Finding Regressions in Projects under Version Control Systems

Jaroslav Bendík, Nikola Beneš[1] and Ivana Černá[1]
[1]*Faculty of Informatics, Masaryk University, Brno, Czech Republic*
{xbendik,xbenes3,cerna}@fi.muni.cz

Keywords: Version Control Systems, Regressions, Regression Points, Code Debugging, Bisection

Abstract: Version Control Systems (VCS) are frequently used to support development of large-scale software projects. A typical VCS repository can contain various intertwined branches consisting of a large number of commits. If some kind of unwanted behaviour (e.g. a bug in the code) is found in the project, it is desirable to find the commit that introduced it. Such commit is called a regression point. There are two main issues regarding the regression points. First, detecting whether the project after a certain commit is correct can be very expensive and it is thus desirable to minimise the number of such queries. Second, there can be several regression points preceding the actual commit and in order to fix the actual commit it is usually desirable to find the latest regression point. Contemporary VCS contain methods for regression identification, see e.g. the *git bisect* tool. In this paper, we present a new regression identification algorithm that outperforms the current tools by decreasing the number of validity queries. At the same time, our algorithm tends to find the latest regression points which is a feature that is missing in the state-of-the-art algorithms. The paper provides an experimental evaluation on a real data set.

1 INTRODUCTION

Version Control Systems (VCS) have become ubiquitous in the area of (not only) software development, from small toy projects to large-scale industrial ones. The recent years saw a rise in the popularity of *Distributed VCS* such as *git* (Git, 2018), *bazaar* (Bazaar, 2018), *Mercurial* (Mercurial, 2018) and many others. These allow for almost seamless cooperation of a large number of developers and support extensive project branching and merging of branches. After a project has been in the development process for some time, the commit graph of its repository may grow to be very large.

As projects grow larger, the appearance of bugs (i.e. unwanted behaviour of the developed product) is going to be inevitable. Software bugs can be usually caught early if the development teams employ extensive testing techniques (unit tests, performance regression tests, etc.); however, from time to time a bug, or a commit that changed properties of the project, may creep into the VCS repository and lie there undetected for some time. Such bug is usually discovered by e.g. extending the coverage of the tests or by employing some other verification technique such as model checking (Clarke et al., 2001). In order to fix the bug it is very useful to identify the commit that introduced the bug as this commit typically contains a relatively small set of source code changes. It is much easier to properly understand and fix a bug when you only need to check a very small set of changes of the source code. Sometimes we are not looking for the commit that introduced a bug, but rather for a commit that caused a change between some "old" and "new" state of the project. As an example, we might be looking for the commit that introduced a particular fix. In such cases it can seem confusing to use the terms "correct" and "buggy" to refer to the state before and after the change, respectively. We thus instead use the terms *valid* and *invalid* commit; we further use the term *regression point* to denote the point where the property of interest has been changed.

The problem of finding regression points has been addressed before and there have been developed tools for solving this problem, such as *git bisect* (Git bisect documentation, 2018). These tools have proved themselves to be very useful and are commonly used during software development nowadays. Yet, there are several issues related to finding regression points and only some of them are targeted by the state-of-the-art tools.

First, the search for regression points consists of several queries of the form: "Given a certain commit, is the bug present in the system after this commit?" Such queries, which we call *validity queries*, may consist of several expensive tasks like running tests, model checking, code inspection, or other forms of verification. It is thus desirable to minimise the number of these queries. Second, the validity of commits does not, in general, have to be monotone. Perhaps a bug was introduced in a certain commit, inadvertently fixed several commits later, and then reintroduced in a yet later commit. This means that there are possibly several regression points preceding the actual invalid commit, but only the identification of the latest regression point can usually help us to fix the bug. The third issue concerns large projects with many branches. If, for example, a new test case is employed, then more than just one active branch can fail the test case and it is desirable to identify a regression point for each of these branches. We can find the regression point for each branch separately. However, dealing with all branches simultaneously can save some validity queries and thus optimise the search.

The state-of-the-art tools target the need to minimise the number of validity queries that are performed. However, they do not tend towards identification of latest regression points, and they deal with only a single invalid branch at a time.

The goal of this paper is to provide a novel algorithm, called the Regression Predecessors Algorithm (RPA), that solves the problem of finding regression points in VCS repositories. The algorithm minimises the number of validity queries and at the same time tries to find the latest regression points both for single and multiple invalid branches. RPA has several variants which we compare on a set of real open-source projects. Moreover, we compare the RPA algorithm with the state-of-the-art tool git bisect and demonstrate that our algorithm outperforms git bisect in all of the three above-mentioned criteria: in the number of performed validity queries, in finding the latest regression points, and in finding regression points for multiple invalid branches.

1.1 Example Use Case

Let us describe a common situation from the area of web development in which RPA can be used. Assume a small company that develops a content management system (CMS) for eCommerce. Such a CMS typically consists of several features, e.g. product management, discount system, or customer management. Many of these features are furthermore divided into sub-features; for example the discount feature can be divided into *buy one, get one free* discounts, *coupon based free shipping* discounts, *order-specific* discounts, and *seasonal discount*. At the beginning, only basic features are available and the remaining features are gradually developed. The company uses a version control system, e.g. git, and uses branching to separate development of individual features.

Some parts of the CMS are based on JavaScript, e.g. an interactive shopping cart page. The usage of JavaScript provides several advantages, but it also brings some disadvantages. One of the major disadvantages is that different web browsers may interpret JavaScript code differently which may result in unacceptable inconsistencies in terms of functionality and interface. At the end of the day, the company has to ensure that all recent versions of all the major web browsers provide the same functionality.

The problem is that JavaScript envolves, i.e. new versions are released, and it usually takes months till all the major web browsers support a particular version of JavaScript. Some features of JavaScript even never get supported by some browsers. Assume that a developer used a JavaScript *directive* in a moment when some browsers support the directive and the others ignore it, i.e. it helps in some browsers and it does no damage in the others. Several months later, a browser from the latter group starts to support the directive, however it interprets it in a different way than the other browsers, which causes a bug. It is very likely that the developer used the directive repeatedly in different pieces of code and that these pieces were subsequently adopted by some of the features that are currently being developed, i.e. the bug is presented in several active branches. It is also easy to imagine that this particular bug was introduced in a certain commit, fixed several commits later during code refactoring, and then maybe reintroduced in a yet another commit.

The rest of the paper is organised as follows. Section 2 defines basic notions and states the problem formally. Section 3 presents the Regression Predecessors Algorithm and illustrates its behaviour on a small example. Section 3.5 reviews the related work and compares RPA with other known algorithms. Section 4 gives an experimental evaluation of different variants of RPA and compares RPA with the state-of-the-art tool git bisect on a set of real benchmarks.

2 PRELIMINARIES AND PROBLEM FORMULATION

Definition 1. *A* rooted directed acyclic graph *is a directed graph $G = (V,E)$ with exactly one root (i.e. a vertex with no incoming edges) and with no cycle (i.e. there is no path $\langle v_0, v_1, \cdots v_k \rangle$ in the graph such that $v_0 = v_k$ and $k > 0$). A* rooted annotated directed acyclic graph (RADAG) *is a pair $(G, valid)$, where G is a rooted directed acyclic graph with root r*

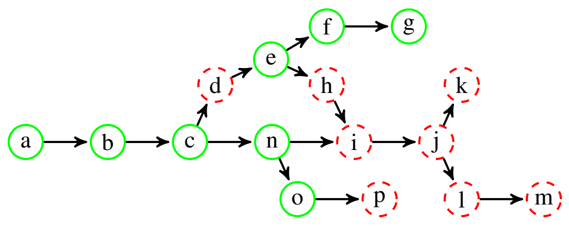

Figure 1: An example of a RADAG, the dashed vertices are invalid. There are three invalid leaves in this example: k, m, p, and four regression points: $(c,d),(e,h),(n,i)$ and (o,p). The first three regression points are regression predecessors of invalid leaves k,m; the fourth regression point is the regression predecessor of the invalid leaf p.

and $valid : V \rightarrow Bool$ *is a* validation function *satisfying* $valid(r) = True$.

We use RADAGs to model structures which arise from using version control systems (VCS). Each vertex in G corresponds to a commit in VCS repository. An edge between two vertices represents two subsequent commits. The root corresponds to the initial commit and the leaves (vertices with no outgoing edges) correspond to the latest commits of individual branches.

The validation function expresses whether a particular commit has the desired property that the system after this commit is correct (i.e. does not contain the bug). We call the vertices with this property (for which the validation function evaluates to *True*) *valid* vertices and the others *invalid* ones. Note that we assume that the graph has only one root and that the root is valid. If this is not the case, the graph can be easily modified by adding a dummy initial valid commit.

Definition 2. *A* regression point *of a RADAG* $((V,E), valid)$ *is a pair of vertices* (u,v) *such that* $(u,v) \in E$*,* $valid(u) = True$*, and* $valid(v) = False$*. A regression point* (u,v) *is a* regression predecessor *of a vertex* $w \in V$ *if* w *is equal to* v *or* w *is reachable from* v*.*

We are now ready to formally state our problem.

Regression Predecessors Problem: Given a RADAG $(G, valid)$ and a set of invalid leaves L of G, find at least one regression predecessor for each leaf from L.

Note that one invalid leaf can have several regression predecessors and one regression point can be a regression predecessor of several invalid leaves. Therefore, the regression predecessors problem may have several different solutions. For an example of such a RADAG see Figure 1.

3 REGRESSION PREDECESSORS ALGORITHM (RPA)

A naive solution to the regression predecessors problem would be to evaluate the function *valid* for each vertex, identify all regression points in G, and find a regression predecessor of each invalid leaf using the reachability relation. Because this approach identifies all regression points we can choose the latest regression predecessor of every invalid leaf. However, the price is crucial; the function *valid* is evaluated for every vertex which is assumed to be extremely time-consuming.

In this section we present a new algorithm, the Regression Predecessors Algorithm (RPA), that substantially decreases the number of vertices for which the function *valid* is evaluated and tends to find the latest regression points at the same time.

3.1 Basic Schema

The main idea of RPA is based on the observation that if a leaf l is invalid then every path starting in a valid vertex and leading to l must contain at least one regression predecessor of l. This reduces the problem to two tasks: finding a path and detecting a regression point on the path.

For the basic description of RPA see Algorithm 1. The algorithm maintains the set *UnprocessedLeaves* which consists of those invalid leaves for which a regression predecessor has not been computed yet. The set *KnownValid* consists of those vertices for which the function *valid* has been evaluated and are valid (initially only the *root* is known to be valid). In each iteration, the algorithm chooses a leaf l from the set *UnprocessedLeaves*. A regression predecessor of l is acquired by building a path p_l which connects a valid vertex $x \in KnownValid$ with l and by finding a regression point (u,v) on this path. While searching for the regression point (u,v), the function *valid* is evaluated for some vertices on the path p_l. The newly detected valid vertices form a set *NewValid* and the set *KnownValid* is updated accordingly.

The algorithm also exploits the fact that one regression point can be a regression predecessor of several invalid leaves. Therefore, every time a regression point (u,v) is found, it is propagated to every invalid leaf $m \in UnprocessedLeaves$ such that m reachable from v. Every such m is removed from *UnprocessedLeaves*. After this propagation step, the procedure also removes from the graph all vertices reachable from v. By removing vertices we avoid propagation of regression points to leaves for which a regression predecessor has already been found and avoid unnecessary traversal of

```
1 function RPA(G, L)
      input: a RADAG $G = ((V,E), valid : V \to Bool)$ with root $r$
      input: a set of invalid leaves $L$
      output: a regression predecessor for each leaf $l \in L$
2     $UnprocessedLeaves \leftarrow L$
3     $KnownValid \leftarrow \{r\}$
4     while $UnprocessedLeaves \neq \emptyset$ do
5         $l \leftarrow$ a leaf from UnprocessedLeaves
6         $UnprocessedLeaves \leftarrow UnprocessedLeaves \setminus \{l\}$
7         $p_l \leftarrow$ a path $\langle x, \dots, l\rangle$ such that $x \in KnownValid$
8         $(u,v) \leftarrow$ find a regression point on $p_l$
9         $KnownValid \leftarrow KnownValid \cup NewValid$
10        output $(u,v)$ is a regression predecessor of $l$
11        propagateRegressionPoint$(v, (u,v))$                    // optional, Alg. 2
```

Algorithm 1: Regression Predecessors Algorithm (basic schema)

```
1 function propagateRegressionPoint(k, (u,v))
      input: a regression point $(u,v)$
      input: a vertex $k$ reachable from $v$ (or $v = k$)
2     for $(k,l) \in E$ do
3         propagateRegressionPoint(l, (u,v))
4     if $k \in UnprocessedLeaves$ then
5         output $(u,v)$ is a regression predecessor of $k$
6         $UnprocessedLeaves \leftarrow UnprocessedLeaves \setminus \{k\}$
7     remove k from the graph
```

Algorithm 2: regression point propagation

the graph. For a complete description of the procedure, see Algorithm 2.

On the one hand, the propagation can result in saving some validation calls. On the other hand, the use of propagation may be in conflict with the desire to identify the latest regression predecessors. Therefore, the usage of propagation is optional. Section 4 demonstrates the behaviour of the algorithm both with and without the propagation step.

There are further three key aspects that affect the efficiency of the algorithm: the order in which leaves are chosen from the set *UnprocessedLeaves*, the method of building the path connecting a valid vertex with the invalid leaf, and the method of regression points identification. We focus on these three aspects in the following text.

3.2 Identification of Regression Points

In this subsection we give the details of our solution to the problem of finding a regression point on a given path $p = \langle v_0, v_1, \dots, v_l\rangle$ connecting a valid vertex v_0 with an invalid vertex v_l.

Linear search The simplest solution to the task is to evaluate the function *valid* for each vertex on the path, starting with v_l and going backwards. As soon as a valid vertex v_i is found, the algorithm outputs (v_i, v_{i+1}) as a regression predecessor of v_l. By starting with v_l and going backwards we guarantee that (v_i, v_{i+1}) is the nearest regression point of v_l along this path. The disadvantage of this approach is that in the worst case all vertices on the path are tested for validity. Because the commit graphs of VCS usually contain hundreds or thousands of commits and the evaluation of the function valid is assumed to be very expensive, the linear search is practically unusable.

Binary search Provided that the first vertex of the path is valid and the last is invalid (which is always our case) we can use binary search to find a regression point. Let $p = \langle v_0, v_1, \dots v_{mid}, \dots, v_l\rangle$ be a path such that v_0 is valid, v_l is invalid, and v_{mid} is the middle vertex of this path. If v_{mid} is valid then there is a regression point on the path $\langle v_{mid}, \dots, v_l\rangle$. Otherwise, there is a regression point on the path $\langle v_0, \dots v_{mid}\rangle$. We can thus always reduce p_l into half and recursively re-

```
1 function multSearch($p$)
      input: a path $p = \langle v_0, \ldots, v_l \rangle$ with valid $v_0$ and invalid $v_l$
      output: a regression point contained in $p$
2     if $l = 1$ then
3         return $(v_0, v_1)$
4     $k = 1$
5     while $l - (2^k - 1) > 0$ do
6         if $valid(v_{l-(2^k-1)})$ then
7             return multSearch($\langle v_{l-(2^k-1)}, \ldots, v_{l-(2^{k-1}-1)} \rangle$)
8         $k = k + 1$
9     return multSearch($\langle v_0, \ldots, l - (2^{k-1} - 1) \rangle$)
```

Algorithm 3: multiplying search

peat the procedure. Eventually, we end up with a path of length 2, thus a regression point is found.

Contrary to the linear search it is not guaranteed that the binary search finds the regression point which is nearest to v_l. The main advantage of the binary search is that it always performs logarithmically many checks, since the path is in each iteration reduced by half. Its efficiency is not much affected by the position of the regression points on the path.

Multiplying search The so-called multiplying search approach combines the advantages of both binary and linear search approaches as it performs asymptotically fewer validity checks than the linear search and at the same time tends to find a regression point which is closer to the last vertex v_l than the regression point found by the binary search.

Let $p = \langle v_0, v_1, \ldots, v_l \rangle$ be a path such that v_0 is valid and v_l is invalid. The *multiplying search* first evaluates the function *valid* for the vertex v_{l-1}. If v_{l-1} is not valid, then the function is stepwise evaluated for vertices $v_{l-(2^2-1)}$, $v_{l-(2^3-1)}$, $v_{l-(2^4-1)}, \ldots$ forming exponentially large gaps between individual vertices. The procedure eventually finds an i such that the vertex $v_{l-(2^{i-1}-1)}$ is invalid and either $v_{l-(2^i-1)}$ is valid or $l - (2^i - 1) < 0$. If the former happens, then the procedure recursively continues with the new path $\overline{p} = \langle v_{l-(2^i-1)}, v_{l-(2^i-2)}, \ldots, v_{l-(2^{i-1}-1)} \rangle$. In the latter case the procedure recursively continues with the path $\overline{p} = \langle v_0, v_1, \ldots, v_{l-(2^{i-1}-1)} \rangle$. The procedure converges to a path containing only two vertices such that the first vertex of the path is valid and the second invalid, i.e., a regression point is found. For the complete description see Algorithm 3.

The number $C(n)$ of vertices on which the function *valid* is evaluated on a path of length n is bounded by the recurrence equation $C(n) \leq C(\frac{n}{2}) + \log n$. In each recursive call the number of evaluations is at most $\log n$ and the length of the path is decreased at least by half. The solution of the recurrence equation (using the Master theorem (Verma, 1994)) gives an upper bound $O(\log^2 n)$ on the number of vertices on which the function *valid* is evaluated.

Note that multiplying search can significantly outperform binary search in many cases. Its performance depends on the distance of the regression points from the leaf. The closer are the regression points to the leaf, the more likely multiplying search outperforms binary search. For example, assume two paths $p_1 = \langle v_0, v_1, \ldots, v_{1023} \rangle$, $p_2 = \langle v_0, v_1, \ldots, v_{1023} \rangle$, where (v_{1007}, v_{1008}) is the only regression point of p_1 and (v_{512}, v_{513}) is the only regression point of p_2. The binary search approach performs 9 validity queries on both paths whereas the multiplying search approach performs 6 queries on p_1 and 17 queries on p_2.

3.3 Leaf Selection and Path Construction

Our next goal is to specify the order in which unprocessed leaves are chosen and determine the method of building a path connecting a valid vertex with the chosen leaf.

We assume that the directed acyclic graphs induced by VCS are represented using adjacency lists (see (Cormen et al., 2009)) in which every vertex is equipped both with a list of its direct successors and a list of its direct predecessors. In the initialisation phase of RPA we compute the length of the shortest paths from v to l for each vertex $v \in V$ and invalid leaf $l \in L$. For every pair $(v, l) \in V \times L$ we maintain a successor of v so that the chain of successors originating at the vertex v runs forward along a shortest path from v to l. This computation is done by running a backwards breadth-first search from each $l \in L$ using the list of predecessors, for details see e.g. (Cormen et al., 2009).

```
1  function priorityBasedRPA(G, L)
       input: a RADAG $G = ((V,E), valid : V \to Bool)$ with root $r$
       input: a set of invalid leaves $L$
       output: a regression predecessor for each leaf $l \in L$
2      for each $(v,l) \in V \times L$ compute the value $dist(v,l)$
3      for each $l \in L$ do
4          $dist(l) \leftarrow dist(r,l)$
5          $start(l) \leftarrow r$
6      $UnprocessedLeaves \leftarrow L$                         // priority queue
7      while $UnprocessedLeaves \neq \emptyset$ do
8          $l \leftarrow UnprocessedLeaves.dequeueMinimum()$
9          $p_l \leftarrow$ a shortest path $\langle x, \ldots, l \rangle$ such that $x = start(l)$
10         $(u,v) \leftarrow$ find a regression point on $p_l$
11         $KnownValid \leftarrow KnownValid \cup NewValid$
12         output $(u,v)$ is the regression predecessor of $l$
13         propagateRegressionPoint($v, (u,v)$)                 // optional, Alg.2
14         updatePriorities($NewValid$)                         // Alg.5
```

Algorithm 4: Regression Predecessor Algorithm

In what follows we use $dist(v,l)$ to denote the length of the shortest path leading from the vertex v to the leaf l; we further define:

$$dist(l) = min\{dist(u,l) \mid u \in KnownValid\}$$
$$start(l) = u \text{ such that } u \in KnownValid$$
$$\text{and } dist(u,l) = dist(l).$$

In other words, $dist(l)$ denotes the length of a shortest path leading to l from a vertex u for which the function *valid* has been evaluated and is valid (i.e. belongs to the set *KnownValid*). The first vertex of such a path is denoted $start(l)$. As the set *KnownValid* changes during the computation, so may the values $dist(l)$ and $start(l)$. Initially, only the root r of the graph is known to be valid, therefore $dist(l) = dist(r,l)$ and $start(l) = r$ for each $l \in L$.

The way in which RPA fixes the order in which invalid leaves are processed and determines which paths should be used for identification of regression points is based on the following observation. The shorter path we process the fewer number of evaluations of the function *valid* is performed, independent on the regression finding approach. For a complete description of the RPA algorithm see Algorithm 4, for an illustrative example see Section 3.4.

The RPA algorithm maintains the set *UnprocessedLeaves* as a priority queue where each $l \in UnprocessedLeaves$ is assigned the priority $dist(l)$. In every iteration the algorithm extracts the leaf l with minimum priority from *UnprocessedLeaves* and constructs the shortest path leading to l. Moreover, each iteration is supplemented by the method *updatePriorities*(*newValid*) that updates the $dist(l)$ and $start(l)$ values (see Algorithm 5).

```
1  function updatePriorities(NewValid)
       input: A set of valid vertices $NewValid$
2      for each $v$ in $NewValid$ do
3          for each $leaf \in UnprocessedLeaves$
             do
4              if $dist(v, leaf) < dist(leaf)$ then
5                  $start(leaf) \leftarrow v$
6                  $dist(leaf) \leftarrow dist(v, leaf)$
```

Algorithm 5: priority update

3.4 Example

Figure 2 demonstrates the execution of RPA with multiplying search and propagation. In this example, the RADAG has only invalid leaves, $L = \{g,k,m,p\}$, and the task is to find the regression predecessors for all leaves. The computation consists of 3 iterations. The function *valid* is evaluated only for 7 out of 16 vertices and 3 regression points are found. We list the values of control variables in each iteration and illustrate them on the graph. The vertices on which the *valid* function has been evaluated are filled with green or red color depending on their validity. The vertices removed from the graph are shaded.

I. iteration
– Removed vertices = $\emptyset$
– $KnownValid$ = $\{a\}$
– Priority queue = $\langle p,g,k,m \rangle$
– $dist(p) = 5$, $dist(g) = 6$, $dist(k) = 6$, $dist(m) = 7$
– $start(p) = a$
– path p_p = $\langle a,b,c,n,o,p \rangle$
– Tested vertices: o,c,n
– Regression point of p_p = (n,o)
– Propagated to: $\{p\}$

II. iteration
– Removed vertices = $\{o,p\}$
– $KnownValid$ = $\{a,c,n\}$
– Priority queue = $\langle k,m,g \rangle$
– $dist(k) = 3$, $dist(m) = 4$, $dist(g) = 4$
– $start(k) = n$
– path p_k = $\langle n,i,j,k \rangle$
– Tested vertices: j,i
– Regression point of p_k = (n,i)
– Propagated to: $\{k,m\}$

III. iteration
– Removed vertices = $\{o,p,i,j,k,l,m\}$
– $KnownValid$ = $\{a,c,n\}$
– Priority queue = $\langle g \rangle$
– $dist(g) = 4$
– $start(g) = c$
– path p_g = $\langle c,d,e,f,g \rangle$
– Tested vertices: f
– Regression point of p_g = (f,g)
– Propagated to: $\{g\}$

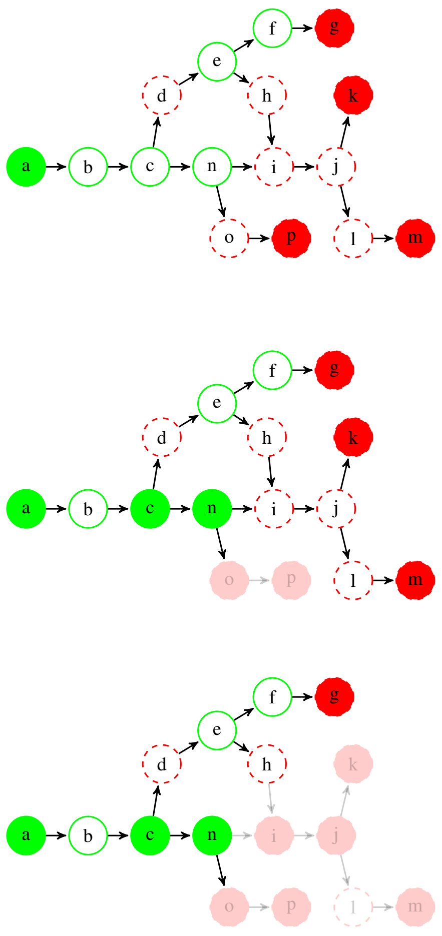

Figure 2: An illustrative example

3.5 Related Work

To the best of our knowledge, the first tool for finding regression points was the *git bisect* tool (Git bisect documentation, 2018) which is a part of the distributed VCS *git* (Git, 2018). The method used in the git bisect tool is called *bisection* and it was subsequently adopted by other VCS like Mercurial (Mercurial, 2018), Subversion (Pilato et al., 2008), and Bazaar (Bazaar, 2018).

The bisection algorithm takes as an input a single invalid commit and finds a regression point that precedes this commit. We provide only a brief description of the bisection algorithm here, for a more elaborated description please refer to (Git bisect algorithm overview, 2018). The algorithm represents the commits using a directed acyclic graph and assumes that the function *valid* is monotone, i.e. that every successor of an invalid commit is also invalid. It starts by taking as an input a single invalid commit called “bad” together with a one or more commits which are known to be valid. Then, it iteratively repeats the following steps:

(i) Keep only the commits that: a) precede “bad” commit (including the “bad” commit itself) and b) do not precede a commit which is known to be valid (excluding the commits which are known to be valid).

(ii) Associate to each commit c a number $r = min\{(x+1), n-(x+1)\}$ where x is the number of commits that precede the commit c and n is the total number of commits in the graph. Roughly speaking, this number represents the amount of information that can be obtained by evaluating the function *valid* on c. If c is valid then all of its predecessors are necessarily also valid (based on the assumption that the function *valid* is monotone). In the other case, if *c* is invalid, then all of its successors are necessarily also invalid.

(iii) Evaluate the function *valid* for the commit *v* with the highest associated number. If *v* is invalid then it becomes the "bad" commit.

Eventually there will be only one invalid commit left in the graph with one of its predecessors in the original graph being valid. This pair of vertices forms the regression predecessor of the original "bad" commit. Although the main idea of the bisection method is based on the monotonicity of the function *valid*, it is guaranteed that the algorithm finds a regression predecessor of the "bad" commit even if the function *valid* is not monotone.

There are two main drawbacks of git bisect comparing to RPA. First, the bisection algorithm does not tend to find the latest regression predecessor. Second, experiments (see the following section) demonstrate that git bisect evaluates more commits than RPA. The reason of this behaviour is that RPA prefers shortest paths while git bisect prefers vertices with the highest associated number. To demonstrate the difference let us consider a graph with one leaf and two paths connecting the root with the leaf. If one path is very short and the second one very long, then RPA prefers the short path while git bisect evaluates vertices on the long one. If a graph contains only one path leading to an invalid leaf, git bisect evaluates the same vertices as RPA combined with binary search.

There is also further related work that deals with problems similar to ours. Heuristics for automated *culprit finding* (Ziftci and Ramavajjala, 2015) are used for isolating one or more code changes which are suspected of causing a code failure in a sequence of project versions. They assume that the codebase is tested/validated regularly (e.g. after every *n* commits) using some test suit. If a bug is detected, they search for the culprit only among the changes to the codebase that have been made since the latest appliance of the test suite. The individual versions are rated according to their potential to cause the failure (e.g. versions with many code changes are rated higher) and versions with high rate are tested as first. The *culprit finding* technique (Ziftci and Ramavajjala, 2015) is efficiently applicable only for searching in a short term history and it assumes that there is only one culprit.

Delta debugging (Zeller, 1999) is a methodology to automate the debugging of programs using the approach of a hypothesis-trial-result loop. For a given code and a test case that detects a bug in the code, the delta debugging algorithm can be used to trim useless functions and lines of the code that are not needed to reproduce to bug. The delta debugging cannot be used for finding regression points in VCS. However, we believe that it can be incorporated into RPA and improve its performance by reducing the portion of code that need to be validated by the function *valid*.

A *regression testing* (Agrawal et al., 1993) and *continuous integration testing* (Duvall, 2007) are types of software testing that verifies that software previously developed and tested still performs correctly even after it was changed or interfaced with other software. These two techniques are suitable for fixing bugs that are detected right after they are introduced. However, if a bug that lied in a codebase for some time is detected, e.g. because of extending the coverage of the tests, a technique like RPA need to be used. That is, RPA and regression testing/continous integration testing are mutually orthogonal techniques

SZZ (Sliwerski et al., 2005; Kim et al., 2006) is an algorithm for identifying commits in a VCS that introduced bugs, however it works in a quite different settings. It assumes, that the bug has been already fixed and that the commit that fixed the bug is explicitly known or can be found in a log file. This allows to identify particular lines of code that fixed the bug and this information is then exploited while searching for the bug-introducing commit. In our settings, the bugs are not fixed yet, thus SZZ cannot be used.

In our previous work (Bendík et al., 2016), a structure similar to RADAG appears. However, that structure is monotone and therefore, the problem formulated in (Bendík et al., 2016) substantially differs from the regression predecessors problem and the algorithm presented in that work cannot be used for finding regression points in RADAGs.

Finally, we relate the regression predecessors problem with well known problems from graph theory. The latest regression point can be found using the breadth-first-search (BFS) algorithm (Jungnickel, 1999). As our goal is to minimize the number of validity queries, BFS is not suitable as it queries every vertex. Therefore, we come with a new, specialized, algorithm.

4 EXPERIMENTAL RESULTS

We demonstrate the performance of the variants of RPA on two types of use cases. We first focus on the problem of finding a regression predecessor of a single invalid leaf. We then focus on the problem of finding regression predecessors of a set of invalid leaves. We also compare the performance of the RPA variants to that of the git bisect tool (Git bisect documentation, 2018; Git bisect algorithm overview, 2018).

As benchmarks we use large real open source projects, taken from the GitHub open source showcases (Github Showcases, 2018), with at least 8 ac-

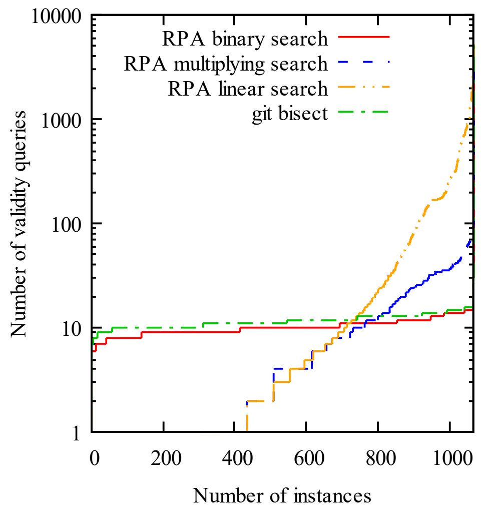

Figure 3: Cumulative distribution plot of performed validity queries for each evaluated algorithm. A point with coordinates [x,y] can be read as “*x* instances were solved by using at most *y* validity queries”.

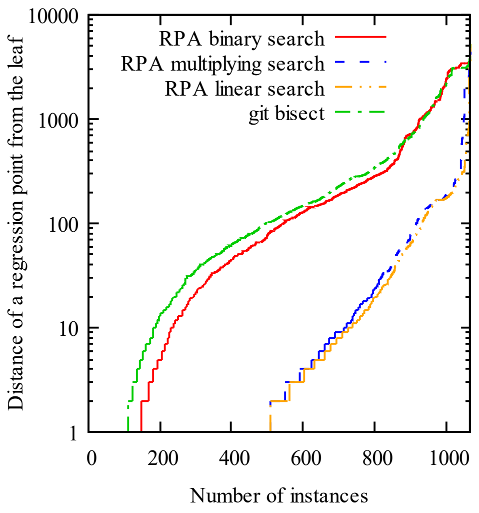

Figure 4: Cumulative distribution plot of distance between regression predecessor and corresponding invalid leaf for each evaluated algorithm. A point with coordinates [x,y] can be read as “in *x* instances the distance was at most *y*”.

tive branches or at least 1000 commits. Due to the size of the projects it would be intractable to build and test all commits in these projects. Therefore we use those projects from (Github Showcases, 2018) that employ TravisCI (Travis CI, 2018). Travis CI is a service used to build and test projects hosted at GitHub and the results of all tests that were run on these projects are publicly available. Whenever our algorithm needs to validate a commit, it acquires the results of the tests from the publicly available Travis CI database instead. Overall we selected 84 projects with 1069 invalid leaves in total. Our selection includes for example the Rails web-application framework (Ruby on Rails, 2018; Bächle and Kirchberg, 2007), the PHP Interpreter (PHP interpreter, 2018), or the ArangoDB (ArangoDB, 2018).

We do not provide any details about the architecture of the computer on which we run the experiments because the computation time is not a relevant criterion in our study. It took a few seconds to run all the experiments because we didn’t actually run the tests. As a main criterion for measuring the efficiency of evaluated algorithms we use the number of performed validity queries and the distance between identified regression points and corresponding invalid leaves (in order to measure the tendency to find the latest regression points). Complete results of all measurements are available at `https://tinyurl.com/y7ps4yyl`.

4.1 Single Invalid Leaf Instances

We first analyse how variants of RPA and git bisect perform while searching for a regression predecessor of a single invalid leaf. In the case of finding a regression predecessor of a single invalid leaf, it makes no sense to use propagation. Therefore, we always build the shortest path from a valid vertex and employ either binary or multiplying search. We also include the naive linear search approach that builds a path and checks one by one individual commits on the path.

The results comparing the number of the performed validity queries are shown in Fig. 3. In this plot, we show the cumulative distributions of the performed validity queries for each evaluated algorithm. The performance of git bisect and binary search was quite stable on all instances since they always perform logarithmically many validity queries. In particular, git bisect and binary search needed to perform at most 17 and 16 validity queries, respectively, to solve the hardest instances. The performance of multiplying search was less stable, since the number of performed validity queries depends on the position of the regression points as we have discussed in Section 3.2. Multiplying search needed to perform from 1 to 101 validity queries. Linear search was negligibly better than multiplying search on instances where the regression point was quite close, however it needed to perform up to 3458 validity queries on the harder instances.

In addition to the plot of cumulative distribution we show in Table 1 the number of instances on which one algorithm performed strictly less validity queries than its competitors. The table shows that binary search was superior to git bisect on most of the instances. This is caused by the nature of these two algorithms. Both of them need to perform just logarithmically many validity queries, but git bisect searches the whole graph

Table 1: The number of instances on which the algorithm named in the row performed strictly less validity queries than the other algorithms. We use the following abbreviations: mult = RPA with multiplying search, bin = RPA with binary search, git = git bisect.

$<$	mult	bin	git
mult	—	736 (69%)	759 (71%)
bin	313 (29%)	—	824 (77%)
git	294 (28%)	25 (2%)	—

Table 2: The number of instances on which the algorithm named in the row found a strictly closer regression predecessor than the other algorithms. We use the following abbreviations: mult = RPA with multiplying search, bin = RPA with binary search, git = git bisect.

$<$	mult	bin	git
mult	—	717 (67%)	843 (79%)
bin	26 (2%)	—	511 (48%)
git	27 (3%)	328 (30%)	—

whereas binary search traverses only a single path. Moreover, RPA always chooses the shortest usable path which might lead to a significant improvement to git bisect. In particular, in some instances, git bisect performed 3 times more validity queries than RPA with binary search. On the other hand, RPA with binary search needed to perform only 1.14 times more validity queries than git bisect in the worst case. Multiplying search was strictly better then its competitors on about 70 percent of instances. It was worse on the instances with large paths where regression points are relatively far from the leaves.

Besides the number of performed validity queries, we also measure the tendency of the algorithms to find the latest regression predecessors. So far we have not precisely defined which regression predecessor is the latest one and there is more than one suitable definition. In the case of binary, multiplying, and linear search we look for a regression predecessor on a path; thus, we can say that the latest regression predecessor is the regression point which is closest to the end of the path (i.e., closest to the leaf). In our experiments, multiplying search found the closest regression predecessor on the path in 86 percent of instances whereas binary search only in 29 percent of instances. This notion of latest regression predecessor is not applicable to git bisect because git bisect does not operate on paths. In order to compare RPA with git bisect, we measured the distance between the found regression predecessor and the corresponding invalid leaf, i.e. the shortest path between these two vertices in the commit graph. Fig. 4 shows a plot of cumulative distributions of distance between regression predecessors and corresponding leaves for each evaluated algorithm. The results demonstrate that multiplying search substantially outperforms binary search and git bisect, and that binary search is better than git bisect. The naive linear search is just negligibly better than multiplying search.

In addition, Table 2 shows the number of instances on which one algorithm found strictly closer regression point than its competitors. The multiplying search strictly dominates both its competitors; the regression predecessor found by multiplying search was closer to the leaf than the one found by git bisect in 79 percent of instances. Binary search performed slightly better, it was dominated by multiplying search only in 67 percent of instances.

4.2 Sets of Invalid Leaves

We now demonstrate the performance of the RPA variants on the problem of finding regression predecessors for a set of invalid leaves. In particular, we evaluate both proposed approaches for finding regression points, i.e., the binary and multiplying search. Moreover, we evaluate both these approaches in two variants: with and without regression point propagation (the optional part of RPA).

We also compare the variants of RPA to the git bisect tool. As mentioned in Section 3.5, git bisect deals with the problem of finding a regression predecessor of a single invalid leaf. Therefore, in order to solve the problem of regression predecessors for a given set of invalid leaves L, git bisect has to be run once per each leaf from L. As all these runs are independent, it might happen that some commits are evaluated repeatedly. In order to avoid the repeated evaluations, we supplement git bisect with a cache saving the results of the previous evaluations. Thus, every commit is evaluated at most once.

As benchmarks we used the 84 projects from GitHub showcases; the goal was to find a regression predecessor for every invalid leaf in every project. In this part of experimental evaluation we focus solely on the number of performed validity queries. Figures 5 and 6 show the cumulative distribution plots of the performed validity queries for the variants of RPA with and without propagation, respectively. In both plots we also include the results achieved by git bisect. Moreover, we include the cumulative distribution of the number of invalid leaves (solid black line), i.e. a point with coordinates $[x,y]$ means that x instances have at most y invalid leaves.

In general, the regression point propagation significantly reduces the overall number of performed valid-

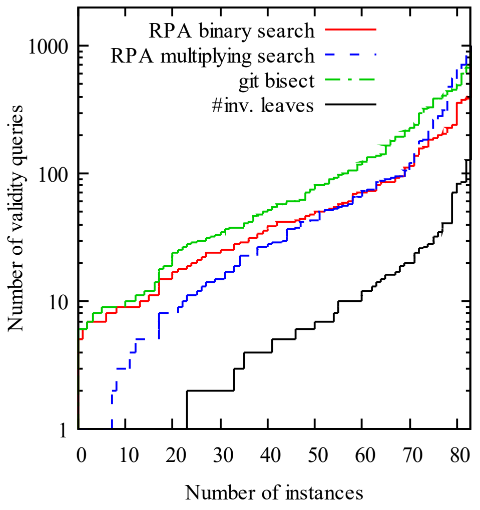

Figure 5: Cumulative distributions of number of performed validity queries for git bisect and variants of RPA without propagation. The black line is the cumulative distribution of number of invalid leaves.

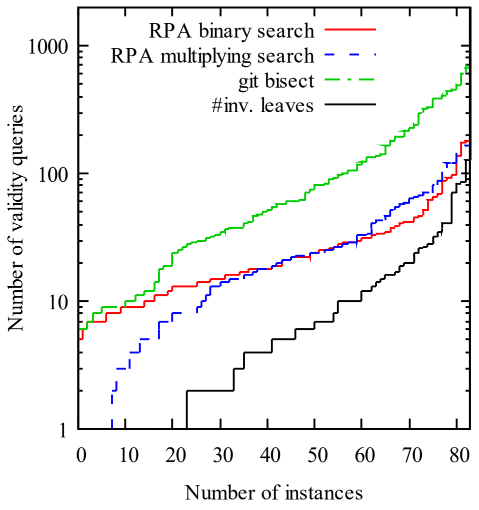

Figure 6: Cumulative distributions of number of performed validity queries for git bisect and variants of RPA with propagation. The black line is the cumulative distribution of number of invalid leaves.

ity queries. Considering the difference in performance between variants of RPA with multiplying and binary search, respectively, we observe the same behaviour as in the case of finding regression predecessors for single invalid leaves. There are some instances on which multiplying search outperformed binary search, and some instances on which binary search outperformed multiplying search. The improvement to git bisect is for both variant of RPA even more significant than in the case of single leaves instances since in this case the priority queue of RPA fully manifested. Note, that all plots are in a logarithmic scale.

4.3 Recommendations

We have presented several variants of the RPA algorithm and the experimental results show that the variants are in general incomparable. There is no variant that would beat all the others independently of the comparison criteria. In the case where the user searches for a regression predecessor of a single invalid leaf it makes no sense to use propagation. If the user prefers finding the closest regression points, we suggest her to use multiplying search. In the other case, where the user rather prefers to minimize the number of performed validity queries, the choice of algorithm depends on the size of the commit graph and also on the (assumed) distance of regression points from the leaf. For large commit graphs with no assumptions about position of regression points we suggest the user to use RPA with binary search since it guarantees that only logarithmically many validity queries will be performed. For small commit graphs or graphs where regression points are assumed to be relatively close to leaves, we suggest the user to use RPA combined with multiplying search.

In the other case, where the user searches for regression predecessors of several invalid leaves, it might be worth to use propagation. If the user prefers finding the closest regression points to minimizing the number of performed validity queries, we suggest her not to use regression point propagation and employ multiplying search. In the opposite case, when the user focus mainly on minimizing the number of validity queries, we suggest her to use regression point propagation and employ either binary search or multiplying search (based on the size of the commit graph and assumed positions of regression points as discussed above).

5 CONCLUSION

We have presented a new algorithm, called the Regression Predecessors Algorithm (RPA), for finding regression points in projects under version control. The algorithm has several variants, the choice of which depends on whether the user prefers to minimise the number of performed validity queries or to find the latest regression points. We have experimentally compared the variants among themselves as well as against the state-of-the-art tool git bisect. The results show that the variants of RPA are in general incomparable as there is no variant that would beat all the others independently of the criteria. The main strength of RPA lies in the ability to minimise the number of validity queries while respecting the requirement to find the latest regression point. In all cases the RPA algorithm is superior to the algorithm used in git bisect.

Acknowledgement

This project has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 692474, project name AMASS. This Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Czech Republic, Germany, Sweden, Austria, Italy, United Kingdom, France.

REFERENCES

Agrawal, H., Horgan, J. R., Krauser, E. W., and London, S. (1993). Incremental regression testing. In *ICSM*, pages 348–357. IEEE Computer Society.

ArangoDB (2018). ArangoDB. `https://github.com/arangodb/arangodb`. Accessed: 2018-04-03.

Bächle, M. and Kirchberg, P. (2007). Ruby on rails. *IEEE Software*, 24(6):105–108.

Bazaar (2018). Bazaar. `http://bazaar.canonical.com/`. Accessed: 2018-04-03.

Bendík, J., Benes, N., Barnat, J., and Cerná, I. (2016). Finding boundary elements in ordered sets with application to safety and requirements analysis. In *SEFM*, volume 9763 of *Lecture Notes in Computer Science*, pages 121–136. Springer.

Clarke, E. M., Grumberg, O., and Peled, D. A. (2001). *Model checking*. MIT Press.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms (3. ed.)*. MIT Press.

Duvall, P. M. (2007). *Continuous integration*. Pearson Education India.

Git (2018). Git. `https://git-scm.com/`. Accessed: 2018-04-03.

Git bisect algorithm overview (2018). Git bisect algorithm overview. `https://git-scm.com/docs/git-bisect-lk2009.html`. Accessed: 2017-04-03.

Git bisect documentation (2018). Git bisect documentation. `https://git-scm.com/docs/git-bisect`. Accessed: 2018-04-03.

Github Showcases (2018). GitHub Showcases. `https://github.com/showcases`. Accessed: 2018-04-03.

Jungnickel, D. (1999). Graphs, networks and algorithms, volume 5 of algorithms and computation in mathematics.

Kim, S., Zimmermann, T., Pan, K., and Jr., E. J. W. (2006). Automatic identification of bug-introducing changes. In *ASE*, pages 81–90. IEEE Computer Society.

Mercurial (2018). Mercurial. `https://www.mercurial-scm.org/`. Accessed: 2018-04-03.

PHP interpreter (2018). PHP interpreter. `https://github.com/php/php-src`. Accessed: 2018-04-03.

Pilato, C. M., Collins-Sussman, B., and Fitzpatrick, B. W. (2008). *Version control with subversion - the standard in open source version control*. O'Reilly.

Ruby on Rails (2018). Ruby on Rails. `https://github.com/rails/rails`. Accessed: 2018-04-03.

Sliwerski, J., Zimmermann, T., and Zeller, A. (2005). When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5.

Travis CI (2018). Travis CI. `https://travis-ci.org/`. Accessed: 2018-04-03.

Verma, R. M. (1994). A general method and a master theorem for divide-and-conquer recurrences with applications. *J. Algorithms*, 16(1):67–79.

Zeller, A. (1999). Yesterday, my program worked. today, it does not. why? In *ESEC / SIGSOFT FSE*, volume 1687 of *Lecture Notes in Computer Science*, pages 253–267. Springer.

Ziftci, C. and Ramavajjala, V. (2015). Heuristics for automated culprit finding. US Patent 9,176,731.