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, 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] 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

1 Introduction

Problem

Insight

Our contributions

2 Background

2.1 CFL-reachability

2.2 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,*
- *$\delta : Q \times \Sigma \to Q$ is a transition function.*

It is well known, that every regular expression can be converted to deterministic FSM without ε-transitions [4].

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.*

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

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.

2.3 Linear algebra

Graph Kronecker product and machines intersection

▶ **Definition 4.** *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 5.** *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 6.** *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 5. Thus, $M(\mathcal{G}) = M(\mathcal{G}_1) \otimes M(\mathcal{G}_2)$, where $\mathcal{G} = \mathcal{G}_1 \otimes \mathcal{G}_2$.

▶ **Definition 7.** *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 [4] an FSM intersection defines the machine for which $L(T) = L(T_1) \cap L(T_2)$.

2.4 Pointer analysis as CFL-reachability problem

Memory alias

Points-to analysis for Java

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 6, we can apply 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 egde from K. Then the new reachable pairs are validated in Lines 12-16: we are interested only in paths from start to final state for some box, therefore some pairs can be excluded from adding to $\mathcal{M}_2$. If $\mathcal{M}_2$ have changed after the insertion of the elements, we calculate the new elements of the Kronecker product and so on.

Algorithm 1 Linear algebra 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
```

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 $\mathcal{M}_1 \otimes \mathcal{A}_2$.

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 [5] 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 [2]. 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 8.** *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 Listing 1. Then for any valid indices i, j and for each nonterminal $A \in N$ the following statement holds: the cell $\mathcal{M}_2[i, j]$ contains $\{A\}$, iff there is a path $i\pi j$ in the graph $\mathcal{G}$ such that $A \xrightarrow{*} l(\pi)$.*

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

Complexity

▶ **Theorem 9.** *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 $\sum_i t(\mathcal{M}_1 \otimes \Delta\mathcal{M}_2^{(i)}) = |\mathcal{M}_1| \sum_i |\Delta\mathcal{M}_2^{(i)}| = (|N| + |\Sigma|)|P|^2 \sum_i |\Delta\mathcal{M}_2^{(i)}| = O((|N| + |\Sigma|)^2 |P|^2 n^2)$.

Now we derive the time complexity of maintaining the dynamic transitive closure. Notice that K has a size of the Kronecker product of $\mathcal{M}_1 \otimes \mathcal{M}_2$, which is equal to $dim(\mathcal{M}_1)n \times dim(\mathcal{M}_1)n = |P|n \times |P|n$ so no more than $|P|^2 n^2$ edges will be added during all iterations of the Algorithm. 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 12-16 of the algorithm from 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 [3]. 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

GraphBLAS

Representation the input as sets of Boolean adjacency matrices

Easy implementation of transitive closure

Custom semirings

5 Evaluation

5.1 Memory alias

5.2 Points-to analysis for Java

6 Related work

6.1 CFL-reachability

Algorithmic aspects

Graph systems for interprocedural static code analysis

6.2 Linear algebra-based graph analysis tools

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 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.

3 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`.

4 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.

5 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`.