Dummy title

Ekaterina Shemetova. Open Access
Dummy University Computing Laboratory, [optional: Address], Country
My second affiliation, Country

Vladimir Kutuev. Public[1]
Department of Informatics, Dummy College, [optional: Address], Country

Rustam Azimov. Public[2]
Department of Informatics, Dummy College, [optional: Address], Country

Egor Orachev. Public[3]
Department of Informatics, Dummy College, [optional: Address], Country

Ilya Epelbaum. Public[4]
Department of Informatics, Dummy College, [optional: Address], Country

Semyon Grigorev. Public[5]
Department of Informatics, Dummy College, [optional: Address], Country

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent convallis orci arcu, eu mollis dolor. Aliquam eleifend suscipit lacinia. Maecenas quam mi, porta ut lacinia sed, convallis ac dui. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse potenti.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases CFL-reachability, interprocedural program analysis, linear algebra

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding *Ekaterina Shemetova. Open Access*: (Optional) author-specific funding acknowledgements
Vladimir Kutuev. Public: [funding]
Rustam Azimov. Public: [funding]
Egor Orachev. Public: [funding]
Ilya Epelbaum. Public: [funding]
Semyon Grigorev. Public: [funding]

Acknowledgements I want to thank ...

1 Introduction

Interprocedural static program analyses are widely applied to support the development of high-quality software. It helps developers detect potential bugs and security vulnerabilities in a program's source code. The popular approach to formualate a large body of interprocedural sratic analysis problems, such as points-to and dataflow analysis, is to use the context-free (CFL) reachability framework [43]. The CFL-reachability problem is to find realizable paths in the graph using a context-free language. The widely used example of such context-free

[1] Optional footnote, e.g. to mark corresponding author
[2] Optional footnote, e.g. to mark corresponding author
[3] Optional footnote, e.g. to mark corresponding author
[4] Optional footnote, e.g. to mark corresponding author
[5] Optional footnote, e.g. to mark corresponding author

language is Dyck language, which treats method calls and returns as pairs of balanced parentheses.

Motivation

The CFL-reachability problem has cubic $O(n^3)$ time complexity in general case, and, despite all efforts, no algorithm faster than $O(n^3/\log n)$ [11] has been obtained. Therefore, the CFL-reachability is known to have a so called "cubic bottleneck", which is often reffered to as "cubic bottleneck in static analysis" [19]. All this leads to the fact that precise CFL-reachability-based analysis can be expensive when applied to large programs.

Our approach

One promising way to achieve high-performance solutions for graph analysis problems is to reduce them to linear algebra operations. To facilitate this approach, the description of basic linear algebra primitives GraphBLAS API [28] was proposed. Evaluation of the libraries that implement this API, such as SuiteSparce [13] and CombBLAS [9], show that reduction to linear algebra is a good way to utilize high-performance parallel and distributed computations for graph analysis. A matrix-based approach to graph algorithms allows the graph algorithms community to leverage the decades of work in creating optimized parallel algorithms for matrix computations. Moreover, the bulk graph/matrix operations allow a serial, parallel, or GPU-based library to optimize the graph operations.

Our contributions

To summarize, we make the following contributions in this paper.

1. We obtain the linear algebra based formulation of the CFL-reachability problem and show that our solution has state-of-the-art theoretical time complexity.
2. We implement the described algorithm on top of pygraphblas library, which is full implementation of GraphBLAS API.
3. To validate scalability, high perfomance and generality, we use our tool for running CFL-reachability based alias analysis for C [68] and field-sensitive points-to analysis for Java [52]. We analyzed large-scale software systems: ???. Our experiments show promising results: ???.

2 Background

2.1 CFL-reachability

Let $G = \langle \Sigma, N, S, P \rangle$ be a context-free grammar, where Σ is a finite set of terminals (or terminal alphabet), N is a finite set of nonterminals (or nonterminal alphabet), $S \in N$ is a start nonterminal, P is a finite set of productions (grammar rules) of form $N_i \to \alpha$ where $N_i \in N$, $\alpha \in (\Sigma \cup N)^*$.

Let $\mathcal{G} = \langle V, E, \Sigma \rangle$ be a directed graph with edges labeled by elements of Σ. The notation (a, i, j) denotes an edge in $\mathcal{G}$ from node i to node j labeled with symbol a. Each path in $\mathcal{G}$ defines a word over Σ by concatenating, in order, the labels of the edges in the path. A path in $\mathcal{G}$ is an S-path if its word is in the language generated by context-free grammar G. A path in $\mathcal{G}$ is an A-path for some $A \in N$ if its word is generated by non-terminal $A \in N$ for some grammar $G = \langle \Sigma, N, S, P \rangle$. The all-pairs *CFL-reachability problem* determines the pairs of vertices (i, j), where there exists an S-path from i to j in $\mathcal{G}$.

2.2 Points-to analysis as CFL-reachability problem

Points-to analysis is traditionally presented as the problem of computing a points-to relation that conservatively maps each pointer variable to the heap objects it can point to at runtime. Two lvalue expressions are *memory aliases* if they might denote the same memory location. Two expressions are *value aliases* if they might evaluate to the same pointer value.

The most commonly used and actively studied formulation of points-to analysis is Andersen's Pointer Analysis [2].

Analysis is *field-sensitive*, when it treats each memory object as a collection of disjoint fields.

In this work we use the CFL-reachability formulation of Andersen's Pointer Analysis for C programming language by Zheng and Rugina [68] (2.2.1) and its field-sensitive variant for Java by Sridharan et al. [52] (2.2.2).

2.2.1 Memory aliases for C

Graph

The graph representation of all expressions and assignments in the program is called *Program Expression Graph* (PEG). The nodes of the graph represent program expressions, and edges are of four kinds:

1. Pointer dereference edges (d): for each dereference $*e$, thereis a d-edge from e to $*e$
2. Assignment edges (a): for each assignment: $*e_1 = e_2$, there is an a-edge from e_2 to $*e_1$
3. For each a-edge, there is a corresponding edge in the opposite direction, denoted by $\bar{a}$
4. For each d-edge, there is a corresponding edge in the opposite direction, denoted by $\bar{d}$.

The example of PEG $\mathcal{G}$ for expressions {`*x, x, &x, *y, y, &y`} and assignment `y = x;` is illustrated in Figure 1.

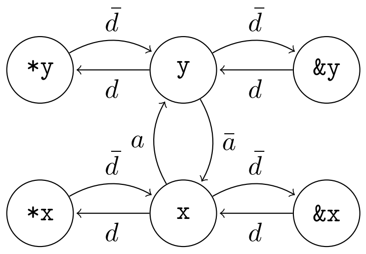

Figure 1 Program Expression Graph $\mathcal{G}$ for program `y = x;`

Grammar

The context-free grammar G_1 for aliasing problems in EBNF notation, where the star symbol is the Kleene star operator, and the question mark indicates an optional term, is as follows:

- $S \to \bar{d}\ V\ d$
- $V \to (S?\bar{a})^*\ S?\ (aS?)^*$

Terminal symbols a and d represent assignments and dereference edges in the expression graph. Terminal symbols $\bar{a}$ ($\bar{d}$) represents a corresponding edge in the opposite direction for a-edge (d-edge respectively). Start non-terminal S models memory aliasing relations, and non-terminal V represents value aliasing relations.

2.2.2 Field-sensitive points-to analysis for Java

Graph

A program is represented by a *Pointer Assignment Graph* (PAG), a directed graph that records pointer flow in a program. Nodes in the graph are either program variables, or heap-object creation sites. Edges of the graph are defined as follows:

1. *assign*-edges: for each assignment of program variable v_1 to other program variable v_2 there is a *assign*-edge from node v_2 to node v_1
2. *alloc*-edges: for each allocation of heap object h_1 and the variable v_1 it is assigned to (i.e. v_1 `= new Obj();`) there is an *alloc*-edge from node h_1 to node v_1
3. $load_f$-edges for all $f \in$ fields: for each reading from a field f of variable v_1 (i.e. $v_2 = v_1.f$) there is a $load_f$-edge from node v_1 to node v_2
4. $store_f$-edges for all $f \in$ fields: for each writing to a field f of variable v_1 (i.e. $v_1.f = v_2$) there is a $store_f$-edge from node v_2 to node v_1

Also, for each edge $n_1 \rightarrow n_2$ labelled l in the graph there is an edge $n_2 \rightarrow n_1$ labelled $\bar{l}$. Given a graph with barred edges, a reverse path $\bar{\pi}$ can be constructed for any path π by reversing the order of the edges in π and replacing each edge in π with its inverse, substituting barred edges for standard edges.

Grammar

The productions of the context-free grammar G_2 in EBNF notation are described as below:

- $alias \rightarrow flowsTo\ \overline{flowsTo}$
- $flowsTo \rightarrow alloc\ (assign \mid store_f\ alias\ load_f)^*$ for all $f \in$ fields
- $\overline{flowsTo} \rightarrow (\overline{assign} \mid \overline{load_f}\ alias\ \overline{store_f})^*\ \overline{alloc}$ for all $f \in$ fields.

A non-terminal $flowsTo$ represents paths between object-creations and variables, a $\overline{flowsTo}$-path represents the standard points-to relation, and an *alias*-path exists between two variables that might be aliases during the program execution.

2.3 Recursive State Machines

In this work we use the notion of *Finite-State Machine* (FSM).

▶ **Definition 1.** *A deterministic finite-state machine without ε-transitions T is a tuple $\langle \Sigma, Q, Q_s, Q_f, \delta \rangle$, where:*

- *Σ is an input alphabet,*
- *Q is a finite set of states,*
- *$Q_s \subseteq Q$ is a set of start (or initial) states,*
- *$Q_f \subseteq Q$ is a set of final states,*
- *$\delta : Q \times \Sigma \rightarrow Q$ is a transition function.*

It is well known, that every regular expression can be converted to deterministic FSM without ε-transitions [23]. Note, that an edge-labeled graph can be viewed as an FSM where edges represent transitions and all vertices are both start and final at the same time.

While a regular expression can be transformed to an FSM, a context-free grammar can be transformed to a *Recursive State Machine* (RSM) in a similar fashion. In our work, we use the following definition of RSM based on [1].

▶ **Definition 2.** *A recursive state machine R over a finite alphabet Σ is defined as a tuple of elements $\langle B, m, \{C_i\}_{i \in B} \rangle$, where:*

- *B is a finite set of labels of boxes,*
- *$m \in B$ is an initial box label,*
- *Set of component state machines or boxes, where $C_i = (\Sigma \cup B, Q_i, q_i^0, F_i, \delta_i)$:*
  - *$\Sigma \cup B$ is a set of symbols, $\Sigma \cap B = \varnothing$,*
  - *Q_i is a finite set of states, where $Q_i \cap Q_j = \varnothing, \forall i \neq j$,*
  - *q_i^0 is an initial state for C_i,*
  - *F_i is a set of final states for C_i, where $F_i \subseteq Q_i$,*
  - *$\delta_i : Q_i \times (\Sigma \cup B) \to Q_i$ is a transition function.*

RSM behaves as a set of finite state machines (or FSM). Each such FSM is called a *box* or a *component state machine*. A box works similarly to the classic FSM, but it also handles additional *recursive calls* and employs an implicit *call stack* to *call* one component from another and then return execution flow back.

▶ **Definition 3.** *The size of RSM $|R|$ is defined as the sum of the number of states in all boxes.*

The size of RSM for some grammar $G = \langle \Sigma, N, S, P \rangle$ does not exceed its size $|G|$, which is defined as the sum of the sizes of its productions $|P|$.

RSM for a grammar G_1 is illustrated in Figure 2.

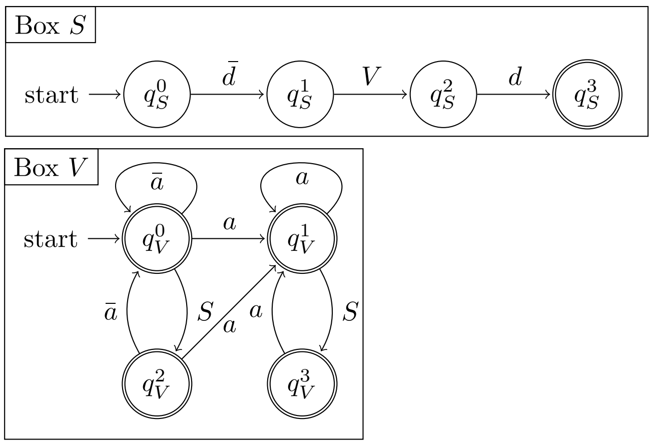

Figure 2 The recursive state machine R for grammar G_1

2.4 Linear algebra

Matrix representation of machines

▶ **Definition 4.** *An adjacency matrix for an edge-labeled directed graph $\mathcal{G} = \langle V, E, L \rangle$ is a matrix M, where:*

- *M has size $|V| \times |V|$*
- *$M[i, j] = \{l \mid e = (i, l, j) \in E\}$*

Adjacency matrix $\mathcal{M}_2$ of the labeled graph (FSM) $\mathcal{G}$ is

$$\mathcal{M}_2 = \begin{pmatrix} \emptyset & \{\bar{d}\} & \emptyset & \emptyset & \emptyset & \emptyset \\ \{d\} & \emptyset & \{\bar{d}\} & \emptyset & \{a\} & \emptyset \\ \emptyset & \{d\} & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \{\bar{d}\} & \emptyset \\ \emptyset & \{\bar{a}\} & \emptyset & \{d\} & \emptyset & \{\bar{d}\} \\ \emptyset & \emptyset & \emptyset & \emptyset & \{d\} & \emptyset \end{pmatrix}.$$

Similarly to an FSM, an RSM can be represented as a labeled graph and, hence, as an adjacency matrix. For our example, $\mathcal{M}_1$ for the RSM R from Figure 2 is:

$$\mathcal{M}_1 = \begin{pmatrix} \emptyset & \{\bar{d}\} & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \{V\} & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \{d\} & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \{\bar{a}\} & \{a\} & \{S\} & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{a\} & \emptyset & \{S\} \\ \emptyset & \emptyset & \emptyset & \emptyset & \{\bar{a}\} & \{a\} & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{a\} & \emptyset & \emptyset \end{pmatrix}$$

Graph Kronecker product and machines intersection

▶ **Definition 5.** *Given two matrices A and B of sizes $m_1 \times n_1$ and $m_2 \times n_2$ respectively, with element-wise product operation $\cdot$, the Kronecker product of these two matrices is a new matrix $C = A \otimes B$ of size $m_1 * m_2 \times n_1 * n_2$ and*

$$C[u * m_2 + v, n_2 * p + q] = A[u,p] \cdot B[v,q].$$

▶ **Definition 6.** *Given two edge-labeled directed graphs $\mathcal{G}_1 = \langle V_1, E_1, L_1 \rangle$ and $\mathcal{G}_2 = \langle V_2, E_2, L_2 \rangle$, the Kronecker product of these two graphs is a edge-labeled directed graph $\mathcal{G} = \mathcal{G}_1 \otimes \mathcal{G}_2$, where $\mathcal{G} = \langle V, E, L \rangle$:*
- $V = V_1 \times V_2$
- $E = \{((u,v), l, (p,q)) \mid (u,l,p) \in E_1 \wedge (v,l,q) \in E_2\}$
- $L = L_1 \cap L_2$

The Kronecker product for graphs produces a new graph with a property that if and only if some path $(u,v)\pi(p,q)$ exists in the result graph then paths $u\pi_1 p$ and $v\pi_2 q$ exist in the input graphs, and $\omega(\pi) = \omega(\pi_1) = \omega(\pi_2)$. These paths π_1 and π_2 can easily be found from π by its definition.

The Kronecker product for directed graphs can be described as the Kronecker product of the corresponding adjacency matrices of graphs, what gives the following definition:

▶ **Definition 7.** *Given two adjacency matrices M_1 and M_2 of sizes $m_1 \times n_1$ and $m_2 \times n_2$ respectively for some directed graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, the Kronecker product of these two adjacency matrices is the adjacency matrix M of some graph $\mathcal{G}$, where M has size $m_1 * m_2 \times n_1 * n_2$ and*

$$M[u * m_2 + v, n_2 * p + q] = M_1[u,p] \cap M_2[v,q].$$

By definition, the Kronecker product for adjacency matrices gives an adjacency matrix with the same set of edges as in the resulting graph in the Definition 6. Thus, $M(\mathcal{G}) = M(\mathcal{G}_1) \otimes M(\mathcal{G}_2)$, where $\mathcal{G} = \mathcal{G}_1 \otimes \mathcal{G}_2$.

▶ **Definition 8.** *Given two finite state machines* $T_1 = \langle \Sigma, Q^1, Q^1_S, Q^1_F, \delta^1 \rangle$ *and* $T_2 = \langle \Sigma, Q^2, Q^2_S, Q^2_F, \delta^2 \rangle$*, the intersection of these two machines is a new FSM* $T = \langle \Sigma, Q, Q_S, Q_F, \delta \rangle$*, where:*

- $Q = Q^1 \times Q^2$
- $Q_S = Q^1_S \times Q^2_S$
- $Q_F = Q^1_F \times Q^2_F$
- $\delta : Q \times \Sigma \to Q$, $\delta(\langle q_1, q_2 \rangle, s) = \langle q'_1, q'_2 \rangle$, *if* $\delta(q_1, s) = q'_1$ *and* $\delta(q_2, s) = q'_2$

According to [23] an FSM intersection defines the machine for which $L(T) = L(T_1) \cap L(T_2)$.

3 CFL-reachability in terms of linear algebra

3.1 Algorithm description

The algorithm is based on the generalization of the FSM intersection for an RSM, and the edge-labeled directed input graph. Since the RSM is composed as a set of FSMs, it could easily be presented as an adjacency matrix for some graph over the set of labels. As shown in the Definition 7, we can apply the Kronecker product for matrices to *intersect* the RSM and the input graph to some extent. But the RSM contains nonterminal symbols with the additional logic of *recursive calls*, which requires a *transitive closure* step to extract such symbols.

The core idea of the algorithm comes from the Kronecker product and transitive closure. The algorithm boils down to the evaluation of the iterative Kronecker product for the adjacency matrix $\mathcal{M}_1$ of the RSM R and the adjacency matrix $\mathcal{M}_2$ of the input graph $\mathcal{G}$, followed by the transitive closure, extraction of new reachability information and updating the graph adjacency matrix $\mathcal{M}_2$. These steps are described in Algorithm 1.

New elements of the Kronecker product are computed in Line 9 of the Algorithm 1. Function $DTC(T, K)$ from Algorithm 1 takes transitive closure matrix T and a matrix K with edges to be inserted, maintains T under edge insertions and returns pairs of vertices (i, j) such that j *became reachable* from i after the insertion of some edge from K. Then the new reachable pairs are validated in the lines 12-18: we are interested only in paths from start to final state of some box, therefore some pairs can be excluded from adding to $\mathcal{M}_2$. If $\mathcal{M}_2$ has changed after the insertion of the elements, we calculate the new elements of the Kronecker product and so on.

Notice that Algorithm 1 naturally allows one to calculate regular reachability or FSM intersection (in this case the main while loop takes only one iteration to actually append data). This feature may be useful for regular over-approximation of the CFL-reachability, for example, when one needs to make the finding of the point-to information less expensive [15, 52].

Graph Kronecker product and machines intersection

To effectively recompute the Kronecker product on each iteration, we employ the fact that it is left-distributive. Let $\mathcal{A}_2$ be a matrix with newly added elements and $\mathcal{B}_2$ be a matrix with all previously found elements, such that $\mathcal{M}_2 = \mathcal{A}_2 + \mathcal{B}_2$. Then by left-distributivity of the Kronecker product we have $K = \mathcal{M}_1 \otimes \mathcal{M}_2 = \mathcal{M}_1 \otimes (\mathcal{A}_2 + \mathcal{B}_2) = \mathcal{M}_1 \otimes \mathcal{A}_2 + \mathcal{M}_1 \otimes \mathcal{B}_2$. Note that $\mathcal{M}_1 \otimes \mathcal{B}_2$ is known from the previous iteration, so it is left to update some elements of K by computing $\mathcal{M}_1 \otimes \mathcal{A}_2$.

Algorithm 1 Kronecker product-based CFL-reachability

```
1:  function LA-CFL-REACHABILITY(G, $\mathcal{G}$)
2:      $R \leftarrow$ Recursive automata for $G$ with $r$ states
3:      $n \leftarrow$ The number of vertices in $\mathcal{G}$
4:      $\mathcal{M}_1 \leftarrow$ Adjacency matrix for $R$
5:      $\mathcal{M}_2 \leftarrow$ Adjacency matrix for $\mathcal{G}$
6:      $\Delta\mathcal{M}_2 \leftarrow \mathcal{M}_2$
7:      $K, T \leftarrow$ The empty matrices of size $rn \times rn$
8:      while Matrix $\mathcal{M}_2$ is changing do
9:          $K \leftarrow \mathcal{M}_1 \otimes \Delta\mathcal{M}_2$                    ▷ Evaluate Kronecker product
10:         $\Delta\mathcal{M}_2 \leftarrow$ The empty matrix
11:         $\Delta T \leftarrow DTC(T, K)$   ▷ Dynamic transitive closure, $\Delta T$ contains new reachable pairs
12:         for $(i, j) \in \Delta T$ do
13:             $s, f \leftarrow \lfloor i/r \rfloor, \lfloor j/r \rfloor$
14:             $x, y \leftarrow i \bmod n, j \bmod n$
15:             if $s$ is start state and $f$ is a final state for box $A$ then   ▷ Getting only accepting runs
16:                 $\Delta\mathcal{M}_2[x, y] \leftarrow \Delta\mathcal{M}_2[x, y] \cup \{A\}$
17:             end if
18:         end for
19:         $\mathcal{M}_2 \leftarrow \mathcal{M}_2 + \Delta\mathcal{M}_2$
20:     end while
21:     return $\mathcal{M}_2$
22: end function
```

Dynamic transitive closure

Note that the adjacency matrix $\mathcal{M}_2$ is changed incrementally i.e. elements (edges) are added to $\mathcal{M}_2$ at each iteration of the algorithm and are never deleted from it. So it is not necessary to recompute the whole product or transitive closure if some appropriate data structure is maintained. The fast computation of transitive closure can be obtained by using an incremental transitive closure technique. Let T be a transitive closure matrix of the graph $\mathcal{G}$ with n vertices. We use an approach by Ibaraki and Katoh [25] to maintain dynamic transitive closure. The key idea of their algorithm is to recalculate reachability information only for those vertices which become reachable after insertion of a certain edge. For each newly inserted edge (i, j) and every node $u \neq j$ of G such that $T[u, i] = 1$ and $T[u, j] = 0$, one needs to perform operation $T[u, v] = T[u, v] \wedge T[j, v]$ for every node v, where $1 \wedge 1 = 0 \wedge 0 = 1 \wedge 0 = 0$ and $0 \wedge 1 = 1$. In this way, transitive closure matrix T can be maintained under edge insertions in $O(n^3)$ total time.

We have modified this algorithm to achieve a logarithmic speed-up on a word RAM with word size $w = \theta(\log n)$. Notice that operations above are equivalent to the element-wise product of two vectors of size n, where multiplication operation is denoted as $\wedge$. To check whether $T[u, i] = 1$ and $T[u, j] = 0$ one needs to multiply two vectors: the first vector represents reachability of the given vertex i from other vertices $\{u_1, u_2, ..., u_n\}$ of the graph and the second vector represents the same for the given vertex j. The operation $T[u, v] \wedge T[j, v]$ also can be reduced to the computation of the element-wise product of two vectors of size n for the given u_k. The first vector contains the information whether vertices $\{v_1, v_2, ..., v_n\}$ of the graph are reachable from the given vertex u_k and the second vector represents the same for the given vertex j. The element-wise product of two vectors can be calculated naively in time $O(n)$. Thus, the time complexity of the transitive closure can be reduced by speeding up the element-wise product of two vectors of size n.

To achieve logarithmic speed-up, we use the Four Russians' trick [5]. Let us assume an architecture with word size $w = \theta(\log n)$. First, we split each vector into $n/\log n$ parts of size $\log n$. Then we create a table $\mathcal{T}$ such that $\mathcal{T}(a, b) = a \wedge b$ where $a, b \in \{0, 1\}^{\log n}$. This takes time $O(n^2 \log n)$, since there are $2^{\log n} = n$ variants of Boolean vectors of size $\log n$ and hence n^2 possible pairs of vectors (a, b) in total, and each component takes $O(\log n)$ time. Assuming constant-time logical operations on words, we can store a polynomial number of lookup tables (arrays) $\mathcal{T}_i$ (one array for each vector of size $\log n$), such that given an index of a table $\mathcal{T}_i$, and any $O(\log n)$ bit vector b, we can look up $\mathcal{T}_i(b)$ in constant time. The index of each array $\mathcal{T}_a$ is stored in array $\mathcal{T}$, which can be accessed in constant time for a given log-size vector a. Thus, we can calculate the product of two parts a and b of size $\log n$ in constant time using the table $\mathcal{T}$. There are $n/\log n$ such parts, so the element-wise product of two vectors of size n can be calculated in time $O(n/\log n)$ with $O(n^2 \log n)$ preprocessing.

3.2 Correctness and complexity

Correctness

▶ **Theorem 9.** *Let $\mathcal{G} = (V, E, L)$ be a graph and $G = \langle \Sigma, N, S, P\rangle$ be a grammar. Let $\mathcal{M}_2$ be a resulting adjacency matrix after the execution of the algorithm in Algorithm 1. Then for any valid indices i, j and for each nonterminal $A \in N$ the following statement holds: the non-terminal $A \in \mathcal{M}_2[i, j]$, iff there is a A-path from node i to node j in the graph $\mathcal{G}$.*

Proof. The main idea of the proof is to use induction on the height of the derivation tree obtained on each iteration. ◀

Complexity

▶ **Theorem 10.** *Let $\mathcal{G} = \langle V, E, L\rangle$ be a graph and $G = \langle \Sigma, N, S, P\rangle$ be a grammar. The Algorithm 1 calculates the resulting matrix $\mathcal{M}_2$ in $O(|P|^3 n^3/\log(|P|n))$ time on a word RAM with word size $w = \theta(\log |P|n)$, where $n = |V|$. Moreover, maintaining of the dynamic transitive closure dominates the cost of the algorithm.*

Proof. The most time-consuming steps of the algorithm are the computations of the Kronecker product and transitive closure.

Let $|\Delta\mathcal{M}_2|$ be the number of non-zero elements in a matrix $\Delta\mathcal{M}_2$. Consider the total time which is needed for computing the Kronecker products. The elements of the matrices $\Delta\mathcal{M}_2^{(i)}$ are pairwise distinct on every i-th iteration of the algorithm because ΔT contains only new reachable pairs of vertices. Therefore the total number of operations is $\sum_i \#$ of operations $(\mathcal{M}_1 \otimes \Delta\mathcal{M}_2^{(i)}) = |\mathcal{M}_1| \sum_i |\Delta\mathcal{M}_2^{(i)}| = (|N| + |\Sigma|)|P|^2 \sum_i |\Delta\mathcal{M}_2^{(i)}| = O((|N| + |\Sigma|)^2 |P|^2 n^2)$.

Now we derive the time complexity of maintaining the dynamic transitive closure. Notice that K has the size of the Kronecker product of $\mathcal{M}_1 \otimes \mathcal{M}_2$, which is equal to $rn \times rn = |P|n \times |P|n$ so no more than $|P|^2 n^2$ edges will be added during all iterations of the Algorithm 1. Checking whether $T[u, i] = 1$ and $T[u, j] = 0$ for every node $u \in V$ for each newly inserted edge (i, j) requires one multiplication of vectors per insertion, thus total time is $O(|P|^3 n^3/\log(|P|n))$. Note that after checking the condition, at least one element $T[u', j]$ changes value from 0 to 1 and then never becomes 0 for some u' and j. Therefore the operation $T[u', v] = T[u', v] \wedge T[j, v]$ for all $v \in V$ is executed at most once for every pair of vertices (u', j) during the entire computation implying that the total time is equal to $O(|P|^2 n^2 |P|n/\log(|P|n)) = O(|P|^3 n^3/\log(|P|n))$, using the multiplication of vectors.

The matrix ΔT contains only new elements, therefore T can be updated directly using only $|\Delta T|$ operations and hence $|P|^2 n^2$ operations in total. The same holds for the lines 12-18 of the Algorithm 1, because operations are performed only for non-zero elements of $|\Delta T|$. Finally, the time complexity of the Algorithm 1 is $O((|N| + |\Sigma|)^2 |P|^2 n^2) + O(|P|^2 n^2) + O(|P|^2 n^2 \log(|P|n)) + O(|P|^3 n^3/\log(|P|n)) + O(|P|^2 n^2) = O(|P|^3 n^3/\log(|P|n))$. ◀

The complexity analysis of the Algorithm 1 shows that the maintaining of the incremental transitive closure dominates the cost of the algorithm. Thus, CFL-reachability can be solved in truly subcubic $O(n^{3-\varepsilon})$ time if there exists an incremental dynamic algorithm for the transitive closure for a graph with n vertices with preprocessing time $O(n^{3-\varepsilon})$ and total update time $O(n^{3-\varepsilon})$. Unfortunately, such an algorithm is unlikely to exist: it was shown that there is no incremental dynamic transitive closure algorithm for a graph with n vertices and at most m edges with preprocessing time $poly(m)$, total update time $mn^{1-\varepsilon}$, and query time $m^{\delta-\varepsilon}$ for any $\delta \in (0, 1/2]$ per query that has an error probability of at most $1/3$ assuming the widely believed Online Boolean Matrix-Vector Multiplication (OMv) Conjecture [21]. OMv Conjecture states that for any constant $\varepsilon > 0$, there is no $O(n^{3-\varepsilon})$-time algorithm that solves OMv with an error probability of at most $1/3$.

4 Implementation

4.1 SuiteSparse:GraphBLAS

GraphBLAS [28] is an API specification that defines standard building blocks for graph algorithms in the language of linear algebra. SuiteSparse:GraphBLAS [13] is a full implementation of the GraphBLAS standard, which defines a set of sparse matrix operations on an extended algebra of semirings using an almost unlimited variety of operators and types. We use pygraphblas[6] [55]: a Python wrapper for SuiteSparse:GraphBLAS.

The building blocks of our implementation are Kronecker product and sparse matrix multiplication, which are built-in primitives of pygraphblas.

4.2 Input representation

GraphBLAS provides a wide range of built-in types and operators, and allows the user application to create new types and operators. In our work we use Boolean and integer matrix representation of the input.

Boolean matrices

Since RSMs and FSMs can be represented as a labeled graph, and, hence, adjacency matrix, one can represent such matrix as a set of Boolean matrices containing a single Boolean matrix for every label. For example, the adjacency matrix $\mathcal{M}_2$ of the graph from Figure 1 can be represented as follows.

$$M_2^a = \begin{pmatrix} . & . & . & . & . & . \\ . & . & . & . & 1 & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \end{pmatrix}, \ M_2^{\bar{a}} = \begin{pmatrix} . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & . & . & . & . \end{pmatrix}, \ M_2^d = \begin{pmatrix} . & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & 1 & . & . & . \\ . & . & . & . & . & . \\ . & . & . & 1 & . & . \\ . & . & . & . & 1 & . \end{pmatrix},$$

[6] GitHub repository of PyGraphBLAS, a Python wrapper for GraphBLAS API: https://github.com/Graphegon/pygraphblas. Access date: 21.11.2021.

$$M_2^{\bar{d}} = \begin{pmatrix} . & 1 & . & . & . & . \\ . & . & . & 1 & . & . \\ . & . & . & . & . & . \\ . & . & . & . & 1 & . \\ . & . & . & . & . & 1 \\ . & . & . & . & . & . \end{pmatrix}.$$

Using Boolean adjacency matrices representation, we can reformulate the Kronecker product of such matrices.

▶ **Definition 11.** *Given two sets of Boolean adjacency matrices $\mathcal{M}_1$ and $\mathcal{M}_2$, the Kronecker product of these matrices is a new matrix $\mathcal{M} = \mathcal{M}_1 \otimes \mathcal{M}_2$, where $\mathcal{M} = \{M_1^a \otimes M_2^a \mid a \in \Sigma\}$ and the element-wise operation is a conjunction over Boolean values ($\wedge$).*

Integer matrices

5 Evaluation

5.1 Memory alias

5.2 Points-to analysis for Java

6 Related work

6.1 CFL-reachability

The CFL-reachability problem was introduced by Yannakakis [63] to describe the Datalog chain query evaluation problem. Later, Reps et al. [24, 43, 46] proposed the CFL-reachability framework for interprocedural program analysis. Since then the CFL-reachability has been used to formulate a variety of static analyses, such as points-to and alias analysis [10, 15, 31, 32, 33, 50, 52, 66, 67, 68], data-dependence analysis [10], type inference analysis [38], type-base flow analysis [42] and program slicing [44].

A cubic $O(n^3)$ algorithm for the CFL-reachability which uses dynamic programming technique, was proposed by Melski and Reps [37]. This result was improved by a logarithmic factor by Chaudhuri [12], giving the worst-case runtime complexity $O(n^3/\log n)$. Unfortunately, no algorithm faster has been discovered, for general graphs with n vertices and general context-free grammars, so the CFL-reachability is known to have a "cubic bottleneck" [19]. Recent result by Chatterjee et al. [10] shows that the CFL-reachability in cubic time is optimal under combinatorial Boolean Matrix Multiplication (BMM) hypothesis. The cubic lower bound under the same hypothesis was also established for Andersen's Pointer Analysis directly [35]. The cubic runtime can be improved substantially in specific cases, by taking advantage of certain properties of the underlying graph (i.e. bidirected graphs) [10, 66] or grammar/context-free language (i.e. Dyck language of 1 parenthesis) [7, 35].

There are some algorithms in the context of database theory, where exists the equivalent problem called Context-Free Path Querying (CFPQ) [6, 18, 20, 36, 39, 47, 54, 56]. It is important to mention that some of these algorithms reduce CFPQ evaluation to linear algebra operations: Azimov et al. [6] reduce CFPQ to matrix multiplication and Orachev et al. [39] reduce CFPQ to Kronecker product. Additionally, recently Sato [48] proposed linear algebraic approach to Datalog evaluation. This approach is based on the transformation of Datalog program to a set of matrix equations, and can be used for Datalog chain queries evaluation which is equivalent to the CFL-reachability problem. Unfortunately, all three mentioned algorithms have worse than cubic $O(n^5)$ theoretical time complexity, whereas our algorithm has state-of-the-art theoretical time complexity, having all the advantages of linear algebra formulation at the same time.

6.2 Graph processing systems

State-of-the-art systems for large graph proccessing use different architectures including single-machine and shared-memory parallel ones [34, 41, 51, 57, 65], multi-core and multi-processor architectures [9, 17, 40], and distributed graph processing systems [14, 16, 26, 29, 30, 45, 49, 59, 61, 64]. However, it is hard to use these engines for the implementation of the interprocedural program analysis tool without ground-up redesign [58].

There are many works which formulate specific graph algorithms in terms of linear algebra, for example, such algorithms as for computing transitive closure and all-pairs shortest paths. Recently this direction was summarized in GraphBLAS API [28] which provides building blocks to develop a graph analysis algorithm in terms of linear algebra. There is a number of implementations of this API, such as SuiteSparse:GraphBLAS [13], CombBLAS [9], GraphBLAST [62], GraphMat [53], GraphPad [3].

We implemented our tool on top of SuiteSparse:GraphBLAS because it gives a very flexible and convenient way to construct graph algorithms by using primitive and highly-optimized building blocks based on the set of of sparse matrix operations.

6.3 CFL-reachability-based code analysis tools

Since CFL-reachability captures a certain sub-class of Datalog, Datalog can be employed as a domain specific language to express custom program analyses, reducing the complexity of developing program analyzers. Such Datalog-powered tools, which are able to run sophisticated static analysis include bddbddb [60], DOOP [8], LogicBlox [4], μZ [22], Soufflé [27]. However, such engines are known to be fundamentally limited by the size of main memory and, therefore, are not able to scale well on a large code systems [70], and experience reduced performance compared to manually implemented tools [27].

A single-machine, disk-based graph systems Grapple [69], Graspan [58] and Chianina [70] turn code analysis into bigdata analytics. The main goal of Graspan is to scale context-free CFL-reachability based analyses to large programs with disk support. A piece of work Chianina [70] supports easy development of any context- and flow-sensitive analysis for C. Unfortunately, massive expensive disk I/Os remain the major performance bottleneck of disk-based graph processing.

7 Conclusion and future work

References

1 Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Analysis of recursive state machines. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, pages 207–220, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

2 Lars Ole Andersen and Peter Lee. Program analysis and specialization for the c programming language. 2005.

3 Michael J. Anderson, Narayanan Sundaram, Nadathur Satish, Md. Mostofa Ali Patwary, Theodore L. Willke, and Pradeep Dubey. Graphpad: Optimized graph primitives for parallel

and distributed platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 313–322, 2016. doi:10.1109/IPDPS.2016.86.

4 Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1371–1382, New York, NY, USA, 2015. Association for Computing Machinery. URL: https://doi.org/10.1145/2723372.2742796, doi:10.1145/2723372.2742796.

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

6 Rustam Azimov and Semyon Grigorev. Context-free path querying by matrix multiplication. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '18, pages 5:1–5:10, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3210259.3210264, doi:10.1145/3210259.3210264.

7 P. G. Bradford. Efficient exact paths for dyck and semi-dyck labeled path reachability (extended abstract). In *2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)*, pages 247–253. IEEE, Oct 2017. doi:10.1109/UEMCON.2017.8249039.

8 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 243–262, New York, NY, USA, 2009. Association for Computing Machinery. URL: https://doi.org/10.1145/1640089.1640108, doi:10.1145/1640089.1640108.

9 Aydın Buluç and John R Gilbert. The combinatorial blas: Design, implementation, and applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, November 2011. URL: https://doi.org/10.1177/1094342011403516, doi:10.1177/1094342011403516.

10 Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. Optimal dyck reachability for data-dependence and alias analysis. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. URL: https://doi.org/10.1145/3158118, doi:10.1145/3158118.

11 Swarat Chaudhuri. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 159–169, New York, NY, USA, 2008. Association for Computing Machinery. URL: https://doi.org/10.1145/1328438.1328460, doi:10.1145/1328438.1328460.

12 Swarat Chaudhuri. Subcubic algorithms for recursive state machines. In *POPL '08*, 2008.

13 Timothy A. Davis. Algorithm 1000: Suitesparse:graphblas: Graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, 45(4), December 2019. URL: https://doi.org/10.1145/3322125, doi:10.1145/3322125.

14 Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17, page 293–304, New York, NY, USA, 2017. Association for Computing Machinery. URL: https://doi.org/10.1145/3087556.3087580, doi:10.1145/3087556.3087580.

15 Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. Giga-scale exhaustive points-to analysis for java in under a minute. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 535–551, New York, NY, USA, 2015. Association for Computing Machinery. URL: https://doi.org/10.1145/2814270.2814307, doi:10.1145/2814270.2814307.

16 Zhisong Fu, Michael Personick, and Bryan Thompson. Mapgraph: A high level api for fast development of high performance graph analytics on gpus. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, page 1–6, New York, NY, USA, 2014. Association for Computing Machinery. URL: `https://doi.org/10.1145/2621934.2621936`, `doi:10.1145/2621934.2621936`.

17 Douglas P. Gregor and Andrew Lumsdaine. The parallel bgl : A generic library for distributed graph computations. 2005.

18 Semyon Grigorev and Anastasiya Ragozina. Context-free path querying with structural representation of result. In *Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia*, CEE-SECR '17, pages 10:1–10:7, New York, NY, USA, 2017. ACM. URL: `http://doi.acm.org/10.1145/3166094.3166104`, `doi:10.1145/3166094.3166104`.

19 Nevin Heintze and David McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, LICS '97, page 342, USA, 1997. IEEE Computer Society.

20 Jelle Hellings. Querying for paths in graphs using context-free path queries. *arXiv preprint arXiv:1502.02242*, 2015.

21 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '15, page 21–30, New York, NY, USA, 2015. Association for Computing Machinery. URL: `https://doi.org/10.1145/2746539.2746609`, `doi:10.1145/2746539.2746609`.

22 Kryštof Hoder, Nikolaj Bjørner, and Leonardo de Moura. μz– an efficient engine for fixed points with constraints. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 457–462, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

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

24 Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '95, page 104–115, New York, NY, USA, 1995. Association for Computing Machinery. URL: `https://doi.org/10.1145/222124.222146`, `doi:10.1145/222124.222146`.

25 T. Ibaraki and N. Katoh. On-line computation of transitive closures of graphs. *Information Processing Letters*, 16(2):95 – 97, 1983. URL: `http://www.sciencedirect.com/science/article/pii/0020019083900339`, `doi:https://doi.org/10.1016/0020-0190(83)90033-9`.

26 Zhihao Jia, Yongkee Kwon, Galen M. Shipman, Patrick S. McCormick, Mattan Erez, and Alexander Aiken. A distributed multi-gpu system for fast graph processing. *Proc. VLDB Endow.*, 11:297–310, 2017.

27 Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.

28 J. Kepner, P. Aaltonen, D. Bader, A. Buluc, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira. Mathematical foundations of the graphblas. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, Sep. 2016. `doi:10.1109/HPEC.2016.7761646`.

29 Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. Cusha: vertex-centric graph processing on gpus. In *HPDC '14*, 2014.

30 Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, apr 2012. URL: `https://doi.org/10.14778/2212351.2212354`, `doi:10.14778/2212351.2212354`.

31 Jingbo Lu, Dongjie He, and Jingling Xue. Eagle: Cfl-reachability-based precision-preserving acceleration of object-sensitive pointer analysis with partial context sensitivity. *ACM Trans.*

Softw. Eng. Methodol., 30(4), jul 2021. URL: `https://doi.org/10.1145/3450492`, `doi:10.1145/3450492`.

32 Jingbo Lu and Jingling Xue. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019. URL: `https://doi.org/10.1145/3360574`, `doi:10.1145/3360574`.

33 Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. An incremental points-to analysis with cfl-reachability. In *Proceedings of the 22nd International Conference on Compiler Construction*, CC'13, page 61–81, 2013. URL: `https://doi.org/10.1007/978-3-642-37051-9_4`, `doi:10.1007/978-3-642-37051-9_4`.

34 Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 527–543, New York, NY, USA, 2017. Association for Computing Machinery. URL: `https://doi.org/10.1145/3064176.3064191`, `doi:10.1145/3064176.3064191`.

35 Anders Alnor Mathiasen and Andreas Pavlogiannis. The fine-grained and parallel complexity of andersen's pointer analysis. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021. URL: `https://doi.org/10.1145/3434315`, `doi:10.1145/3434315`.

36 Ciro M. Medeiros, Martin A. Musicante, and Umberto S. Costa. Efficient evaluation of context-free path queries for graph databases. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, pages 1230–1237, New York, NY, USA, 2018. ACM. URL: `http://doi.acm.org/10.1145/3167132.3167265`, `doi:10.1145/3167132.3167265`.

37 David Melski and Thomas Reps. Interconvertbility of set constraints and context-free language reachability. *SIGPLAN Not.*, 32(12):74–89, December 1997. URL: `https://doi.org/10.1145/258994.259006`, `doi:10.1145/258994.259006`.

38 Ana Milanova, Wei Huang, and Yao Dong. Cfl-reachability and context-sensitive integrity types. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, page 99–109, New York, NY, USA, 2014. Association for Computing Machinery. URL: `https://doi.org/10.1145/2647508.2647522`, `doi:10.1145/2647508.2647522`.

39 Egor Orachev, Ilya Epelbaum, Rustam Azimov, and Semyon Grigorev. Context-free path querying by kronecker product. In Jérôme Darmont, Boris Novikov, and Robert Wrembel, editors, *Advances in Databases and Information Systems*, pages 49–59, Cham, 2020. Springer International Publishing.

40 Roger Pearce, Maya Gokhale, and Nancy M. Amato. Scaling techniques for massive scale-free graphs in distributed (external) memory. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 825–836, 2013. `doi:10.1109/IPDPS.2013.72`.

41 Yonathan Perez, Rok Sosič, Arijit Banerjee, Rohan Puttagunta, Martin Raison, Pararth Shah, and Jure Leskovec. Ringo: Interactive graph analytics on big-memory machines. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1105–1110, New York, NY, USA, 2015. Association for Computing Machinery. URL: `https://doi.org/10.1145/2723372.2735369`, `doi:10.1145/2723372.2735369`.

42 Jakob Rehof and Manuel Fähndrich. 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 '01, page 54–66, New York, NY, USA, 2001. Association for Computing Machinery. URL: `https://doi.org/10.1145/360204.360208`, `doi:10.1145/360204.360208`.

43 Thomas Reps. Program analysis via graph reachability. In *Proceedings of the 1997 International Symposium on Logic Programming*, ILPS '97, page 5–19, Cambridge, MA, USA, 1997. MIT Press.

44 Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '94, page 11–20, New York, NY, USA, 1994. Association for Computing Machinery. URL: https://doi.org/10.1145/193173.195287, doi:10.1145/193173.195287.

45 Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 472–488, New York, NY, USA, 2013. Association for Computing Machinery. URL: https://doi.org/10.1145/2517349.2522740, doi:10.1145/2517349.2522740.

46 Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1):131–170, 1996. URL: https://www.sciencedirect.com/science/article/pii/0304397596000722, doi:https://doi.org/10.1016/0304-3975(96)00072-2.

47 Fred C. Santos, Umberto S. Costa, and Martin A. Musicante. A bottom-up algorithm for answering context-free path queries in graph databases. In Tommi Mikkonen, Ralf Klamma, and Juan Hernández, editors, *Web Engineering*, pages 225–233, Cham, 2018. Springer International Publishing.

48 TAISUKE SATO. A linear algebraic approach to datalog evaluation. *Theory and Practice of Logic Programming*, 17(3):244–265, 2017. doi:10.1017/S1471068417000023.

49 Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey S. Young, Matthew Wolf, and Karsten Schwan. Graphin: An online high performance incremental graph processing framework. In *Euro-Par*, 2016.

50 Lei Shang, Yi Lu, and Jingling Xue. Fast and precise points-to analysis with incremental cfl-reachability summarisation: Preliminary experience. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, page 270–273, New York, NY, USA, 2012. Association for Computing Machinery. URL: https://doi.org/10.1145/2351676.2351720, doi:10.1145/2351676.2351720.

51 Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, page 135–146, New York, NY, USA, 2013. Association for Computing Machinery. URL: https://doi.org/10.1145/2442516.2442530, doi:10.1145/2442516.2442530.

52 Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. *SIGPLAN Not.*, 40(10):59–76, oct 2005. URL: https://doi.org/10.1145/1103845.1094817, doi:10.1145/1103845.1094817.

53 Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, jul 2015. URL: https://doi.org/10.14778/2809974.2809983, doi:10.14778/2809974.2809983.

54 Arseniy Terekhov, Artyom Khoroshev, Rustam Azimov, and Semyon Grigorev. Context-free path querying with single-path semantics by matrix multiplication. In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA'20, New York, NY, USA, 2020. Association for Computing Machinery. URL: https://doi.org/10.1145/3398682.3399163, doi:10.1145/3398682.3399163.

55 Timothy A Davis Timothy Mattson, Michel Pelletier. Graphblas programmability: Python and matlab interfaces. HPEC '20, 2020.

56 Ekaterina Verbitskaia, Semyon Grigorev, and Dmitry Avdyukhin. Relaxed parsing of regular approximations of string-embedded languages. In Manuel Mazzara and Andrei Voronkov, editors, *Perspectives of System Informatics*, pages 291–302, Cham, 2016. Springer International Publishing.

57 Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

58 Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 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, page 389–404, New York, NY, USA, 2017. Association for Computing Machinery. URL: `https://doi.org/10.1145/3037697.3037744`, `doi:10.1145/3037697.3037744`.

59 Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the gpu. *SIGPLAN Not.*, 51(8), feb 2016. URL: `https://doi.org/10.1145/3016078.2851145`, `doi:10.1145/3016078.2851145`.

60 John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, page 97–118, Berlin, Heidelberg, 2005. Springer-Verlag. URL: `https://doi.org/10.1007/11575467_8`, `doi:10.1007/11575467_8`.

61 Da Yan, Yuzhen Huang, Miao Liu, Hongzhi Chen, James Cheng, Huanhuan Wu, and Chengcui Zhang. Graphd: Distributed vertex-centric graph processing beyond the memory limit. *IEEE Transactions on Parallel and Distributed Systems*, 29:99–114, 2018.

62 Carl Yang, Aydin Buluc, and John D. Owens. Graphblast: A high-performance linear algebra-based graph framework on the gpu, 2019. `arXiv:1908.01407`.

63 Mihalis Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, pages 230–242, New York, NY, USA, 1990. ACM. URL: `http://doi.acm.org/10.1145/298514.298576`, `doi:10.1145/298514.298576`.

64 Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, USA, 2010. USENIX Association.

65 Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, page 183–193, New York, NY, USA, 2015. Association for Computing Machinery. URL: `https://doi.org/10.1145/2688500.2688507`, `doi:10.1145/2688500.2688507`.

66 Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 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, page 435–446, New York, NY, USA, 2013. Association for Computing Machinery. URL: `https://doi.org/10.1145/2491956.2462159`, `doi:10.1145/2491956.2462159`.

67 Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. Efficient subcubic alias analysis for c. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages Applications*, OOPSLA '14, page 829–845, New York, NY, USA, 2014. Association for Computing Machinery. URL: `https://doi.org/10.1145/2660193.2660213`, `doi:10.1145/2660193.2660213`.

68 Xin Zheng and Radu Rugina. Demand-driven alias analysis for c. *SIGPLAN Not.*, 43(1):197–208, January 2008. URL: `http://doi.acm.org/10.1145/1328897.1328464`, `doi:10.1145/1328897.1328464`.

69 Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. Grapple: A graph system for static finite-state property checking of large-scale systems code. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery. URL: `https://doi.org/10.1145/3302424.3303972`, `doi:10.1145/3302424.3303972`.

70 Zhiqiang Zuo, Yiyu Zhang, Qiuhong Pan, Shenming Lu, Yue Li, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. Chianina: An evolving graph system for flow- and context-sensitive analyses of million lines of c code. In *Proceedings of the 42nd ACM SIGPLAN*

International Conference on Programming Language Design and Implementation, PLDI 2021, page 914–929, New York, NY, USA, 2021. Association for Computing Machinery. URL: `https://doi.org/10.1145/3453483.3454085`, `doi:10.1145/3453483.3454085`.