Using δ-NFGs to identify and eliminate dead code in C# programs

Dead code is any code in a computer program which does not affect the application. It is prevalent in computer programs, can be harmful and pose significant risks. This research aims to detect and eliminate dead code. To do this, we used a newly introduced data structure, called δ-NFG, derived from the Program Dependency Graph (PDG). To fulfil our objective, we created a δ-NFG for each commit of a code repository and then identified two types of dead code: unreachable code and unused variables. This was done by using an altered Breadth-First Search (BFS) and by analyzing the graph's data and name flow. We have found that δ-NFGs are useful in the detection of revived code and can be helpful in future projects. For standard dead code detection, a simple PDG is sufficient.

Additional Key Words and Phrases: δ-NFG, dead code detection, dead code elimination, data flow analysis

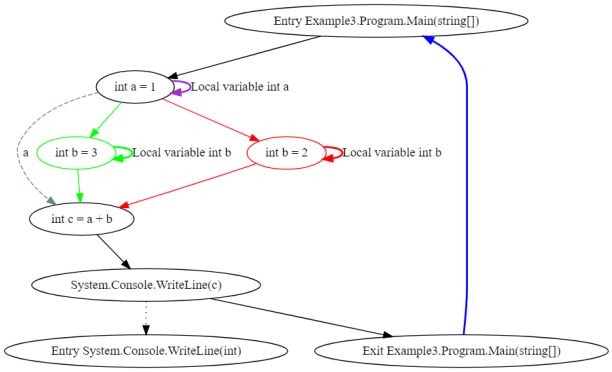
1 INTRODUCTION

Dead or inactive code is any code that has no effect on the application's behavior [14]. If executed, it could impact performance and create security risks [3, 18]. Furthermore, it impedes the comprehension of the rest of the source code [14, 17, 18]. It is thus a severe problem during the development and the publishing of the application. However, it occurs surprisingly often [3, 18] and research shows it matters to software professionals [20].

Some methods exist to detect and eliminate the presence of dead code. For example, Boomsma et al [2] researched how to eliminate dead code from PHP systems. Unlike static systems, PHP systems are dynamic, meaning that the identification of dead code is more difficult. To identify dead code in PHP or dynamic systems, Boomsma et al analyzed which files were used over time, and removed the unused files, classifying them as "potentially dead".

Another approach is by using program dependency graphs (PDG), introduced by Ferrante et al in 1984 [5]. A program dependence graph (PDG) shows both the data and control flow for each operation in a program [5]. Ferrante et al also describe how to identify and eliminate dead code using PDGs [5].

In 2018, Dash et al used the concept of PDG to create a new data structure, the NFG (Name Flow Graph) [4], used in their tool called RefiNym[1]. A name flow graph, or NFG, is a standard data flow graph, with the edges annotated with lexemes that flow across them. A data flow graph is a graph that shows data dependencies in code. The purpose of its creation was to separate common types, such as a string, into multiple types, such as "Password" or "Username", in order to have fewer errors and bugs during the development of an application. Two years later, Pârtachi et al [16] created the δ-NFG in their tool called Heddle, where they use the δ-NFG in their technique called Flexeme[2]. The δ-NFG^{p,q} is the disjoint union of all



(a) δ-NFG generated from the code in (b)

```
1    namespace Example3
2    {
3        internal class Program
4        {
5            public static void Main(string[] args)
6            {
7                int a = 1;
-                int b = 2;
8    +           int b = 3;
9                int c = a + b;
10               System.Console.WriteLine(c);
11           }
12       }
13   }
```

(b) Code with the changes annotated from a commit

Fig. 1. A small example of a δ-NFG

nodes and edges across all code versions in [p,q] [16]. An example can be found in Figure 1. In Figure 1(a), a δ-NFG is shown, generated from the code in Figure 1(b). In this δ-NFG, we can observe the name and data flow with the dashed edges while the control flow is shown with bold edges. Green and red elements of the graph represent the different code versions. We will explain δ-NFGs more thoroughly in Section 3. Heddle was created to untangle commits. For example, if a commit contains both a refactoring and a bug fix, Heddle will separate the issues into different commits.

We will use Heddle to create the δ-NFG. As both tools currently only work on C# code, we will focus on dead code elimination in C# code.

In this research, we will explore the newly introduced data structure to create a solution to the widespread problem of the presence of dead code.

Our goal can then be defined as using the δ-NFG to identify and eliminate dead code in C# programs.

The following research questions (RQ) will help achieve our goal:

- RQ 1: How can δ-NFGs be used to identify and eliminate dead code?
- RQ 2: What are the characteristics of a δ-NFG that make it easier to use than a standard program dependency graph?

[1] https://github.com/askdash/refinym
[2] https://github.com/PPPI/Flexeme/tree/0.2

TScIT 37, July 8, 2022, Enschede, The Netherlands

This research is structured as follows. First, we will discuss related work in the fields of multiversion code representation and dead code identification and elimination. Then, we will explain how we retrieved the δ-NFG from Heddle before going into the details of our implementation. Lastly, we will describe the threats to validity of our research and conclude by answering our research questions.

2 RELATED WORK

In this section, we will discuss the previous work related to our research project. First, we will consider the representation of multiversion code. Then, we will discuss PDG-based techniques before conferring dead code detection.

Software analytics is the general area on which our research is focused. It considers source code, static and dynamic characteristics as well as related processes of their development and evolution. It is a critical area to research as the cost for maintaining all lifecycles of software must be low. Many techniques exist in this area. For example, dead code detection and clone detection are software analytics techniques.

To track the evolution of code, software repositories are often used. An interesting application of using software repositories is the multiversion representations of code. Le et al [12] proposed a Multiversion Interprocedural Control Flow Graph (MVICFG) for patch verification. This graph integrates and compares the control flow of multiple versions of programs. Five years later, in 2019, Alexandru et al [1] proposed using the MVICFG to produce the Lean Language-Independent Software Analyzer (LISA), a generic framework for representing and analyzing multi-revisioned software artifacts. Unlike them, our research uses another novel technique, the δ-NFG [16], representing both control and data flow.

As we have seen before, PDGs are relatively old as they were created in 1984 by Ferrante et al [5]. These are important to this research as the δ-NFG was derived from it. So which techniques did program dependency graphs help develop? In 2006, Liu et al used PDGs to detect software plagiarism [13], since PDGs are invariant during plagiarism. PDGs are also often used for clone detection [6, 8–10, 15, 21], which is a form of software analytics. As NFGs are simply PDGs augmented with lexemes, all those techniques will also work with NFGs.

Graphs are often used for the detection of dead code. An example is the data flow graph. Data flow analysis is performed by creating a data flow graph and then analyzing this graph to find, in our case, dead code. In 1994, Knoop et al [11] implemented a new technique based on data flow analysis called assignment sinking to determine and eliminate partially dead code. Partially dead code is code that is dead on some branch of the code but not on another. Assignment sinking eliminates assignments that compute values that are dead. In 1997, Gupta et al [7] improved this technique by adding resource availability and path profiling information.

Techniques to detect and eliminate dead code without using graphs exist as well. A previously discussed example is Boomsma et al's work [2], which is an example of accessibility analysis. Furthermore, in 2017, Wang et al [19] detected dead code using program slicing, which is the computation of the set of program statements, the program slice, that may affect the values at some point of interest, referred to as a slicing criterion. If graphs are used, then slicing-based approaches still work, but become reachability analysis problems.

3 δ-NFG CONSTRUCTION

We had to retrieve the δ-NFG from the source code of Heddle. As the source code was on the GitHub repository, it was easy to retrieve all the code. Before going into the details of what we changed, we will first explain how Heddle created its δ-NFG.

3.1 Heddle's δ-NFG construction

Heddle created its δ-NFG by first extracting both name flow and PDG (Program Dependency Graph) in Roslyn, the open-source compiler for C# and Visual Basic from Microsoft. The NFGs are stored in GraphViz Dot format, in a temporary folder. Then, Heddle marks each NFG. This is done by considering each node in it, and if it was removed or added by a commit, the node is colored in red or green respectively. The same is done with each edge. Finally, Heddle merges the NFGs belonging to certain commits into one NFG for each file, creating a δ-NFG per file.

In the δ-NFG, each node has the following information:

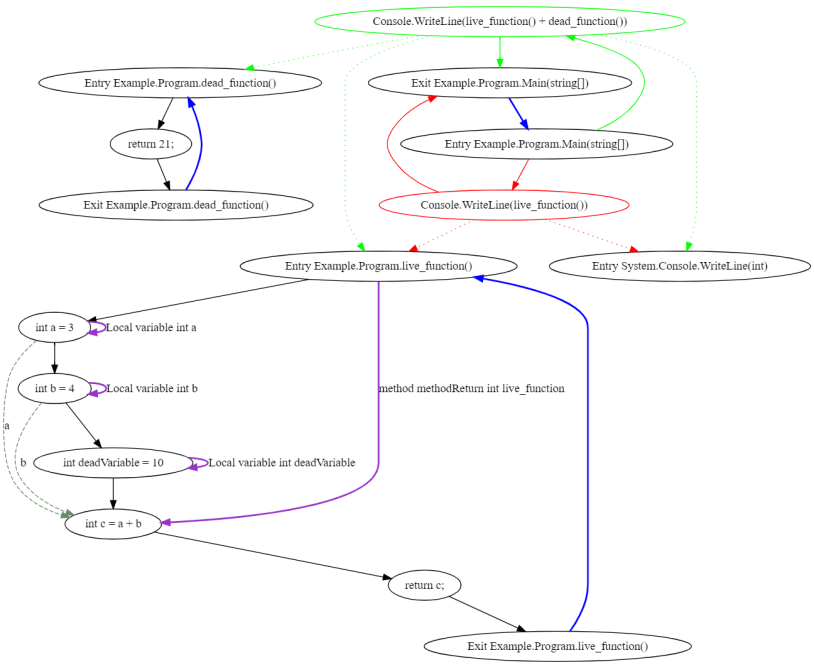
- **Cluster**: typically, it consists of the form "[namespace].-[class].[method]". It shows where in the file it belongs.
- **Label**: code expression. It is also the text shown on the node in the graph. Often does not show the entirety of the expression.
- **Span**: line-span. It represents the line number(s) on which the corresponding expression is.
- **Color**: a node can have three colors. If a node is black (default), the node is in all versions of the code. If a node is green, the node was added in one version of the code. If a node is red, the node was removed in one version of the code.

Furthermore, each edge has the following information:

- **Style**: an edge can be solid, dotted, dashed or bold. A solid edge shows the control flow of the graph. A dashed edge shows the data flow of the graph. A dotted edge connects an expression to the entry node of its corresponding called method. A bold edge connects the entry and exit nodes of a method.
- **Label**: often represents the name flows. It can also show other information such as names of local variables or return statements.
- **Color**: an edge can have five colors. If an edge is green, one of the nodes it is incident to was added in one code version. If an edge is red, one of the nodes it is incident to was removed in one version of the code. A purple edge is similar to a bold edge. A blue edge connects the exit and entry nodes of each method. Black is the default edge color.

The δ-NFGs are also stored in GraphViz Dot format so that they can be easily accessible. This is done via the NetworkX library[3]. The GraphViz Dot format also helps visualizing the graphs, so it is easy to manually spot mistakes.

[3] https://networkx.org/



(a) δ-NFG generated from the code in (b)

```
1   using System;
2
3   namespace Example
4   {
5       internal class Program
6       {
7           public static int dead_function()
8           {
9               return 21;
10          }
11
12          public static int live_function()
13          {
14              int a = 3;
15              int b = 4;
16              int deadVariable = 10;
17              int c = a + b;
18              return c;
19          }
20
21          public static void Main(string[] args)
22          {
-               Console.WriteLine(live_function());
23  +           Console.WriteLine(live_function() + dead_function());
24          }
25      }
26  }
```

(b) Code with the changes annotated from a commit

Fig. 2. An example of a δ-NFG

An example of δ-NFG can be found in Figure 2. In Figure 2(b) we can find the code that is represented in the δ-NFG found in Figure 2(a). The code is taken from Git where the addition and removal of lines of the corresponding commit are displayed. Here, we can see that line 23 has been changed to add `+ dead_function()`. This is represented in the graph by coloring the node representing the removed line in red and coloring the node representing the added line in green. The changes in control flow are also colored in their respective color. We also notice that for each method, entry and exit nodes exist. These are nodes that help visualize when we enter or exit a method. On the bottom left side of the graph, we can see two examples of data flow. These edges are represented in dashed style and have a variable name as the label. For example, the integer `a` is used to define the integer `c`, so there is a data flow connecting these two variables.

3.2 Our changes to Heddle's implementation

Heddle considers commits that: "

(1) Have been committed by the same developer within 14 days of each other with no other commit by the same developer in between them.
(2) Change namespaces whose names have a large prefix match.
(3) Contain files that are frequently changed together.
(4) Do not contain certain keywords (such as 'fix', 'bug', 'feature', 'implement') multiple times." [16]

Our implementation considers all commits as we do not need the conditions (unlike Heddle).

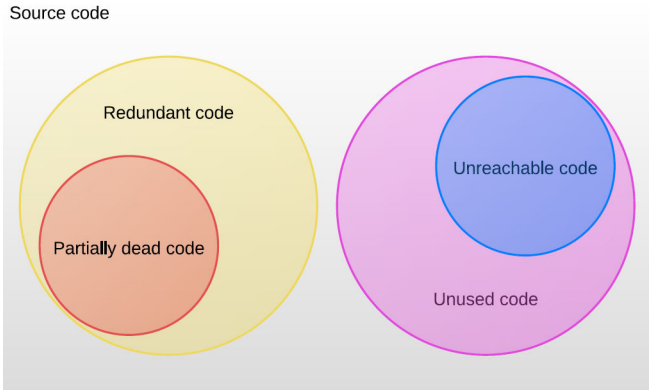


Fig. 3. Dead code's broad definition. Source: Boomsma et al, 2012 [2]

Furthermore, Heddle uses chains of SHAs while computing their δ-NFGs, while we use one commit per δ-NFG. We did this as it is easier to work with since for revived code detection, we need to know exactly which changes belong to which commits, something that is more difficult to discern when working with chains of commits.

4 IMPLEMENTATION

In this section we will describe our implementation of dead and revived code detection, as well as dead code elimination.

4.1 Dead code detection

Dead code is any code that has no effect on the application's behavior [14]. There are several types of dead code (Figure 3). First, redundant code is code that is unnecessarily executed. Inside this set is a subset called partially dead code. That is code that is dead only on some program paths, such as a variable that is only used in the "else" part of an if-statement. Then, unreachable code is code that is never reached or executed by the program, which is a subset of unused code, which is code that is never triggered.

Our implementation does not consider all types of dead code. Instead, we only detect unreachable code and unused variables. For example, when only considering the red and black elements of the graph in Figure 2, the top left cluster is unreachable code as it will never be reached. In practice, this is a function that is never called, `dead_function`. When only considering the green and black elements, the function `dead_function` is not dead as it is called in the "main" method. Furthermore, the integer `deadVariable` is an unused variable, as it is never used in the rest of the code. This can be seen with the data flow: there is no data flow edge coming out of the corresponding node.

```
def bfs(graph, start):
    """
    Breadth-first search algorithm that only visits green
      and black nodes and edges
    :param graph: delta-NFG
    :param start: starting node
    :return: set of reachable nodes
    """

    visited, queue = dict(), [start]
    while queue:
        vertex = queue.pop(0)
        if vertex not in visited and ('color' not in
     graph.nodes[vertex].keys() or not graph.nodes[vertex
     ]['color'] == "red"):
            visited[vertex] = len(visited)
            queue.extend(set(graph[vertex]) - set(visited
     ))
    return visited
```

Listing 1. Breadth-first search algorithm

We implemented the detection of unreachable code by using a breadth-first search (BFS), altered such that we only consider green and black elements. If we also considered red elements, we might detect unreachable code that is dropped in the newest commit, thus falsely detecting unreachable code. In Listing 1 we can find our implementation of the breadth-first search algorithm. To find the starting node for the BFS, we take the entry node of the "main" method. If that method does not exist, we use a random entry node in the biggest connected component of the graph. Currently, the detection algorithm only works on files in which methods are not called in other files. We therefore only detect unreachable code in a file itself and not in the whole codebase.

```
def unused_variables(G):
    """
    Detect unused variables of given delta-NFG G
    :param G: delta-NFG
    :return: array of nodes whose labels are unused
     variables
    """

    labels = nx.get_node_attributes(G, "label")

    # Get all variables in the delta-NFG
    variables = []
    for vertex in G:
        try:
            label_split = labels[vertex].split(" ")
            if label_split[2] == "=":
                variables.append(vertex)
        except IndexError:
            pass

    # Find all used variables
    used_variables = []
    for variable in variables:
        out_ = G.out_edges(variable, data="style",
     default="pink")

        for edge in out_:
            if edge[2] == "dashed":
                used_variables.append(variable)
            elif edge[2] == "solid":
                out_node = edge[1]
                label_split = labels[variable].split(" ")
                if label_split[1] in labels[out_node]:
                    used_variables.append(variable)

    unused = set(variables) - set(used_variables)
    return unused
```

Listing 2. Unused variables detection

The implementation of the detection of unused variables is shown in Listing 2. We first identify all the variables in one file using the label attribute of each node. We then find which of them are used. This is done by first checking if there is data flow coming out of the corresponding node. Data flow edges are dashed while control flow edges are solid. If there is, it is used. If there is not, we check if the next node following control flow uses the variable. If that is not the case, the variable is unused. Data flow edges between two nodes are only shown when there is at least one node between those nodes.

4.2 Revived code detection

The detection of revived code is more complex than the detection of dead code. We first compute unused variables and unreachable code as described in the previous subsection. Then, we calculate for each piece of dead code in which commit it is found dead. Finally, we construct two graphs. The first graph is the δ-NFG of the file and commit in which the corresponding variable or code is found dead. The second graph is the δ-NFG of the newest commit for which the corresponding file is present. When the graphs are constructed, we find a node in the second graph that is equivalent to the dead node in the first graph. If the found node is in the list of dead code previously computed, the piece of code is still dead. If not, it has been revived. The algorithm returns two arrays: one containing the revived variables, and one containing the variables that are still dead in the newest commit.

For example, in Figure 2, the top left cluster is revived code in that commit.

4.3 Dead code elimination

We thus obtain four distinct types of code: dead variables that have not been revived, unreachable code that has not been revived, revived variables and revived code (code that was previously unreachable). For this part of the implementation, we only used the dead variables.

Suppose we have retrieved the array of the still dead variables from the revived code detection algorithm. We then iterate through each variable in that array to find the line span and we compute the path to the file using information stored in the corresponding node. After determining all necessary information, we delete the lines in the file.

It is more challenging to eliminate dead code in general, such as methods. Indeed, each method has its own entry and exit nodes which are not specified in the files. Due to time constraints, we have not implemented the elimination of dead code in general.

5 TESTING

In this section we will discuss the tools we used for testing our implementation of the detection and elimination of dead code.

5.1 Dead and revived code detection

To test the implementation of dead and revived code detection, we artificially created a Git repository having revived variables, revived methods, and dead code. It is a small repository, so it is easy to manually check whether the δ-NFGs are correct. It is also easier to manually detect all types of dead code.

We then wrote four unit tests to confirm whether the detection functions gave the correct results. One of such tests is found in Listing 3.

```
def test_revived_functions():
    """
    Check whether the function reports the correct revived nodes
    """

    # Get revived nodes
    repository_name = "Example"
    revived_functions = revived_code(repository_name, 2)[0]

    assert len(revived_functions) == 3

    # These should be the labels of the nodes that are revived
    correct_labels = [
        '"return 21;"',
        '"Entry Example.Program.dead_function()"',
        '"Exit Example.Program.dead_function()"'
    ]

    for var in revived_functions:
        commit = var[0]
        node = var[1]
        dot = var[2]

        # Construct graph to get the cluster and line span of each node
        graph = get_graph(repository_name, commit, dot)

        # Get node label
        label = graph.nodes[node]["label"]

        assert label in correct_labels
        correct_labels.remove(label)  # Remove it so no other node can use the label
```

Listing 3. Revived code test

In this test, we first compute code that was unreachable but is now revived. We manually check in the δ-NFGs what they should be and save the labels of the corresponding nodes in an array called `correct_labels`. First, we assert the number of unreachable nodes. Then, for each computed revived node, we check whether its label is in `correct_labels`. If all labels in that array have been used, the test passes successfully.

Similar tests have been written for revived variables, unreachable nodes, and unused variables. Every unit test tests one of the four outcomes of the method `revived_code` which uses all other methods used for revived and dead code detection.

5.2 Dead code elimination

To test our implementation of dead code elimination, we used the same repository as for the testing of our implementation of dead and revived code detection. We wrote one unit test for this part of the implementation.

First, from the tests done before, we know that there is only one dead variable in the latest commit. This variable is represented in the node labeled `int deadVariable2 = 20`. In the actual .cs file, the line is `int deadVariable2 = 20;`, so we first check if that line is in the file. Then, we call the elimination method and check whether that line is still in the .cs file. If not, the test continues. Finally, we check whether no other line has been removed and then we reinsert the deleted line so we can run the test multiple times.

The line coverage of all unit tests combined can be found in Table 1.

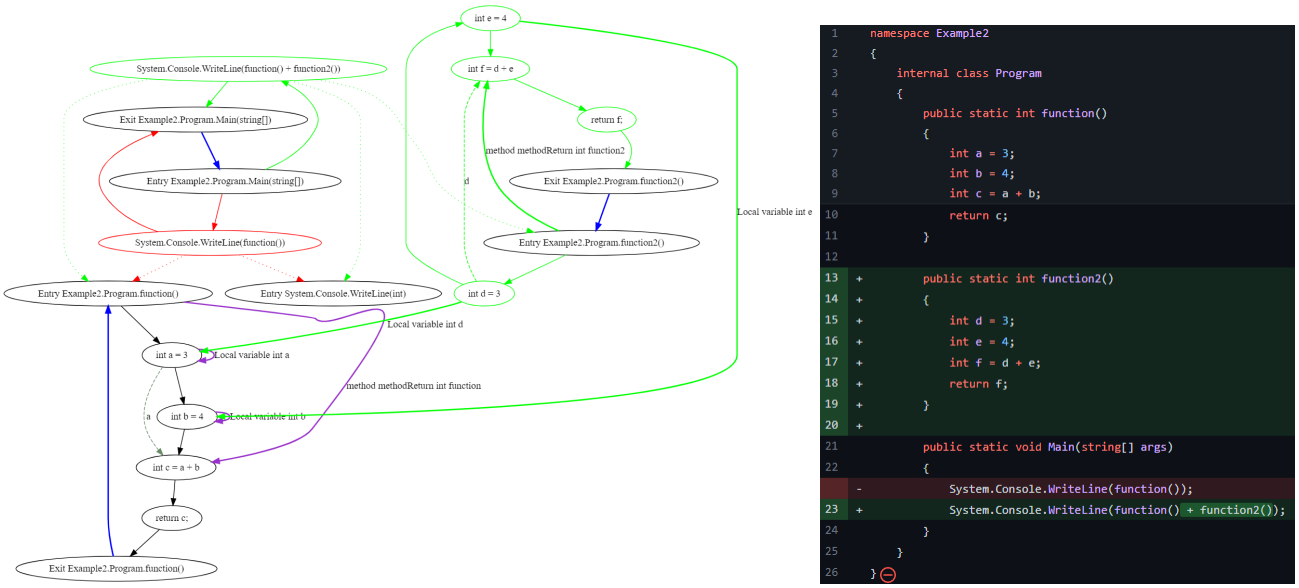
File	Line coverage
dead_code_detection.py	92%
dead_code_elimination.py	100%
dead_util.py	87%

Table 1. Line coverage of each file used for dead code elimination and detection

6 THREATS TO VALIDITY

In this section we will discuss both internal and external threats to the validity of our research.

Our implementation faces an important threat to its external validity: our testing conditions. To test our implementation, we use one repository that is self-made. This impacts the testing results, first because it is small, and second because it might not reflect the ground truth. We did not validate our implementation against real-world repositories. This is mostly because our implementation can only accurately detect dead code in the file in which that code is present: it will falsely detect dead code when it is dead in the file itself but is used in another file. Since most repositories use methods



(a) Incorrect δ-NFG generated from the code in (b)

(b) Code with the changes annotated from a commit

Fig. 4. An example of an incorrect δ-NFG

in multiple files, we could not use them. However, this could be fixed by merging all δ-NFGs of a commit into one file and performing the detection on the merged file.

Furthermore, our implementation faces a threat to its internal validity, namely the generation of the δ-NFGs. Indeed, Pârtachi et al's implementation can generate wrong δ-NFGs for some repositories. The artificially created repository we currently use for testing does not create errors. However, it generated wrong δ-NFGs for previously used repositories. An example can be found in Figure 4. In this commit, we have copied the method `function` and gave it the name `function2`. We have changed the variables `a`, `b`, and `c` to `d`, `e`, and `f` respectively. However, the δ-NFG generated wrong edges. Indeed, we can see that the bold edges with the labels "Local variable int d" and "Local variable int e" point towards the wrong nodes. Due to time constraints, we have not been able to fix issues related to the generation of the δ-NFGs.

7 CONCLUSION

We have presented a novel way to detect and eliminate dead code, using a new data structure called δ-NFG, taken from Flexeme [16].

To answer our first research question, we can conclude that δ-NFGs can be used to detect and eliminate dead code by using a breadth-first search with the control flow, and the data flow helped us find dead variables. These are properties that a standard PDG (Program Dependency Graph) also has, which brings us to our second research question. The δ-NFG's most valuable property is that it combines distinct code versions, which we have used to find revived code. Furthermore, the elimination of dead code is with fewer risks, since if the code makes a mistake and removes the wrong code, the user can simply roll back the code.

We conclude by suggesting that the δ-NFG has much potential in future projects using repositories. However, for standard dead code detection, a PDG is sufficient.

REFERENCES

[1] C.V. Alexandru, S. Panichella, S. Proksch, and H.C. Gall. 2019. Redundancy-free analysis of multi-revision software artifacts. *Empirical Software Engineering* 24 (2019), 332–380. Issue 1. https://doi.org/10.1007/s10664-018-9630-9

[2] H. Boomsma and H.-G. Gross. 2012. Dead code elimination for web systems written in PHP: Lessons learned from an industry case. *IEEE International Conference on Software Maintenance, ICSM*, 511–515. https://doi.org/10.1109/ICSM.2012.6405314

[3] D. Caivano, P. Cassieri, S. Romano, and G. Scanniello. 2021. An exploratory study on dead methods in open-source java desktop applications. *International Symposium on Empirical Software Engineering and Measurement.* https://doi.org/10.1145/3475716.3475773

[4] S.K. Dash, M. Allamanis, and E.T. Barr. 2018. RefiNym: Using names to refine types. *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 107–117. https://doi.org/10.1145/3236024.3236042

[5] J. Ferrante, K.J. Ottenstein, and J.D. Warren. 1984. *The program dependence graph and its use in optimization.* Vol. 167 LNCS. 125–132 pages. https://doi.org/10.1007/3-540-12925-1_33

[6] P. Gautam and H. Saini. 2017. Non-trivial software clone detection using program dependency graph. *International Journal of Open Source Software and Processes* 8 (2017), 1–24. Issue 2. https://doi.org/10.4018/IJOSSP.2017040101

[7] Rajiv Gupta, David A. Berson, and Jesse Z. Fang. 1997. Resource-sensitive profile-directed data flow analysis for code optimization. *Proceedings of the Annual International Symposium on Microarchitecture*, 358–368. https://doi.org/10.1109/MICRO.1997.645834

[8] Y. Higo and S. Kusumoto. 2009. Enhancing quality of code clone detection with Program Dependency Graph. *Proceedings - Working Conference on Reverse Engineering, WCRE*, 315–316. https://doi.org/10.1109/WCRE.2009.39

[9] Y. Higo and S. Kusumoto. 2011. Code clone detection on specialized PDGs with heuristics. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, 75–84. https://doi.org/10.1109/CSMR.2011.12

[10] C.M. Kamalpriya and P. Singh. 2017. Enhancing program dependency graph based clone detection using approximate subgraph matching. *IWSC 2017 - 11th IEEE International Workshop on Software Clones, co-located with SANER 2017.* https://doi.org/10.1109/IWSC.2017.7880511

[11] J. Knoop, O. Rüthing, and B. Steffen. 1994. Partial dead code elimination. *ACM SIGPLAN Notices* 29 (1994), 147–158. Issue 6. https://doi.org/10.1145/773473.

178256

[12] W. Le and S.D. Pattison. 2014. Patch verification via multiversion interprocedural control flow graphs. *Proceedings - International Conference on Software Engineering*, 1047–1058. Issue 1. https://doi.org/10.1145/2568225.2568304

[13] C. Liu, C. Chen, J. Han, and P.S. Yu. 2006. GPLAG: Detection of software plagiarism by program dependence graph analysis. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* 2006, 872–881.

[14] M. Mäntylä, J. Vanhanen, and C. Lassenius. 2003. A Taxonomy and an Initial Empirical Study of Bad Smells in Code. *IEEE International Conference on Software Maintenance, ICSM*, 381–384. https://doi.org/10.1109/icsm.2003.1235447

[15] H. Nasirloo and F. Azimzadeh. 2018. Semantic code clone detection using abstract memory states and program dependency graphs. *2018 4th International Conference on Web Research, ICWR 2018*, 19–27. https://doi.org/10.1109/ICWR.2018.8387232

[16] P.-P. Pârtachi, S.K. Dash, M. Allamanis, and E.T. Barr. 2020. Flexeme: Untangling commits using lexical flows. *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 63–74. https://doi.org/10.1145/3368089.3409693

[17] S. Romano, C. Vendome, G. Scanniello, and D. Poshyvanyk. 2016. Are unreachable methods harmful? Results from a controlled experiment. *IEEE International Conference on Program Comprehension* 2016-July. https://doi.org/10.1109/ICPC.2016.7503723

[18] S. Romano, C. Vendome, G. Scanniello, and D. Poshyvanyk. 2020. A Multi-Study Investigation into Dead Code. *IEEE Transactions on Software Engineering* 46 (2020), 71–99. Issue 1. https://doi.org/10.1109/TSE.2018.2842781

[19] X. Wang, Y. Zhang, L. Zhao, and X. Chen. 2017. Dead Code Detection Method Based on Program Slicing. *Proceedings - 2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, CyberC 2017* 2018-Janua, 155–158. https://doi.org/10.1109/CyberC.2017.69

[20] A. Yamashita and L. Moonen. 2013. Do developers care about code smells? An exploratory survey. *Proceedings - Working Conference on Reverse Engineering, WCRE*, 242–251. https://doi.org/10.1109/WCRE.2013.6671299

[21] Y. Zou, B. Ban, Y. Xue, and Y. Xu. 2020. CCGraph: a PDG-based code clone detector with approximate graph matching. *Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020*, 931–942. https://doi.org/10.1145/3324884.3416541