CFL/Dyck Reachability: An Algorithmic Perspective

Andreas Pavlogiannis, Aarhus University, Denmark

CFL/Dyck reachability is a simple graph-theoretic problem: given a CFL/Dyck language $\mathcal{L}$ over an alphabet Σ, a graph $G = (V, E)$ of Σ-labeled edges, and two distinguished nodes $s, t \in V$, does there exist a path from s to t that spells out a word in $\mathcal{L}$? This simple notion of language-based graph reachability serves as the algorithmic formulation of a large number of problems in diverse domains, such as graph databases and program static analysis. This paper takes an algorithmic perspective on CFL/Dyck reachability, and overviews several recent advances concerning the decidability and complexity of the problem and some its close variants, as realized in the areas of automata theory and program verification.

1. INTRODUCTION

CFL/Dyck reachability, and variants thereof, have played a key role in the practice of automated program analysis for several decades. In particular for static analyses, a directed graph $G = (V, E)$ serves as the program model, and paths in G capture control-flow/data-flow information in an over-approximate manner. As this over-approximation might be too coarse-grained to yield useful analysis results, the graph program model is typically enhanced with a language component $\mathcal{L}$ over some alphabet Σ; the latter also labels (some of) the edges of G. Now, paths in G are considered to represent valid program behavior only if the labels on their edges produce a word in $\mathcal{L}$. The language $\mathcal{L}$ is predominantly a CFL language, and often the Dyck language of properly-balanced parenthesis words, and is used to faithfully model programming constructs such as matching function calls with returns, or matching get- and set- field accesses in composite objects. An example illustrating the use of Dyck languages for modeling field accesses is given in Figure 1.

Once this graph-and-language theoretic modeling has taken place, the program analysis reduces to a language-respecting reachability question in G. E.g., a path from a node s to a node t that matches parentheses representing calling contexts is considered as a valid execution of the program from control node s to control node t while respecting function calls. Thus we have an instance of the CFL/Dyck reachability problem on G. This language-based tightening of the program model, by filtering out invalid paths not

```
1  ...
2  f.x=a;
3  b=f.y;
4  c=f.x;
5  ...
```

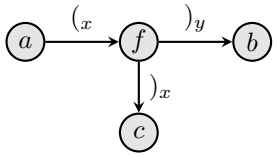

Fig. 1: Modeling program behavior with language graph reachability. Parentheses are used to match field accesses on the composite object f. The balanced path $a \xrightarrow{(_x} f \xrightarrow{)_x} c$ captures the fact that a flows into c. On the other hand, as the path $a \xrightarrow{(_x} f \xrightarrow{)_y} b$ is imbalanced, it does not witness the flow of a into b.

respecting $\mathcal{L}$, has found application in a truly numerous types of static analyses, such as dataflow analysis, alias and pointer analysis, slicing, and many others; we relegate our discussion on related work to Section 6. Naturally, as the problem is of vital importance to practitioners, it has been a subject of very active study, in two respects.

(1) *Algorithmic complexity*, as faster CFL/Dyck reachability algorithms yield faster static analyses.
(2) *Problem variations* that have different modeling power, thereby increasing the analysis precision, while sub-classes often allow for faster algorithms

In this paper we overview recent algorithmic results (both positive and negative) for the standard Dyck reachability problem, as well as some of its popular variants. Table I, Table II and Table III summarize the main results.

Table I: The complexity of Dyck reachability wrt the number of parenthesis symbols k, on graphs of n nodes and m edges. The upper bounds for $k \geq 1$ solve the all-pairs problem. BMM refers to Boolean Matrix Multiplication.

$\mathcal{D}_k$ reachability	Upper bound	Lower Bound
$k = 0$	$O(n+m)^{\dagger}$	$\Omega(n+m)^{\dagger}$
$k = 1$	$O(n^{\omega} \cdot \log^2 n)^{\ddagger}$	BMM-hard††
$k \geq 2$	$O(n^3/\log n)^{\ddagger\ddagger}$	2NPDA-hard†††

† This is plain graph reachability.
‡ [Mathiasen and Pavlogiannis 2021].
†† [Cetti Hansen et al. 2021].
‡‡ [Chaudhuri 2008].
††† [Heintze and McAllester 1997].

Table I concerns Dyck reachability, denoted $\mathcal{D}_k$ reachability, where k stands for the number of different parenthesis symbols over which the Dyck language is defined. The setting of $k = 0$ denotes plain graph reachability as studied in standard textbooks on algorithms, which can be solved in linear time. The case of $k \geq 2$ represents general Dyck reachability, which is solvable in $O(n^3/\log n)$ time and is known to be complete for the class of two-way nondeterministic pushdown automata (2NPDA-complete). The setting of $k = 1$ lies in between the two and corresponds to the computational model known as counter automata (or one-dimensional vector addition systems). This case was recently shown to be solvable in essentially matrix-multiplication time, and this bound is tight.

Table II: The complexity of Dyck reachability on bidirected graphs of n nodes and m edges. The upper bounds refer to the all-pairs problem.

$\mathcal{D}_k$ reachability†	Upper bound	Lower Bound
Worst-case	$O(m + n \cdot \alpha(n))$	$\Omega(m + n \cdot \alpha(n))$
Expected	$O(n+m)$	$\Omega(n+m)$

† All results from [Chatterjee et al. 2018].

Table II concerns Dyck reachability on a special class of graphs called *bidirected graphs*. These graphs have the property that all edges come in both directions and are labeled with complementary parenthesis symbols. This symmetry is reminiscent of undirect-

edness in plain graphs, and indeed turns reachability to an equivalence relation. This restriction was recently shown sufficient to reduce the algorithmic complexity of Dyck reachability down to almost linear, with matching lower bounds.

Table III: The complexity of interleaved Dyck reachability on graphs of n nodes and m edges.

$\mathcal{D}_k$ reachability	General	Bidirected
$\mathcal{D}_1 \odot \mathcal{D}_1$	NL-complete[†]	$O(n^3 \cdot \alpha(n))$[‡]
$\mathcal{D}_k \odot \mathcal{D}_1$	Decidability Open	PSPACE[††]
$\mathcal{D}_k \odot \mathcal{D}_k$	Undecidable[‡‡]	Undecidable[‡]

[†] [Englert et al. 2016].
[‡] [Kjelstrøm and Pavlogiannis 2022].
[††] [Ganardi et al. 2022].
[‡‡] [Reps 2000].

Finally, Table III concerns *interleaved* Dyck reachability. Here we have two Dyck languages, over different parenthesis alphabets, and reachability witnesses must respect both languages. This is a very expressive formalism, essentially corresponding to emptiness-testing for intersections of CFLs; as such the reachability problem is undecidable. However, the restriction of interleaved reachability to bidirected graphs was only recently explored, showing that the problem remains undecidable, but decidability is obtained if we also restrict one (or both) languages to single parentheses.

Next, we take a closer look at the results listed in Table I, Table II and Table III, as well as at some other related results.

2. PRELIMINARIES

General notation. Given a finite alphabet Σ, we denote by $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$. Given a natural number n, we let $[n] = \{1, \ldots, n\}$.

Edge-labeled graphs. We consider edge-labeled graphs $G = (V, E)$, where V is a set of nodes and $E \subseteq V \times V \times \Sigma_\epsilon$ is a set of edges, partially labeled with symbols from an alphabet Σ (unlabeled edges receive the special label ϵ). Hence, an edge e is of the form $e = (u, v, \lambda)$ where $u, v \in V$ and $\lambda \in \Sigma$ (sometimes written as $u \xrightarrow{\lambda} v$). Sometimes we are only interested on the endpoints of an edge e, in which case we represent $e = (u, v)$, and we often write $\Sigma(u, v) = \lambda$ to refer to the label λ of the edge (u, v). A *path* P is a sequence of edges $(e_1, \ldots, e_r)$ and each $e_i = (x_i, y_i, \lambda_i)$ is such that for all $1 \leq i \leq r - 1$, we have $y_i = x_{i+1}$. The length of P is $|P| = r$. We extend the edge labeling to paths, and denote by $\Sigma(P) = \Sigma(e_1)\Sigma(e_2) \ldots \Sigma(e_r)$ the label of P. We say that a node u is *reachable* from node v if there exists a path $P \colon u \rightsquigarrow v$.

Language reachability. Given some alphabet Σ, consider a Σ-labeled graph $G = (E, V)$ and some language $\mathcal{L} \subseteq \Sigma^*$. Two nodes $u, v \in V$ are $\mathcal{L}$-reachable, denoted as $u \overset{\mathcal{L}}{\rightsquigarrow} v$, if there exists a path $P \colon u \rightsquigarrow v$ such that $\Sigma(P) \in \mathcal{L}$. Thus, language reachability refines the standard notion of graph reachability by requiring that a path witnessing reachability also produces a word that belongs to the corresponding language. This notion is closely related to the emptiness of the intersection of $\mathcal{L}$ with a regular language. Indeed, we may treat G as a non-deterministic finite automaton with u being the

initial state and v being the final state, defining some regular language $\mathcal{L}_G$. Hence, the language reachability question for u and v is deciding whether $\mathcal{L}_G \cap \mathcal{L} = \emptyset$. We will maintain our view of language reachability as opposed to language emptiness, to emphasize the fact that the problem is studied with respect to graphs G of varying size and properties, while $\mathcal{L}$ is some fixed language with a constant-size syntactic description.

CFL and Dyck reachability. When $\mathcal{L}$ is a context-free language (CFL), the language-reachability problem is known as CFL reachability. Here we assume without loss of generality that CFL is specified via a corresponding context-free grammar (CFG) $\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{R}, \mathcal{S})$, where N is a set of non-terminals, R is a set of production rules, and $\mathcal{S} \in \mathcal{N}$ is the initial non-terminal. We also assume that $\mathcal{G}$ is given in Chomsky normal form, which is also a frequent assumption of algorithms solving the CFL memberhip problem. Given a non-terminal $A \in \mathcal{N}$ and a word $\sigma \in \Sigma^*$, we denote by $A \vdash \sigma$ the fact that A produces σ according the rules of $\mathcal{G}$.

One specific class of CFL languages that is frequently used in CFL reachability is that of Dyck languages. A Dyck language $\mathcal{D}_k$, parameterized by some natural number $k \in \mathbb{N}$, is defined with respect to a *matched alphabet* $\Sigma = \Sigma^O \cup \Sigma^C$, where $\Sigma^O = \{\alpha_1, \ldots, \alpha_k\}$ and $\Sigma^C = \{\overline{\alpha}_1, \ldots, \overline{\alpha}_k\}$. Every α_i is matched by the corresponding $\overline{\alpha}_i$. It is common to interpret α_i as an opening a parenthesis $(_i$, and $\overline{\alpha}_i$ as a closing parenthesis $)_i$, and Dyck languages be called parenthesis languages. For notational clarity, we will stick to α_i and $\overline{\alpha}_i$, but call them "parentheses". The Dyck language is the language of all properly-balanced parenthesis words, generated by the following grammar.

$$\mathcal{S} \to \mathcal{S}\,\mathcal{S} \mid \mathcal{A}_1\,\overline{\mathcal{A}}_1 \mid \ \ldots\ \mid \mathcal{A}_k\,\overline{\mathcal{A}}_k \mid \epsilon\ ; \qquad \mathcal{A}_i \to \alpha_i\,\mathcal{S}\ ; \qquad \overline{\mathcal{A}}_i \to \mathcal{S}\,\overline{\alpha}_i$$

As a corner case, we take $\mathcal{D}_0 = \Sigma^*$, i.e., we have plain graph reachability.

3. CFL AND DYCK REACHABILITY

In this section we present the main results on standard CFL and Dyck reachability.

3.1. An Algorithm for CFL/Dyck Reachability

We begin with the standard algorithm on CFL reachability, which naturally also solves reachability over Dyck languages. The input is a labeled graph $G = (V, E)$, and a CFG $\mathcal{G}$ defining the respective CFL $\mathcal{L}$.

The basic algorithm for CFL/Dyck reachability. The basic algorithm for computing CFL reachability can be seen as a generalization of the standard CYK algorithm that solves the membership problem for CFLs. The algorithm expects the input grammar $\mathcal{G}$ in Chomsky normal form, and performs a fixpoint computation on G, by inserting an edge $u \xrightarrow{C} v$ in G whenever it discovers that there is a path $P\colon u \rightsquigarrow v$ such that $C \vdash \Sigma(P)$. In the end, we have that $u \overset{\mathcal{L}}{\rightsquigarrow} v$ iff the algorithm has produced an edge $u \xrightarrow{\mathcal{S}} v$, where $\mathcal{S}$ is the initial non-terminal symbol of $\mathcal{G}$. Algorithm 1 gives the algorithm in pseudocode, while Figure 2 shows the main saturation steps of Line 16 and Line 23.

Correctness and complexity. The soundness of the algorithm is captured in the following statement: *for every triplet (u, v, A) inserted in $\mathcal{W}$, there exists a path $P\colon u \rightsquigarrow v$ such that $A \vdash \Sigma(P)$*. This can be shown by a straightforward induction on the elements inserted in $\mathcal{W}$. The completeness is captured by an analogous invariant, namely: *for every pair of nodes u, v such that there exists a path $P\colon u \rightsquigarrow v$ such that $A \vdash \Sigma(P)$, the algorithm will insert a triplet (u, v, A) in $\mathcal{W}$*. This can be shown by an induction on the depth of the derivation tree that witnesses $A \vdash \Sigma(P)$. Regarding the running time,

Algorithm 1: The basic algorithm for CFL/Dyck reachability.

Input: A Σ-labeled graph $G = (V, E)$, a CFG $\mathcal{G}$ for a language $\mathcal{L} \subseteq \Sigma^*$.
Output: $\{(u, v) \colon u \overset{\mathcal{L}}{\rightsquigarrow} v\}$.

```
// Initialization
Initialize a worklist W
foreach rule A → a in G do
    foreach edge u -a-> v in G do
        Insert u -A-> v in G
        Insert (u, v, A) in W
    end
    if a = ε then
        foreach node u do
            Insert u -A-> u in G
            Insert (u, v, A) in W
        end
    end
end
// Computation
while W ≠ ∅ do
    Extract a triplet (u, v, A) from W
    foreach rule C → AB in G do
        foreach edge v -B-> w in G do
            if u -C-> w is not an edge in G then
                Insert u -C-> w in G
                Insert (u, w, C) in W
            end
        end
    end
    foreach edge w -B-> u in G do
        foreach rule C → BA in G do
            if w -C-> v is not an edge in G then
                Insert w -C-> v in G
                Insert (w, v, C) in W
            end
        end
    end
end
return {(u, v) : u -S-> v is an edge in G}
```

Fig. 2: Combining a newly-discovered edge $u \xrightarrow{A} v$ with the grammar rules $C \to AB$ (left) and $C \to BA$ (right).

if G has n nodes then the algorithm takes $O(n^3)$ time in the worst case. To see this, observe that every triplet (u, v, A) is inserted in $\mathcal{W}$ at most once, hence we can bound the number of times that the while loop is executed by $O(n^2)$. Each iteration takes time $O(n)$, by iterating over the neighbors of u and v in G, leading to a total bound of $O(n^3)$.

A logarithmic speedup. The cubic bound of Algorithm 1 can be slightly improved to $O(n^3/\log n)$ by making use of the standard "four Russians" technique [Arlazarov et al. 1970]. In algorithmic parlance, the technique is also known under the term "word tricks". The key idea is that sets representing the edges adjacent to a node and labeled with a specific symbol A can be represented as bit sets. In the RAM model, a single machine word consists of $\Theta(\log n)$ bits, hence each set is represented using $O(n/\log n)$ machine words. Basic set operations such as union, intersection and complementation amount to bit-wise logical OR, AND and XOR operations, which, when applied on machine words, require $O(n/\log n)$ time, as opposed to $O(n)$ time, yielding a logarithmic speedup. Moreover, the set difference $X = Y \setminus Z$ can be computed in time $O(|X| + n \log n)$, by first computing X in $O(n \log n)$ time using set operations, and then repeatedly retrieving the most significant bit of X in $O(1)$ time, setting it to zero, and repeating.

These fast operations on sets are directly applicable to Algorithm 1. Instead of iterating over w with $v \xrightarrow{B} w$ in Line 16, we iterate over all such w for which we don't have $u \xrightarrow{C} w$. This amounts to computing the difference between the set storing the edges $v \xrightarrow{B} w$ and $u \xrightarrow{C} w$, using word tricks. The same applies to Line 23, yielding a total running time of $O(n^3/\log n)$, by counting $O(n^2)$ time for adding new edges to G, plus $O(n^2 \cdot n/\log n)$ time for computing these set differences. This speedup of Algorithm 1 by using word tricks was observed in [Chaudhuri 2008], resulting in the following theorem.

THEOREM 3.1 ([CHAUDHURI 2008]). *Dyck reachability on connected graphs of n nodes can be solved in $O(n^3/\log n)$ time.*

A sub-quadratic bound for output-sparse graphs. In several application domains, it has been observed that the graph G remains sparse at the end of Algorithm 1, i.e., *after* all new edges have been added to it [Sridharan and Fink 2009; Zhang et al. 2013]. It is thus natural to consider the complexity of the algorithm as a function of the number of edges at the end of the algorithm. The key insight is that the total running time of Algorithm 1 can be bounded by the number of edges plus the number of triangles in G at the end of the algorithm. Indeed, observe that every saturation step of Figure 2 produces a triangle in G, and the number of triangles precisely counts the total work performed in Line 16 and Line 23. It is well known that a graph of m edges has $O(m^{3/2})$ triangles [Suri and Vassilvitskii 2011], which results in the following theorem.

THEOREM 3.2. *Algorithm 1 takes $O(m^{3/2})$ time on graphs that have m edges at the end of the algorithm.*

Theorem 3.2 yields the following corollary for output-sparse graphs, i.e., when $m = O(n)$.

COROLLARY 3.3. *Algorithm 1 takes $O(n^2)$ time on output-sparse graphs.*

3.2. Lower Bounds

The cubic bound of Theorem 3.1, and the lack of any considerable improvement on it over the years, has lead researchers into investigating *lower bounds* for the problem. As usual, such lower bounds are conditional instead of absolute, i.e., they have the general form "problem A cannot be solved in time faster than t_A as long as problem B cannot be solved in time faster than t_B".

2NPDA completeness. The first such lower bound was shown for CFL reachability in [Heintze and McAllester 1997].

THEOREM 3.4 ([HEINTZE AND MCALLESTER 1997]). *CFL reachability is* 2NPDA*-complete.*

2NPDA is the class of languages (or problems) defined by two-way nondeterministic pushdown automata. It is known that problems in 2NPDA are solvable in cubic time [Aho et al. 1968], but no truly subcubic algorithm is known to date. Rytter obtained a $\log n$ improvement for such problems [Rytter 1985], and further a $\log^2 n$ improvement for the subclass of 2NPDA recognized by loop-free automata [Rytter 1986]. Naturally, these logarithmic improvements translate to CFL reachability. The $\log n$ improvement has already been described in Section 3.1; the $\log^2 n$ improvement was obtained in [Chaudhuri 2008], established for the class of bounded-stack Recursive State Machines (RSMs), that can be seen as an analogue to loop free two-way automata.

Dyck reachability and Boolean Matrix Multiplication. The complexity of CFL parsing has been studied extensively. The textbook-standard CYK algorithm follows

a dynamic-programming approach and solves the problem in $O(n^3)$ time, for a word of length n [Hopcroft et al. 2001]. Valiant's famous parser [Valiant 1975] reduces the CFL-recognition to Boolean Matrix Multiplication (BMM), achieving a the improved bound of $O(n^\omega)$, where $\omega \simeq 2.372$ is the BMM exponent. Note that this subcubic bound is based on fast-matrix-multiplication algorithms. If we only focus on combinatorial algorithms, the parsing problem is known to admit no $O(n^{3-\varepsilon})$ algorithms, for an fixed $\varepsilon > 0$, under the combinatorial BMM hypothesis (i.e., assuming that BMM has no truly subcubic combinatorial algorithm) [Lee 2002]. As CFL reachability can be seen as a generalization of CFL parsing (a word can be encoded as a path-graph), the BMM lower-bound holds for CFL reachability as well. On the other hand, recognizing Dyck languages can be done in $O(n)$ time, by a simple left-to-write pass on the input word. Thus, one might hope that Dyck reachability on graphs that are "path-like", Dyck reachability can be performed in sub-cubic time. Unfortunately, this turns out to not be the case, at least if we interpret "path-like" graphs as graphs that have bounded pathwidth [Robertson and Seymour 1983].

THEOREM 3.5 ([CHATTERJEE ET AL. 2018]). *Dyck reachability is BMM-hard, even on graphs with bounded pathwidth.*

The proof of Theorem 3.5 reduces the general CFL-parsing problem, which is known to be BMM-hard [Lee 2002], to Dyck reachability. It is conceptually close to the Chomsky–Schützenberger representation theorem [Chomsky and Schützenberger 1963], where the graph G plays the role of the regular language in the theorem.

SETH and subcubic certificates. The area of fine-grained complexity is concerned with lower bounds of problems in PTime, and in particular, in bounding the degree of the polynomial expressing the complexity of the problem. Reducibility between problems in PTime is done using *fine-grained reductions*. Given two problems A and B, and functions $t_A, t_B \colon \mathbb{N} \to \mathbb{N}$, a fine-grained reduction from (A, t_A) to (B, t_B) is an algorithm that transforms every instance $\mathcal{I}$ of A to an instance $\mathcal{J}$ of B such that

(1) $\mathcal{I}$ is positive iff $\mathcal{J}$ is positive,
(2) the running time of the reduction is $t_A(|\mathcal{I}|)^{1-\gamma}$, for some fixed $\gamma > 0$, and
(3) for any fixed $\varepsilon > 0$ there is a fixed $\delta > 0$ such that $t_B(|\mathcal{J}|)^{1-\varepsilon} = O(t_A(|\mathcal{I}|))^{1-\delta}$.

Such a reduction implies that if B admits a polynomial complexity improvement over t_B (i.e., a time bound sublinear in t_B), then A also admits a polynomial complexity improvement over t_A. Thus, taking as a hypothesis that t_A is a lower bound for A, we obtain that B has a $\Omega(t_B)$ is lower bound.

One of the most popular hypotheses in fine-grained complexity theory is the Strong Exponential Time Hypothesis (SETH), which roughly conjectures that for any fixed $\varepsilon > 0$, as ℓ increases, ℓ-SAT over n variables cannot be solved in time $2^{(1-\varepsilon)n}$. A very natural question is, thus, whether CFL/Dyck reachability can be shown to have a cubic lower bound based on SETH. A recent work investigated this relationship, showing that CFL/Dyck reachability admits *subcubic certificates*, which has negative implications for the question at hand [Chistikov et al. 2022]. A problem A admits positive (resp., negative) certificates of size s, if there is an algorithm M such that

— for every positive (resp., negative) instance $\mathcal{I}$ of A there exists a word c of length s such that $M(\mathcal{I}, s)$ accepts in time s, and
— for every negative (resp., positive) instance $\mathcal{I}'$ of A, for every word c' of length s, we have that $M(\mathcal{I}, s)$ rejects in time s.

The following theorem is based on showing that (i) positive instances of Dyck reachability admit certificates of size $O(n^2)$, while (ii) negative instances admit certificates of size $O(n^\omega)$. Roughly speaking, the certificates for positive instances are similar to the usual backpointers stored in the dynamic-programming table of CYK parsing in order to recover a parse tree [Hopcroft et al. 2001], relating every pair of nodes (u, v) of G with a non-terminal $\mathcal{N}_{u,v}$ that can be produced via a path $u \rightsquigarrow v$. For the certificate to be verifiable in $O(n^2)$ time, it also stores in (u, v) a rule that produces $\mathcal{N}_{u,v}$, as well as the intermediate node w for which $\mathcal{N}_{u,w}$ and $\mathcal{N}_{w,v}$ are the right-hand sides of that rule (i.e., we have $\mathcal{N}_{u,v} \to \mathcal{N}_{u,w}\mathcal{N}_{w,v}$). The certificates for negative instances are somewhat more complex. The intuition is to capture in the certificate the state of Algorithm 1 at saturation, i.e., when no more edges can be inserted in G, while $s \xrightarrow{S} t$ is not an edge of G. Naturally, this can be achieved by iterating over all rules $C \to AB$ and all paths of length two $u \xrightarrow{A} v \xrightarrow{B} v$, and verifying that $u \xrightarrow{C} v$ is also in G; however, naively computing this requires $\Theta(n^3)$ time. The trick is to encode this computation as a fixed number of BMMs, and rely on fast BMM to reduce the cubic bound to n^ω.

THEOREM 3.6 ([CHISTIKOV ET AL. 2022]). *$\mathcal{D}_k$ reachability admits certificates of length $O(n^\omega)$.*

The Nondeterministic SETH (NSETH) is a hypothesis analogous to SETH, which roughly excludes $2^{(1-\varepsilon)n}$ algorithms for solving ℓ-TAUT, as ℓ increases [Carmosino et al. 2016]. An important implication of NSETH is that, if a problem admits certificates of size s, it cannot be shown to have a SETH-based lower bound of size polynomially larger than s. Concretely, for our setting, Theorem 3.6 implies the following corollary.

COROLLARY 3.7 ([CHISTIKOV ET AL. 2022]). *For any fixed $\varepsilon > 0$, there is no fine-grained reduction from $(\mathrm{SAT}, 2^n)$ to ($\mathcal{D}_k$ reachability, $n^{\omega+\varepsilon}$), under NSETH.*

3.3. The Case of $k = 1$

In this section we turn our attention to Dyck reachability over $k = 1$ parenthesis symbol. The conceptual stack that is associated with the language now acts as a counter, where an opening-parenthesis edge $u \xrightarrow{\alpha} v$ increases the counter and a closing parenthesis edge $u \xrightarrow{\overline{\alpha}} v$ decreases the counter. A path witnessing reachability must ensure that (i) the counter remains non-negative along the path, and (ii) the counter becomes 0 at the end of the path. Under this interpretation, we use $+1$ (resp., -1) to represent an opening (resp., closing) parenthesis. The setting is also known as counter automata, or one-dimensional vector addition systems with states. Naturally, the general cubic upper bound of Theorem 3.1 applies for $k = 1$. However, this setting admits an improved upper bound, which is further matched by an almost-tight lower bound.

Upper bound. One important property of $\mathcal{D}_1$ reachability is that reachability witnesses are, without loss of generality, polynomially bounded in length. In particular, if v is $\mathcal{D}_1$-reachable from u, then there exists a witness path of length $L = O(n^2)$ [Deleage and Pierre 1986]. In contrast, shortest witnesses for $\mathcal{D}_2$ reachability can be exponentially long [Pierre 1992]. The key idea behind the algorithm for $\mathcal{D}_1$ reachability is to reduce the problem to $O(\log^2 n)$ plain, all-pairs reachability instances, and use fast matrix multiplication to solve each of them. The algorithm comes in two steps.

Step 1: Reachability with respect to the language $(+1)^\ell(-1)^\ell$. Consider the language $\mathcal{L}_= = (+1)^\ell(-1)^\ell$, i.e., all counter increments precede counter decrements. Pictorially, paths P with $\lambda(P) \in \mathcal{L}_=$ exhibit a single local maximum on the counter value. There is a simple algorithm to solve reachability with respect to $\mathcal{L}_=$, that is a combination of

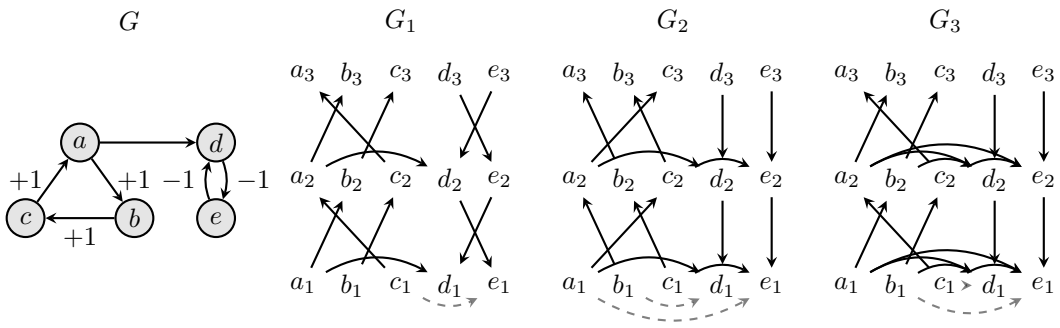

Fig. 3: Illustration of Algorithm 2 on the Dyck graph G (left). $\mathcal{L}_=$ reachability in G as witnessed by paths $P\colon x \rightsquigarrow y$ with maximum counter value $\leq 2^i - 1$ is captured in graph G_i (right) by the path $x_1 \rightsquigarrow y_1$. Dashed edges in G_i represent the summarization of the path, which is carried over to G_{i+1} as a single edge.

saturation with successive doubling. The algorithm constructs a sequence of $O(\log n)$ plain (i.e., not labeled) digraphs $(G_i = (K, R_i))_i$. The node set K is common to all G_i and consists of three copies x_1, x_2, x_3 for every node $x \in V$. In iteration i, the algorithm performs all-pairs reachability in G_i, and using this information, constructs the edge set R_{i+1}. Each G_i consists of three copies of G, where $\mathcal{L}_=$-labeled paths of maximum counter value 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 counters reaching 2^i (resp., -2^i). We refer to Algorithm 2 for a detailed description and to Figure 3 for an illustration.

Algorithm 2: Algorithm for $\mathcal{L}_=$ reachability.

Input: A Dyck graph $G = (V, E)$
Output: A set $\{(x, y)\}_{x,y \in V}$ such that $x \overset{\mathcal{L}_= y}{\rightsquigarrow}$.

```
// 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 -ε-> y ∈ E
    Insert x_j → y_{j+1} ∈ R_1 iff x -(+1)-> y ∈ E
    Insert y_{j+1} → x_j ∈ R_1 iff x -(-1)-> y ∈ E
end
Construct the graph G_1 = (K, R_1)
Let L ← an upper bound on d(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}
```

Step 2: $\mathcal{D}_1$ reachability. Observe that $\mathcal{L}_= \subseteq \mathcal{D}_1$, i.e., if two nodes are $\mathcal{L}_=$-reachable then they are also $\mathcal{D}_1$-reachable. Of course the opposite is not true, as the counter along reachability witnesses may exhibit many local maxima. Nevertheless, consider that we apply Algorithm 2 for reachability with respect to $\mathcal{L}_=$ on G. We may attempt to identify all such reachable pairs $x \overset{\mathcal{L}_=}{\rightsquigarrow} y$, and insert edges $x \overset{\epsilon}{\to} y$ in G, in order to directly represent $\mathcal{L}_=$-reachability information. Let G^2 be the resulting graph, and let

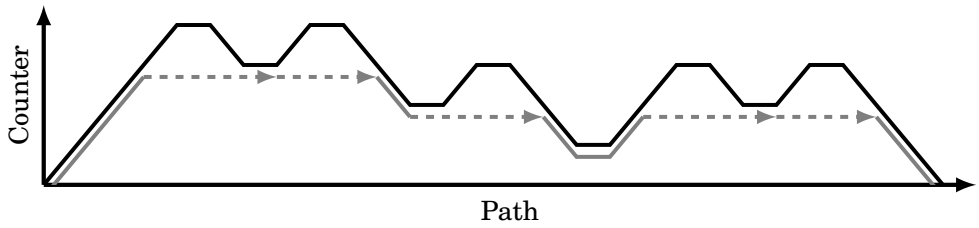

Fig. 4: Illustration of a path P in graph G^i (black) and its summarization path P' in graph G^{i+1} (gray). The number of local maxima in P' is at most half of that in P.

$G^1 = G$. Naturally, $\mathcal{D}_1$ reachability on G^1 coincides with that on G^2. Observe, however, that certain nodes that were $\mathcal{D}_1$-reachable in G^1 but not $\mathcal{L}_=$-reachable, now become $\mathcal{L}_=$-reachable in G^2 (e.g., this occurs when the witness path in G^1 exhibits the counter sequence $+1, -1, +1, +1, -1, -1$). We may thus feel tempted to repeat this process on G^2, thereby obtaining another graph G^3, and so on. The obvious question is: *is there a bound on how many iterations we have to go through until we have discovered all $\mathcal{D}_1$-reachable pairs?* The crucial observation is as follows: for all i, if $x \overset{\mathcal{D}_1}{\rightsquigarrow} y$ is witnessed in G^i by a path with $\ell > 1$ local maxima, then $x \overset{\mathcal{D}_1}{\rightsquigarrow} y$ is witnessed in G^{i+1} by a path with $\leq \lceil \ell/2 \rceil$ local maxima. Figure 4 provides an illustration.

Since $\mathcal{D}_1$-reachability witnesses are polynomially bounded in length, and the number of local maxima of a path cannot exceed its length, it follows that after $\lceil \log L \rceil$ iterations, $x \overset{\mathcal{D}_1}{\rightsquigarrow} y$ resorts to reachability with respect to $\mathcal{L}_=$ in $G^{\lceil \log L \rceil}$. Overall, the algorithm for $\mathcal{D}_1$ reachability simply performs $\lceil \log L \rceil = O(\log n)$ repetitions of $\mathcal{L}_=$-reachability operations. We thus have the following theorem.

THEOREM 3.8 ([MATHIASEN AND PAVLOGIANNIS 2021]). *$\mathcal{D}_1$ reachability on graphs of n nodes can be solved in $O(n^\omega \cdot \log^2 n)$ time, where $\omega \leq 2.373$ is the matrix-multiplication exponent.*

An interesting aspect of Theorem 3.8 is that, similarly to Theorem 3.1 concerning the general case, the upper bound also holds for all-pairs reachability.

Lower bounds. Observe that Theorem 3.8 solves all-pairs Dyck reachability (over $k = 1$) in essentially the same time as all-pairs plain reachability (over $k = 0$). As the latter problem is at least as hard as BMM, it follows that the bound of Theorem 3.8 is essentially tight for the all-pairs version. On the other hand, single-pair plain reachability can be solved in time linear in the size of the graph. It is thus natural to ask whether the $O(n^\omega \cdot \log^2 n)$ bound for *single pair* Dyck reachability (over $k = 1$) is tight. Here we sketch a recent proof [Cetti Hansen et al. 2021] that this is indeed the case.

The proof follows a fine-grained reduction from the problem of *triangle-detection*: given an undirected graph $H = (R, T)$, does H contain a triangle (i.e., a clique of size 3)? Triangle detection can be reduced to matrix multiplication, and can thus be solved in $O(n^3)$ time by combinatorial algorithms, and in $O(n^\omega)$ time in general. Interestingly, it was recently shown that neither bound admits polynomial improvements, under the (combinatorial, for the former) BMM hypothesis [Williams 2019].

Reduction. Consider an instance $H = (R, T)$ of the triangle detection problem, and we construct a labeled graph $G = (V, E)$ as follows. We assume without loss of generality that $R = [\ell]$, i.e., it is the set of integers $\{1, \ldots, \ell\}$. The set of nodes of G is $V = \{s, t\} \cup \bigcup_{i \in [\ell]} \{a_i, b_i, c_i, d_i\}$ i.e., we have four nodes in V per node $i \in R$, plus two auxiliary

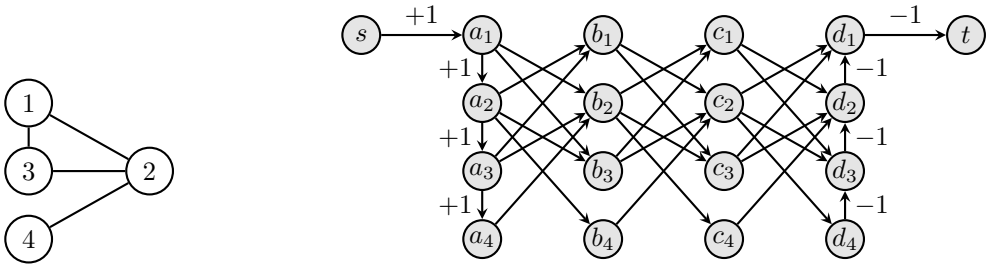

Fig. 5: An input graph H (left) and the labeled graph G constructed in our reduction (right). The path s, a_1, b_2, c_3, d_1, t in G is a witness of the triangle $(1, 2, 3)$ in H.

nodes. The edge relation is $E = A \cup B \cup C$, where $A = \{s \xrightarrow{+1} a_1, d_1 \xrightarrow{-1} t\}$, and

$$B = \bigcup_{(i,j)\in E} \{a_i \xrightarrow{0} b_j, b_i \xrightarrow{0} c_j, c_i \xrightarrow{0} d_j\} \qquad C = \bigcup_{i\in[\ell-1]} \{a_i \xrightarrow{+1} a_{i+1}, d_{i+1} \xrightarrow{-1} d_i\}$$

Figure 5 provides an illustration. It is easy to verify that t is reachable from s iff H has a triangle, which leads to the following theorem.

THEOREM 3.9 ([CETTI HANSEN ET AL. 2021]). *$\mathcal{D}_1$ reachability on graphs of n nodes requires $\Omega(n^\omega)$ time, where ω is the matrix multiplication exponent.*

Given Theorem 3.9, the upper bound established in Theorem 3.8 is thus optimal (modulo polylogarithmic improvements). Finally, as the reduction towards Theorem 3.9 is a combinatorial one, it also implies a combinatorial cubic lower bound for the problem.

COROLLARY 3.10. *For any fixed $\varepsilon > 0$, there is no combinatorial algorithm for $\mathcal{D}_1$ reachability operating in $O(n^{3-\varepsilon})$ time on graphs of n nodes, under the combinatorial BMM hypothesis.*

3.4. Reachability on Recursive State Machines

One important class of CFL reachability is reachability on Recursive State Machines (RSMs) [Alur and Madhusudan 2004]. From a static analysis perspective, RSMs naturally model the organization of a program into functions, and capture analysis sensitivity in calling contexts. Although RSM reachability is equivalent to CFL reachability, one can express the complexity of RSM reachability with respect to certain parameters of the RSM, as opposed to the generic cubic bound $O(n^3)$ for CFL reachability on graphs. This expression often yields a sub-cubic (or even linear!) upper bound when these parameters remain small, which is a common case in static analyses. In the following we give a formal account of RSMs and their reachability problem.

Recursive State Machines. A *recursive state machine (RSM)* is a tuple $\mathcal{R} = \langle \mathcal{M}_1, \ldots, \mathcal{M}_k \rangle$, where every *module* $\mathcal{M}_i = \langle B_i, Y_i, N_i, \delta_i \rangle$ is given by

— a finite set B_i of *boxes*,
— a mapping $Y_i : B_i \mapsto [k]$,
— a finite set $N_i = In_i \cup En_i \cup Ex_i \cup Call_i \cup Ret_i$ of *nodes*, partitioned into
  — *internal* nodes In_i,
  — *entry* nodes En_i,
  — *exit* nodes Ex_i,
  — *call* nodes $Call_i = \{\langle b, e\rangle \mid b \in B_i \text{ and } e \in En_{Y_i(b)}\}$,
  — *return* nodes $Ret_i = \{\langle b, x\rangle \mid b \in B_i \text{ and } x \in Ex_{Y_i(b)}\}$,

— a *transition relation* $\delta_i \subseteq (In_i \cup En_i \cup Ret_i) \times (In_i \cup Ex_i \cup Call_i)$,

We write B for $\bigcup_{i=1}^{k} B_i$, and similarly for N, In, En, Ex, $Call$, Ret, δ. An important parameter of RSMs is the number of entry and exit nodes of the modules. In particular, we let $\theta_e = \max_{1 \leq i \leq k} |En_i|$ and the *exit bound* $\theta_x = \max_{1 \leq i \leq k} |Ex_i|$, i.e., the maximum number of entries and exits, respectively, over all modules. A *stack* is a sequence of boxes $S = b_1 \dots b_r$, where b_1 is the top; and ε is the empty stack. For a box b and a stack S, we denote with bS the concatenation of b and S, i.e., a push of b onto the top of S.

Configurations and transitions. A *configuration of an RSM* $\mathcal{R}$ is a tuple $\langle u, S \rangle$, where $u \in In \cup En \cup Ret$ is an internal, entry, or return node, and S is a stack. For $S = b_1 \dots b_r$, where $b_i \in B_{j_i}$ for $1 \leq i \leq r$ and some j_i, we require that $Y_{j_i}(b_i) = j_{i-1}$ for $1 < i \leq r$, as well as $u \in N_{Y_{j_1}(b_1)}$. This captures the case that the control is inside the module of node u, which was entered via box b_1 from module $\mathcal{M}_{j_1}$, which was entered via box b_2 from module $\mathcal{M}_{j_2}$, and so on. We define a transition relation $\Rightarrow$ over configurations such that $\langle u, S \rangle \Rightarrow \langle u', S' \rangle$ iff there exists a transition $t \in \delta_i$ and one of the following holds:

(1) *Internal transition:* $u' \in In_i$, $t = \langle u, u' \rangle$, and $S' = S$.
(2) *Call transition:* $u' = e \in En_{Y_i(b)}$ for some box $b \in B_i$, $t = \langle u, \langle b, e \rangle \rangle$, and $S' = bS$.
(3) *Return transition:* $u' = \langle b, x \rangle \in R_i$ for some box $b \in B_i$ and exit node $x \in Ex_{Y_i(b)}$, $t = \langle u, x \rangle$, and $S = bS'$.

For any module $\mathcal{M}_i$ and two nodes $s, t \in N_i$, we say that t is *reachable* from s if $\langle s, \epsilon \rangle \Rightarrow^* \langle t, \epsilon \rangle$, where $\Rightarrow^*$ is the reflexive transitive closure of $\Rightarrow$. Observe that RSM reachability coincides with Dyck reachability, if we interpret the RSM as a graph, and call and return transitions are labeled with opening and closing parenthesis, respectively, parameterized by the box they start from, and end to, respectively.

THEOREM 3.11 ([ALUR AND MADHUSUDAN 2004; CHATTERJEE ET AL. 2017]). *RSM reachability can be solved in $O(m \cdot \theta + n \cdot \theta^2)$ time, on an RSM with n nodes and m edges, where $\theta = \min(\theta_e, \theta_x)$ and θ_e (resp., θ_x) is the entry bound (resp., exit bound).*

When $\theta = \Omega(n)$, Theorem 3.11 yields the familiar cubic bound. However, in several static analyses, the modeling yields $\theta = O(1)$, which leads to a linear bound.

4. DYCK REACHABILITY ON BIDIRECTED GRAPHS

In this section we turn our attention to Dyck reachability on a special class of graphs called *bidirected*. Recall that Dyck languages are defined over a parentheses alphabet $\Sigma = \Sigma^O \cup \Sigma^C$, where $\Sigma^O = \{\alpha_1, \dots, \alpha_k\}$ are the opening parentheses, and $\Sigma^C = \{\overline{\alpha}_1, \dots, \overline{\alpha}_k\}$ are the closing parentheses. A graph $G = (V, E)$ is called *bidirected* if for all $u, v \in V$ and $f \in \Sigma_\epsilon$, we have $u \xrightarrow{f} v \in E$ iff $v \xrightarrow{\overline{f}} u \in E$, where $\overline{\overline{\alpha}}_i = \alpha_i$ and $\overline{\epsilon} = \epsilon$. Bidirectedness is the analogue of undirectedness on plain graphs, as it turns reachability to an equivalence: any path $P \colon u \rightsquigarrow v$ now can be reversed to $\overline{P} \colon v \rightsquigarrow u$ with $\lambda(P) = \lambda(\overline{P})$. Thus, if P witnesses the Dyck reachability of v from u, $\overline{P}$ witnesses the Dyck reachability of u from v. The nodes of G are hence partitioned into *Dyck Strongly Connected Components (DSCCs)*, which are maximal subsets of V containing inter-reachable nodes. In contrast to standard SCCs, all paths witnessing reachability between two nodes in a DSCC might have to traverse nodes outside the DSCC.

For economy of presentation, in this section we only refer to edges labeled with a closing parenthesis symbol (or have an empty label), with the understanding that the reverse, complementary-labeled edges are implied.

Upper Bound. The equivalence property hints on a simple principle for computing DSSCs. The idea is that for any two distinct nodes u, v to belong to some DSCC X, there must exist two (not necessarily distinct) nodes x, y that belong to some DSCC Y, and a closing parenthesis $\overline{\alpha}_i$ such that $x \xrightarrow{\overline{\alpha}_i} u, y \xrightarrow{\overline{\alpha}_i} v \in E$. See Figure 6 for an illustration.

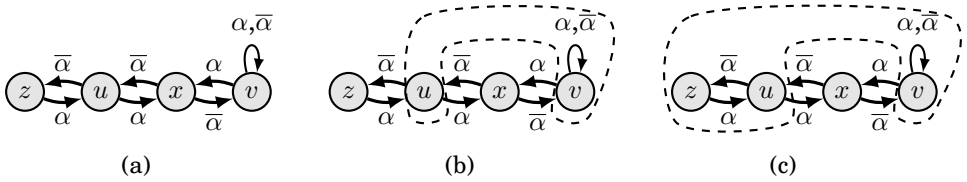

Fig. 6: Illustration of the merging principle on bidirected graphs . **(6a)** Nodes u and v are in the same DSCC since node x has an outgoing edge to each of u and v labeled with $\overline{\alpha}$. **(6b)** Similarly, nodes z and v belong to the same DSCC, since there exist two nodes u and v such that (i) u and v belong to the same DSCC, (ii) u has an outgoing edge to z, and v has an outgoing edge to itself, and (iii) both outgoing edges are labeled with the same closing parenthesis symbol. **(6c)** The final DSCC.

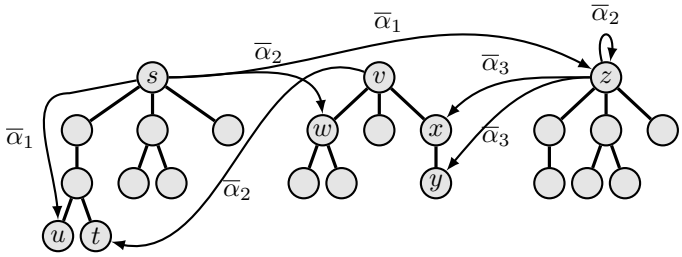

Fig. 7: A state of Algorithm 3 consists of a set of trees, with outgoing edges coming only from the root of each tree.

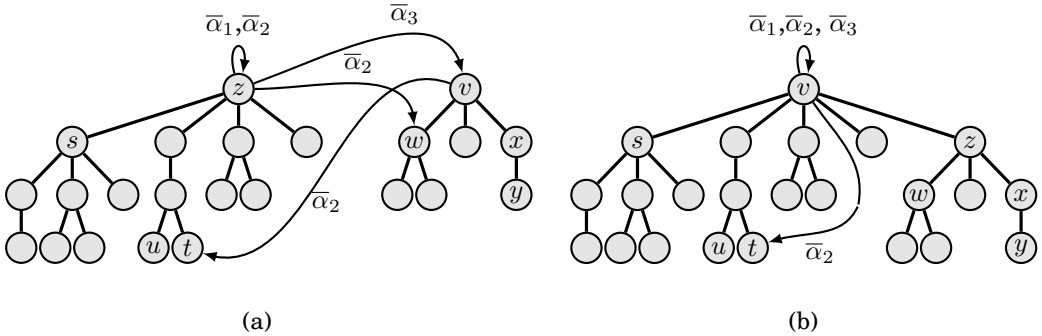

Fig. 8: The intermediate stages of Algorithm 3 starting from the state of Figure 7.

Algorithm 3: Algorithm for $\mathcal{D}_k$ reachability

Input: A Σ_k-labeled bidirected graph $G = (V, E)$
Output: A DisjointSets map of DSCCs

```
   // Initialization
 1 Q ← an empty queue
 2 Edges ← a map V × Σ^C → V* implemented as a linked list
 3 DisjointSets ← a disjoint-sets data structure over V
 4 foreach u ∈ V do
 5 |   DisjointSets.MakeSet(u)
 6 |   for i ← 1 to k do
 7 |   |   Edges[u][ᾱ_i] ← (v : (u, v, ᾱ_i) ∈ E)
 8 |   |   if |Edges[u][ᾱ_i]| ≥ 2 then Insert (u, ᾱ_i) in Q
 9 |   end
10 end
   // Computation
11 while Q is not empty do
12 |   Extract (u, ᾱ_i) from Q
13 |   if u = DisjointSets.Find(u) then
14 |   |   Let S ← {DisjointSets.Find(w) : w ∈ Edges[u][ᾱ_i]}
15 |   |   if |S| ≥ 2 then
16 |   |   |   Let x ← some arbitrary element of S \ {u}
17 |   |   |   Make DisjointSets.Union(S, x)
18 |   |   |   for j ← 1 to k do
19 |   |   |   |   foreach v ∈ S \ {x} do
20 |   |   |   |   |   if u ≠ v or i ≠ j then
21 |   |   |   |   |   |   Move Edges[v][ᾱ_j] to Edges[x][ᾱ_j]
22 |   |   |   |   |   else
23 |   |   |   |   |   |   Append (x) to Edges[x][ᾱ_j]
24 |   |   |   |   |   end
25 |   |   |   |   end
26 |   |   |   |   if |Edges[x][ᾱ_j]| ≥ 2 then Insert (x, ᾱ_j) in Q
27 |   |   |   end
28 |   |   else
29 |   |   |   Let x ← the single node in S
30 |   |   end
31 |   |   if u ∉ S or |S| = 1 then Edges[u][ᾱ_i] ← (x)
32 end
33 return DisjointSets
```

The above principle can be directly turned into an algorithm that repeatedly computes DSCCs (see Algorithm 3). For efficiency, we use a Disjoint Sets data structure to maintain DSCCs discovered so far. Each DSCC is represented as a tree T rooted on some node $x \in V$, and x is the only node of T that has outgoing edges. However, any node of T can have incoming edges. See Figure 7 for an illustration. Upon discovering that a root node x of some tree T has two or more outgoing edges $x \xrightarrow{\overline{\alpha}_i} u_1, x \xrightarrow{\overline{\alpha}_i} u_2, \ldots, x \xrightarrow{\overline{\alpha}_i} u_r$, for some $\overline{\alpha}_i$, the algorithm uses r Find operations of the Disjoint Sets data structure to determine the trees T_i that the nodes u_i belong to. Afterwards, a Union operation is performed between all T_i to form a new tree T, and all the outgoing edges of the root of each T_i are merged to the outgoing edges of the root of T. Figure 8 illustrates this step.

The running time of Algorithm 3 is $O(m)$ time to process all edges of G, plus $O(n)$ union operations, each amortizing time $\alpha(n)$ (or $O(1)$ time in expectation). We thus arrive at a significant improvement over the cubic bound of Theorem 3.1.

THEOREM 4.1 ([CHATTERJEE ET AL. 2018]). *Dyck reachability on bidirected graphs of n nodes and m edges can be computed in $O(m + n \cdot \alpha(n))$ time, and $O(n + m)$ expected time.*

Lower Bound. Given the closeness of bidirected Dyck reachability to plain reachability on undirected graphs, it is perhaps tempting to remove the $\alpha(n)$ factor in Theorem 4.1, leading to the truly linear bound $O(n + m)$. Unfortunately, it turns out that this is not the case, i.e., the factor $\alpha(n)$ is necessary. The key insight is that Dyck reachability on bidirected graphs can encode the Union-Find problem, which suffers the $\Omega(n \cdot \alpha(n))$ lower bound [Tarjan 1979; Banachowski 1980].

THEOREM 4.2 ([CHATTERJEE ET AL. 2018]). *Any Dyck-reachability algorithm for bidirected graphs with n nodes and $m = \Omega(n)$ edges requires $\Omega(m + n \cdot \alpha(n))$ time.*

5. INTERLEAVED BIDIRECTED DYCK REACHABILITY

In this section we turn our attention to *interleaved* bidirected Dyck reachability. This setting concerns two Dyck languages, on separate alphabets, that are *interleaved*: a path witnessing reachability must produce, simultaneously, properly parenthesis words, one for each Dyck language. We first give a formal definition of this interleaving, and then outline some recent results concerning various classes of this problem.

Interleaved Dyck languages. Given natural numbers $k_1, k_2 \in \mathbb{N}$, consider two disjoint matched (parenthesis) alphabets Σ_1, Σ_2, and let $\mathcal{D}(\Sigma_i)$ be the Dyck language with respect to Σ_i. Given some word $w \in (\Sigma_1 \cup \Sigma_2)^*$, we denote by $w \restriction \Sigma$ the projection of w on the alphabet Σ_i. The interleaved Dyck language over Σ_1, Σ_2 is defined as

$$\mathcal{D}(\Sigma_1) \odot \mathcal{D}(\Sigma_2) = \left\{\sigma \in (\Sigma_1 \cup \Sigma_2)^* : \sigma \restriction \Sigma_i \in \mathcal{D}(\Sigma_i) \text{ for each } i \in [2]\right\} .$$

For example, given $\Sigma_1 = \{\alpha^1, \overline{\alpha^1}\}$ and $\Sigma_2 = \{\alpha^2, \overline{\alpha^2}\}$, we have $\alpha^1\alpha^2\overline{\alpha^1\alpha^2} \in \mathcal{D}(\Sigma_1) \odot \mathcal{D}(\Sigma_2)$. As per standard so far, we typically ignore the alphabets Σ_i and write $\mathcal{D}_{k_1} \odot \mathcal{D}_{k_2}$ for the interleaved Dyck language over two implicit alphabets, each of size k_i.

Interleaved Dyck reachability arises naturally in various static analysis settings, where each Dyck language is used to capture different kind of analysis sensivities, i.e., precision with respect to specific programming features, such as calling contexts and field accesses [Reps 2000; Späth et al. 2019; Kjelstrøm and Pavlogiannis 2022]. See Figure 9 for an illustration. It is not hard to see that the underlying reachability problem becomes undecidable on general graphs [Reps 2000]. Here we outline some results for interleaved Dyck reachability on *bidirected* graphs, as well as on subclasses where one Dyck langauge is over one parenthesis symbol (i.e., a counter, similarly to Section 3.3).

$\mathcal{D}_1 \odot \mathcal{D}_1$ reachability. Perhaps the simplest instance of interleaving occurs when we only have two Dyck languages, each over a single parenthesis symbol. Hence, each Dyck language acts as a counter, and the setting is known as two-dimensional VASS. Even for non-bidirected graphs, the reachability problem is known to be NL-complete (and thus in PTime) [Englert et al. 2016].

Bidirectedness allows to simplify $\mathcal{D}_1 \odot \mathcal{D}_1$ reachability, thereby obtaining a small polynomial complexity bound. The key technical underpinning of bidirectedness is the following. If a node t is $\mathcal{D}_1 \odot \mathcal{D}_1$-reachable from a node s, then there exists a witness path along which both counters remain bounded by $O(n^2)$ [Kjelstrøm and Pavlogiannis 2022]. This leads to a simple algorithm for the problem. We first flatten the input graph G on the first counter, i.e., we create a graph G' in which every node also stores one of the $O(n^2)$ values that the first counter can take, and edges in G manipulating the first counter become ϵ-labeled in G', while connecting nodes with the corresponding counter

```
...
void setX(Point p, int v){
  p.x = v;
}
int getX(Point r){
  return r.x;
}
...
```

```
...
int a,b;
Point q;
setX(q, a);
b=getX(q);
...
```

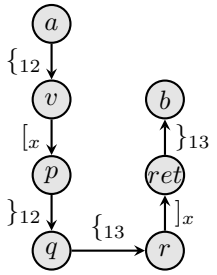

Fig. 9: *(Left):* A program on which to perform context-sensitive and field-sensitive alias analysis. *(Right):* A graph where the curly braces model context sensitivity and square brackets model field sensitivity. The path $P\colon a \rightsquigarrow b$ produces two interleaved words, $\{_{12}\}_{12}\{_{13}\}_{13}$ and $[_x]_x$. As both words are balanced, P witnesses that b may alias a.

changes (e.g, an edge $u \xrightarrow{+1} v$ in G connects all nodes $(u, i) \to (v, i+1)$ in G'). Note that G' has n^3 nodes, but it is a simple bidirected counter graph (i.e., with respect to the counter that we did not flatten). It thus suffices to use Theorem 4.1 to solve bidirected reachability on G'. We may further assume that G (and thus also G') is sparse, as any node with 3 or more outgoing edges has two edges labeled with the same alphabet, and thus the corresponding endpoints can be contracted to a single node (recall the bidirectedness algorithm in Section 4). We thus arrive at the following theorem.

THEOREM 5.1. *Bidirected $\mathcal{D}_1 \odot \mathcal{D}_1$ reachability can be computed in $O(n^3 \cdot \alpha(n))$ time, where $\alpha(n)$ is the inverse Ackermann function.*

$\mathcal{D}_k \odot \mathcal{D}_1$ reachability. The next step is to allow one of the two Dyck languages be over multiple parentheses, giving rise to $\mathcal{D}_k \odot \mathcal{D}_1$ reachability. The setting is known as one-dimensional Pushdown VASS (PVASS). Interestingly, the decidability of reachability for non-bidirected PVASS has remained a long-standing open problem [Schmitz and Zetzsche 2019; Ganardi et al. 2022]. The more general *coverability* problem asks whether a node t is reachable from a node s, as witnessed by paths whose counter can have any value at the end (but must remain non-negative along the path), and is known to be decidable [Leroux et al. 2015]. Focusing on bidirected graphs, the problem turns out to be decidable. The crux of the proof is on the insight that coverability, together with bidirectedness, implies reachability. This observation was first made in [Kjelstrøm and Pavlogiannis 2022], and was combined with the decidability of coverability [Leroux et al. 2015] to obtain the decidability of reachability on bidirected graphs. Later, this result was improved to PSPACE [Ganardi et al. 2022].

THEOREM 5.2 ([GANARDI ET AL. 2022]). *Bidirected $\mathcal{D}_k \odot \mathcal{D}_1$ reachability is in* PSPACE.

$\mathcal{D}_k \odot \mathcal{D}_k$ reachability. Finally, we look at bidirected $\mathcal{D}_k \odot \mathcal{D}_k$ reachability. The non-bidirected case is well-known to be undecidable [Reps 2000], as the two Dyck languages can encode the intersection of CFLs, for which the emptiness of intersection is undecidable [Hopcroft et al. 2006]. The bidirected case was studied recently, showing that undecidability remains. The proof is based on a reduction from the corresponding non-bidirected case [Kjelstrøm and Pavlogiannis 2022].

THEOREM 5.3 ([KJELSTRØM AND PAVLOGIANNIS 2022]). *Bidirected $\mathcal{D}_k \odot \mathcal{D}_k$ reachability is undecidable.*

6. RELATED WORK

The CFL/Dyck reachability problem has applications to a very wide range of static analyses, such as interprocedural data-flow analysis [Reps et al. 1995], slicing [Reps et al. 1994], shape analysis [Reps 1995], impact analysis [Arnold 1996], type-based flow analysis [Rehof and Fähndrich 2001], taint analysis [Huang et al. 2015], data-dependence analysis [Tang et al. 2017], alias/points-to analysis [Lhoták and Hendren 2006; Zheng and Rugina 2008; Xu et al. 2009], and many others. RSM reachability has also been studied under the lens of parameterized complexity, and in particular under the assumption that modules have low treewidth [Chatterjee et al. 2015; Chatterjee et al. 2016; Chatterjee et al. 2020] a property that is known to hold for the control-flow graphs of most programs [Thorup 1998; Chatterjee et al. 2017].

Bidirected graphs as program models have been a standard approach to handle mutable heap data [Sridharan and Bodík 2006; Xu et al. 2009] – though it can sometimes be relaxed for read-only accesses [Milanova 2020], and the de-facto formulation of demand-driven points-to analyses [Sridharan et al. 2005; Zheng and Rugina 2008; Yan et al. 2011; Vedurada and Nandivada 2019]. Bidirectedness is also used for CFL-reachability formulations of pointer analysis [Reps 1997]. The algorithmic benefit of bidirectedness was highlighted in [Yuan and Eugster 2009], where an $O(n \cdot \log n)$ algorithm was presented when the underlying graph is a bidirected tree. Later this bound was improved to $O(n)$ for trees, while the problem was shown to take $O(n^2)$ time (and $O(n \cdot \log n)$ expected time) on general bidirected graphs, thereby breaking below the cubic bound [Zhang et al. 2013]. This sequence of results ended with the work of [Chatterjee et al. 2018], the results of which have been presented here.

Reachability in VASS has been a long-studied problem. Though its decidability has been known for many decades [Mayr 1981], its complexity was settled only recently to Ackermann complete [Leroux and Schmitz 2019; Czerwiński and Orlikowski 2022]. In the case of PVASS, the decidability of reachability in one dimension is open [Schmitz and Zetzsche 2019]. On the other hand, it was recently shown that bidirectedness suffices to make the problem decidable in all dimensions [Ganardi et al. 2022].

7. CONCLUSION AND FUTURE DIRECTIONS

CFL/Dyck reachability is a fascinating problem with truly numerous applications in static program analysis. In this paper we have focused on algorithmic aspects of the problem, for which there has been a lot of progress recently. In particular, we have looked into the following classes.

(1) Traditional Dyck reachability $\mathcal{D}_k$, for which we have seen upper and lower bounds depending on k.
(2) Dyck reachability $\mathcal{D}_k$ on bidirected graphs, which is solved faster than traditional Dyck reachability.
(3) Interleaved Dyck reachability on general and bidirected graphs, which is a significantly harder problem, also closely connected to VASS.

A number of exciting questions are still open. Is $\mathcal{D}_1$ reachability truly harder to solve than plain reachability, or can the $\log^2 n$-factor of the former be improved? Does general $\mathcal{D}_k$ reachability admit a truly sub-cubic algorithm? Although many researchers find this unlikely, it would be insightful to strengthen this belief with other complexity lower bounds, relating the problem to popular hypotheses in fine-grained complexity theory. Finally, in the context of VASS, is $\mathcal{D}_k \odot \mathcal{D}_1$ reachability decidable? And does $\mathcal{D}_k \odot \mathcal{D}_1$ reachability on bidirected graphs admit a polynomial time-bound?

The demand for more precise static analyses, particularly while programming languages are becoming more feature-rich, suggests further progress on graph-modeling aspects. In particular, one may consider moving one level up in the Chomsky hierarchy, i.e., lifting CFL-reachability to context-*sensitive* language (CSL) reachabiliy. The provably higher expressive power that comes with CSLs makes them a potentially more useful tool than CFLs. If such modeling indeed proves useful, it would further open up a new collection of algorithmic problems, similarly to the case of context-free models.

ACKNOWLEDGMENTS

This work was partially supported by a research grant (VIL42117) from VILLUM FONDEN.

References

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1968. Time and tape complexity of pushdown automaton languages. *Information and Control* 13, 3 (1968), 186–206. DOI: http://dx.doi.org/https://doi.org/10.1016/S0019-9958(68)91087-5

Rajeev Alur and P. Madhusudan. 2004. Visibly Pushdown Languages. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing (STOC '04)*. Association for Computing Machinery, New York, NY, USA, 202–211. DOI: http://dx.doi.org/10.1145/1007352.1007390

Vladimir L'vovich Arlazarov, Yefim A Dinitz, MA Kronrod, and IgorAleksandrovich Faradzhev. 1970. On economical construction of the transitive closure of an oriented graph. In *Doklady Akademii Nauk*, Vol. 194. Russian Academy of Sciences, 487–488.

Robert S. Arnold. 1996. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA.

Lech Banachowski. 1980. A complement to Tarjan's result about the lower bound on the complexity of the set union problem. *Inform. Process. Lett.* 11, 2 (1980), 59 – 65.

Marco L. Carmosino, Jiawei Gao, Russell Impagliazzo, Ivan Mihajlin, Ramamohan Paturi, and Stefan Schneider. 2016. Nondeterministic Extensions of the Strong Exponential Time Hypothesis and Consequences for Non-Reducibility. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science (ITCS '16)*. Association for Computing Machinery, New York, NY, USA, 261–270. DOI: http://dx.doi.org/10.1145/2840728.2840746

Jakob Cetti Hansen, Adam Husted Kjelstrøm, and Andreas Pavlogiannis. 2021. Tight bounds for reachability problems on one-counter and pushdown systems. *Inform. Process. Lett.* 171 (2021), 106135. DOI: http://dx.doi.org/https://doi.org/10.1016/j.ipl.2021.106135

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, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2020. Optimal and Perfectly Parallel Algorithms for On-demand Data-Flow Analysis. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 112–140.

Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. 2017. JTDec: A Tool for Tree Decompositions in Soot. In *Automated Technology for Verification and Analysis*, Deepak D'Souza and K. Narayan Kumar (Eds.). Springer International Publishing, Cham, 59–66.

Krishnendu Chatterjee, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. 2016. Optimal Reachability and a Space-Time Tradeoff for Distance Queries in Constant-Treewidth Graphs. In *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*. 28:1–28:17.

Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Andreas Pavlogiannis, and Prateesh Goyal. 2015. Faster Algorithms for Algebraic Path Properties in Recursive State Machines with Constant Treewidth. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 97–109. DOI: http://dx.doi.org/10.1145/2676726.2676979

Swarat Chaudhuri. 2008. Subcubic Algorithms for Recursive State Machines. *SIGPLAN Not.* 43, 1 (Jan. 2008), 159–169. DOI: http://dx.doi.org/10.1145/1328897.1328460

Dmitry Chistikov, Rupak Majumdar, and Philipp Schepper. 2022. Subcubic Certificates for CFL Reachability. *Proc. ACM Program. Lang.* 6, POPL, Article 41 (jan 2022), 29 pages. DOI: http://dx.doi.org/10.1145/3498702

Noam Chomsky and Marcel-Paul Schützenberger. 1963. The Algebraic Theory of Context-Free Languages*. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 35. Elsevier, 118–161. DOI: http://dx.doi.org/https://doi.org/10.1016/S0049-237X(08)72023-8

Wojciech Czerwiński and Łukasz Orlikowski. 2022. Reachability in Vector Addition Systems is Ackermann-complete. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. 1229–1240. DOI: http://dx.doi.org/10.1109/FOCS52979.2021.00120

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

Matthias Englert, Ranko Lazić, and Patrick Totzke. 2016. Reachability in Two-Dimensional Unary Vector Addition Systems with States is NL-Complete. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '16)*. Association for Computing Machinery, New York, NY, USA, 477–484. DOI: http://dx.doi.org/10.1145/2933575.2933577

Moses Ganardi, Rupak Majumdar, Andreas Pavlogiannis, Lia Schütze, and Georg Zetzsche. 2022. Reachability in Bidirected Pushdown VASS. In *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022) (Leibniz International Proceedings in Informatics (LIPIcs))*, Mikołaj Bojańczyk, Emanuela Merelli, and David P. Woodruff (Eds.), Vol. 229. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 124:1–124:20. DOI: http://dx.doi.org/10.4230/LIPIcs.ICALP.2022.124

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

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2001. Introduction to Automata Theory, Languages, and Computation, 2nd Edition. *SIGACT News* 32, 1 (mar 2001), 60–65. DOI: http://dx.doi.org/10.1145/568438.568455

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and Precise Taint Analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 106–117. DOI: http://dx.doi.org/10.1145/2771783.2771803

Adam Husted Kjelstrøm and Andreas Pavlogiannis. 2022. The Decidability and Complexity of Interleaved Bidirected Dyck Reachability. *Proc. ACM Program. Lang.* 6, POPL, Article 12 (jan 2022), 26 pages. DOI: http://dx.doi.org/10.1145/3498673

Lillian Lee. 2002. Fast Context-Free Grammar Parsing Requires Fast Boolean Matrix Multiplication. *J. ACM* 49, 1 (jan 2002), 1–15. DOI: http://dx.doi.org/10.1145/505241.505242

Jérôme Leroux and Sylvain Schmitz. 2019. Reachability in Vector Addition Systems is Primitive-Recursive in Fixed Dimension. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '19)*. IEEE Press, Article 50, 13 pages.

Jérôme Leroux, Grégoire Sutre, and Patrick Totzke. 2015. On the Coverability Problem for Pushdown Vector Addition Systems in One Dimension. In *Automata, Languages, and Programming*, Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 324–336.

Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *Proceedings of the 15th International Conference on Compiler Construction (CC)*. 47–64.

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 (Jan. 2021), 29 pages. DOI: http://dx.doi.org/10.1145/3434315

Ernst W. Mayr. 1981. An Algorithm for the General Petri Net Reachability Problem. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing (STOC '81)*. Association for Computing Machinery, New York, NY, USA, 238–246. DOI: http://dx.doi.org/10.1145/800076.802477

Ana Milanova. 2020. FlowCFL: Generalized Type-Based Reachability Analysis: Graph Reduction and Equivalence of CFL-Based and Type-Based Reachability. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 178 (Nov. 2020), 29 pages. DOI: http://dx.doi.org/10.1145/3428246

Laurent Pierre. 1992. Rational indexes of generators of the cone of context-free languages. *Theoretical Computer Science* 95, 2 (1992), 279 – 305. DOI: http://dx.doi.org/https://doi.org/10.1016/0304-3975(92)90269-L

Jakob Rehof and Manuel Fähndrich. 2001. Type-base Flow Analysis: From Polymorphic Subtyping to CFL-reachability. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 54–66.

Thomas Reps. 1995. Shape Analysis As a Generalized Path Problem. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '95)*. ACM, 1–11.

Thomas Reps. 1997. Program Analysis via Graph Reachability. In *Proceedings of the 1997 International Symposium on Logic Programming (ILPS)*. 5–19.

Thomas Reps. 2000. Undecidability of Context-sensitive Data-dependence Analysis. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 162–186.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*. ACM, New York, NY, USA.

Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. 1994. Speeding Up Slicing. *SIGSOFT Softw. Eng. Notes* 19, 5 (1994), 11–20.

Neil Robertson and P.D. Seymour. 1983. Graph minors. I. Excluding a forest. *Journal of Combinatorial Theory, Series B* 35, 1 (1983), 39–61. DOI: http://dx.doi.org/https://doi.org/10.1016/0095-8956(83)90079-5

Wojciech Rytter. 1985. The Complexity of Two-Way Pushdown Automata and Recursive Programs. In *Combinatorial Algorithms on Words*, Alberto Apostolico and Zvi Galil (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 341–356.

Wojciech Rytter. 1986. Fast Recognition of Pushdown Automaton and Context-Free Languages. *Inf. Control* 67, 1–3 (oct 1986), 12–22. DOI: http://dx.doi.org/10.1016/S0019-9958(85)80024-3

Sylvain Schmitz and Georg Zetzsche. 2019. Coverability Is Undecidable in One-Dimensional Pushdown Vector Addition Systems with Resets. In *Reachability Problems*, Emmanuel Filiot, Raphaël Jungers, and Igor Potapov (Eds.). Springer International Publishing, Cham, 193–201.

Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, Flow-, and Field-Sensitive Data-Flow Analysis Using Synchronized Pushdown Systems. *Proc. ACM Program. Lang.* 3, POPL, Article 48 (Jan. 2019), 29 pages. DOI: http://dx.doi.org/10.1145/3290361

Manu Sridharan and Rastislav Bodík. 2006. Refinement-based Context-sensitive Points-to Analysis for Java. *SIGPLAN Not.* 41, 6 (2006), 387–400.

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*.

Siddharth Suri and Sergei Vassilvitskii. 2011. Counting Triangles and the Curse of the Last Reducer. In *Proceedings of the 20th International Conference on World Wide Web (WWW '11)*. Association for Computing Machinery, New York, NY, USA, 607–614. DOI: http://dx.doi.org/10.1145/1963405.1963491

Hao Tang, Di Wang, Yingfei Xiong, Lingming Zhang, Xiaoyin Wang, and Lu Zhang. 2017. Conditional Dyck-CFL Reachability Analysis for Complete and Efficient Library Summarization. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 880–908.

Robert Endre Tarjan. 1979. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. System Sci.* 18, 2 (1979), 110 – 127.

Mikkel Thorup. 1998. All Structured Programs Have Small Tree Width and Good Register Allocation. *Information and Computation* (1998).

Leslie G. Valiant. 1975. General Context-Free Recognition in Less than Cubic Time. *J. Comput. Syst. Sci.* 10, 2 (apr 1975), 308–315. DOI: http://dx.doi.org/10.1016/S0022-0000(75)80046-8

Jyothi Vedurada and V. Krishna Nandivada. 2019. Batch Alias Analysis. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*. IEEE Press, 936–948. DOI: http://dx.doi.org/10.1109/ASE.2019.00091

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

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.

Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven Context-sensitive Alias Analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA)*. 155–165.

Hao Yuan and Patrick Eugster. 2009. An Efficient Algorithm for Solving the Dyck-CFL Reachability Problem on Trees. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009 (ESOP)*. 175–189.

Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 2013. Fast Algorithms for Dyck-CFL-Reachability with Applications to Alias Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 435–446. DOI: http://dx.doi.org/10.1145/2491956.2462159

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.