \in \mathcal{D}_1$, and we argue that D_1-ReachAlgo returns that y is D-reachable from x. Recall that $\gamma_i(x, y)$ is the maxima-distance from x to y in the graph G_i constructed by the algorithm in the i-th iteration. We have

$$\gamma_1(x, y) = \gamma(x, y) \leq \delta(x, y) \leq L\,,$$

where the last inequality follows from our choice of L as an upper-bound on $\delta(x, y)$. By Lemma 6.5, we have $\gamma_{i+1}(x, y) \leq \gamma_i(x, y)/2$ for each $i \in [\lceil \log L \rceil]$, hence after $i = \lceil \log L \rceil$ iterations, we have $\gamma_i(x, y) = 1$. Thus, in the last iteration of the algorithm, y is bell-shape-reachable from x, and by the correctness of the routine BellReachAlgo (Lemma 6.2), BellReachAlgo will return that y is D-reachable from x. Thus D_1-ReachAlgo will return that y is D-reachable from x in G, as desired.

We now turn our attention to the complexity of BellReachAlgo. The algorithm performs $O(\log L)$ invocations to the routine BellReachAlgo. In each invocation, BellReachAlgo performs $O(\log L) = O(\log n)$ transitive closure operations on graphs with $O(n)$ nodes. Using fast BMM [Munro 1971], each transitive closure takes $O(n^\omega)$ time. The total running time of D_1-ReachAlgo is $O(n^\omega \cdot \log^2 L) = \tilde{O}(n^\omega)$, as by Lemma 6.1, we have $L = O(n^2)$.

The desired result follows. □

A comparison note with [Bradford 2018]. A sub-cubic bound for D_1-Reachability was recently established independently in [Bradford 2018]. The crux of that algorithm is an elegant algebraic matrix encoding of *flat* D_1-Reachability to $O(\log n)$ AGMY matrix multiplications, each performed in $O(n^\omega \cdot \log n)$ time [Alon et al. 1997]. Intuitively, flat D_1-Reachability concerns reachability witnessed by sequentially composing bell-shaped paths, and the above reduction gives a $O(n^\omega \cdot \log^2 n)$ bound for the problem. A second step solves flat D_1-Reachability for $O(\log n)$ iterations, with some special treatment needed in each iteration for ensuring correct AGMY representation. Composing the two steps yields a $O(n^\omega \cdot \log^3 n)$ bound for D_1-Reachability [Bradford 2018, Theorem 2]. In comparison, the algorithm presented here is a $\log n$-factor faster, and relies on a purely combinatorial reduction to $O(\log n^2)$ BMMs, which are then performed in $O(n^\omega)$ time using algebraic techniques.

6.2 Bounded Andersen's Pointer Analysis in Sub-cubic Time

In the previous section we saw that D_1-Reachability can be solved in nearly BMM time, i.e., $\tilde{O}(n^\omega)$. In this section we show how we can use this result to speed-up bounded Exhaustive APA, towards Theorem 3.3. Recall that, given some $i \in [4]$ and $j \in \mathbb{N}$, the (i, j)-bounded APA asks to compute

all memory locations b that a pointer a may point to, as witnessed by straight-line programs (under the operational semantics of Table 1) that use statements of type i at most j times. We start with a simple lemma that allows us to consider instances of APA which contain only linearly many statements of type 4.

Lemma 6.6. *Wlog, we have* $|S_4| \leq n$.

Proof. Consider any pointer $a \in A$ such that we have statements $\{*a = b_i\}_i$ in S_4. We introduce a new pointer c, and (i) we insert a new type-4 statement $*a = c$, and (ii) we replace each $*a = b_i$ statement with $c = b_i$. Performing the above process for each a, we create a new instance (A', S') such that for every $a, b \in A$, we have $b \in [\![a]\!]$ in (A, S) iff the same holds in (A', S'). Finally note that $|A'| \leq 2 \cdot |A|$ and $|S'| \leq |S| + |A|$, while $|S'_4| \leq |A'|$ as now every pointer appears in the left-hand side of a type 4 statement at most once.

The desired result follows. □

Algorithm BoundedAPAAlgo**.** We now present our algorithm BoundedAPAAlgo which solves$(4, j)$ Exhaustive APA for an instance (A, S) and some given $j \geq 0$. The algorithm performs $j + 1$ iterations of D_1-Reachability on graphs $G_i = (A, E_i)$, for $i \in [j + 1]$, where initially G_1 is the Dyck graph in the representation (G_1, S_4) of the APA instance (A, S). In iteration i, the algorithm solves D_1-Reachability in G_i, and then computes all pointers c that flows into some pointer a in G_i for which there is a statement $*a = b$ in S_4. Then the algorithm resolves the statement by inserting an edge (b, c, ϵ) in G_{i+1}. See Algorithm 4 for a detailed description. We conclude this section with the proof of Theorem 3.3.

Algorithm 4: BoundedAPAAlgo

Input: An instance (A, S) of APA, a bound j on statements of type 4
Output: A set (a, b) of all points-to relationships $b \in [\![a]\!]$ witnessed by $(4, j)$-bounded programs.

```
// Initialization
// |S_4| ≤ n wlog
Let (G_1 = (A, E_1), S_4) be the Dyck-graph representation of (A, S)
Let V = {a_1, a_2 : a ∈ A} be a node set
// Computation
foreach i ∈ [j + 1] do
    Solve D_1-Reachability in G_i using D_1-ReachAlgo
    Let Z_i^1 = {(a_1, b_2) : b = &a is a statement in S}
    Let Z_i^2 = {(a_2, b_2) : b is D-reachable from a in G_i}
    Solve all-pairs reachability in H_i = (V, Z_i^1 ∪ Z_i^2)
    Let E_{i+1} = E_i
    foreach statement *a = b in S_4 do
        foreach c ∈ A with c_1 ⇝ a_2 in H_i do
            Insert (b, c, ε) in E_{i+1}
        end
    end
    Construct the graph G_{i+1} = (A, E_{i+1})
end
return {a, b : b_1 ⇝ a_2 in H_{j+1}}
```

Proof of Theorem 3.3. The correctness follows directly from the correctness of D_1-ReachAlgo (Theorem 3.5). By induction, at the end of iteration i, BoundedAPAAlgo has solved $(4, i-1)$-bounded Exhaustive APA, hence at the end of iteration $j+1$ the algorithm has solved $(4, j)$-bounded Exhaustive APA.

We now turn our attention to complexity. In each iteration of the main loop in Line 3, we have an invocation to D_1-ReachAlgo in Line 4, which, by Theorem 3.5, takes $\tilde{O}(n^\omega)$ time. In addition, the all-pairs reachability in Line 7 can be performed using fast BMM [Munro 1971] in $O(n^\omega)$ time. Finally, by Lemma 6.6, the loop in Line 9 is executed at most n times, while the inner loop in Line 10 is clearly executed at most n times as well. Hence, in each iteration, the running time is dominated by the invocation to D_1-ReachAlgo, and thus the total time for all iterations is $\tilde{O}(n^\omega \cdot j)$.

The desired result follows. □

Impact of bounding type-4 statements. Theorem 3.3 targets $(4, j)$-bounded Exhaustive APA which limits the number of applications of type-4 statements. This bounding might miss points-to relationships created by repeatedly nested aliasing. In practice, the level of indirection is typically small, and thus we expect the above algorithm to be relatively complete. Given the algorithmic benefits of this approach, an experimental evaluation of its precision is interesting future work.

7 THE QUADRATIC HARDNESS OF BOUNDED ANDERSEN'S POINTER ANALYSIS

In this section we continue to study bounded APA and prove Theorem 3.4, i.e., if we restrict our attention to points-to relationships as witnessed by programs of length $\tilde{O}(1)$, even the on-demand problem has a quadratic (conditional) lower bound. Our reduction is from the problem of Orthogonal Vectors [Williams 2019].

Orthogonal Vectors (OV). The input to the problem is two sets X, Y, each containing n' vectors in $\{0,1\}^D$, for some dimension $D = \omega(\log n')$. The task is to determine if there exists a pair $(x, y) \in (X \times Y)$ that is orthogonal, i.e., for each $j \in D$, we have $x[j] \cdot y[j] = 0$. The respective hypothesis states that the problem cannot be solved in time $O(n'^{2-\epsilon})$, for any fixed $\epsilon > 0$.

Reduction from OV. Consider an instance X, Y of OV, and assume wlog that D is even. Here we show how to construct a On-demand APA instance with two distinguished pointers s and t such that $s \in [\![t]\!]$ iff there exists an orthogonal pair of vectors in $X \times Y$.

Intuition. We start with a high-level intuition, while Figure 7 provides an illustration. Consider the first two vectors $x_1 \in Y$ and $y_1 \in Y$, and focus on the first coordinate. Recall the representation of APA as a Dyck graph G and a set of type-4 statements S_4, and let $\overline{G}$ be the resolved Dyck graph. We introduce two pointers a_1^1 and u_1^1, with the goal that a_1^1 flows into u_1^1 in the resolved Dyck graph $\overline{G}$ iff $x_1[1] \cdot y_1[1] = 0$, i.e., x_1 and y_1 are orthogonal as far as the first coordinate is concerned. We achieve this by introducing a distinguished node z, and having two edges $a_1^1 \xrightarrow{\&} z$ and $z \xrightarrow{*} u_1^1$. Moreover, if $x_1[1] = 0$, we also create a path $P_1^x \colon a_1^1 \overset{\&\&}{\rightsquigarrow} z$. Similarly, if $y_1[1] = 0$, we also create a path $P_1^y \colon z \overset{\epsilon}{\rightsquigarrow} u_1^1$ (i.e., P_1^y is simply an ϵ-labeled edge). Observe that a_1^1 flows into u_1^1 iff $x_1[1] \cdot y_1[1] = 0$. If $x_1[1] \cdot y_1[1] = 1$ then nothing happens and the process stops here. Otherwise, x_1 and y_1 are potentially orthogonal, so we proceed with the second coordinate. We create a node v_1^1 and a type-4 statement $*u_1^1 = v_1^1$; since a_1^1 flows into u_1^1, the resolved graph $\overline{G}$ also has an edge $v_1^1 \xrightarrow{\epsilon} u_1^1$. The contents of x_1 and y_1 on the second coordinate are encoded via paths to new pointers a_2^1 and u_2^1, respectively. In particular, we have two edges $a_1^1 \xrightarrow{*} a_2^1$ and $u_2^1 \xrightarrow{\&} v_1^1$.

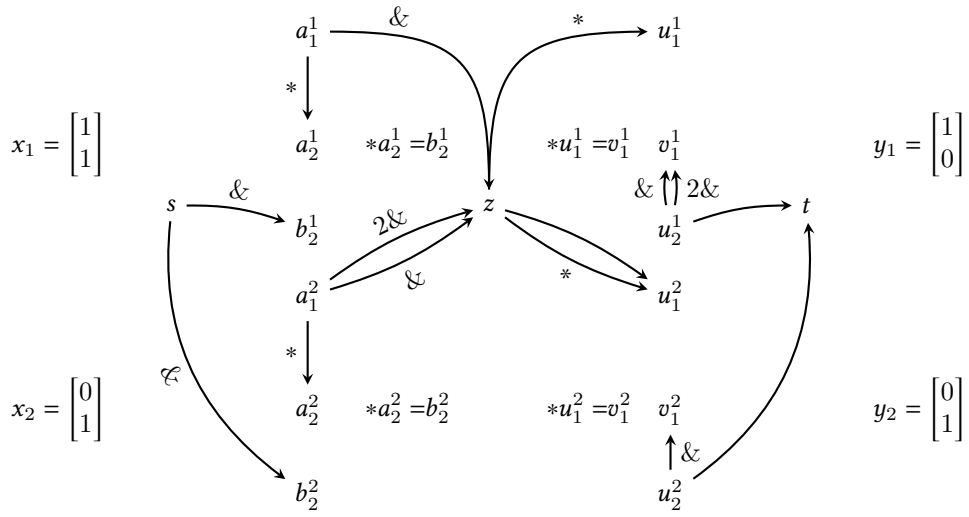

Fig. 7. Reduction from OV with vector sets $A = \{x_1, x_2\}$ and $B = \{y_1, y_2\}$ to $(\text{All}, O(\log n))$-bounded APA on the pair $s \in [\![t]\!]$.

Moreover, if $x_1[2] = 0$, we also create a path $P_2^x \colon a_1^1 \overset{\epsilon}{\rightsquigarrow} a_2^1$ (i.e., P_2^x is simply an ϵ-labeled edge). Similarly, if $y_1[2] = 0$, we also create a path $P_2^y \colon u_2^1 \overset{\&*}{\rightsquigarrow} v_1^1$. Observe that u_1^2 flows into a_2^1 iff $v_1^1 \xrightarrow{\epsilon} a_1^1$ (which is established by the previous step, as $x_1[1] \cdot y_1[1] = 0$) and also $x_1[2] \cdot y_1[2] = 0$, thereby establishing that x_1 and y_1 appear orthogonal on the first two coordinates. In that case, we have another pointer b_2^1 and type-4 statement $*a_2^1 = b_2^1$, which inserts an edge $b_2^1 \xrightarrow{\epsilon} u_2^1$ in $\overline{G}$. From here on, the process repeats as before, with b_2^1 playing the role of pointer z initially, and the contents of $x_1[3]$ captured in paths from $a_3^1 \rightsquigarrow b_2^1$, and the contents of $y_1[3]$ captured in paths $u_2^1 \rightsquigarrow u_3^1$. In the end, we have that u_D^1 is D-reachable from b_D^1 iff $x_1[j] \cdot y_1[j] = 0$ for all $j \in [D]$.

Finally, to capture all potential pairs of vectors $x_{i_1}, y_{i_2} \in X \times Y$, we connect the pointer z to all pointers $a_1^{i_1}$ and $u_1^{i_2}$, as above. To complete the reduction, we make $s \xrightarrow{\&} b_D^{i_1}$ and $u_D^{i_2} \xrightarrow{\epsilon} t$, and thus s flows into t via the D-reachable path $u_D^{i_1} \rightsquigarrow b_D^{i_2}$ iff x_{i_1} and y_{i_2} are orthogonal.

Formal construction We now proceed with the formal construction, as follows. First, we introduce a pointer z.

For every vector $x^i \in X$, we introduce pointers $a_1^i, \ldots, a_D^i$ and $b_2^i, b_4^i, \ldots, b_D^i$.

(1) We have $z = \&a_1^i$. If $x^i[1] = 0$, we also introduce a new pointer $\hat{a}_1^i$ and two assignments $z = \&\hat{a}_1^i$ and $\hat{a}_1^1 = \&a_1^i$.
(2) For every even $j \in [D]$, we have $*a_j^i = b_j^i$ and $a_j^i = *a_{j-1}^i$. If $x^i[j] = 0$, we also have $a_j^i = a_{j-1}^i$.
(3) For every odd $j \in [D]$ with $j > 1$, we have $b_{j-1}^i = \&a_j^i$. If $x^i[j] = 0$, we also introduce a new pointer $\hat{a}_j^i$ and two assignments $b_{j-1}^i = \&\hat{a}_j^i$ and $\hat{a}_j^i = \&a_j^i$.

For every vector $y^i \in Y$, we introduce pointers $u_1^i, \ldots, u_D^i$ and $v_1^i, v_3^i, \ldots, v_{D-1}^i$.

(1) We have $u_1^i = *z$. If $y^i[1] = 0$, we also have $u_1^i = z$.
(2) For every odd $j \in [D]$, we have $*u_j^i = v_j^i$ and $u_j^i = *u_{j-1}^i$. If $y^i[j] = 0$, we also have $u_j^i = u_{j-1}^i$.

(3) For every even $j \in [D]$, we have $v^i_{j-1} = \&u^i_j$. If $y^i[j] = 0$, we also introduce a new pointer $\hat{u}^i_j$ and two assignments $v^i_{j-1} = \&\hat{u}^i_j$ and $\hat{u}^i_j = \&u^i_j$.

Finally, we introduce two pointers s and t. For every $i \in [n']$, we have $b^i_D = \&s$ and $t = u^i_D$. The on-demand question is whether $s \in [\![t]\!]$. Observe that we have used $n = O(n' \cdot D)$ pointers, and the above construction can be easily carried out in $O(n)$ time.

Correctness. We now establish the correctness of the above construction. The key idea is as follows. Recall our definition of the Dyck-graph representation $(G = (A, E), S_4)$ of the APA instance, and the resolved Dyck graph $\overline{G}$ (see Section 2.2). The resolved graph $\overline{G}$ is constructed from G by iteratively (i) finding three pointers a, b, c such that a flows into b and we have a type 4 statement $*b = c$, and (ii) inserting an edge $c \xrightarrow{\epsilon} a$ in G. The above construction guarantees that, for two integers $i_1, i_2 \in [n']$, the following hold by induction on $j \in [D]$.

(1) If j is odd, we have $v^{i_2}_j \xrightarrow{\epsilon} a^{i_1}_j$ iff $\sum_{j' \le j} x^{i_1}[j'] \cdot y^{i_2}[j'] = 0$.
(2) If j is even, we have $b^{i_1}_j \xrightarrow{\epsilon} u^{i_2}_j$ iff $\sum_{j' \le j} x^{i_1}[j'] \cdot y^{i_2}[j'] = 0$.

Once such an ϵ-labeled edge is inserted for some j, it creates a path that leads to a flows-into relationship that leads to inserting the next ϵ-labeled edge for $j + 1$ iff $x^{i_1}[j+1] \cdot y^{i_2}[j+1] = 0$. Note that s flows into t iff there exist $i_1, i_2 \in [n']$ such that $b^D_{i_1} \xrightarrow{\epsilon} u^D_{i_2}$, which, by the above, holds iff x^{i_1} and y^{i_2} are orthogonal. Finally, since $D = \Theta(\log n)$, the witness program for $s \in [\![t]\!]$ has length $\tilde{O}(1)$.

The above idea is formally captured in the following two lemmas

Lemma 7.1. *If there exist $i_1, i_2 \in [n']$ such that x^{i_1} and y^{i_2} are orthogonal, then s flows into t in $\overline{G}$. Moreover, there exists a witness program $\mathcal{P}$ of length $O(\log n)$ that results in $s \in [t]$.*

Lemma 7.2. *If s flows into t in $\overline{G}$, there exist $i_1, i_2 \in [n']$ such that x^{i_1} and y^{i_2} are orthogonal.*

We conclude this section with the proof of Theorem 3.4.

Proof of Theorem 3.4. Lemma 7.1 and Lemma 7.2, together with Lemma 2.2, state the correctness of the reduction. Note that the APA instance we constructed has $n = O(n' \cdot D)$ pointers and $m = O(n' \cdot D)$ assignments, hence it is a sparse instance. Moreover the time for the construction is $O(n' \cdot D)$. Assume that there exists some fixed $\epsilon > 0$ such that On-demand APA can be solved in $O(n^{2-\epsilon})$ time. Then we have a solution for the OV instance in time $O((n' \cdot D)^{2-\epsilon})$ time, which violates the Orthogonal-Vectors hypothesis.

The desired result follows. □

8 THE PARALLELIZABILITY OF ANDERSEN'S POINTER ANALYSIS

In this section we address the parallelizability of APA, and show the following results. In Section 8.1 we prove Theorem 3.6, which shows that APA is not parallelizable under standard hypotheses in complexity theory. In Section 8.2 we prove Theorem 3.7, which shows that bounded APA is efficiently parallelizable as long as we focus on poly-logarithmically (i.e., $O(\log^c n)$, for some constant c) many applications of type 4 statements.

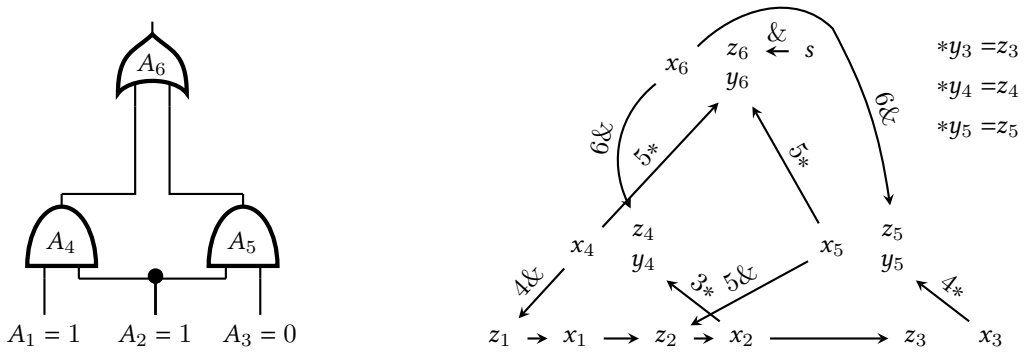

Fig. 8. A Monotone CVP instance (left) and the corresponding On-demand APA instance for $s \in [\![x_6]\!]$? (right). Squiggly arrows represent paths of unique nodes, where the path has the corresponding path label.

8.1 Andersen's Pointer Analysis Is Not Parallelizable

In this section we prove Theorem 3.6, i.e., that On-demand APA is P-complete. This implies that the problem is unlikely to be parallelizable. Our reduction is from Monotone CVP.

The problem Monotone CVP**.** The input to the problem is a circuit represented as a sequence of assignments $(A_1, \ldots, A_{n'})$, such that for all $i \in [n']$, A_i has one of the following types, where $j < k < i$.

$$A_i = 0 \qquad A_i = 1 \qquad A_i = A_j \wedge A_k \qquad A_i = A_j \vee A_k$$

The condition $j < k < i$ ensures acyclicity. The assignments $A_i = 0$ and $A_i = 1$ are the *inputs*, whereas all other assignments are the *gates*. The task is to compute whether $A_{n'}$ evaluates to 1 under the standard Boolean algebra interpretation of the operators $\wedge$ and $\vee$. It is known that Monotone CVP is P-complete even when every assignment has fan-out 2 (except $A_{n'}$ and inputs A_i) [Greenlaw et al. 1995].

Reduction from Monotone CVP**.** Consider an instance $(A_1, \ldots, A_{n'})$ of Monotone CVP, and we construct an instance (A, S) of On-demand APA. An illustration is given in Figure 8.

(1) For every input A_i, we introduce two pointers $x_i, z_i \in A$. We have a statement $x_i = z_i$ in S iff $A_i = 1$.
(2) For every gate $A_i = A_j \wedge A_k$, with $j < k < i$, we introduce (i) three pointers x_i, y_i, z_i, (ii) i pointers $x_i^1, \ldots, x_i^i$, and (iii) $i-1$ pointers $y_i^1, \ldots, y_i^{i-1}$. We have the following statements in S.

$$x_i^1 = \&x_i \qquad x_i^2 = \&x_i^1 \qquad \cdots \qquad x_i^i = \&x_i^{i-1} \qquad z_j = \&x_i^i \qquad \text{and}$$
$$y_i^{i-1} = *y_k \qquad y_i^{i-2} = *y_i^{i-1} \qquad \cdots \qquad y_i^1 = *y_i^2 \qquad y_i = *y_i^1$$

For every gate $A_i = A_j \vee A_k$, with $j < k < i$, we introduce (i) three pointers x_i, y_i, z_i, (ii) i pointers $x_i^1, \ldots, x_i^i$ and i pointers $\hat{x}_i^1, \ldots, \hat{x}_i^i$, and (iii) $i-1$ pointers $y_i^1, \ldots, y_i^{i-1}$ and $i-1$ pointers

$\hat{y}_i^1, \ldots, \hat{y}_i^{i-1}$. We have the following statements in S.

$$x_i^1 = \&x_i \quad x_i^2 = \&x_i^1 \quad \cdots \quad x_i^i = \&x_i^{i-1} \quad z_j = \&x_i^i \quad \text{and}$$
$$\hat{x}_i^1 = \&x_i \quad \hat{x}_i^2 = \&\hat{x}_i^1 \quad \cdots \quad \hat{x}_i^i = \&\hat{x}_i^{i-1} \quad z_k = \&\hat{x}_i^i \quad \text{and}$$
$$y_i^{i-1} = *y_j \quad y_i^{i-2} = *y_i^{i-1} \quad \cdots \quad y_i^1 = *y_i^2 \quad y_i = *y_i^1 \quad \text{and}$$
$$\hat{y}_i^{i-1} = *\hat{y}_k \quad \hat{y}_i^{i-2} = *\hat{y}_i^{i-1} \quad \cdots \quad \hat{y}_i^1 = *\hat{y}_i^2 \quad y_i = *\hat{y}_i^1$$

(3) For every gate A_i, we add a statement $*y_i = z_i$.
(4) Finally, we introduce a special pointer s, and a statement $z_{n'} = \&s$. The on-demand question is whether $s \in [\![x_{n'}]\!]$.

Note that the size of our APA instance is $\Theta(n'^2)$ i.e., the construction leads to a quadratic blow-up. Nevertheless, it is easy to carry out the above construction in logarithmic space, as required for a log-space reduction.

Correctness. We now establish the correctness of the reduction. We first present an intuitive insight and then the formal steps of the proof.

Intuitive argument of correctness. Let $(G = (A, E), S_4)$ be the Dyck-graph representation of the constructed APA instance (A, S), and let $\overline{G}$ be the resolved Dyck graph (see Section 2.2). Recall that the resolved graph $\overline{G}$ is constructed from G by iteratively (i) finding three pointers a, b, c such that a flows into b and we have a type 4 statement $*b = c$, and (ii) inserting an edge $c \xrightarrow{\epsilon} a$ in G.

The correctness of the construction relies on the following invariant. For every i such that A_i is a gate with inputs A_j, A_k, we have that x_i is D-reachable from z_i iff (i) $z_j \xrightarrow{\epsilon} x_j$ and $z_k \xrightarrow{\epsilon} x_k$ (if A_i is an AND gate), or (ii) $z_j \xrightarrow{\epsilon} x_j$ or $z_k \xrightarrow{\epsilon} x_k$ (if A_i is an OR gate). Observe that in $\overline{G}$ we have potentially other nodes u that are D-reachable from z_i. For example, in Figure 8, we have $z_4 \xrightarrow{\epsilon} x_5^1$ in $\overline{G}$, where x_5^1 is the second node in the path $x_5 \overset{5\&}{\rightsquigarrow} z_2$ (not explicitly shown in the figure). This occurs because we have a statement $*y_4 = z_4$, and x_5^1 flows into y_4. The key insight is then that the invariant holds despite such "unwanted" edges. Given the invariant, the correctness proof follows by an induction on the depth of the circuit.

Formal Correctness. We now make the above insights formal. We start with a technical lemma, which shows that if x_i is D-reachable from z_i in $\overline{G}$, then, in fact, $z_i \to x_i$, as x_i flows into y_i and we have a statement $*y_i = z_i$.

Lemma 8.1. *For all $i \in [n']$, x_i is D-reachable from z_i in $\overline{G}$ iff x_i flows into y_i in $\overline{G}$.*

The following lemma establishes our main invariant, which relates the encoding of the output of a gate in $\overline{G}$ to the encoding of its inputs.

Lemma 8.2. *Consider a gate A_i. We have that x_i is D-reachable from z_i in $\overline{G}$ iff*

(1) A_i is an AND gate $A_i = A_j \wedge A_k$, and x_j is D-reachable from z_j and x_k is D-reachable from z_k, or
(2) A_i is an OR gate $A_i = A_j \vee A_k$, and x_j is D-reachable from z_j or x_k is D-reachable from z_k.

The following lemma establishes the correctness of the construction. Its proof follows by an induction on the depth of the circuit, and using Lemma 8.2 on the respective gate.

Lemma 8.3. *We have that $x_{n'}$ is D-reachable from $z_{n'}$ iff $A_{n'}$ evaluates to* 1.

We conclude this section with the proof of Theorem 3.6.

Proof of Theorem 3.6. Membership in P is known (e.g., Theorem 3.1), so we need to argue that the problem is P-hard. Observe that s flows into $x_{n'}$ iff $x_{n'}$ is D-reachable from $z_{n'}$. By Lemma 8.3, we have that s flows into $x_{n'}$ iff $A_{n'}$ evaluates to 1. By Lemma 2.2, we have that $s \in [\![x_{n'}]\!]$ iff $A_{n'}$ evaluates to 1. The desired result follows. □

8.2 Bounded Andersen's Pointer Analysis Is Parallelizable

Finally, in this section we develop an algorithm for solving bounded Exhaustive APA, and thus prove Theorem 3.7.

Parallel bounded APA**.** The algorithm is a parallelization of BoundedAPAAlgo (Algorithm 4) for sequential bounded APA. The parallel algorithm performs the iterations of the main loop of Line 3 sequentially, while the body of the loop is run in parallel. We outline the steps of the parallelization.

(1) In Line 4, BoundedAPAAlgo invokes the routine D_1-ReachAlgo for computing D_1-Reachability (Algorithm 3). Recall, by Lemma 6.1, that the distance between two reachable nodes is $O(n^2)$, and hence the same bound holds for the maximum stack height of the shortest path that witnesses reachability between such nodes. It follows that D_1-Reachability can be reduced to standard graph reachability on a graph with $O(n^3)$ nodes of the form (u, i), where u is a node of the Dyck graph and $i \in [O(n^2)]$ encodes the stack height. Finally, graph reachability is solved in parallel [Papadimitriou 1993].
(2) In Line 7, BoundedAPAAlgo constructs another graph H and solves all-pairs reachability on H. Now, we perform the construction of H in parallel, while all-pairs reachability in H is also computed in parallel, as in the previous item.
(3) Finally, we execute the two nested loops in Line 9 and Line 10 in parallel, by using one processor per triplet of pointers (a, b, c).

The correctness of the parallelization follows directly from the correctness of the sequential version (Theorem 3.3). We conclude with the complexity analysis, which establishes Theorem 3.7.

Proof of Theorem 3.7. In each iteration, the parallel running time is the time required to compute the transitive closure of a graph of $O(n^3)$ nodes. Since graph reachability is in NC^2 [Papadimitriou 1993], each iteration takes $O(\log^2 n)$ time. Hence, executing the main loop sequentially for j iterations yields $O(j \cdot \log^2 n)$ time. In particular, for $j = \log^i n$ iterations, the algorithm solves $O(4, \log^i n)$-bounded Exhaustive APA in $O(\log^{i+2} n)$ parallel time, hence the problem is in NC^{i+2}.

The desired result follows. □

9 CONCLUSION

Andersen's Pointer Analysis is a standard approach to static, flow-insensitive pointer analysis. Despite its long history and practical importance, the complexity of the analysis had remained illusive. In this work, we have drawn a rich fine-grained and parallel complexity landscape based on various aspects of the problem. We have shown that even deciding whether a single pointer may point to a specific heap location is unlikely to have sub-cubic complexity, and additionally, the problem is not parallelizable. These results strongly characterize the hardness of the problem. On

the positive side, we have presented a bounded version of the problem that becomes solvable in nearly matrix-multiplication time, and have established a conditional quadratic lower bound for the bounded version of the problem.

Our positive results build some stable ground for further practical improvements for Andersen's pointer analysis. We expect that our solution to the bounded version of the problem, which essentially reduces to a small number of standard transitive closure operations, can provide the basis for faster practical approaches: graph transitive closure solvers have been heavily optimized over the years in both software and hardware, while the full parallelizability of the bounded problem opens itself up to more efficient multithreaded implementations. As our focus in this work has been on characterizing the tractability landscape of the problem, we have left the practical realizations of our results for interesting future work.

ACKNOWLEDGMENTS

We wish to thank Phillip G. Bradford for bringing to our attention his result on D_1-Reachability and for helpful comments in comparing his algorithm to ours, as well as anonymous reviewers for their constructive feedback in an earlier version of this manuscript.

REFERENCES

2003. T. J. Watson Libraries for Analysis (WALA). https://github.com. (2003).

Amir Abboud and Virginia Vassilevska Williams. 2014. Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems. In *FOCS*. 434–443.

Alexander Aiken, Manuel Fahndrich, and Jeffrey S Foster. 1997. *Flow-Insensitive Points-to Analysis with Term and Set Constraints.* Technical Report. EECS UC Berkeley.

Noga Alon, Zvi Galil, and Oded Margalit. 1997. On the Exponent of the All Pairs Shortest Path Problem. *J. Comput. System Sci.* 54, 2 (1997), 255 – 262. https://doi.org/10.1006/jcss.1997.1388

Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Ph.D. Dissertation.

N. Bansal and R. Williams. 2009. Regularity Lemmas and Combinatorial Algorithms. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*. 745–754.

Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. 2003. Points-to Analysis Using BDDs. *SIGPLAN Not.* 38, 5 (May 2003), 103–114. https://doi.org/10.1145/780822.781144

Sam Blackshear, Bor-Yuh Evan Chang, Sriram Sankaranarayanan, and Manu Sridharan. 2011. The Flow-Insensitive Precision of Andersen's Analysis in Practice. In *Proceedings of the 18th International Conference on Static Analysis (SAS'11)*. Springer-Verlag, Berlin, Heidelberg, 60–76.

Thorsten Blaß and Michael Philippsen. 2019. GPU-Accelerated Fixpoint Algorithms for Faster Compiler Analyses. In *Proceedings of the 28th International Conference on Compiler Construction (CC 2019)*. Association for Computing Machinery, New York, NY, USA, 122–134. https://doi.org/10.1145/3302516.3307352

Mark Boddy. 1991. Anytime Problem Solving Using Dynamic Programming. In *Proceedings of the Ninth National Conference on Artificial Intelligence - Volume 2 (AAAI'91)*. AAAI Press, 738–743.

Phillip G. Bradford. 2018. Efficient Exact Paths For Dyck and semi-Dyck Labeled Path Reachability. (2018). arXiv:cs.DS/1802.05239

Venkatesan T. Chakaravarthy. 2003. New Results on the Computability and Complexity of Points–to Analysis. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. Association for Computing Machinery, New York, NY, USA, 115–125. https://doi.org/10.1145/604131.604142

Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. 2018. Optimal Dyck Reachability for Data-Dependence and Alias Analysis. *Proc. ACM Program. Lang.* 2, POPL, Article Article 30 (Dec. 2018), 30 pages.

Krishnendu Chatterjee, Wolfgang Dvorák, Monika Henzinger, and Veronika Loitzenbauer. 2016. Conditionally Optimal Algorithms for Generalized Büchi Games. In *41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Piotr Faliszewski, Anca Muscholl, and Rolf Niedermeier (Eds.), Vol. 58. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 25:1–25:15.

Krishnendu Chatterjee, Andreas Pavlogiannis, and Yaron Velner. 2015. Quantitative Interprocedural Analysis. *SIGPLAN Not.* 50, 1 (Jan. 2015), 539–551. https://doi.org/10.1145/2775051.2676968

Manuvir Das. 2000. Unification-Based Pointer Analysis with Directional Assignments. *SIGPLAN Not.* 35, 5 (May 2000), 35–46. https://doi.org/10.1145/358438.349309

Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. 2001. Estimating the Impact of Scalable Pointer Analysis on Optimization. In *Static Analysis*, Patrick Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 260–278.

Nirav Dave, Kermin Fleming, Myron King, Michael Pellauer, and Muralidaran Vijayaraghavan. 2007. Hardware Acceleration of Matrix Multiplication on a Xilinx FPGA. In *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE '07)*. IEEE Computer Society, USA, 97–100.

Jean-Luc Deleage and Laurent Pierre. 1986. The Rational Index of the Dyck Language D1. *Theor. Comput. Sci.* 47, 3 (Nov. 1986), 335–343.

Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. 2015. Giga-Scale Exhaustive Points-to Analysis for Java in under a Minute. *SIGPLAN Not.* 50, 10 (Oct. 2015), 535–551. https://doi.org/10.1145/2858965.2814307

Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. 1998. Partial Online Cycle Elimination in Inclusion Constraint Graphs. *SIGPLAN Not.* 33, 5 (May 1998), 85–96. https://doi.org/10.1145/277652.277667

Rakesh Ghiya, Daniel Lavery, and David Sehr. 2001. On the Importance of Points-to Analysis and Other Memory Disambiguation Methods for C Programs. *SIGPLAN Not.* 36, 5 (May 2001), 47–58. https://doi.org/10.1145/381694.378806

Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. 1995. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, Inc., USA.

Ben Hardekopf and Calvin Lin. 2007. The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code. *SIGPLAN Not.* 42, 6 (June 2007), 290–299. https://doi.org/10.1145/1273442.1250767

Ben Hardekopf and Calvin Lin. 2011. Flow-Sensitive Pointer Analysis for Millions of Lines of Code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, USA, 289–298.

Nevin Heintze and David McAllester. 1997. On the Cubic Bottleneck in Subtyping and Flow Analysis. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS '97)*. IEEE Computer Society, Washington, DC, USA, 342–. http://dl.acm.org/citation.cfm?id=788019.788876

Nevin Heintze and Olivier Tardieu. 2001a. Demand-Driven Pointer Analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 24–34. https://doi.org/10.1145/378795.378802

Nevin Heintze and Olivier Tardieu. 2001b. Ultra-Fast Aliasing Analysis Using CLA: A Million Lines of C Code in a Second. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 254–263. https://doi.org/10.1145/378795.378855

Nevin Charles Heintze. 1992. *Set Based Program Analysis*. Ph.D. Dissertation. USA.

Michael Hind. 2001. Pointer Analysis: Haven'T We Solved This Problem Yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*. ACM, 54–61.

Martin Hirzel, Amer Diwan, and Michael Hind. 2004. Pointer Analysis in the Presence of Dynamic Class Loading. In *ECOOP 2004 – Object-Oriented Programming*, Martin Odersky (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 96–122.

Susan Horwitz. 1997. Precise Flow-Insensitive May-Alias Analysis is NP-Hard. *ACM Trans. Program. Lang. Syst.* 19, 1 (Jan. 1997), 1–6. https://doi.org/10.1145/239912.239913

Jianyu Huang, Tyler M. Smith, Greg M. Henry, and Robert A. van de Geijn. 2016. Strassen's Algorithm Reloaded. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Article Article 59, 12 pages.

Steven Huss-Lederman, Elaine M. Jacobson, Anna Tsao, Thomas Turnbull, and Jeremy R. Johnson. 1996. Implementation of Strassen's Algorithm for Matrix Multiplication. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (Supercomputing '96)*. IEEE Computer Society, USA, 32–es. https://doi.org/10.1145/369028.369096

Dongseok Jang and Kwang-Moo Choe. 2009. Points-to Analysis for JavaScript. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC '09)*. Association for Computing Machinery, New York, NY, USA, 1930–1937.

Igor Kaporin. 1999. A practical algorithm for faster matrix multiplication. *Numerical Linear Algebra with Applications* 6, 8 (1999), 687–700.

John Kodumal and Alex Aiken. 2004. The Set Constraint/CFL Reachability Connection in Practice. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*. Association for Computing Machinery, New York, NY, USA, 207–218. https://doi.org/10.1145/996841.996867

Julian Laderman, Victor Pan, and Xuan-He Sha. 1992. On practical algorithms for accelerated matrix multiplication. *Linear Algebra Appl.* 162-164 (1992), 557 – 588. https://doi.org/10.1016/0024-3795(92)90393-O

William Landi and Barbara G. Ryder. 1991. Pointer-induced Aliasing: A Problem Classification. In *POPL*. ACM.

François Le Gall. 2014. Powers of Tensors and Fast Matrix Multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation (ISSAC)*. 296–303.

Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction (CC'03)*. Springer-Verlag, Berlin, Heidelberg, 153–169.

Bozhen Liu, Jeff Huang, and Lawrence Rauchwerger. 2019. Rethinking Incremental and Parallel Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 41, 1, Article Article 6 (March 2019), 31 pages. https://doi.org/10.1145/3293606

Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. 2013. An Incremental Points-to Analysis with CFL-Reachability. In *Proceedings of the 22nd International Conference on Compiler Construction (CC'13)*. Springer-Verlag, Berlin, Heidelberg, 61–81. https://doi.org/10.1007/978-3-642-37051-9_4

Steven Lyde, William E. Byrd, and Matthew Might. 2015. Control-Flow Analysis of Dynamic Languages via Pointer Analysis. *SIGPLAN Not.* 51, 2 (Oct. 2015), 54–62. https://doi.org/10.1145/2936313.2816712

Anders Alnor Mathiasen and Andreas Pavlogiannis. 2020. The Fine-Grained and Parallel Complexity of Andersen's Pointer Analysis. (2020). arXiv:cs.PL/2006.01491

David McAllester. 1999. On the Complexity Analysis of Static Analyses. In *Static Analysis*, Agostino Cortesi and Gilberto Filé (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 312–329.

David Melski and Thomas Reps. 2000. Interconvertibility of a Class of Set Constraints and Context-free-language Reachability. *Theor. Comput. Sci.* 248, 1-2 (Oct. 2000), 29–98. https://doi.org/10.1016/S0304-3975(00)00049-9

Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. 2012. A GPU Implementation of Inclusion-Based Points-to Analysis. *SIGPLAN Not.* 47, 8 (Feb. 2012), 107–116. https://doi.org/10.1145/2370036.2145831

Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. 2010. Parallel Inclusion-Based Points-to Analysis. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. Association for Computing Machinery, New York, NY, USA, 428–443. https://doi.org/10.1145/1869459.1869495

Anders Møller and Michael I. Schwartzbach. 2018. *Static Program Analysis.* Technical Report. Department of Computer Science, Aarhus University. http://cs.au.dk/~amoeller/spa/

Ian Munro. 1971. Efficient determination of the transitive closure of a directed graph. *Inform. Process. Lett.* 1, 2 (1971), 56 – 58.

Christos H. Papadimitriou. 1993. *Computational Complexity.* Addison-Wesley.

David J. Pearce, Paul H. J. Kelly, and Chris Hankin. 2004. Online Cycle Detection and Difference Propagation: Applications to Pointer Analysis. *Software Quality Journal* 12, 4 (2004), 311–337. https://doi.org/10.1023/B:SQJO.0000039791.93071.a2

Edgar Pek and P. Madhusudan. 2014. Explicit and Symbolic Techniques for Fast and Scalable Points-to Analysis. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis (SOAP '14)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/2614628.2614632

G. Ramalingam. 1994. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.* 16, 5 (Sept. 1994), 1467–1471.

Thomas Reps. 1996. On the sequential nature of interprocedural program-analysis problems. *Acta Informatica* 33, 5 (01 Aug 1996), 739–757. https://doi.org/10.1007/BF03036473

Atanas Rountev and Satish Chandra. 2000. Off-Line Variable Substitution for Scaling Points-to Analysis. *SIGPLAN Not.* 35, 5 (May 2000), 47–56. https://doi.org/10.1145/358438.349310

Marc Shapiro and Susan Horwitz. 1997. Fast and Accurate Flow-Insensitive Points-to Analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/263699.263703

Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (April 2015), 1–69. https://doi.org/10.1561/2500000014

Manu Sridharan and Stephen J. Fink. 2009. The Complexity of Andersen's Analysis in Practice. In *Proceedings of the 16th International Symposium on Static Analysis (SAS '09)*. Springer-Verlag, Berlin, Heidelberg, 205–221.

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven Points-to Analysis for Java. In *OOPSLA*.

Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. Association for Computing Machinery, New York, NY, USA, 32–41. https://doi.org/10.1145/237721.237727

Volker Strassen. 1969. Gaussian Elimination is Not Optimal. *Numer. Math.* 13, 4 (1969), 354–356.

Yu Su, Ding Ye, and Jingling Xue. 2014. Parallel Pointer Analysis with CFL-Reachability. In *Proceedings of the 2014 Brazilian Conference on Intelligent Systems (BRACIS '14)*. IEEE Computer Society, USA, 451–460. https://doi.org/10.1109/ICPP.2014.54

Zhendong Su, Manuel Fähndrich, and Alexander Aiken. 2000. Projection Merging: Reducing Redundancies in Inclusion Constraint Graphs. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*. Association for Computing Machinery, New York, NY, USA, 81–95. https://doi.org/10.1145/325694.325706

Yulei Sui and Jingling Xue. 2016. On-Demand Strong Update Analysis via Value-Flow Refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 460–473. https://doi.org/10.1145/2950290.2950296

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *CASCON '99*. IBM Press.

J. Vedurada and V. K. Nandivada. 2019. Batch Alias Analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 936–948.

Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A Single-Machine Disk-Based Graph System for Interprocedural Static Analyses of Large-Scale Systems Code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 389–404. https://doi.org/10.1145/3037697.3037744

John Whaley and Monica S. Lam. 2002. An Efficient Inclusion-Based Points-To Analysis for Strictly-Typed Languages. In *Static Analysis*, Manuel V. Hermenegildo and Germán Puebla (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 180–195.

Ryan Williams. 2005. A New Algorithm for Optimal 2-Constraint Satisfaction and Its Implications. *Theor. Comput. Sci.* 348, 2 (Dec. 2005), 357–365. https://doi.org/10.1016/j.tcs.2005.09.023

Virginia Vassilevska Williams. 2019. *On some fine-grained questions in algorithms and complexity.* Technical Report.

Virginia Vassilevska Williams and R. Ryan Williams. 2018. Subcubic Equivalences Between Path, Matrix, and Triangle Problems. *J. ACM* 65, 5, Article Article 27 (Aug. 2018), 38 pages. https://doi.org/10.1145/3186893

Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming (Genoa)*. 98–122.

Qirun Zhang. 2020. Conditional Lower Bound for Inclusion-Based Points-to Analysis. *arXiv preprint arXiv:2007.05569* (2020).

Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 2013. Fast Algorithms for Dyck-CFL-reachability with Applications to Alias Analysis *(PLDI)*. ACM.

Xin Zheng and Radu Rugina. 2008. Demand-driven Alias Analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, 197–208.