Automatic generation of test cases for Massive Open Online Courses (MOOCs)

João Alexandre
joaopmalexandre@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

July 2017

Abstract

Massive open online courses have recently attracted a lot of interest. An online class-room that provides special attention to each student. In order to evaluate the students, one of the assessments is made through a project implementation and its deliveries are in a network platform. Since MOOCs involve thousands of students and manual feedback is not possible, the feedback provided is based in "yes/no" . The "yes" feedback appears if a test passes, otherwise "no" appears. With this feedback, students do not get any information about their failing tests, which leads to time wasted debugging their projects.

In this work, we develop a software that provides better feedback to students in an introductory algorithms course. To create this improved feedback our material was based on test cases, inputs given to student's projects in order to test them, used in the automatic evaluation as reference. However, some of them are large and complex which causes difficulties for students to debug their projects. With graph testing techniques, we show that it is possible to reduce test cases through a process that isolates the bug. In this work, we also generate automatic test cases to work on which helps to provide good feedback.

To evaluate our software, we measure the efficiency and the effectiveness of our algorithms with a set of students projects, from an introductory algorithm's course. Experimental results show that for most projects, a small test case can be generated where the student project still fails. Moreover, in most cases, our tool is able to produce an answer in less than one minute.

Therefore, the proposed solution is able to generate personalized feedback to students in an automated framework from online learning.

Keywords: graph, graph generation, education, MOOC

1. Introduction

Massive Online Open Courses (MOOCs) [13, 10], is a known concept in the online education field. An online course is aimed at unlimited participation and open access via web. This platform was researched and developed with the goal to educate people who could not learn through physical schools or where certain subjects were not taught. Coursera[1] and Khan-Academy[2] are examples of important MOOCs with millions of users, where thousands of them learn physics, engineering, humanities, medicine, biology, social sciences, mathematics, business, computer science, digital marketing, data science, and other subjects every day. In order to evaluate the students, online learning platforms make assessments using exercises, tests, projects, etc. This concept has benefits but there are still a lot of issues to be dealt. Providing good feedback, to the assessments mentioned, is important for students to learn more about their courses which is a feature to improve in MOOCs. A good feedback also keeps the students motivated to learn and helps them through their difficulties.

Introductory algorithms courses have an important focus on graphs [4]. They can represent all sorts of networks, from the natural kingdom, to human relations, to the Internet and social networks. Problems from different fields can also be mapped to a graph problem [1, 8] and solved with the existing graph analysis techniques. Of course, as the problems become more complex, so do the graphs and the algorithms necessary to handle them. That is also why a solid understanding of elementary graph theory is necessary to every computer science student.

Debuggers are computer programs that can be used to test and debug other programs. These tools examine a program code and for every instruction

[1] `https://pt.coursera.org/`
[2] `https://pt-pt.khanacademy.org/`

analyze if there is a bug in it, displaying an error message if this happen. Debuggers are tools used by the students in MOOCs to review projects, but the feedback is extensive and not intuitive.

Given the above paragraphs, the challenge about this work is to evaluate student's projects submitted to online platforms and provide a personalized feedback for introductory algorithms courses. This can help them in their learning process and improve their projects. The goal is to help students debugging their code by having access to small and failing test cases specific for their implementation.

1.1. Motivation

Although tutors use the same definitions and general explanation to introduce a new subject, when it comes to practice, giving a better a feedback to each student can be harder. Even in a traditional classroom, faculty members are usually burdened with the task of helping, every semester, each student learning this new definitions which can lead to discouragement and lack of concern. What if an automatic tool could help teacher to provide automatic feedback exposing the students project bugs?

A graph generation tool would also be useful to students, since they could use it to verify which test cases they are failing and get a simpler version of it.

Algorithms have already been developed to isolate failure causes automatically. However, the final properties of this solutions are not facing graphs problems. What if a tutor needs a tool that can provide better feedback by isolate a graph program failure? If we could create a graph generation tool that receiving a test case where the student project fails, which will come from a online platform, we could isolate the failure by providing a minimal graph meaning that a graph reduction would lead to a passing test. With this solution we would help the students and the faculty of a course to provide and receive feedback allowing them to dedicate more time to another tasks.

1.2. Objectives

Based on software testing techniques, the purpose of this work, is to develop a set of algorithms and implementing them, capable of identifying failing tests in the students projects and automatically generate, through a manipulation and a simplification process, a simpler test case where the project still fails but the bug is exposed.

The bug is exposed in a simple way to allow that electronics deliveries, such as MOOCs delivery system, provide a better feedback. As a result, students should be able to easily understand why and where their project is failing leading to more time to improve their projects.

The software that we implemented, is specific for an introductory algorithms course.

2. Related Work

A graph is a pair (V, E) where V denotes the set of vertexes and E the set of edges. A vertex or node is the fundamental unit of which graphs are formed and an edge is a link between two vertexes. A path is a sequence of alternating nodes and edges with an origin node and an end node.A subgraph is a graph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$ of a graph $G = (V, E)$. The subgraphs complements are graphs $Gc = (Vc, Ec)$ where $Vc \cap V' = \emptyset$ and $Ec \cap E' = \emptyset$. A minimum size graph is a graph which only contains two vertexes and one edge between them. This is the smallest structure entitled graph. An adjacency list of a vertex is a set of vertexes that have an edge to this vertex.

Breadth-first search (BFS) and Depth-first search (DFS) are algorithms that consists in search a target node in a graph through a certain process. In the BFS case, it starts in a given node and explores the neighbor nodes first, before moving to the next level neighbors. In DFS case, starts in a given node but explores as far as possible along each branch before backtracking.

2.1. Models for graph generation

ErdõsRényi[11, 12] is a random graph generation model. Introduced in 1959 was one of the first graph generation models. Erdõs and Rényi name that derives from the creators names Erdõs and Rényi is composed by two models, one of them was exclusively research by this team (The $G(N, M)$ model). The second one (The $G(N, p)$ model) was also produced and researched by Gilbert[6].

Algorithm 2.1: ErdõsRényi (N, p)

```
1  V ← N;
2  E ← ∅;
3  foreach i ∈ N do
4      foreach j ← i + 1.....N do
5          chance ← random(0, 1);
6          if p > chance then
7              if graphUndirected then
8                  E ← E ∪ {(i, j)};
9              else
10                 chance ← random(0, 1);
11                 if d > Chance then
12                     E ← E ∪ {(i, j)};
13                     E ← E ∪ {(j, i)};
14                 else
15                     E ← E ∪ {(j, i)};
16                     h ← randomNode(N);
17                     E ← E ∪ {(i, h)};
18 return G = (V, E)
```

Morphing[5] is a technique to introduce structure or randomness into a wide variety of problems. Moreover, it provides a powerful tool to study topological structures like small-worlds networks. Morphing algorithm starts by receiving a graph $G = (V_g, E_g)$ and a random graph $R = (V_r, E_r)$. Given these structures, the Morphing Technique takes a fraction $1 - p$ of the substructures from G, and a fraction p of the substructures from R. In the graphs morph, called type B, the substructures are the edges and gaps (absence of edges) between nodes. To morph between two vertexes (v_1, v_2) such as $v_1 \in V_G$ and $v_2 \in V_R$, it gathers all the edges in common and then gathers a fraction $1-p$ in remaining edges from E_G and fraction p from the E_R.The Watts-Strogatz algorithm uses a rewiring method to achieve a small-world network by starting with a ring lattice. Ring lattice have edges that are increasing distance apart and are rewired. Morphing provides a much simpler and more general mechanism for this small world graphs assemble. Using a a clustered graph with large path lengths (lattice ring) and a random graph that have short path lengths but little clustering, we can simple morph it in the way described previously. One of Morphing problems is that needs to generate a random graph with some characteristics, that makes this technique dependent on other graph generation techniques.

2.2. Input Testing

Input is the act of inserting data into a computer. An input can consist in one or more variables, which have the same or different types. For each input variable corresponds a variable in the corresponding function, that is needed to make it work.

To provide an improved feedback to students, we aim to gather all the failing test inputs and perform an input simplification. Input simplification is a process to change the data imported to a function with two purposes. One is reducing the time that a function takes to compute. The second one is making an easy way to debugging. For example, a graph $G = (V, E)$ with $|V| = 50$ and $|E| = 100$ can be really hard to debug. Through an input simplification, one can reduce vertexes and edges to lower numbers where the bug still happens but is easier to analyze. Input simplification is often implemented by simple algorithms where the program decrements an input variable to check if the bug still exists. The simplification process stops when it decrements all variables to a minimal test case where the input cannot be further reduced. However, there are some algorithms that choose to study the input in order to get a smart and efficient simplification. Failure induction is a concept in a minimal test case which defines the code that makes a successful test turn into a failing one.

Delta Debugging[14, 7, 2, 3] generalizes and simplifies a failing test case into a minimal test case which means that the program still fails and moreover isolates the difference between a passing and a failing test case. This minimal test case not only allows a description of the problem and a valuable problem insight (bug exposure), but it also classifies current and future bug reports. Meaning that a future bug with the same problem, for example a wrong formulation in SCCs, will be as report the same problem description as a previous SCCs wrong formulation. The Delta debugging algorithm receives a failing test case I, which is successively tested in order to see where the test is failing. To minimize the test, this algorithm uses subsets as the main concept. First, two subsets of an input set I are created to simplify this test case. Consider two subsets A and B such that $A \subset I$, $B \subset I$ and $A \cup B = I$. Delta debugging minimizing algorithm first tests the A subset. If A test case still fails, then we have to reduce the input set I to A subset and keep doing partitions in the subset. If A passes or stays unresolved (for instance, due to a timeout), then we have to test B subset instead, doing the same thing if B test case fails. This approach is similar to "divide and conquer", which is based on recursively divide the problem into two or more subproblems, until these become simple enough to be solved directly. But what if both A and B subsets pass or stay unresolved? What Delta Debugging does in this case, is trying to get some knowledge about the nature of our input and if this knowledge is not enough, then it tests larger subsets of the main group to increase the chances that the test fails. On the other hand, it can test smaller subsets that get us a faster progression, but the chances of our test keeps failing is smaller. This algorithm has some problems related to the number of tests that needs to get a minimal test case if it does not have a good knowledge about the nature of the input.

In Fig. 1[14], the minimizing delta debugging algorithm (ddmin) is described. As previously mentioned, this algorithm is based in test subset. When neither Δ_1 or Δ_2 are conclusive, the algorithm uses granularity as a tool to keep testing. As previously mentioned, ddmin deals with inconclusive subsets by testing larger or smaller subsets. Therefore, granularity is a unit that defines how many subsets a test should have to test. When both subsets keep passing, the granularity is increased.

Fig.2[14] illustrates the use of the algorithm presented in Fig. 1 using the granularity when subsets are inconclusive. The illustration shows that a input set starts by being divided in two subsets Δ_1 and Δ_2, where $\Delta_1 = \{1, 2, 3, 4\}$ and $\Delta_2 = \{5, 6, 7, 8\}$. Testing these two subsets, the algorithm did not find a failing one. The granularity

Minimizing Delta Debugging Algorithm

Let *test* and $c_{\mathbf{X}}$ be given such that $test(\emptyset) = \checkmark \wedge test(c_{\mathbf{X}}) = \mathbf{X}$ hold.
The goal is to find $c'_{\mathbf{X}} = ddmin(c_{\mathbf{X}})$ such that $c'_{\mathbf{X}} \subseteq c_{\mathbf{X}}$, $test(c'_{\mathbf{X}}) = \mathbf{X}$, and $c'_{\mathbf{X}}$ is 1-minimal.
The *minimizing Delta Debugging algorithm* $ddmin(c)$ is

$$ddmin(c_{\mathbf{X}}) = ddmin_2(c_{\mathbf{X}}, 2) \quad \text{where}$$

$$ddmin_2(c'_{\mathbf{X}}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(\Delta_i) = \mathbf{X} \text{ (“reduce to subset”)} \\ ddmin_2(\nabla_i, \max(n-1, 2)) & \text{else if } \exists i \in \{1, \ldots, n\} \cdot test(\nabla_i) = \mathbf{X} \text{ (“reduce to complement”)} \\ ddmin_2(c'_{\mathbf{X}}, \min(|c'_{\mathbf{X}}|, 2n)) & \text{else if } n < |c'_{\mathbf{X}}| \text{ (“increase granularity”)} \\ c'_{\mathbf{X}} & \text{otherwise (“done”).} \end{cases}$$

where $\nabla_i = c'_{\mathbf{X}} - \Delta_i$, $c'_{\mathbf{X}} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$, all Δ_i are pairwise disjoint, and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{\mathbf{X}}|/n$ holds.
The recursion invariant (and thus precondition) for $ddmin_2$ is $test(c'_{\mathbf{X}}) = \mathbf{X} \wedge n \leq |c'_{\mathbf{X}}|$.

Figure 1: Minimizing Delta Debugging Algorithm.

Step	Test case								*test*	
1	$\Delta_1 = \nabla_2$	1	2	3	4	.	.	.	.	?	Testing Δ_1, Δ_2
2	$\Delta_2 = \nabla_1$	.	.	.	.	5	6	7	8	?	⇒ Increase granularity
3	Δ_1	1	2	.	.	.	.	.	.	?	Testing $\Delta_1, \ldots, \Delta_4$
4	Δ_2	.	.	3	4	.	.	.	.	✓	
5	Δ_3	.	.	.	.	5	6	.	.	✓	
6	Δ_4	.	.	.	.	.	.	7	8	?	
7	∇_1	.	.	3	4	5	6	7	8	?	Testing complements
8	∇_2	1	2	.	.	5	6	7	8	✗	⇒ Reduce to $c'_{\mathbf{X}} = \nabla_2$; continue with $n = 3$
9	Δ_1	1	2	.	.	.	.	.	.	?*	Testing $\Delta_1, \Delta_2, \Delta_3$
10	Δ_2	.	.	.	.	5	6	.	.	✓*	* same *test* carried out in an earlier step
11	Δ_3	.	.	.	.	.	.	7	8	?*	
12	∇_1	.	.	.	.	5	6	7	8	?	Testing complements
13	∇_2	1	2	.	.	.	.	7	8	✗	⇒ Reduce to $c'_{\mathbf{X}} = \nabla_2$; continue with $n = 2$
14	$\Delta_1 = \nabla_2$	1	2	.	.	.	.	.	.	?*	Testing Δ_1, Δ_2
15	$\Delta_2 = \nabla_1$	.	.	.	.	.	.	7	8	?*	⇒ Increase granularity
16	Δ_1	1	.	.	.	.	.	.	.	?	Testing $\Delta_1, \ldots, \Delta_4$
17	Δ_2	.	2	.	.	.	.	.	.	✓	
18	Δ_3	.	.	.	.	.	.	7	.	?	
19	Δ_4	.	.	.	.	.	.	.	8	?	
20	∇_1	.	2	.	.	.	.	7	8	?	Testing complements
21	∇_2	1	.	.	.	.	.	7	8	✗	⇒ Reduce to $c'_{\mathbf{X}} = \nabla_2$; continue with $n = 3$
22	Δ_1	1	.	.	.	.	.	.	.	?*	Testing $\Delta_1, \ldots, \Delta_3$
23	Δ_2	.	.	.	.	.	.	7	.	?*	
24	Δ_3	.	.	.	.	.	.	.	8	?*	
25	∇_1	.	.	.	.	.	.	7	8	?	Testing complements
26	∇_2	1	.	.	.	.	.	.	8	?	
27	∇_3	1	.	.	.	.	.	7	.	?	Done
Result		1	.	.	.	.	.	7	8		

Figure 2: Minimizing Delta Debugging test case.

is increased, $n = 4$, therefore Δ_1 and Δ_2 are divided. Having now four subsets where $\Delta_1 = \{1, 2\}$, $\Delta_2 = \{3, 4\}$, $\Delta_3 = \{5, 6\}$ and $\Delta_4 = \{7, 8\}$, the process is repeated being that the complements are also tested. The complement $\nabla_2 = \{1, 2, 5, 6, 7, 8\}$ fails so delta debugging stops testing the other following complements. The test input is reduced to it and the granularity $n = 3$. Now that ∇_2 is the input, the simplification proceeds and three subsets are created $\Delta_1 = \{1, 2\}$, $\Delta_2 = \{5, 6\}$ and $\Delta_3 = \{7, 8\}$. Next, the tests in these subsets and its corresponding complements are executed. Considering that $\nabla_2 = \{1, , 2, 7, 8\}$ fails, the input is again reduced to a failing subset complement and granularity $n = 2$. When granularity is equal 2, the subsets are only two. $\Delta_1 = \{1, 2\}$, $\Delta_2 = \{7, 8\}$ are tried but since none fails, the granularity is increased $n = 4$. Both Δ_1 and Δ_2 are divided in two subsets, where now $\Delta_1 = \{1, \}$, $\Delta_2 = \{2\}$, $\Delta_3 = \{7\}$ and $\Delta_4 = \{8\}$. The complement of Δ_2 , $\nabla_2 = \{1, 7, 8\}$, fails. Finally the input is reduced to this complement and granularity $n = 3$. The process is repeated, being tried subsets $\Delta_1 = \{1, \}$, $\Delta_2 = \{8\}$, $\Delta_3 = \{8\}$ and the complements $\nabla_1 = \{7, 8\}$, $\nabla_2 = \{1, 8\}$, $\nabla_3 = \{1, 7\}$. Neither subsets or complements fails, so being impossible to reduce more the subsets, the input simplification is done and the final result is $I = \{1, 7, 8\}$.

3. Software

This software integrates the course of Analysis and Synthesis of Algorithms (ASA) from the degree on information systems and computer engineering at Instituto Superior Tcnico (IST). ASA subject focus on the development of algorithms using graphs. The course projects are related to graphs algorithms which is suitable for our software.

3.1. Software Architecture

In our thesis proposal we proposed to implement a software composed by two major algorithms, Delta Debugging and Morphing. As it was described in the related work, Delta Debugging use a series of testing, subset and their complements, to reduce an input sample, in our case related to graphs. On the other hand, Morphing is meant to be applied when Delta Debugging cannot reduce the sample to a minimal case test, rewiring the graph in order to give Delta Debugging another reduction attempt.

When implementing our software, we created an architecture with the components illustrated in Fig.3.

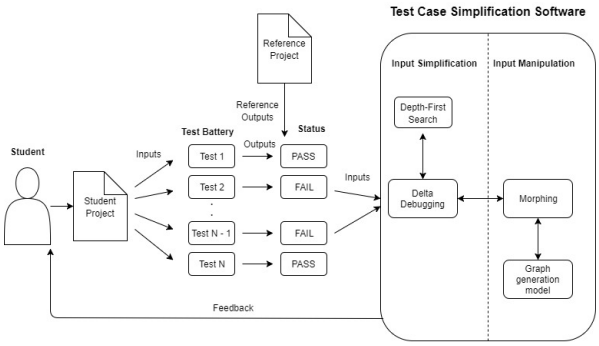

Figure 3: Software Architecture Components.

In order to build the software, we had to know the users who would use it. Since this software is directed to graphs, we choose to use students from Analysis and Synthesis of Algorithms course to be our users. This course subject receives a student project and test it with a test battery. This test battery is composed by N tests and an online platform, Mooshak[9], runs the student's project for every single one of them. Each test is an input file that contains a graph information (vertexes, edges and root vertex) with specific characteristics made by the course faculty in order to test student's projects. These test cases are sorted by size and when a test fails, our software saves the failing test to be used as argument for our program. To check a test status our software uses a reference project, created by the course faculty, that also receives the test cases and produces reference outputs. With these output files and student output files, a test passes if the difference between them is a blank file and fails

if there are differences between the outputs which means that the difference file will not be empty.

Our software receives the failing test case and uses Delta Debugging as a test case simplification algorithm. In order to reduce a test case, Delta Debugging needs an auxiliary algorithm to divide the graph. We decided to use Depth-First Search algorithm in order to produce not only the subgraphs but their complementaries as well. So as the figure 3 represents, Delta Debugging uses Depth-First Search to divide the graph, then it tries to reduce the graph to one of its subgraphs if they still fail in the student project. Making it cyclic until we reach a minimal test case.

But what happens when we cannot reduce a test case to a minimum size test? For those cases, we introduce two more algorithms in our architecture. The Morphing algorithm receives the graph that cannot be reduced and a random graph created with the same number of vertexes as the original graph. Giving this two graphs, Morphing produces a morphed graph using an edge rewire, based on probabilities. This morphed graph should be similar to the original graph so that students receive feedback from it and by fixing a bug in their projects pass the test case that contains the original graph. However, to accomplish this algorithm, we need an auxiliary algorithm to produce a random graph. So another architecture component is Graph Generation Model that we describe, in the next subsection.

3.2. Software Components

Starting the Delta Debugging algorithm we need to divide the initial test case graph into two subgraphs, resulting in two test cases. Using Depth-First Search (DFS) algorithm we can accomplish that. Knowing that DFS algorithm searches unvisited vertexes until all of them are visited, we assure that there is not a vertex that remains unvisited. DFS algorithm searches in depth, this means that it always tries to visit the first vertex in the adjacency list belonging to the vertex being currently visited. This algorithm starts searching at some root vertex and cyclically does a depth search among all vertexes. This algorithm was smoothly modified by us. Since we needed to divide the graph in a certain number of subgraphs, a granularity variable was created in order to manage the graph division. For example, if a graph has $|V| = 24$ and $granularity = 2$, the graph should be divided into two subgraphs each with $|V| = 12$, except for special cases. These special cases happen when the DFS reaches a point where all its visited vertexes do not have unvisited vertexes in their adjacency list. In these cases what happens is that more than two subgraphs are generated for $granularity = 2$. Algorithm 3.1 illustrates the pseudo-code of our Delta Debugging procedure.

Algorithm 3.1: Delta Debugging
$DeltaDeb(G = (V, E), granularity)$

```
1  if |V|/granularity ≤ 2 then
2      return G
3  else
4      SG ← DFS(G, granularity);
5      foreach sg ∈ SG do
6          if checktest(sg) = FAIL then
7              if isMinimal(sg) then
8                  return sg;
9              else
10                 if granularity > 2 then
11                     granularity ←
                         granularity − 1;
12                 return
                     DeltaDeb(sg, granularity);
13     granularity ← granularity + 1;
14     return Deltadeb(G, granularity);
```

As you can see, in the pseudo-code, there is a condition (line 1) before the DFS function call. This condition is needed in order to avoid the excess of testing in our algorithm, which will be mention further in this subsection. Receiving a graph and the granularity, the DFS returns a list of subgraphs SG (line 4). What Delta Debugging algorithm does is test the new test cases (the subgraphs in SG) where if there is not a difference between the output obtained by the student project and the reference project, that means that the test case passes the test and we need to continue to other test cases. If there is a difference, the test fails. This procedure is represented in the pseudo-code in the function $checktest(sg)$ (line 6), which checks the outputs from the reference project and the student project through a certain case test, sg. If the test fails, we reduce our original test case to this failing test case making Delta Debugging algorithm recursive (line 12) until we reach out a test that cannot further be reduced. The $isMinimal$ (line 7) function checks if the test case is a minimal test case. Since Delta Debugging reaches a minimal test case then this is the test case that we send to the student (line 8). But what if all the test cases pass? In this situation, our program increases the granularity (line 13) and calls out recursively the Delta Debugging algorithm (line 14). As a result, the DFS will divide the graph in more subgraphs. Therefore, if the granularity was 2 it will increase to 3 and the original graph will be divided into three subgraphs. Moreover, if a test fails and $granularity > 2$, then

the granularity is decremented (line 11) in order to reduce the number of generated subgraphs.

We chose Depth-First Search algorithm as an auxiliary algorithm to Delta Debugging and use it to identify subgraphs. This algorithm is modified to reach our goal that is get subgraphs instead of only doing a search through a graph as a typical DFS does. In algorithm 3.2, we receive a graph, $G = (V, E)$ and a granularity. In the beginning all vertexes are marked unvisited (line 2-3). The algorithm finds the first unvisited vertex and calls a function $Visit$ (line 6) that starts a graph transversal visiting vertexes recursively and returns a list of visisted vertexes (See Algorithm 3.3).

Algorithm 3.2: Depth-First Search $DFS(G = (V, E), granularity)$

```
1  SG ← ∅;
2  foreach v ∈ V do
3      color[v] ← white;
4  foreach v ∈ V do
5      if color[v] = white then
6          vertexes ←
             Visit(v, (|V|/granularity), ∅);
7          sg ← (vertexes, {(u, v) ∈ E : u ∈
             vertexes ∧ v ∈ vertexes)});
8          SG ← SG ∪ {sg};
9  return SG
```

Algorithm 3.3: Depth-First Search Auxiliary $Visit(v, maxVisit, vertexes)$

```
1  color[v] ← black;
2  vertexes ← vertexes ∪ {v};
3  if |vertexes| = maxVisit then
4      return vertexes
5  foreach u ∈ adj[v] do
6      if color[u] = white then
7          vertexes ←
             Visit(u, maxVisit, vertexes);
8          if |vertexes| = maxVisit then
9              return vertexes
10 return vertexes
```

In order to increase the chance of getting a failing test and have a better test coverage, we decided to test the subgraph complements as well. First, we tried to test the complement by using the vertexes from the original graph that were not in the subgraph but this can result in disconnected graphs[3]. Being this a problem, we had to separate this complement graph into more than one graph, meaning that instead of having a disconnected graph, we would have two or more connected graphs. This solution was achieve by using a modified DFS to find the complementary graphs. DFS was modified to start with a certain number of vertexes already visited (vertexes from the subgraph). Thus, only vertexes that not belong to the subgraph will be visited and we will get our complement graphs. The complement function, $complement(G, sg)$, receives the original graph and a subgraph obtained with DFS. Like in DFS, it searches the unvisited vertexes and keeps doing it recursively returning one or more complements of the subgraph received.

Since the number of generated subgraphs can be more than two, even if the $granularity = 2$, this leads to complements that have almost the same size as the original test case. This results in taking more time to test a buggy student project. For example, in a test case with 10000 vertexes, using the DFS to return the subgraphs can lead to dozens of tiny subgraphs with 3 to 20 vertexes which will lead to complements with a number of vertexes similar to the original test case. This can lead to a sequential low reduction of vertexes in failing test cases. Therefore, to avoid this situation, we start by applying a condition when calling the complement DFS, $complement(G, sg)$. The condition that avoids these situations is when the $((|V|/granularity) * 0.1) > |S|$, where $|V|$ is the number of vertexes of the original graph and the $|S|$ is the number of vertexes of the subgraph. So if $|V| = 10000$ and $granularity = 2$ means that for subgraphs with $|S| < 500$ no complements will be tested. With this, we can avoid the redundancy of reducing a test case to a test case slightly smaller. This technique also allows to only test large size complements when the granularity is higher. Hence, if the original test case cannot be reduced to a smaller graph, eventually we will be testing these complements to reduce the graph to a smaller one even if the difference is not that large.

Our software can reduce test cases if they keep failing and reach a minimal test case. But there are situations where applying Delta Debugging does not result in finding a failing test case. At a certain point, granularity will reach a value where the number of vertexes $|V|$ divided by the granularity will only provide minimum size graphs to test. This occurs when $|V|/granularity < 2$. Delta Debugging will only work with graphs that have two vertexes and one edge or single vertexes at this point. Since we solve our cache problem by ignoring test cases with the same number of vertexes and number of

[3]We assume each graph must be connected

edges as the ones that are in our already tested list, our Delta Debugging at this point returns the simplest failing test case that it found.

However, our objective is to reduce test cases and maximize the number of test cases that can be reduced through our software. As a result, we have implemented other algorithms that will help us out maximizing this number. Morphing is an algorithm used to smoothly mix two things that are in the same context, creating one with similarities from both but a little different from the original ones. In our software context we use this technique to create a morphed graph. When our original graph cannot be reduced to a minimal case test, we try to slightly modify the original graph by creating a random graph, with an auxiliary algorithm graph generation, and mixing it with the original graph.

First we have to create a random graph. In our software, we use ErdõsRényi algorithm for that purpose. ErdõsRényi algorithm receives a number of vertexes and randomly assigns edges between them. Since we need to control the number of edges created so that this random graph does not have a huge amount of edges and keeps similar to the original graph, we consider the graph density. The graph density is calculated as follows: $graphdensity = (|E|/((|V| * (|V| - 1))/2))$ which give us a number between 0 and 1. As mentioned in section 2, the ErdõsRényi pseudo-code (Algorithm 2.1) illustrate how we can create a random directed or undirected graph. Since our software is dedicated to undirected graphs, in the algorithm 3.4 we show the pseudo-code for the undirected graph creation, hiding the directed part from the other pseudo-code. As previously mentioned, before we apply the ErdõsRényi algorithm, we calculate the graph density and that defines the variable p. With the number of vertexes N and the graph density p we can create a random graph in a way that for each possible edge between two vertexes, we get a random value from a random value generator (line 5).

Algorithm 3.4: ErdõsRényi (N, p)

```
1 V ← N;
2 E ← ∅;
3 foreach i ∈ N do
4 |   foreach j ← i + 1.....N do
5 |   |   chance ← random(0, 1);
6 |   |   if p > chance then
7 |   |   |   E ← E ∪ {(i, j)};
8 return G = (V, E)
```

Having also a random value, if the graph density is higher than this random value, then the ErdõsRényi algorithm adds an edge to the new random graph (lines 6-7). Otherwise, the edge is not created. After verifying this procedure for all the possible edges between the vertexes (line 3-4), we obtain a random graph which has a number of edges similar to our original graph but that does not take in consideration the edges that the original graph has. We use this simple algorithm as a creation tool of a graph but there is several alternative algorithms than can be introduced here as a random graph generator. The reason that we chose ErdõsRényi algorithm is that we wanted a simple and easy way to implement random graph generator which we did not have to specify any characteristics for it to work, since our graph restriction are only about having the same number of vertexes as the original graph.

Algorithm 3.5: Morphing $(G_o = (V_o, E_o), G_r = (V_r, E_r), p)$

```
1  V ← V_o;
2  E ← ∅;
3  foreach (u, v) ∈ E_o do
4  |   chance ← random(0, 1);
5  |   if (u, v) ∈ E_r then
6  |   |   E = E ∪ {(u, v)};
7  |   else
8  |   |   if chance > p then
9  |   |   |   E = E ∪ {(u, v)};
10 foreach (u, v) ∈ E_r do
11 |   chance ← random(0, 1);
12 |   if (u, v) ∉ E_o ∧ chance < p then
13 |   |   E = E ∪ {(u, v)};
14 return G = (V, E)
```

Now that we have our random graph, we use Morphing to rewire the edges of the original graph taking into account the random graph edges. Receiving these two graphs (an original graph G_o and a random graph G_r) and a variable p with a number that we define. The edge rewire process can start. For every edge in our original graph we verify if we can add an edge between them or not. For this to happen, two things must occur. First, the random graph $(u, v) \in E_r$ also has this edge between the vertexes being analyzed. If the random graph contains it, so we straightly add an edge since all the common edges between the G_r and the G_o will be on the morphed graph. If there is not the edge on the G_r, we make use of the probability variable *chance* (line 4). This variable can have values between 0 and 100. Its function is to guarantee that the morphed graph results similar or different from the G_o. If we choose to put, as input, a lower number in p, it should be similar to the morphed graph but it can also look exactly the same and we do not want that

to happen. If we put just assign a higher value, this can lead to a problem where the morphed graph is so different from the G_o, that reducing the test case will not help the student solving his problem with the original case test. Therefore, we defined p equal to 10 if the graph has more than 100 vertexes and equal to 25 otherwise. With this variable p and a random value *chance*, if the random value is higher than the probability then we create the edge even if the G_r does not have it. Since that probability has a low value, it should be more likely that the edge will is added. Next, we should move on to analyze the G_r and its edges. Given that all the edges existing in both graphs were added in the morphed graph, we do not need to check this again. Hence, we only have to verify if a vertex does not exist in the original graph $(u, v) \notin E_o$ and if that happens, then we check if $chance < p$ (line 12). This is more unlikely to happen since the random value has to be lower than p which has a low value. But if this happens anyway, then we add a new edge between the vertexes being analyzed. These are minor modifications that let the morphed graph slightly similar to the G_o.

Finally, when we have the morphed graph, we should send it to the test case simplification algorithms to try a new reduction in the test case.

4. Results

We started by checking how many student projects had bugs. Projects without bugs were not included in our subject sample. This is achieved through a script created to verify if a student project fails in our test battery. If it fails, then we can use our software to reduce the failing test case into a minimal one. In the following graphs and tables, we show these general results. In table 1 we show the sample that is used to test our software, which includes the number of students projects that have a failing test case and the number of student projects that our software was able to reduce the failing test case into a minimal one.

Student Projects with a failing test case	618
Student Projects with a minimal test case	570

Table 1: Software testing sample.

Through intensive testing of our software, we reach some results about its effectiveness and efficiency. Our total of student projects submitted to testing were 618. With these submissions, we applied our software to each one of them. We produced a minimal test case to most of the submissions. However, in some cases, the reduction was smaller than expected. In other situations, we also

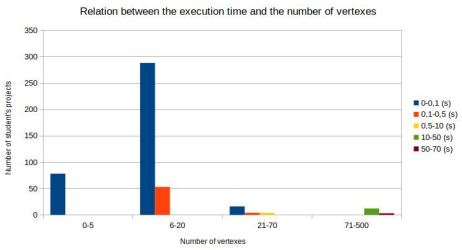

Figure 4: Relation between the students projects execution time and its number of vertexes. (G1)

sent some submissions that contained a graph with the same number of vertexes as the original failing test case. We see this in more detail in the test case simplification section through the vertexes reduction that we have as results. Even when the minimal test case is not small enough to a student write on a paper, the student project still contains a bug with that minimal test case as input which can be used to some debug measures.

In order to analyze the effectiveness of our software, we made a comparison between the number of submissions which have a failing test case and the number of submissions that have a minimal test case which was sent to a student. These results shows that our software goal was achieve by helping out the students with their projects and providing to them a simpler test case which they can analyze and fix the problem. The results are presented in the table 1. As we can see, they were positive in a way that in 618 student projects with a failing test case, 570 submissions contain a minimal test case that was send to a student. This reveals that our software had a 92,23% of success reducing a graph, meaning that this is our software effectiveness towards this sample.

Another general criteria that we should test is the efficiency of our software, meaning that we have to analyze the resources consumed to reduce the failing test cases into a minimal test case. Using a script that writes on a file the execution time of our software for a given failing test case from a student project, we relate the time with the graph size (number of vertexes). With this relation we will be able to see if the graph size is directly proportional to the execution time of our software since that a higher number of vertexes could mean that more graphs are tested.

In figures 4 and 5, we can see that the execution time is higher for larger graphs. The reason is that it relies on this is because the number of testing done by Delta Debugging is a higher for test cases with higher number of vertexes. For example, a graph with 10 vertexes and 15 edges is easy to test the subgraphs and the complementary subgraphs

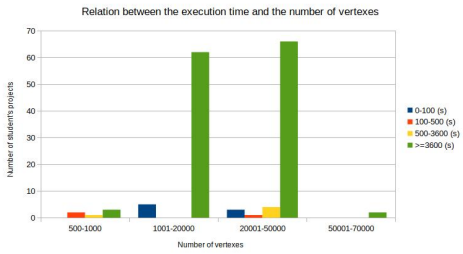

Figure 5: Relation between the students projects execution time and its number of vertexes. (G2)

because even though we cannot find a smaller graph that fails in the student project, the granularity is increasing by $2 * granularity$ and reachs a point where granularity can no longer be increased, meaning that the granularity is equal to the number of vertexes divided by the granularity is equal or higher than two, $\|V\|/granularity > 2$. And even when the student project takes some time to provide output for each test, the number of graphs tested is so low that it is almost irrelevant. So, with a low number of vertexes, the granularity reach that condition quickly which leads to a faster execution of our application. On the other hand, a graph with 5000 vertexes and 15000 edges, will take a lot more time to run the subgraphs and complementary subgraphs, when the granularity is 100, this means that there will be 100 subgraphs or more and a lot of complements. This is one of the reasons that the execution time of our software takes longer. Another reason is that a student graph can be very buggy and takes some time to run a certain test case (subgraph), which even with our time limit (two seconds) will make our software slower. For example, 100 subgraphs in an inefficient student project can take 200 seconds. Finally, one reason that we consider and that makes our software more inefficient is the time that a huge graph takes to be randomized and morphed. Seeing that in the morphing process, we have to search thousands of edges, this process makes our software slower than excepted.

5. Conclusions

Our goal was to reduced failing test cases for a series of student project submissions in a platform. Since that a student submitted his project in Mooshak (an online platform), we should check the first failing test case in that submission and with the student project, the failing test case and a reference project, test the student project for smaller graphs, through Delta Debugging algorithm and check if the test remains as a failing test case reaching a minimal test case after doing this cyclic. After testing this functionality and present the results in section 4, we concluded that for the majority of the student projects we were able to provide a minimal test case which have a considerable size of analyze for testing it through a paper and a pencil and easily find the project bug. We also concluded that for bigger graphs, TCSS takes more time to reduce it and for most of the times we only can reach a minimal test case that still have a lot of vertexes and edges and which sometimes has a similar size than a passing test case. For example, in a student project that has a failing test case with 500 vertexes and 5000 edges, we found out that some of these cases get a minimal test case with 100-300 edges and that have a similar graph size to a passing test case that has 70 vertexes which leads to a conclusion that is very unlikely to discover a graph with less than 70 vertexes that keeps failing in the student project. But even this case is possible to happen, because a failure can be related to the number of edges and the paths that exist in a graph. The minimal test case of a failing test case will help the student in different ways. It helps the student finding his project bug given that for smaller minimal graphs they can just draw the graph in a paper and do a manual iteration of their code checking where is the bug. However, for bigger minimal graphs, the students can also do some prints in their console to analyze easily where is the bug and if it is related to wrong path analysis, since that the student project search the shortest path between two vertexes or if it has something to do with memory allocation/memory access. Considering the previous example, if the original test case had 500 vertexes and TCSS could only find a minimal test case which had 101 vertexes, this could mean that the student have a structure with a maximum 100 positions for vertexes and with more vertexes leads to a failing test case. With this kind of feedback the students are allowed to receive a test case even that the minimal one is not that much smaller than the original test case, but at least gives him the an idea on which type of graph he is failing. This would not happen if the original test case could not be reduced (since we cannot send an original test case to a student).

In a more external perspective, we conclude that TCSS is a good tool for providing feedback to an user that is doing a project involving undirected graphs and has the need to test his application. For the environment that TCSS were used it provides a more detailed feedback for students that aim to pass a certain test in the test battery provided by the course faculty. It will help students to understand their project bugs and allow them to understand the course lessons and the graphs notions involved. With this an advantage is brought, this advantage is that students improved their debugging skills such as manual debugging (paper and pencil) and program variables debugging (asserts, prints,

breakpoints, etc). For a faculty point of view, this tool makes them not waste time with several emails that students send to them hoping to get feedback. Also helps them to provide feedback to a student since that they can now deliver minimal test cases to students which has forbidden or hard to tell only by knowing the test case that the student was failing and did not having the option to send the original test case.

Finally we concluded that TCSS was a success since we have a high rate of test cases reduction, 92,23%, that was our goal and with this we can help the students in getting a better feedback about their bugs.

5.1. Future Work

As previously mentioned, TCSS is only functional for test cases that contain an undirected graph which these test cases have to be a specific order. First line the number of vertexes and the number of edges, second line for the root vertex and then one line per each edge which contains the two vertexes which this edge links. For future work, we though that it would be good to make some changes on TCSS to make it functional to test cases that contains a directed graph with the same specific input order. TCSS at the moment, helps the students to receive a minimal graph but only for undirected graphs projects, with some modifications as said before this software should be able to work for directed graphs as well. As a reminder the changes should include a modification in the file reader since it should only write only add the destination vertex in the adjacency list of the origin vertex and not in both adjacency lists. Also, our function that inserts an edge in a graph should be changed to only insert the destination vertex in the adjacency list of the origin vertex. This should be modified both for the algorithm that produces a random graph and our Morphing algorithm. Not only directed graphs can easily be functional in this software but also weighted graphs can be adapted to get a minimal graph through TCSS. To work around this, we should modify the same things as the directed graphs but in a way that a edge now has a weight which means that if the weighted graph is undirected we should include in the read file and the insert edge function an weight associated with the adjacency list. For example for an edge 1 5 with origin vertex 1 and the destination node 5, if the graph is undirected TCSS should need to include the 5 in the adjacency list of the origin vertex 1 and the vertex 1 in the adjacency list of the vertex 5 but this adjacency lists should also be modified so that instead of a list of integers, it could associate a weight with the vertex. For example a pair of integers where the first is the vertex and the second is the weight to move from the origin vertex to that vertex should be working. Of course that also the algorithm that creates a random graph and morphing had to be modified in order to create an weighted random or morphed graph. As you can see the differences from an undirected and a directed weighted graphs, it would be the same as the undirected and directed graphs in a way that only would affect the way that the edges are inserted in a graph. For Delta Debugging since it only divide the graph, with Depth-First Search algorithm, in a certain number of subgraphs and test them as well as the complementary subgraphs nothing would have to change because a search for divide the graph is equal in the different graphs mentioned. Also, TCSS is modular, facilitating the removal and addition of new algorithms. For different cases with different types of graphs, we can change algorithms such as graph searching, random graph generator or graph edges rewire for algorithm that could adjust better for certain situations.

References

[1] B. Bollobás. *Random graphs*. Springer, 1998.

[2] R. Brummayer and A. Biere. Fuzzing and delta-debugging smt solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pages 1–5. ACM, 2009.

[3] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th international conference on Software engineering*, pages 342–351. ACM, 2005.

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

[5] I. P. Gent, H. H. Hoos, P. Prosser, and T. Walsh. Morphing: Combining structure and randomness. *Aaai/Iaai*, 99:654–660, 1999.

[6] E. N. Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, 1959.

[7] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 263–272. ACM, 2005.

[8] S. Janson, T. Luczak, and A. Rucinski. *Random graphs*, volume 45. John Wiley & Sons, 2011.

[9] J. P. Leal and F. Silva. Mooshak: A web-based multi-site programming contest system. *Software: Practice and Experience*, 33(6):567–581, 2003.

[10] T. R. Liyanagunawardena, A. A. Adams, and S. A. Williams. Moocs: A systematic study of the published literature 2008-2012. *The International Review of Research in Open and Distributed Learning*, 14(3):202–227, 2013.

[11] B. Prettejohn, M. Berryman, and M. McDonnell. Methods for generating complex networks with selected structural properties for simulations: a review and tutorial for neuroscientists. 2011.

[12] P. Révész. *Random walk in random and non-random environments*. World Scientific, 2005.

[13] L. Yuan, S. Powell, J. CETIS, et al. Moocs and open education: Implications for higher education, 2013.

[14] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.