Program Analysis for Adaptive Data Analysis

ANONYMOUS AUTHOR(S)

An adaptive data analysis is based on multiple queries over a data set, in which some queries rely on the results of some other queries. The error of each query is usually controllable and bound independently, but the error can propagate through the chain of different queries and bring to high generalization error. To address this issue, data analysts are adopting different mechanisms in their algorithms, such as Gaussian mechanism, etc. To utilize these mechanisms in the best way one needs to understand the depth of chain of queries that one can generate in a data analysis. Unfortunately, this depth which relies on the program(implementation) itself is costly in human efforts, and how to statically obtain this information is not well studied to support data analysts.

In this work, we define a programming framework which can provide, through analysis on some transformation of the target program implementing an adaptive data analysis, an upper bound on the adaptivity depth (the length of the longest chain of queries) along with another upper bound on the total number of queries requested. We show how this framework can help to analyze the generalization error of two data analyses with different adaptivity structures.

Additional Key Words and Phrases: data analysis, program analysis, dependency graph

1 INTRODUCTION

Consider a dataset X consisting of n independent samples from some unknown population P. How can we ensure that the conclusions drawn from X *generalize* to the population P? Despite decades of research in statistics and machine learning on methods for ensuring generalization, there is an increased recognition that many scientific findings generalize poorly (e.g.). While there are many reasons a conclusion might fail to generalize, one that is receiving increasing attention is *adaptivity*, which occurs when the choice of method for analyzing the dataset depends on previous interactions with the same dataset.

Adaptivity can arise from many common practices, such as exploratory data analysis, using the same data set for feature selection and regression, and the re-use of datasets across research projects. Unfortunately, adaptivity invalidates traditional methods for ensuring generalization and statistical validity, which assume that the method is selected independently of the data. The misinterpretation of adaptively selected results has even been blamed for a "statistical crisis" in empirical science.

A recent line of work initiated by Dwork *et al.* [Dwork et al. 2015] and Hardt and Ullman posed the question: Can we design *general-purpose* methods that ensure generalization in the presence of adaptivity, together with guarantees on their accuracy? This line of work has identified many new algorithmic techniques for ensuring generalization in adaptive data analysis, leading to algorithms with greater statistical power than all previous approaches. It has also identified problematic strategies for adaptive analysis, showing limitations on the statistical power one can hope to achieve.

A key development in this line of work is that the best method for ensuring generalization in an adaptive data analysis depends to a large extent on the number of *rounds of adaptivity*. That is, not only does the analysis of the generalization error depend on the number of rounds, but knowing the number of rounds actually allows one to choose methods that lead to the smallest possible generalization error.

It is obviously promising if this number of *rounds of adaptivity* can be estimated statically. Nevertheless, the corresponding study on static analysis over programs implementing adaptive

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

analysis is not well explored, even the formal definition of this *rounds of adaptivity* is not clear. Our goal is to study adaptivity and conduct analysis over programs to find the number of rounds. To this end, we design a programming framework, which statically provides an upper bound of the number of rounds of adaptivity for programs implementing adaptive data analysis algorithms. The first question we need to decide on is what language the target programs to be analyzed are written, in a functional programming language or an imperative one?

We choose the imperative programming language, then the following question is what does this imperative language look like? A principle of the ideal language in our mind is supposed to be simple enough to alleviate the burden of analysis on complex components that is unnecessary to adpativity, and expressive enough to support most of the adaptive data analysis algorithms. With this principle in mind, we introduce an imperative language called the loop language, with one finite loop construct, allowing data analysts recursively request queries in their programs. The query request is also supported in the language.

For the number of rounds of adaptivity of program in the loop language, a definition of "one query relies on another query", or "one query may depend on the other" is the next big thing. We choose to define this "may-dependency" with the help of a trace-based operational semantics. The trace is a list of queries generated along with the execution of the program, a history of the execution only caring about query requests.

Our framework includes transformations of the target programs in loop language, to "ssa-form" programs. The programs in single static assignment form, with all the variables assigned once (variables in the loop is also handled and will be discussed in later section), greatly helps our analysis avoid the complexity of handling variables overwriting. The direct analysis on program in ssa form then allows our framework to provide the upper bound on the number of rounds of adaptivity. To be more specific, our framework is equipped with algorithms to record the necessary information and construct a dependency graph to reach a final estimation of the adaptivity.

Our contribution lies in the static analysis on programs implementing adaptive analysis for a tight estimation of its number of rounds of adaptivity through a framework. Specifically, our contributions can be broken into the following:

(1) A formal definition of dependency between queries in adaptive data analysis programs.
(2) A formal definition of adaptivity through a query-based dependency graph, bases on a trace-based operational semantics for the loop language.
(3) A transformation between programs in loop language and its ssa form language, with the soundness of the transformation shows that every program in loop language can be appropriately transformed to its ssa form.
(4) An matrix-and-vector-based algorithm to track the dependency between unique variables of ssa form programs (including loop) and construct a variable-based dependency graph for our static estimation of rounds of adaptivity.
(5) A soundness proof shows that the predicted upper bound from our algorithm strictly bounds the actual number of rounds of adaptivity in execution.

2 OVERVIEW

In this section, we start with a review of the prior work of adaptive data analysis, which motivates our work, a framework to statically give an upper bound on the rounds of adaptivity . We then show the architecture of the framework and give our readers a taste from simple examples.

2.1 Review of Prior Work

To explain the key concepts and motivate our work, we review the model of adaptive data analysis of [??]. We remark that many of the details of the model are immaterial for our work, and are included for illustrative purposes and concreteness.

In the model there is some distribution P over a domain $\mathcal{X}$ that an analyst would like to study. For our purposes, the analyst wishes to answer *statistical queries* (also known as *linear queries* or *linear functionals*) on the distribution. A statistical query is defined by some function $f : \mathcal{X} \to [-1, 1]$. The analyst wants to learn the *population mean*, which (abusing notation) is defined as

$$f(P) = \mathop{\mathbb{E}}_{X \sim P}[f(X)].$$

However, the distribution P can only be accessed via a set of *samples* $X_1, \ldots, X_n$ drawn identically and independently from P. These samples are held by a mechanism $M(X_1, \ldots, X_n)$ who receives the query f and computes an answer $a \approx f(P)$.

In this work we consider analysts that ask a sequence of k queries $f_1, \ldots, f_k$. If the queries are all chosen in advance, independently of the answers then we say they are *non-adaptive*. If the choice of each query f_j may depend on the prefix $f_1, a_1, \ldots, f_{j-1}, a_{j-1}$ then they are *fully adaptive*. An important intermediate notion is *r-round adaptive*, where the sequence can be partitioned into r batches of non-adaptive queries. Note that non-interactive queries are 1-round and fully adaptive queries are k rounds.

We now review what is known about the problem of answering r-round adaptive queries. Given the samples, the naïve way to approximate the population mean is to use the *empirical mean*, which (abusing notation) is defined as

$$f(X_1, \ldots, X_n) = \frac{1}{n} \sum_{i=1}^{n} f(X_i).$$

Theorem 2.1. *For any distribution P, and any k* non-adaptive *statistical queries, the naïve mechanism satisfies*

$$\max_{j=1,\ldots,k} |a_j - f_j(P)| = O\left(\sqrt{\frac{\log k}{n}}\right)$$

For any $r \geq 2$ and any r-round adaptive *statistical queries, it satisfies*

$$\max_{j=1,\ldots,k} |a_j - f_j(P)| = O\left(\sqrt{\frac{k}{n}}\right)$$

And there exists analysts that make these bounds tight (up to constant factors).

Note that even allowing one extra round of adaptivity leads to an exponential increase in the generalization error from $\log k$ to k.

Perhaps surprisingly, Dwork *et al.* [?] and Bassily *et al.* [?] showed that an alternative mechanism can actually achieve much stronger generalization error as a function of the number of queries, specifically.

Theorem 2.2 ([??]). *For any k, there exists a mechanism such that for any distribution P, and any $r \geq 2$ any r-round* adaptive *statistical queries, it satisfies*

$$\max_{j=1,\ldots,k} |a_j - f_j(P)| = O\left(\frac{\sqrt[4]{k}}{\sqrt{n}}\right)$$

And there is an analyst that make this bound tight (up to constant factors).

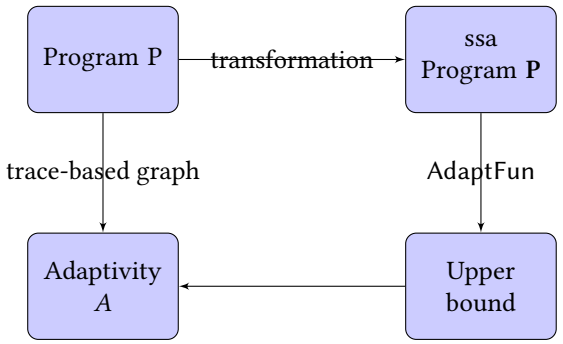

Fig. 1. High level architecture

Notice that Theorem 2.2 has different quantification in that the optimal choice of mechanism depends on the number of queries. Thus, we need to know the number of queries *a priori* to choose the best mechanism.[1]

Later work by Dwork gave more refined bounds in terms of the number of rounds of adaptivity. Similarly, if one knows a good *a priori* upper bound on the number of rounds of adaptivity, one can get a much better guarantee of generalization error, but only by using an appropriate choice of the mechanism. Specifically,

Theorem 2.3 ([**?**]). *For any r and k, there exists a mechanism such that for any distribution P, and any $r \geq 2$ any* r-round adaptive *statistical queries, it satisfies*

$$\max_{j=1,\ldots,k} |a_j - f_j(P)| = O\left(\frac{r\sqrt{\log k}}{\sqrt{n}}\right)$$

And there is an analyst that make this bound tight (up to constant factors).

2.2 Static analysis framework for adaptivity

We show a framework that gives a good *a priori* upper bound on the number of rounds of adpativity, to achieve the aforementioned refined bounds.

The architecture of the framework is shown in Figure 1. We are interested in the number of rounds of adpativity (we use adaptivity as a short in the rest of the paper) of a program P in the loop language. We use a simplified adaptivite data analysis algorithm- two round strategy TR to illustrate how the framework works to produces a upper bound on adpativity in Figure 2. The two round algorithm consists of two main steps. The first one is asking the database through query request $q()$ for k times and storing results in a list a. The second step is to construct another query $q_2(a)$ using the list a, and asks for the result from database by requesting the new query $q_2(a)$. In the loop language, the query request is assigned by $x \leftarrow q(e)$. In the above example, we simplify the query request, for example, query $q()$ in the first step does not have argument and the new constructed query is just $q(a)$ for the sake of brevity. In TR, we also add the line number. We do not go deep into it and more details about the syntax of the loop language are presented in Section 3.

As shown in the left hand side of Figure 1, our framework generates a query-based dependency graph to achieve the actual adaptivity A of program P. This query-based dependency graph requires two main components.

[1] One can, in principle, avoid knowing the number of queries and rounds *a priori* using a "guess-and-double" strategy, however this would weaken the bound on generalization error considerably.

$$TR(k) \triangleq \begin{array}{l} [a \leftarrow []]^1; \\ \texttt{loop}\ [k]^2\ \texttt{do} \\ \Big([x \leftarrow q_1()]^3; \\ [a \leftarrow x :: a]^4 \Big); \\ [l \leftarrow q_2(a)]^5 \end{array}$$

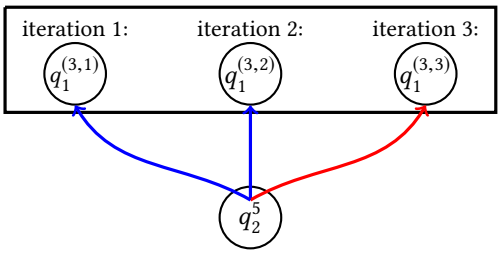

Fig. 2. A query-based dependency graph for simplified two round algorithm

(1) A trace-based operational semantics to keep the history of the execution in record.
(2) A formal definition of may-dependency between two query requests.

The trace-based operational semantics of the loop language have the shape of $\langle m, c, t, w\rangle \rightarrow \langle m', c', t', w'\rangle$. It works on a configuration of memory m, a program c, a trace t, a loop map w. The trace t is a list of annotated queries. One annotated query is a query with the annotation (l, w), l is the line number and w is used for loop and tells the iteration number. For example, the query $q()$ at line 3 in the example TR at the first iteration is represented as the annotated query $q()^{(3,[2:1])}$. The loop map w is a map, from the line number of the loop counter ($[k]^2$) to its iteration number. The trace tracks the query asked during the execution, still use the TR as an example. The trace of TR starting with an empty loop map $\emptyset$ has the following trace t_{tr}, supposing $k = 3$.

$$t_{tr} = [q()^{(3,[2:1])}, q()^{(3,[2:2])}, q()^{(3,[2:3])}, q(a)^{(5,\emptyset)}]$$

The second component is the defintion of may-dependency between queries. In particular, what we need to be clear is what does it mean by saying one query may depend on another query. To this end, we look at trace. Now whether one query q depends on another query p can be checked, by witnessing q in the new generated trace upon change of p. For example, if we change the result of $q()^{(3,[2:1])}$, we have a new trace t'_{tr}, and a may change as well. In this case, $q(a)^{(5,\emptyset)}$ may not show up in the new generated trace t'_{tr}. So we think $q(a)^{(5,\emptyset)}$ may depend on $q()^{(3,[2:1])}$.

With the two components ready, a query-based dependency graph is built, in which the nodes are the annotated queries in the trace and the edge is the may dependency. We show a simple query-based graph below. The adpativity is obtained by finding out the longest path in the graph.

Now, we look at the right hand side of Figure 1. We find the loop language is not straightforward for analysis due to the complexity of variables overwriting.

To solve this dilemma, we transform the program P to its single static assignment(short for ssa) form **P**. We call it ssa language. More details about the transformation as well as the ssa language is in Section **??**. We briefly show our two round example in its ssa form TR^{ssa} to illustrate the ssa language.

$$TR^{ssa}(k) \triangleq \begin{array}{l} [a_1 \leftarrow []]^1; \\ \texttt{loop}\ [k]^2\ \texttt{do} \\ \Big(a_3 = \phi(a_1, a_2); \\ [x_1 \leftarrow q()]^3; \\ [a_2 \leftarrow x :: a_3]^4 \Big); \\ [l_1 \leftarrow q(a_3)]^5 \end{array}$$

We notice there is one more command without label $a_3 = \phi(a_1, a_2)$ in TR^{ssa} compared to its loop variant. It adds a new variable a_3 whose value may come from a_1 or a_2. In the syntax of our ssa

Arithmetic Expr.	a	::=	$n \mid x \mid \chi \mid a \oplus_a a \mid [] \mid [a, \ldots, a]$
Boolean Expr.	b	::=	$\texttt{true} \mid \texttt{false} \mid \neg b \mid b \oplus_b b \mid a \sim a$
Expressions	e	::=	$a \mid b$
commands	c	::=	$\texttt{skip} \mid x \leftarrow e \mid x \leftarrow q(e) \mid \texttt{loop}\ a\ \texttt{do}\ c \mid c; c \mid \texttt{if}(b, c, c)$

Fig. 3. Syntax of loop language

language, we use $[a_3, a_1, a_2]$ to represent this extra command in the loop command, will be discussed in Section **??**.

Then we have our analysis algorithm AdaptFun to directly anlayze the ssa program **P** to get an upper bound. This algorithm construct a variable-based dependency graph and shows the may-dependency between annotated variables. We show the graph of our two round example.

Finally, we show that our upper bound from the variable-based dependency graph is sound with respect to our adaptivity.

3 LOOP LANGUAGE

In this section, we formally introduce the language we will focus on for writing data analyses. This is a simple loop language with some primitives for calling queries. [MG: I would just call the language Loop language]. After defining the syntax of the language and showing an example, we will define its trace-based operational semantics. This is the main technical ingredient we will use to define program's adaptivity. We will conclude this section by discussing the limitation of this language with respect to static analysis for adaptivity.

3.1 Syntax

We introduce the syntax of the Loop language we will use to write our data analyses in Figure 3. It is standard that expressions can be either arithmetic expressions or boolean expressions. An arithmetic expression can be a constant n denoting integer, a variable x from some countable set Var, the special variable χ representing a row of the database, a combination of arithmetic expressions by means of the symbol $\oplus_a$, denoting basic operations including addition, product, subtraction, etc., the empty list [], a list $[a_1, \ldots, a_k]$ of arithmetic expressions. A boolean expression can be true or false, the negation of a boolean expression, or a combination of boolean expressions by means of $\oplus_b$, denoting basic boolean connectives, or the result of some basic comparison *sym* between arithmetic expressions, e.g. $\leq, =, <$, etc. A command c can either be skip, an assignment command $x \leftarrow e$, the composition of two commands $c; c$, an if statement $\texttt{if}(b, c, c)$, a loop statement loop a do c.

The main novelty of the syntax is the query request command $x \leftarrow q(e)$. As a reminder, the aforementioned tight bound of adaptive data analysis in Section2 focuses on linear queries. So our framework targets the linear queries as well. The linear query is specified by a function from rows to $[0, 1]$ or $[-1, +1]$. To express these functions, we introduce the special variable χ to represent the rows of the database in the arithmetic expressions. In this sense, a simple linear query which returns the first element of the row is written as $q(\chi(1))$, as a notation for a query $q(\chi) = \chi(1)$.

[MG: I don't think that this corresponds exactly to our approach. I think that we want to focus on linear queries, these are the ones for which we have the bounds. A linear query is specified by a function from rows to [0,1] or [-1,+1]. So, in some sense, we want to have a language that describes these functions. Cannot we use $q(r) = e$ where r is a special variable denoting the given row, and e is an expression as we have right now? Example: $q_j(x) = x(i) \cdot x(j)$ can be written as $q(\chi(i) \cdot \chi(j))$ which is a notation for $q = \lambda\chi\chi(i) \cdot \chi(j)$.]

$\mathbf{TRC(k)}$	$\mathbf{TRC^{ssa}(k)}$
$[a \leftarrow []]^1$;	$[a_1 \leftarrow []]^1$;
$[j \leftarrow 0]^2$;	$[j_1 \leftarrow 0]^2$;
loop $[k]^3$ do	loop $[k]^3, 0,$ do$[(j_3, j_1, j_2), (a_3, a_1, a_2)]$
$\Big([x \leftarrow q(\chi(j) \cdot \chi(k))]^4$;	$\Big([x_1 \leftarrow q(\chi(j_3) \cdot \chi(k))]^4$;
$[j \leftarrow j+1]^5$;	$[j_2 \leftarrow j_3+1]^5$;
$[a \leftarrow x :: a]^6 \Big)$;	$[a_2 \leftarrow x_1 :: a_3]^6 \Big)$;
$\Big[l \leftarrow q(\text{sign}(\sum_{i\in[k]} \chi(i) \times \ln \frac{1+a[i]}{1-a[i]}))\Big]^7$	$\Big[l_1 \leftarrow q(\text{sign}(\sum_{i\in[k]} \chi(i) \times \ln \frac{1+a_3[i]}{1-a_3[i]}))\Big]^7$

Fig. 4. Two round algorithm complete version

We have seen a simplified version of the two round algorithm in Section 2. We show its complete version *TRC* expressed in our loop language on the left hand side in Figure 4.

As described before, the complete version still has two steps. The query asked in the first step now depends on the iteration number so that the query ask at the j iteration is $(q_j(x) = x(j) \cdot x(k))$, expressed as $q(\chi(j) \cdot \chi(k))$. The iteration counter is initialized to 0, $j = 0$. In the second step, the final query is more complicated. It uses an auxiliary function $\text{sign}(y) = \begin{cases} +1 & \text{if } y \geq 0 \\ -1 & \text{otherwise} \end{cases}$ The input of this function is $\sum_{i\in[k]} \chi(i) \times \ln \frac{1+a[i]}{1-a[i]}$, which sums the product of $\chi(i)$ and $\ln \frac{1+a[i]}{1-a[i]}$ that uses the value at index i of the list a.

In a word, we go through a two-round adaptive analysis algorithm and shows how we can represent it in the loop language. It reveals the expressiveness of this language for most of adaptive data analysis algorithms.

3.2 Trace-based Operational semantics

In order to capture the dependency relations between queries we will evaluate programs in our Loop language by means of a trace-based operational semantics. For distinguishing among different (occurrences of) queries, our trace-based semantics will generate traces that record also the line and loop numbers for different queries. For this reason we will consider a labelled version of the Loop language. This can be described as follows: [MG: Change "while map" to "Loop maps" everywhere.]

Label	l	:=	$\mathbb{N}$
Loop Maps	w	$\in$	Label $\times \mathbb{N}$
Labelled commands	c	::=	$[x \leftarrow e]^l \mid [x \leftarrow q\ (e)]^l \mid$ loop $[a]^l$ do $c \mid c; c \mid$ if$([b]^l, c, c) \mid [\text{skip}]^l$
Memory	m	::=	$[] \mid m[x \rightarrow v]$
Trace	t	::=	$[] \mid q(v)^{(l,w)} :: t$
Annotated Query	$\mathcal{AQ}$	::=	$\{q(v)^{(l,w)}\}$

The commands are now labelled — we assume that labels are unique and correspond to the line of code where they appear. We denote the label l, which is the natural number standing for the line number of command in a program. Here, we put the label l to the conditional b in the if statement and to the loop counter a in the loop command, which is helpful when we present the Loop Maps w.

A memory is standard, a map from variables to values. An important component of the labelled loop language is w, what we call "Loop Maps", designed for loop specifically as an complementary

annotation to the label. With the label and Loop Maps defined, we can show annotated queries $\mathcal{AQ}$ have unique annotations.

A trace t is supposed to accumulate along with the execution of the program, it is a list of annotated queries $\mathcal{AQ}$, in which the query $q(v)$ has the unique annotation (l, w). We can understand the annotation in such a way. The label l locates arbitrary query request when we look at the execution path of a program with no loop. However, when loop repeats queries in its body, these query requests from varied iterations share the same line number. Hence, the label l is not enough to distinguish queries in loops. For this purpose, another symbol is needed to represent the iteration number of a loop. A simplified approach is to use natural number n for the iteration number so that a pair (l, n) can help, but it fails to support nested loop. This accounts for the appearance of loop maps w into the annotation (l, w). As a map from label l to the iteration number n (a natural number), w gives accurate information on which loop specified by its label and which iteration n the query belongs to. We define some operations on w below.

$$\begin{array}{lll} w \setminus l & = w & l \notin Keys(w) \\ & = w_l & Otherwise \\ w + l & = w[l \rightarrow 1] & l \notin Keys(w) \\ & = w[l \rightarrow w(l) + 1] & Otherwise \end{array}$$

We denote $w \setminus l$ to remove the mapping of the key l in the loop maps w, used when exiting the loop at the line l. We register a label to w in the first iteration of a loop marked by label l and assign l to iteration 1, and increase its iteration number by one mapped by the label l when going into another iteration of the loop at line l. We denote $Keys(w)$ to return all the keys of the loop maps w.

A trace can be regarded as the history of query requests during the execution, generated by the trace-based small-step operational semantics. The evaluation of labelled commands are presented in Figure 5, of the form $\langle m, c, t, w\rangle \rightarrow \langle m', \texttt{skip}, t', w'\rangle$, stating a configuration $\langle m, c, t, w\rangle$ evaluates to another configuration with the trace and while map updated and the command c evaluated to `skip`. The configuration contains a memory m, a starting trace t, a starting while map w. Most of the time, the while map remains empty until the evaluation goes into the loop. We also have the evaluation of arithmatic expressions $\langle m, a\rangle \rightarrow_a a'$, evaluating an arithmatic expression a in the memory m, and similar for the boolean expressions $\langle m, b\rangle \rightarrow_b b'$.

The rule **low-query-e** evaluates the argument of a query request. After the query arguments in the normal form, the rule **low-query-v** modifies the starting memory m to $m[v_q/x]$ using the return results v_q of the query $q(v)$ request from a hidden database protected by a mechanism. We also notice that the trace expands by appending this query request $q(v)$ with the current annotation (l, w). The rule for assignment is standard and the trace remains unchanged. The sequence rule keeps tracking the modification of the trace, and the evaluation rules for if conditional goes into one branch based on the result of the conditional b. The rules for loop are interesting. In rule **low-loop**, the while map w is updated by $w + l$ because the execution goes into another iteration when $v_N > 0$. When v_N reaches 0, the loop exits and the while map w eliminates the label l of this loop command by $w \setminus l$ in the rule **low-loop-exit**.

3.3 Query-based may-dependency graph

we give our attempt to define the number of rounds of adaptivity in our loop language, through a query-based dependency graph. A query-based dependency graph of a program consists of nodes, which are the queries during the execution of the program, and directed edges showing "may dependency" between two nodes(queries).

To this end, the definition of one query may depend on the other is fundamental to construct the query-based dependency graph. We first look at two possible candidates:

$\boxed{\langle m, c, t, w\rangle \rightarrow \langle m', c', t', w'\rangle}$

$$\frac{\langle m, e\rangle \rightarrow e'}{\langle m, [x \leftarrow q(e)]^l, t, w\rangle \rightarrow \langle m, [x \leftarrow q(e')]^l, t, w\rangle}\ \textbf{low-query-e}$$

$$\frac{q(v) = v_q}{\langle m, [x \leftarrow q(v)]^l, t, w\rangle \rightarrow \langle m[v_q/x], \texttt{skip}, t +\!\!+ [q(v)^{(l,w)}], w\rangle}\ \textbf{low-query-v}$$

$$\frac{}{\langle m, [x \leftarrow v]^l, t, w\rangle \rightarrow \langle m[v/x], [\texttt{skip}]^l, t, w\rangle}\ \textbf{low-assn}$$

$$\frac{\langle m, c_1, t, w\rangle \rightarrow \langle m', c_1', t', w'\rangle}{\langle m, c_1; c_2, t, w\rangle \rightarrow \langle m', c_1'; c_2, t', w'\rangle}\ \textbf{low-seq1} \qquad \frac{}{\langle m, [\texttt{skip}]^l; c_2, t, w\rangle \rightarrow \langle m, c_2, t, w\rangle}\ \textbf{low-seq2}$$

$$\frac{\langle m, b\rangle \rightarrow_b b'}{\langle m, [\texttt{if}(b, c_1, c_2)]^l, t, w\rangle \rightarrow \langle m, [\texttt{if}(b', c_1, c_2)]^l, t, w\rangle}\ \textbf{low-if}$$

$$\frac{}{\langle m, [\texttt{if}(\texttt{true}, c_1, c_2)]^l, t, w\rangle \rightarrow \langle m, c_1, t, w\rangle}\ \textbf{low-if-t}$$

$$\frac{}{\langle m, [\texttt{if}(\texttt{false}, c_1, c_2)]^l, t, w\rangle \rightarrow \langle m, c_2, t, w\rangle}\ \textbf{low-if-f}$$

$$\frac{v_N > 0}{\langle m, [\texttt{loop}\ v_N\ \texttt{do}\ c]^l, t, w\rangle \rightarrow \langle m, c; [\texttt{loop}\ (v_N - 1)\ \texttt{do}\ c]^l, t, (w + l)\rangle}\ \textbf{low-loop}$$

$$\frac{v_N = 0}{\langle m, [\texttt{loop}\ v_N\ \texttt{do}\ c]^l, t, w\rangle \rightarrow \langle m, [\texttt{skip}]^l, t, (w \setminus l)\rangle}\ \textbf{low-loop-exit}$$

Fig. 5. Trace-based Operational semantics

(1) One query depends on the other if and only if the change of the return results of one query may also change the results of the other query.
(2) One query depends on the other if and only if the change of the return results of one query may also change the appearance of the other query.

The first one looks reasonable to witnessing the result of one query according to the change of return result of another query. We can easily find that the two queries have nothing to do with each other in a simple example $p = x \leftarrow q(\chi(1)); y \leftarrow q(\chi(2))$. However, it does not consider that one query may not even being executed according the change of another query, as shown in program p_1, which means we may fail to witness the change of result.

$$p_1 = x \leftarrow q(\chi(1)); \texttt{if}(x > 2, y \leftarrow q(\chi(2)), \texttt{skip})$$

We choose the second definition, witnessing the appearance of one query $q(\chi(2))$ upon the change of results of another query $q(\chi(1))$. One benefit of focusing on the appearance of queries forces us to study the effect of one query on the control flow, and then the execution. Still use $p = x \leftarrow q(\chi(1))$ $; y \leftarrow q(\chi(2))$ as an example, we can still easily conclude that $q(\chi(1))$ has nothing to do with $q(\chi(2))$

because the appearance of $q(\chi(2))$ is always there during the execution(or in the trace). Let us look at p_1 again, in which control flow plays its role. We think the two queries may have dependency because the change of the result of the previous query $q(\chi(1))$ affects the control flow, hence the appearance of $q(\chi(2))$ in the corresponding trace.

Another benefit of the appearance definition is it also takes the arguments of the queries into account. Let us look at another variant of program p, marked as p_2, in which the queries equipped with functions using previous assigned variables.

$$p_2 = x \leftarrow q(\chi(2)); y \leftarrow q(x + \chi(3))$$

As a reminder, in the loop language, the query request is composed by two components: a symbol q representing a linear query type and the argument e, which represents the function specifying what the query asks. So we do think $q(\chi(1))$ is different from $q(\chi(2))$. Informally, we think $q(x + \chi(3))$ may depend on the query $q(\chi_{(2)})$, because equipped function of the former $x + \chi(3)$ is affected by the latter $q(\chi(2))$. The appearance definition catches this case, since $q(x + \chi(2))$ will be different queries if x is changed along with the change of the $q(\chi(2)$.

We give a formal definition of query may dependency based on the trace-based operational semantics as follows.

Definition 1 (Query may dependency). *One query $q(v_1)$ may depend on another query $q(v_2)$ in a program c, with a starting while map w, denoted as $\mathsf{DEP}(q(v_1)^{(l_1,w_1)}, q(v_2)^{(l_2,w_2)}, c, w, m, D)$ is defined as below.*

$$\begin{aligned}
&\forall t.\exists m_1, m_3, t_1, t_3.\langle m, c, t, w\rangle \rightarrow^* \langle m_1, [x \leftarrow q(v_1)]^{l_1}; c_2, t_1, w_1\rangle \rightarrow \\
&\langle m_1[q(v_1)(D)/x], c_2, t_1 + +[q(v_1)^{(l_1,w_1)}], w_1\rangle \rightarrow^* \langle m_3, \mathtt{skip}, t_3, w_3\rangle \\
&\wedge\Big(q(v_1)^{(l_1,w_1)} \in (t_3 - t) \wedge q(v_2)^{(l_2,w_2)} \in (t_3 - t) \implies \exists v \in \mathtt{codom}(q(v_1)), m_3', t_3', w_3'. \\
&\langle m_1[v/x], c_2, t_1 + +[q(v_1)^{(l_1,w_1)}], w_1\rangle \rightarrow^* \langle m_3', \mathtt{skip}, t_3', w_3'\rangle \wedge (q(v_2)^{(l_2,w_2)}) \notin (t_3' - t)\Big) \\
&\wedge\Big(q(v_1)^{(l_1,w_1)} \in (t_3 - t) \wedge q(v_2)^{(l_2,w_2)} \notin (t_3 - t) \implies \exists v \in \mathtt{codom}(q(v_1)), m_3', t_3', w_3'. \\
&\langle m_1[v/x], c_2, t_1 + +[q(v_1)^{(l_1,w_1)}], w_1\rangle \rightarrow^* \langle m_3', \mathtt{skip}, t_3', w_3'\rangle \wedge (q(v_2)^{(l_2,w_2)}) \in (t_3' - t)\Big)
\end{aligned}$$

We give a formal definition of the query-based dependency graph with the formal definition of may-dependency between two queries.

Definition 2 (Query-based Dependency Graph). *Given a program c, a database D, a starting memory m, an initial while map w, the query-based dependency graph $G(c, D, m, w) = (V, E)$ is defined as:*

$V = \{q(v)^{l,w} \in \mathcal{AQ} \mid \forall t.\exists m', w', t'.\langle m, c, t, w\rangle \rightarrow^* \langle m', \mathtt{skip}, t', w'\rangle \wedge q(v)^{l,w} \in (t' - t)\}$.

$E = \left\{(q(v)^{(l,w)}, q(v')^{(l',w')}) \in \mathcal{AQ} \times \mathcal{AQ} \;\middle|\; \mathsf{DEP}(q(v)^{(l,w)}, q(v')^{(l',w')}, c, w, m, D) \wedge \mathsf{To}(q(v')^{(l',w')}, q(v)^{(l,w)}\right\}$.

The function $\mathsf{To}(q(v')^{(l',w')}, q(v)^{(l,w)}$ tells that the query request $q(v')^{(l',w')}$ appears after the query request $q(v)^{(l,w)}$ in the trace, by comparing the annotation (l', w') and (l, w). It helps to decide on the direction of one edge.

The query-based dependency graph only considers the newly generated annotated queries during the execution of the program c, so we see the nodes comes from the trace $t' - t$. The previous trace before the execution of c is excluded when constructing the graph. To summary, for every execution of a program c staring with different starting configuration, we can construct a corresponding dependency graph.

Finally, we reach the definition of what we want – the number of rounds of adaptivity , by means of a query-based dependency graph.

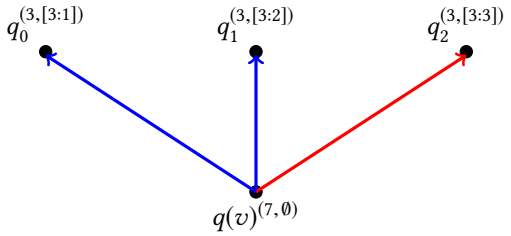

Fig. 6. A query-based dependency graph for two round algorithm complete version

Definition 3 (Adaptivity). *Given a program c, and a meory m, a database D, a starting while map w, the adaptivity of the dependency graph $G(c, D, m, w) = (V, E)$ is the length of the longest path in this graph. We denote the path from $q(v)^{(l,w)}$ to $q(v')^{(l',w')}$ as $p(q(v)^{(l,w)}, q(v')^{(l',w')})$. The adaptivity denoted as $A^*(c, D, m, w)$.*

$$A(c, D, m, w) = \max_{q(v)^{(l,w)}, q(v')^{(l',w')} \in V} \{|p(q(v)^{(l,w)}, q(v')^{(l',w')})|\}$$

We give an example to illustrate the aforementioned contents to collect the trace, build the dependency graph and get the adaptivity, via our familiar adaptive data analysis algorithm - two round strategy.

Given a specific database $D = [[1, 1], [0, 0], [1, 1], [1, 1]]$, the execution trace is generated along with the operational semantics as follows:

$$t = \left\{ q(\chi(0) \cdot \chi(3))^{(4,[3:1])},\ q(\chi(1) \cdot \chi(3))^{(4,[3:2])},\ q(\chi(2) \cdot \chi(3))^{(4,[3:3])},\ q(v)^{(7,\emptyset)} \right\}$$

For the sake of brevity, we use $q(v)$ to represent $q(\text{sign}(\sum_{i\in[k]} \chi(i) \times \ln \frac{1+a[i]}{1-a[i]}))$ where $a = [1.0.1]$. We use q_0, q_1, q_2 to represent $q(\chi(0)\cdot\chi(3))$, $q(\chi(1)\cdot\chi(3))$, $q(\chi(2)\cdot\chi(3))$. Then we have the dependency graph generated in Figure 6.

We can notice the complete version generates the similar query-based dependency graph as its simplified version in Figure 2.

4 THE ANALYSIS ALGORITHM ON SSA PROGRAMS

In this section, we clarify the algorithm AdaptFun that analyzes the adpativity of a target program in ssa language, which consists of three auxiliary algorithms: AG and AD for generating a weighted variable-based dependency graph and AP to find the most weighted path in the graph. We do not show details of AP, it is quite standard path-finding algorithm in a weighted graph. We go through the two round example to illustrate the algorithm. Finally, we show the soundness of our algorithm with respect to the adaptivity.

4.1 The ideas behind the algorithm

In consideration of the definition of adaptivity from the query-based dependency graph, our analysis whose goal is a sound upper bound on the adaptivity, is supposed to take care of paths(possible adaptivity candidates) in all the possible dependency graphs (per configuration). To this end, our algorithm aims to syntactically construct a dependency graph, in which the nodes are annotated ssa variables and the directed edges showing one annotated variable may depend on the other if there exists an edge between them. Intuitively, query requests in the query-based dependency graph is assigned to variables that appears in the predicted ssa-variable-based dependency graph.

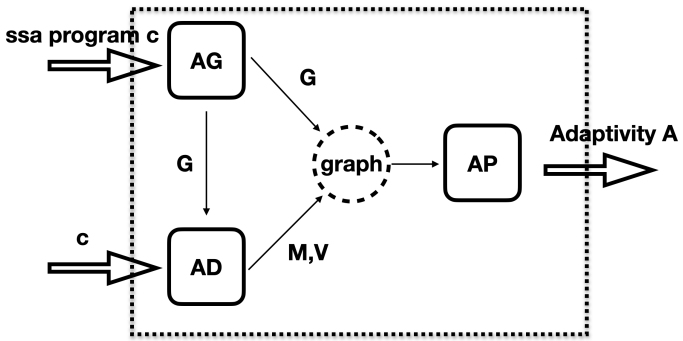

Fig. 7. The overview of AdaptFun

The algorithm AdaptFun estimates the adaptivity from the weighted variable-based dependency graph, whose structure is shown in Figure 7. We have a look at the structure of AdaptFun. We start with the dependency graph, represented in the form of a matrix M. To know which variable is associated with a query request in the matrix, an extra vector V is produced in the algorithm, of the size of all estimated assigned variables in the target program $\mathbf{c}$. Naturally, the first question comes to us, what is the size of the matrix M and V, or the estimated assigned variables? To solve this, the analysis goes through the ssa program $\mathbf{c}$ twice, estimating the assigned variables to be tracked and adding the appropriate annotation (l, w), in the first time scan. Those annotated variables are collected in a global list G, whose size determines the size of the matrix and vector. This first scan generating G is conducted by the algorithm AG in Figure 7, which takes the target program $\mathbf{c}$ as input. The global list G is fed into the second scan, which decides the size of the matrix and vector. For example, if G of size N, then the matrix used in the second round is of size $N \times N$, and vector of size N. It maintains an unique mapping from variables in G to the matrix M and the vector V. For example, the ith row, jth column of the matrix $M[i][j] = 1$ represents the may-dependency from variable $G[i]$ to $G[j]$, $M[i][j] = 0$ means no dependency. In a similar way, $V[i] = 1$ means the variable $G[i]$ is assigned to a query request. The second scan of $\mathbf{c}$, implemented by the algorithm AD, then records the may-dependency in M, the weight is tracked in V. We think variable associated with a query request of weight 1 in the graph and others of weight 0. After the analysis of AG and AD, a ssa-variable-based weighted dependency graph is constructed. We use another auxiliary algorithm AP to find the path with the most weights in the graph. Then the weight is the adaptivity A AdaptFun estimates.

4.2 Variable Estimation algorithm

We first show how G will be collected, through a variable estimating algorithm AG of the form $\mathrm{AG}(G; w; \mathbf{c}) \rightarrow (G'; w')$ in Figure 8. The input of AG is a list of estimated annotated variables G collected before the program $\mathbf{c}$, a while map w consistent with previous estimation, a program $\mathbf{c}$. The output of the algorithm is the updated global list G', along with the updated while map w for later estimation. The assignment is the source of variables AG estimates, in the case **ag-asgn** and **ag-query**, the output global list is extended by $\mathbf{x}^{(l,w)}$. When it comes to the if command in the rule **ag-if**, variables assigned in the then branch $\mathbf{c}_1$, as well as the variables assigned in the else branch $\mathbf{c}_2$, and the new generated variables $\bar{\mathbf{x}}, \bar{\mathbf{y}}, \bar{\mathbf{z}}$ in $[\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]$, $[\bar{\mathbf{y}}, \bar{\mathbf{y}_1}, \bar{\mathbf{y}_2}]$, $[\bar{\mathbf{z}}, \bar{\mathbf{z}_1}, \bar{\mathbf{z}_2}]$. The sequence is standard by accumulating the predicted variables in the two commands $\mathbf{c}_1$ and $\mathbf{c}_2$ in a

$$\frac{}{\mathsf{AG}(G; w; [\mathbf{x} \leftarrow \mathbf{e}]^{\mathbf{l}}) \rightarrow (G + +[\mathbf{x}^{(l,w)}]; w)}\ \textbf{ag-asgn}$$

$$\frac{}{\mathsf{AG}(G; w; [\mathbf{x} \leftarrow q(\mathbf{e})]^{l}) \rightarrow (G + +[\mathbf{x}^{(l,w)}]; w)}\ \textbf{ag-query}$$

$$\frac{\begin{array}{c}\mathsf{AG}(G; w; \mathbf{c_1}) \rightarrow (G_1; w_1) \qquad \mathsf{AG}(G_1; w; \mathbf{c_2}) \rightarrow (G_2; w_2) \\ G_3 = G_2 + +[\bar{\mathbf{x}}^{(\mathbf{l},\mathbf{w})}] + +[\bar{\mathbf{y}}^{(\mathbf{l},\mathbf{w})}] + +[\bar{\mathbf{z}}^{(\mathbf{l},\mathbf{w})}]\end{array}}{\mathsf{AG}(G; w; [\mathtt{if}(\mathbf{b}, [\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}], [\bar{\mathbf{y}}, \bar{\mathbf{y}_1}, \bar{\mathbf{y}_2}], [\bar{\mathbf{z}}, \bar{\mathbf{z}_1}, \bar{\mathbf{z}_2}], \mathbf{c_1}, \mathbf{c_2})]^{l}) \rightarrow (G_3; w)}\ \textbf{ag-if}$$

$$\frac{\mathsf{AG}(G; w; \mathbf{c_1}) \rightarrow (G_1; w_1) \qquad \mathsf{AG}(G_1; w_1; \mathbf{c_2}) \rightarrow (G_2; w_2)}{\mathsf{AG}(G; w; (\mathbf{c_1}; \mathbf{c_2})) \rightarrow (G_2; w_2)}\ \textbf{ag-seq}$$

$$\frac{\begin{array}{c}G_0 = G \quad w_0 = w \qquad \forall 0 \leq z < N.\mathsf{AG}(G_z + +[\bar{\mathbf{x}}^{(\mathbf{l},\mathbf{w_z+1})}]; (w_z + l); \mathbf{c}) \rightarrow (G_{z+1}; w_{z+1}) \\ G_f = G_N + +[\bar{\mathbf{x}}^{(\mathbf{l},\mathbf{w_N \setminus l})}] \qquad \mathbf{a} = N\end{array}}{\mathsf{AG}(G; w; [\mathtt{loop}\ \mathbf{a}, n, [\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]\ \mathtt{do}\ \mathbf{c}]^{l}) \rightarrow (G_f; w_N \setminus l)}\ \textbf{ag-loop}$$

Fig. 8. The algorithm AG

sequence $\mathbf{c_1}; \mathbf{c_2}$. The loop considers the loop iterations as well by assuming the loop counter $\mathbf{a}$ to be certain natural number N in the rule **ag-loop**. The algorithm counts the assigned variables in every iteration, including those new assigned variables in $\bar{\mathbf{x}}$, those variables assigned in the body $\mathbf{c}$, with the appropriate annotation showing the iteration number of the variables.

4.3 Matrix and Vector based algorithm

We have a global list G after the first scan of the ssa programs $\mathbf{c}$. We develop a matrix and vector based approach based on the global list G to get an estimated upper bound on the adaptivity of the program. In this approach, a matrix is constructed according to those estimated annotated variables from the global list, in which every row and every column corresponds to the unique variable in G by position. To be precise, in this matrix, 0 means no dependency while non-zero value in the matrix shows may-dependency between corresponding variables. If the value of $M[i][j]$ in the matrix M is greater than zero, we know annotated variable $G[i]$ may depend on $G[j]$. A vector V has the same size as G and it records whether the corresponding variable $G[i]$ is assigned with a query request. $V[i] = 1$ means assigned with a query request and otherwise $V[i] = 0$. We call this algorithm which tracks may-dependency in the ssa programs and query information, AD. The algorithm of the form $\mathsf{AD}(\Gamma; \mathbf{c}; i) \rightarrow (M; V; i')$, the input is a tuple consisting of three elements: (1) a one-row-N-column matrix Γ storing the dependency from previous program, it is used when handling the if command. (2) the ssa program $\mathbf{c}$ to be analyzed (3) an index i specifying the location of the first assigned variable of the program $\mathbf{c}$ in the global list G. The output of AD also consists of three elements ; (1) A matrix M showing the may-dependency in $\mathbf{c}$ (2) A vector V records the query requests in $\mathbf{c}$ (3) the index i' that refers to the next position of the last assigned variable in $\mathbf{c}$, if exists. The existence of the index i' helps to locate the first assigned variable if we need to analyze programs after $\mathbf{c}$.

We first define some functions which use the indices in G. The function $\mathsf{L(i)}$ generates one-column-N-rows matrix, where only the $i - th$ row is 1 and all the other rows are 0. This function is used to locate the right row when calculate the matrix when analyzing assignment and query.

$$\frac{M = \mathsf{L}(i) * (\mathsf{R}(\mathbf{e}, i) + \Gamma)}{\mathsf{AD}(\Gamma; [\mathbf{x} \leftarrow \mathbf{e}]^l; i) \rightarrow (M; V_\emptyset; i+1)}\ \textbf{ad-asgn} \qquad \frac{M = \mathsf{L}(i) * (\mathsf{R}(\mathbf{e}, i) + \Gamma) \qquad V = \mathsf{L}(i)}{\mathsf{AD}(\Gamma; [\mathbf{x} \leftarrow q(\mathbf{e})]^l; i) \rightarrow (M; V; i+1)}\ \textbf{ad-query}$$

$$\frac{\begin{array}{c}\mathsf{AD}(\Gamma + \mathsf{R}(\mathbf{b}, i_1); \mathbf{c}_1; i_1) \rightarrow (M_1; V_1; i_2) \qquad \mathsf{AD}(\Gamma + \mathsf{R}(\mathbf{b}, i_1); \mathbf{c}_2; i_2) \rightarrow (M_2; V_2; i_3) \\ \mathsf{AD}(\Gamma; [\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]; i_3) \rightarrow (M_x; V_\emptyset; i_3 + |\bar{\mathbf{x}}|) \qquad \mathsf{AD}(\Gamma; [\bar{\mathbf{y}}, \bar{\mathbf{y}_1}, \bar{\mathbf{y}_2}]; i_3 + |\bar{\mathbf{x}}|) \rightarrow (M_y; V_\emptyset; i_3 + |\bar{\mathbf{x}}| + |\bar{\mathbf{y}}|) \\ \mathsf{AD}(\Gamma; [\bar{\mathbf{z}}, \bar{\mathbf{z}_1}, \bar{\mathbf{z}_2}]; i_3 + |\bar{\mathbf{x}}| + |\bar{\mathbf{y}}|) \rightarrow (M_y; V_\emptyset; i_3 + |\bar{\mathbf{x}}| + |\bar{\mathbf{y}}| + |\bar{\mathbf{z}}|) \\ M = (M_1 + M_2) + M_x + M_y + M_z\end{array}}{\Gamma \vdash^{(i_1, i_3 + |\bar{x}| + |\bar{y}| + |\bar{z}|)}_{M, V_1 \uplus V_2} [\mathtt{if}(\mathbf{b}, [\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}], [\bar{\mathbf{y}}, \bar{\mathbf{y}_1}, \bar{\mathbf{y}_2}], [\bar{\mathbf{z}}, \bar{\mathbf{z}_1}, \bar{\mathbf{z}_2}], \mathbf{c}_1, \mathbf{c}_2)]^l}\ \textbf{ad-if}$$

$$\frac{\mathsf{AD}(\Gamma; \mathbf{c}_1; i_1) \rightarrow (M_1; V_1; i_2) \qquad \mathsf{AD}(\Gamma; \mathbf{c}_2; i_2) \rightarrow (M_2; V_2; i_3)}{\mathsf{AD}(\Gamma; (\mathbf{c}_1; \mathbf{c}_2); i_1) \rightarrow ((M_1; M_2); V_1 \uplus V_2; i_3)}\ \textbf{ad-seq}$$

$$\frac{\begin{array}{c}B = |\bar{\mathbf{x}}| \qquad A = |\mathbf{c}| \\ \forall 0 \le j < N. \mathsf{AD}(\Gamma; [\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]; i + j * (B + A)) \rightarrow (M_{1j}; V_{1j}; i + B + j * (B + A)) \\ \mathsf{AD}(\Gamma; \mathbf{c}; i + B + j * (B + A)) \rightarrow (M_{2j}; V_{2j}; i + B + A + j * (B + A)) \\ \mathsf{AD}(\Gamma; [\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]; i + N * (B + A)) \rightarrow (M; V; i + N * (B + A) + B) \\ \mathbf{a} = N \qquad M' = M + \sum_{0 \le j < N} (M_{1j} + M_{2j}) \qquad V' = V \uplus \sum_{0 \le j < N} (V_{1j} \uplus V_{2j})\end{array}}{\Gamma \vdash^{(i, i + N*(B+A)+B)}_{M', V'} [\mathtt{loop}\ \mathbf{a}, 0, [\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]\ \mathtt{do}\ \mathbf{c}]^l}\ \textbf{ad-loop}$$

Fig. 9. The algorithm AD

The function R(e, i) generates a one-row-N-column matrix. For every variable used in e, it finds the corresponding index i in G so that $G[i]$ maps to the variable and mark the ith column as 1. If it is not found, we do not mark. When we say $G[i]$ maps to a target variable, we take off the annotation of $G[i]$ and check if the left variable is the same as the target variable. To handle loop, for instance, a variable y appears many times in G, but with different annotations(iteration numbers), the argument i helps to find most recent assignment variable y before the index i in G. It is still used when analyzing assignment and query. Thanks to our ssa language, our choice of the most recent assigned variable is reasonable because the variable used in the loop refers to the most recent assignment and every variable is uniquely assigned in its ssa form.

We define $M_1; M_2$ to combine two matrix, where $M_1 + M_2$ is the standard sum of two matrix.

$$M_1; M_2 := M_2 \cdot M_1 + M_1 + M_2$$

And define the operator $\uplus$ to combine two vectors.

$$V_1 \uplus V_2 := \begin{cases} 1 & (V_1[i] = 1 \vee V_2[i] = 1) \wedge i = 1, \cdots, N \wedge |V_1| = |V_2| \\ 0 & o.w. \end{cases}$$

For the sake of brevity, we also define some annotations as follows. We show how the algorithm AD handles the extra part $[\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]$ in the if and loop commands. First, we give a unique name for variables in lists $\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}$ respectively, as follows: $\forall 0 \le z < |\bar{\mathbf{x}}|. \bar{\mathbf{x}}(z) = \mathbf{x_z}, \bar{\mathbf{x}_1}(z) = \mathbf{x_{1z}}, \bar{\mathbf{x}_2}(z) = \mathbf{x_{2z}}$. And then we treat every tuple $(\mathbf{x_z}, \mathbf{x_{1z}}, \mathbf{x_{2z}})$ in $[\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]$ as the simple may dependency case : $\mathbf{x_z}$ may depend on both $\mathbf{x_{1z}}$ and $\mathbf{x_{2z}}$, just like $\mathbf{x_z} \leftarrow \mathbf{x_{1z}} + \mathbf{x_{2z}}$, defined as follows. $\mathsf{AD}(\Gamma; [\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]; i) \rightarrow (M; V_\emptyset; i + |\bar{\mathbf{x}}|) \triangleq \forall 0 \le z < |\bar{\mathbf{x}}|. \mathsf{AD}(\Gamma; \mathbf{x_z} \leftarrow \mathbf{x_{1z}} + \mathbf{x_{2z}}; i + z) \rightarrow (M_{x_z}; V_\emptyset; i + z + 1)$ where $M = \sum_{o \le z < |\bar{\mathbf{x}}|} M_{x_z}$. One of the key idea under algorithm AD is to track the indices i,i' both in the input and output

to synchronize with its previous algorithm AG. The index in AD increases as the same way as the global list expands after the analysis of a program $\mathbf{c}$, which helps AD record the dependency relation from the program $\mathbf{c}$ in the right place of the matrix. For example, in the case **ad-asgn** and **ad-query**, the index increases by 1, which corresponds to their counterparts of algorithm **AG**. The if and loop commands have the extra part $[\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]$ and we find that the output index increases by also considering this part as we do in collecting the global list.

Another interesting point is the construction of the matrix. The fundamental case is the assignment and query cases. We use a function $L(i)$ to generate a N-row-one-column matrix L to guarantee the resulting matrix only has non-zeros at row i. The intuition behind is that one single assignment or query request can only reveal the dependency of its assigned variable (corresponding to one row of the matrix) to the variables used on the right hand sides. Thanks to the index i, we know which row this assignment should be in the matrix. The function $\mathsf{R}(\mathbf{e}, i)$ gets a one-row-N-column matrix marking the variables used in the right hand side. The Γ is designed for the if command and we will discuss it later. We can see one simple example sa to get a taste.

$$sa \triangleq \begin{array}{l} [x_1 \leftarrow 2]^1 ; \\ [x_2 \leftarrow x_1 + 2]^2 ; \\ [x_3 \leftarrow x_1 + x_2]^3 \end{array}$$

In the program sa, only simple assignment is involved. When we assume Γ is empty, for the assignment at line 3, the matrix is built as follows.

$$\textbf{line3: } [x_3 \leftarrow x_1 + x_2]^3 : \quad \begin{matrix} x_1 \\ x_2 \\ x_3 \end{matrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} * \begin{matrix} x_1 & x_2 & x_3 \end{matrix} \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} = \begin{matrix} \\ x_1 \\ x_2 \\ x_3 \end{matrix} \begin{matrix} x_1 & x_2 & x_3 \\ \end{matrix} \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

We use a one-row-N-column matrix Γ as one of the input of AD, to handle the cases when the control flow diverges in the labelled if command $[\mathtt{if}(\mathbf{b}, [\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}], [\bar{\mathbf{y}}, \bar{\mathbf{y}_1}, \bar{\mathbf{y}_2}], [\bar{\mathbf{z}}, \bar{\mathbf{z}_1}, \bar{\mathbf{z}_2}], \mathbf{c_1}, \mathbf{c_2})]^l$, where the execution of either branch may depend on the conditional guard $\mathbf{b}$. Follow this intuition, the analysis of either branch is supposed to consider the variables used in the conditional $\mathbf{b}$, tracked in Γ. In the case **ad-if**, we can see the analysis of the two branches $\mathbf{c_1}$ and $\mathbf{c_2}$ share the same input $\Gamma + \mathsf{R}(\mathbf{b}, i)$ and $\mathsf{R}(\mathbf{b}, i)$ tells the variables assigned before the if command and used in the conditional $\mathbf{b}$.

We compose the matrix and vectors in the case of sequence in **ad-seq**. The non-zeros or we call it effect range of the matrix and vector is decided by its input and output indices. From the case **ad-seq**, two programs $\mathbf{c_1}$ and $\mathbf{c_2}$ have disjoint effect ranges $[i_1, i_2)$ and $[i_1, i_3)$, it is safe to combine them without lose information.

The loop case is handled in a well organized way. We use $B = |\bar{\mathbf{x}}|$ and $A = |\mathbf{c}|$ to estimate the size of variables assigned in $\bar{\mathbf{x}}$ and $\mathbf{c}$. And $|\mathbf{c}|$ is defined by the help of the algorithm AG, defined as $|\mathbf{c}| = |G|$ when $\mathsf{AG}([]; \mathbf{c}; \emptyset) \rightarrow (G; \emptyset)$. The algorithm then gets how many iterations N the loop may executes from the loop counter $\mathbf{a}$. For every iteration, it first records the dependency relations between variables in $[\bar{\mathbf{x}}, \bar{\mathbf{x}_1}, \bar{\mathbf{x}_2}]$ by constructing a corresponding matrix M_{1j} (j is the iteration number) and an empty vector V_{1j}, and analyze the loop body $\mathbf{c}$ with a resulting matrix M_{2j} and vector V_{2j}. We give an extra analysis of those new assigned variables as what AG does, it works well when the loop is executed ($N = 0$) or not. We know that for all the possible iteration number j, M_{1j} and M_{2j} have disjoint effect ranges so we combine them, similar as the vectors V_{1j} and V_{2j}.

Finally, we are able to construct a variable-based weighted dependency graph based on G,M and V generated by the framework. The definition of the estimated adaptivity is the weight of the most weighted path in the graph defined as follows.

Definition 4 (Estimated Adaptivity). *Given a program* $\mathbf{c}$*, the global list* G*, and* $\mathsf{AD}(\Gamma; \mathbf{c}; i_1) \rightarrow (M, V, i_2)$*, the weighted dependency graph* $G_{ssa}(M, V, G, i_1, i_2) = (Nodes, Edges, Weights)$ *is defined as:*
Nodes $Vt = \{G(j) \in \mathcal{LV} \mid i_1 \leq j < i_2\}$
Edges $E = \{(G(j_1), G(j_2)) \in \mathcal{LV} \times \mathcal{LV} \mid M[j_1][j_2] \geq 1 \wedge i_1 \leq j_1, j_2 < i_2\}$
Weights $Wt = \{(G(j), 1) \in \mathcal{LV} \times \mathcal{N} | i_1 \leq j < i_2 \wedge V[j] = 1\} \cup \{(G(j), 0) \in \mathcal{LV} \times \mathcal{N} | i_1 \leq j < i_2 \wedge V[j] = 0\}$.

Adaptivity of the program defined according to the graph is as:

$$Adapt(M, V, i_1, i_2) := \max_{vt_1, vt_2 \in Vt} \{\mathsf{Weight}(p(vt_1, vt_2), Wt)\},$$

where $p(k, l)$ *is the path in graph* $G_{ssa}(M, V, i_1, i_2)$ *starting from* k *to* l*,* $\mathsf{Weight}(p(vt_1, vt_2), Wt)$ *get the total sum of weights along the path* $p(vt_1, vt_2)$.

4.4 Analysis on two round algorithm

We show how AdaptFun analyze the two round algorithm. For the sake of brevity, we conduct the analysis on the simplified two round algorithm TR^{ssa}.

AdaptFun first runs the algorithm AG to generate the global list G. We assume the input $k = 2$ and have the following.

$$G_{k=2} = \left[a_1^{(1,\emptyset)}, a_3^{(2,[2:1])}, x_1^{(3,[2:1])}, a_2^{(4,[2:1])}, a_3^{(2,[2:2])}, x_1^{(3,[2:2])}, a_2^{(4,[2:2])}, a_3^{(2,\emptyset)}, l_1^{(5,\emptyset)}\right]$$

We denote a_1^1 short for $a_1^{(1,\emptyset)}$ and $a_3^{(2,1)}$ short for $a_3^{(2,[2:1])}$. Then the resulting matrix M_{tr} and V_{tr} of the algorithm AD as follows.

$$M_{tr} = \begin{array}{c} \\ a_1^1 \\ a_3^{(2,1)} \\ x_1^{(3,1)} \\ a_2^{(4,1)} \\ a_3^{(2,2)} \\ x_1^{(3,2)} \\ a_2^{(4,2)} \\ a_3^2 \\ l_1^5 \end{array} \begin{array}{c} \begin{array}{ccccccccc} a_1^1 & a_3^{(2,1)} & x_1^{(3,1)} & a_2^{(4,1)} & a_3^{(2,2)} & x_1^{(3,2)} & a_2^{(4,2)} & a_3^2 & l_1^5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}, V_{tr} = \begin{bmatrix} a_1^1 & 0 \\ a_3^{(2,1)} & 0 \\ x_1^{(3,1)} & 1 \\ a_2^{(4,1)} & 0 \\ a_3^{(2,2)} & 0 \\ x_1^{(3,2)} & 1 \\ a_2^{(4,2)} & 0 \\ a_3^2 & 0 \\ l_1^5 & 1 \end{bmatrix}$$

4.5 Soundness of AdaptFun

We would like to show that the query-based dependency graph generated from the trace of the execution of the target ssa program is a subgraph of the variable-based dependency graph predicted from our algorithm AdaptFun, and the query requested during the execution is also bounded by an estimation from our algorithm.

We first give a definition of subgraph of a query-based dependency graph with respect to a variable-based dependency graph.

Definition 5 (Subgraph). *Given a query-based dependency graph* $G_s = (V_1, E_1)$*, a variable-based dependency graph* $G_{ssa} = (V_2, E_2)$*,* $G_s \subseteq G_{ssa}$ *iff:*
$\exists f$ *so that*
1. for every $v \in V_1$*,* $f(v) \in V_2$.
2. $\forall e = (v_i, v_j) \in E_1$*, there exists a path from* $f(v_i)$ *to* $f(v_2)$ *in* G_{ssa}.

Then we show the soundness of AdaptFun. In the theorem, we use some definition. $G \vDash M, V$ says that G and M, V have the corresponding size. $G; w \vDash (\mathbf{c}, i_1, i_2)$ checks if the variables assigned in $\mathbf{c}$ calculated by AG matches the variables in G from index i_1 to i_2.

Theorem 4.1 (Soundness of AdaptFun). *Given* $\mathsf{AD}(\Gamma; \mathbf{c}; i_1) \to (M; V; i_2)$*, for any global list* G*, loop maps* w *such that* $G; w \vDash (\mathbf{c}, i_1, i_2) \wedge G \vDash (M, V)$*.* K *is the number of queries inquired during the execution of the piece of program* $\mathbf{c}$ *and* $|V|$ *gives the number of non-zeros in* V*. Then,*

$$K \leq |V| \wedge \forall D, \mathbf{m}. G_s(\mathbf{c}, D, \mathbf{m}, w) \subseteq G_{ssa}(M, V, G, i_1, i_2)$$

5 MORE EXAMPLES

The two round strategy works well in our framework, we explore further to look at an advanced adaptive data analysis algorithm - multiple round algorithm.

Algorithm 1 A multi-round analyst strategy for random data (Algorithm 5 in ...)

Example 5.1 (Multiple Round Algorithm). **Require:** Mechanism $\mathcal{M}$ with a hidden state $X \in [N]^n$ sampled u.a.r., control set size c
Define control dataset $C = \{0, 1, \cdots, c-1\}$
Initialize $Nscore(i) = 0$ for $i \in [N]$, $I = \emptyset$ and $Cscore(C(i)) = 0$ for $i \in [c]$
for $j \in [k]$ **do**
  let $p = \mathtt{uniform}(0, 1)$
  define $q(x) = \mathtt{bernoulli}(p)$.
  define $qc(x) = \mathtt{bernoulli}(p)$.
  let $a = \mathcal{M}(q)$
  for $i \in [N]$ **do**
    $Nscore(i) = Nscore(i) + (a - p) * (q(i) - p)$ if $i \notin I$
  for $i \in [c]$ **do**
    $Cscore(C(i)) = Cscore(C(i)) + (a - p) * (qc(i) - p)$
  let $I = \{i | i \in [N] \wedge Nscore(i) > \max(Cscore)\}$
  let $X = X \setminus I$
return X.

$$MR(k) \triangleq \begin{array}{l} [I \leftarrow []]^1; \\ [ns \leftarrow 0]^2; \\ [cs \leftarrow 0]^3; \\ \mathtt{loop}\ [k]^4\ \ \mathtt{do} \\ \Big([p \leftarrow c]^5; \\ [a \leftarrow q(f(p, I))]^6; \\ [ns \leftarrow update\_nscore(p, a);]^7 \\ [cs \leftarrow update\_cscore(p, a);]^8 \\ [I \leftarrow \mathtt{update}(I, ns, cs)]^9\Big) \end{array} \Rightarrow MR^{ssa}(k) \triangleq \begin{array}{l} [I_1 \leftarrow []]^1; \\ \mathtt{loop}\ [k]^2, 0, [I_3, I_1, I_2] \\ \ \mathtt{do} \\ \Big([p_1 \leftarrow c]^3; \\ [a_1 \leftarrow q(p, I_3)]^4; \\ [I_2 \leftarrow \mathtt{update}\ (I_3, (a_1, p_1))]^5\Big) \end{array}$$

Description: The multiple round algorithm starts from an initialized empty tracking list I, a score called Nscore $ns = 0$, another score Cscore $cs = 0$. There is a hidden database X as well.It goes k rounds and every round, the two scores ns and cs are updated by a query result. Then the list I is updated by the two scores for every round. After the r rounds, the algorithm returns the columns of the hidden database X not specified in the tracking list I, which is $X \setminus I$.

The algorithm is written in the loop language as MR. It uses a loop for the k rounds computation and. We use functions $update\_nscore(p, a)$,$update\_cscore(p, a)$,$update(I, ns, cs)$ to simplify the complex update computation of Nscore, Cscore and the tracking list I. It will not change our analysis

Fig. 10. A query-based dependency graph for multi round algorithm

because these functions provides enough information through their arguments. In comparison with the two round algorithm, the query asked in each iteration is not independent in the multiple round one any more. The query in one iteration j now depends on the tracking list I from its previous iteration $j-1$, which is updated by the query result at the same iteration $j-1$. We can easily see the connection between queries from different iterations.

In MR, the tracking list I is initialized to an empty list. It appears inside the function of query $q(f(p,I))$ and updated in each iteration. We choose $k=3$, and assume the tracking list becomes I_a, I_b, I_c after 3 rounds, respectively. the execution trace is generated as:

$$t_{mr} = \left\{q(p,I_a)^{4,[2:1]}, q(p,I_b)^{4,[2:2]}, q(p,I_c)^{4,[2:3]}\right\}$$

Then we have the dependency graph generated as following, with the graph shown in Figure 10.
$V = \left\{q(p,I_a)^{(6,[4:1])}, q(p,I_b)^{(6,[4:2])}, q(p,I_c)^{(6,[4:3])}\right\}$
$E = \left\{(q(p,I_b)^{(6,[4:2])}, q(p,I_a)^{(6,[4:1])}, (q(p,I_c)^{(6,[4:3])}, q(p,I_b)^{(6,[4:2])}), (q(p,I_c)^{(6,[4:3])}, q(p,I_a)^{(6,[4:1])}), \right\}$
Then we have the adaptivity calculated from the graph as follows:

$$\begin{aligned} A^*(c,D,m,w) &= \max_{q(v)^{(l,w)}, q(v')^{(l',w')} \in V} \{|p(q(v)^{(l,w)}, q(v')^{(l',w')})|\} \\ &= \{|(q(p,I_c)^{(6,[4:3])} \rightarrow q(p,I_b)^{(6,[4:2])} \rightarrow q(p,I_a)^{(6,[4:1])})|\} \\ &= 3 \end{aligned}$$

Using AdaptFun, we first generate a global list G from an empty list [] and empty whlemap $\emptyset$.

$$[];\emptyset;MR^{ssa} \rightarrow G; w \wedge w = \emptyset$$

.

$$G_{k=2} = \left[I_1{}^{(1,\emptyset)}, I_3{}^{(2,[2:1])}, p_1{}^{(3,[2:1])}, a_1{}^{(4,[2:1])}, I_2{}^{(5,[2:1])}, I_3{}^{(2,[2:2])}, p_1{}^{(3,[2:2])}, a_1{}^{(4,[2:2])}, I_2{}^{(5,[2:2])}, I_3{}^{(2,\emptyset)}\right]$$

We denote I_1^1 short for $I_1{}^{(1,\emptyset)}$ and $I_3{}^{(2,1)}$ short for $I_3{}^{(2,[2:1])}$, where the label $(2,1)$ represents at line number 2 and in the 1 st iteration.

$$M = \begin{bmatrix} I_1^1 & I_3^{(2,1)} & p_1^{(3,1)} & a_1^{(4,1)} & I_2^{(5,1)} & I_3^{(2,2)} & p_1^{(3,2)} & a_1^{(4,2)} & I_2^{(5,2)} & I_3^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}, V = \begin{bmatrix} I_1^1 & 0 \\ I_3^{(2,1)} & 0 \\ p_1^{(3,1)} & 0 \\ a_1^{(4,1)} & 1 \\ I_2^{(5,1)} & 0 \\ I_3^{(2,2)} & 0 \\ p_1^{(3,2)} & 0 \\ a_1^{(4,2)} & 1 \\ I_2^{(5,2)} & 0 \\ I_3^2 & 0 \end{bmatrix}$$

The adaptivity is 1 computed from the graph. The query-based dependency graph is a subgraph of the variable dependency graph for multi round algorithm.

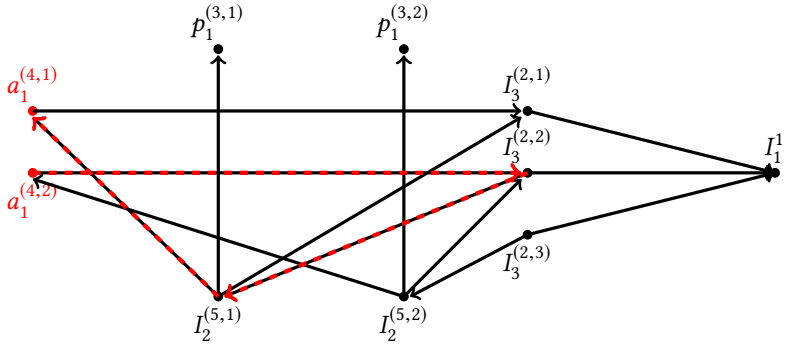

Fig. 11. the variable dependency graph for multi round algorithm

6 RELATED WORKS

REFERENCES

Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Leon Roth. 2015. Preserving statistical validity in adaptive data analysis. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*. 117–126.

A APPENDIX

Text of appendix …