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

João Pedro Mateus Alexandre

Thesis to obtain the Master of Science Degree in Information Systems and Computer Engineering

Supervisor: Professor Vasco Miguel Gomes Nunes Manquinho

Examination Committee

Chairperson: Professor Luís Antunes Veiga
Supervisor: Professor Vasco Miguel Gomes Nunes Manquinho
Member of the Committee: Professor Luís Manuel Silveira Russo

July 2017
I would like to thank my Thesis supervisor, Vasco Manquinho for the opportunity to work on such an interesting project such as this one and for all the guidance provided throughout the whole process. I would also like to thank my family, specially my parents, for all the support given, not only through this year but also throughout my life. And finally I would like to thank my friends whose company and support managed to keep me sane and in a good mood no matter what.
Acknowledgments

This work was partially supported by national funds through FCT with reference UID/CEC/50021/2013
Resumo

Cursos Online Abertos e Massivos (MOOCs) têm despertado um grande interesse recentemente. Uma sala de aula online que fornece especial atenção a cada aluno. Implementações de projectos são umas das formas de avaliar os alunos e as entregas desses projectos são feitas numa plataforma online. Visto que as MOOCs envolvem milhares de alunos e que um feedback manual não é possível, o feedback é fornecido à base do "yes/no". O feedback "yes" aparece se o teste é passado, caso contrário aparece "no". Com este feedback, os alunos não retiram informação necessária à cerca dos seus testes falhados, o que leva a um desperdício de tempo ao analizar os bugs.

Neste trabalho desenvolvemos um software que fornece um melhor feedback aos alunos de uma cadeira de introdução de algoritmos. Para criar este feedback melhorado, o nosso material é baseado em casos de teste, inputs dados aos projectos dos alunos de modo a testá-los, usados numa avaliação automática. Mas alguns destes casos de teste são grandes e complexos o que causa dificuldades aos alunos que desejam dar debug nos seus projectos. Com técnicas de teste a grafos, nós mostramos que é possível reduzir casos de teste através de um processo que isola o bug. Também neste trabalho, iremos gerar automaticamente grafos que ajudam a fornecer um melhor feedback.

O primeiro algoritmo, relacionado com teste de inputs, reduz um caso de teste inicial para outro, o mais pequeno possível, onde o projecto do aluno continue a falhar.

O segundo algoritmo, relacionado com reordenação de grafos, é um algoritmo auxiliar ao primeiro. Este algoritmo utiliza um processo de reordenação de arestas para criar um novo grafo e através disso tentar uma redução do grafo criado utilizando o primeiro algoritmo.

Finalmente, para avaliar o nosso software, nós medimos a eficiência e a eficácia dos nossos algoritmos com uma amostra de projectos de alunos, de um curso de introdução aos algoritmos. Testando se estes projectos falham em algum caso de teste, qual o caso que falha e se consegue ser reduzido para um caso de teste que contenha um grafo mais pequeno do qual o projecto do aluno continua a falhar. Os resultados foram de certa forma positivos e permitiu-nos fornecer um feedback específico para cada aluno. Os resultados experimentais mostram que para a maioria dos projectos, um caso de teste curto, onde o projecto do aluno ainda falha, pode ser gerado. Além disso, na maioria dos casos, a nossa ferramenta é capaz de produzir uma resposta em menos de um minuto.

Assim sendo, a solução proposta é capaz de gerar um feedback personalizado para os alunos, numa framework automática para aprendizagem online.

Palavras-chave: algoritmos, grafo, geração de grafos, educação, MOOC
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.

The first algorithm, related with input testing, reduces an initial test case to a smaller one where the student project still fails.

The second algorithm, related to graph rewire, is an auxiliary algorithm to the first one. It uses rewire process to create a new graph in order to try a reduction on it with our first algorithm.

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: algorithms, graph, graph generation, education, MOOC
Contents

Acknowledgments .......................................................... v
Resumo ........................................................................ vii
Abstract ......................................................................... ix
List of Algorithms .......................................................... xiii
List of Figures .................................................................. xvi
List of Tables .................................................................... xvii

1 Introduction ................................................................. 1
  1.1 Motivation ............................................................... 2
  1.2 Objectives ............................................................... 3
  1.3 Structure ................................................................. 3

2 Preliminaries and Related Work ........................................ 5
  2.1 Graph generation ...................................................... 5
    2.1.1 Graphs .............................................................. 5
    2.1.2 Models for graph generation ................................. 7
  2.2 Input Testing ........................................................... 11
    2.2.1 Input simplification ............................................ 11
    2.2.2 Black-Box Testing ............................................. 11
    2.2.3 Delta Debugging ................................................ 12
    2.2.4 Fuzzing ........................................................... 14

3 Automatic generation of test cases for MOOCs .................. 15
  3.1 Software Architecture ............................................... 15
  3.2 Software Components .............................................. 17
    3.2.1 Input .............................................................. 17
    3.2.2 Test Case Simplification Algorithms ....................... 17
    3.2.3 Test Case Generation Algorithms .......................... 21
    3.2.4 Simplifying generated test cases ............................ 24
    3.2.5 Output ............................................................ 24
  3.3 Delivery Problem ...................................................... 25
  3.4 Segmentation Fault Problem ......................................... 26
3.5 Resource Limits Problem ........................................ 26
3.6 Discussion ....................................................... 27

4 Evaluation ......................................................... 31
  4.1 Software General Results .................................... 31
  4.2 Test Case Simplification Detailed Results ................. 35
    4.2.1 Delta Debugging ........................................ 35
  4.3 Test Case Generation Detailed Results .................... 40
  4.4 Test Case Simplification VS Test Case Generation ....... 41

5 Conclusions ...................................................... 45
  5.1 Future Work .................................................... 47

Bibliography ........................................................ 50
List of Algorithms

2.1 Erdős–Rényi \((N, p)\) ................................................................. 7
2.2 Random spanning tree algorithm with a root \(r\). ................................. 9

3.1 Delta Debugging \(\text{Delta Deb}(G = (V, E), \text{granularity})\) ..................... 18
3.2 Depth-First Search \(\text{DFS}(G = (V, E), \text{granularity})\) ......................... 19
3.3 Depth-First Search Auxiliary \(\text{Visit}(v, \text{maxVisit}, \text{vertexes})\) ................. 19
3.4 Erdős–Rényi \((N, p)\) ................................................................. 22
3.5 Morphing \((G_o = (V_o, E_o), G_r = (V_r, E_r), p)\) ........................... 23
List of Figures

2.1 Strongly connected components. ................................................. 6
2.2 Lattice ring network(A) converted into a small-world network(B). ........ 8
2.3 Black-box testing. ................................................................. 12
2.4 Minimizing Delta Debugging Algorithm. ..................................... 13
2.5 Minimizing Delta Debugging test case. ..................................... 14

3.1 Software Architecture Components. ......................................... 16
3.2 Input parameters. ................................................................. 17
3.3 Example of graph isomorphism ................................................ 21
3.4 Comparison between two graphs edges. ..................................... 21
3.5 Deliver process. ................................................................. 26
3.6 Input file. ................................................................. 28
3.7 Depth-First Search diagram. ................................................ 29

4.1 Software execution time to get a minimal test case for students projects in G1. .... 33
4.2 Software execution time to get a minimal test case for students projects in G2. .... 33
4.3 Relation between the students projects execution time and its number of vertexes. (G1) .... 34
4.4 Relation between the students projects execution time and its number of vertexes. (G2) .... 34
4.5 Relation between test cases vertexes reduction and number of vertexes. (G1) .... 36
4.6 Relation between test cases vertexes reduction and number of vertexes. (G2) .... 36
4.7 Relation between test cases edges reduction and number of edges. (G1) .... 37
4.8 Relation between test cases edges reduction and number of edges. (G2) .... 37
4.9 Relation between the number of failing subgraphs of a test case and the number of vertexes. (G1) .... 38
4.10 Relation between the number of failing subgraphs of a test case and the number of vertexes. (G2) .... 39
4.11 Relation between the number of failing complements of a test case and the number of vertexes. (G1) .... 40
4.12 Relation between the number of failing complements of a test case and the number of vertexes. (G2) .... 41
4.13 Relation between a test case number of test cases already tested and number of vertexes.
   (G1) .......................................................... 42
4.14 Relation between a test case number of test cases already tested and number of vertexes.
   (G2) .......................................................... 43
4.15 Graph reduced in order to obtain the minimal test case. (G1) ................................. 43
4.16 Graph reduced in order to obtain the minimal test case. (G2) ................................. 44
List of Tables

4.1 Software testing sample. ......................................................... 32
Chapter 1

Introduction

Massive Online Open Courses (MOOCs) [25, 13], is a known concept in the online education field. An online course is aimed at unlimited participation and open access via web. Introduced in 2008, the first MOOC emerged from the open educational resources movement, and the objective was to bring freely accessible, openly licensed documents and media that are useful for teaching, learning, and assessing as well as for research purposes. 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. Coursera¹ and Khan-Academy² 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. Moreover, the feedback must be generated automatically, since the course faculty is not available to answer to thousands of students.

Introductory algorithms courses have an important focus on graphs [5]. 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, 10] 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 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

¹https://pt.coursera.org/
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.

**Significance of Random Graph Generation.** Providing feedback to thousands of students can be a tedious task for the tutor [21] of a traditional course, but becomes an impossible task in a MOOC. Although a simple feedback or the original failing test case can be send for every student, personalized feedback avoid the potential of a reference output and increase the student’s motivation.

For MOOC graph projects, it is important to be able to generate a large quantity of different graphs that can be used to provide feedback to the students.

### 1.1 Motivation

Massive Open Online Courses are a new way of learning and teaching. They allow the online world to seat in the classroom with the best and brightest professors. Programming is a recurring subject on these classes \(^3\), which naturally leads to studying algorithms, data structures, and, of course, graphs.

Although tutors use the same definitions and general explanation to introduce a new subject, when it comes to practice, giving a better 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.

Student’s projects most of the times have different varieties of bugs such as wrong answers, segmentation fault, time-limit exceeded. In this thesis, we show how to deal with this problems in order to get minimal test cases which will be delivered to students.

\(^3\)See [www.edx.org/course?search_query=programming](http://www.edx.org/course?search_query=programming) or [www.coursera.org/courses?query=programming](http://www.coursera.org/courses?query=programming) for just a few examples.
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. In particular, for projects that solve graph problems and focus on elementary graph algorithms, such as single-source shortest path's algorithms on unweighted graphs. One important parameter our generators should be able to handle is the size of the graph. Our tools should be able to reduce large graphs to small graphs for paper and pencil exercises and to allow students to trace their algorithms looking for problems. Likewise, our tool should be able to generate larger graphs similar to the original test cases graphs.

1.3 Structure

This thesis has 5 chapters. In this first chapter, we introduce, motivate and present the goals of our thesis. In chapter 2, we present the most important terminology used in this thesis and describe the related work that was done in the thesis subject. In chapter 3, we propose and show the implementation of our new algorithms for test cases simplification and test cases random generation. In chapter 4, we evaluate our algorithms in a real world scenario such as a introductory graph course. Finally, chapter 5 presents the conclusions and future work.
Chapter 2

Preliminaries and Related Work

In this section the concepts related to graphs such as its features and characteristics will be reviewed. Graph generation algorithms such as Erdős–Rényi, tree generation and complex graph generation will be described and analyzed as related work.

2.1 Graph generation

2.1.1 Graphs

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 graph is said to be directed if each edge is associated with a direction. Otherwise, it is said to be undirected. A graph \(G\) is acyclic if it does not have cycles between his nodes. Also, an undirected graph is connected if every vertex is reachable from all others vertexes. The degree of a vertex is the number of incident edges to it. A directed graph has out-edges and in-edges which leads to a distinction between out-degree and in-degree. The degree sequence of an undirected graph is the non-increasing sequence of its vertex degrees. The weight of an edge is a function that maps edges to real-values. And the weight of a path \(w(p)\) is the sum of the weights of its \(k\) edges \(\sum_{i=1}^{k} w(v_{i-1}, v)\). Paths can have different weights according to its chosen edges, which had prompt to many path search algorithm. 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 \(G_c = (V_c, E_c)\) where \(V_c \cap V' = \emptyset\) and \(E_c \cap E' = \emptyset\). 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.

A strongly connected component of a directed graph \(G\), is a maximal set of vertexes \(C \subseteq V\) such that for every pair of vertexes \(u\) and \(v\) in \(C\), we have both a path, \(u \rightarrow v\) and a path \(v \rightarrow u\), meaning that this vertexes are reachable from one to each other. This characteristic reveals subgraphs that are themselves strongly connected and test the strong connectivity of a graph.

A tree is a connected, acyclic, undirected graph. This type of graphs is characterized by its vertexes
that are connected by exactly one path. A tree might have a root (rooted tree) and leaves, where root is the main vertex where the tree begins and leaves are the end-vertexes.

A spanning tree $T$ of an undirected graph $G$ is a subgraph that is a tree which includes all of the vertexes of $G$. Connected, weighted and undirected graphs $G = (V, E)$ can have a large number of spanning trees and one of them is the minimum spanning tree (MST), an acyclic subgraph (tree) that connects all the vertexes $V$ together, minimizing the total weight $w(V) = \sum_{(u,v) \in V} w(u, v)$ where $u$ is the initial vertex and $v$ the end vertex. In order to minimize the cost of power networks, wiring connections, piping, automatic speech recognition, etc. [12], people often use algorithms that gradually build a spanning tree (or many such trees) as intermediate steps in the process of finding this MST.

A Small-world network is a graph where the nodes are highly clustered leading to small path lengths between them. The node distance $L$ (path) between two randomly chosen nodes grows proportionally to the logarithm (in a slowly way) of the number of nodes $N$ in the network $L \propto \log N$.

### 2.1.1 Graphs and Tree Algorithms

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.

Dijkstra’s algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. Its main idea consists in starting in a given node and explore the next node based in which neighbor have the lower cost. As it is easy to see, a spanning tree, refereed in the section above, is internally build as an intermediate step in solving this problem.

These algorithms are taught in the course subject related to this work, Analysis and synthesis of algorithms (ASA) from bologna degree in information systems and computer engineering at Instituto Superior Técnico (IST).
2.1.2 Models for graph generation

This subsection assembles the beginnings of graph generation, reviewing algorithms such as Erdős–Rényi[17, 19]. Models for tree and complex graph generation are analyzed as related work.

2.2.1 Early graph generation

Erdős–Rényi[17, 19] is a random graph generation model. Introduced in 1959 was one of the first graph generation models. Erdős–Rényi name that derives from the creators names Erdős 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[8].

The $G(N, M)$ model consists in uniformly and randomly select a graph from a graph set with $N$ nodes and $M$ edges. While the $G(N, p)$ model selects a graph based in the probabilities $p$ of a pair of nodes. These two models show that random networks (graphs) have shorter average path length than ordered networks, one of the transition problems between graphs theory and real world complex networks. Algorithm 2.1, presents a pseudo-code for Erdős–Rényi $G(N, p)$ algorithm.

```
Algorithm 2.1: Erdős–Rényi $(N, p)$
1  $V \leftarrow N$
2  $E \leftarrow \emptyset$
3  foreach $i \in N$ do
4      foreach $j \leftarrow i + 1.....N$ do
5          chance $\leftarrow \text{random}(0, 1)$
6          if $p > \text{chance}$ then
7              if graphUndirected then
8                  $E \leftarrow E \cup \{(i, j)\}$
9              else
10                  chance $\leftarrow \text{random}(0, 1)$
11                  if $d > \text{Chance}$ then
12                      $E \leftarrow E \cup \{(i, j)\}$
13                      $E \leftarrow E \cup \{(j, i)\}$
14                  else
15                      $E \leftarrow E \cup \{(j, i)\}$
16                      $h \leftarrow \text{randomNode}(N)$
17                      $E \leftarrow E \cup \{(i, h)\}$
18  return $G = (V, E)$
```

This algorithm iterates all nodes $i \in N$. For each $i$, the relation with all the other nodes $j$ is randomly assigned. If $G$ is undirected and the probability $p$ associated with $i$ is greater than a random variable between 0 and 1 (chance), then create an undirected edge between $i$ and $j$. If $G$ is directed and the probability $p$ is also greater than chance, then this chance is randomized again to compare with the probability $d$ associated with $j$ node. Finally if $d$ is greater than chance, two directed edges are created. One edge from $i$ to $j$ and another from $j$ to $i$. But if this does not happen, then two edges are also
created, one from $i$ to $j$ and another from a random chosen node $h$ to $i$.

In 1998, a new random graph generation model was proposed by Watts-Strogatz [17, 23, 15]. Introduced after Erdős–Rényi, it aims at solving the problem related to the transition between graph theory and real world complex networks, namely the tendency of nodes in real world networks to form highly interconnected groups, or clusters, and yet still retain irregular connectivity patterns. The Watts-Strogatz model consists in having an undirected ring lattice network, which is a ring of nodes with edges divided evenly between its closest left and right neighbors. Where this ring have a high clustering coefficient, a measure of the degree to which nodes in a graph tend to cluster together. And through a random re-wiring process (i.e. edge modification) each edge has an arbitrary probability $p$ of being re-wired. This algorithm modifies only one vertex in each edge. Given an edge $e_1 = (n_1, n_2)$ at most one of its vertexes is changed. Moreover, for any given vertex, at most half of its edges are modified. The Watts-Strogatz algorithm became known by introducing the concept of small-world network as a result.

Figure 2.2[17] illustrates a re-wiring procedure achieved by Watts-Strogatz algorithm.

Since neither the Erdős–Rényi network nor the Watts-Strogatz network reproduce networks that have a scale-free approach, a small number of nodes with a much larger than average number of edges to/from other nodes. A third model has introduced to solve this problem, Barabási and Albert[17], in 1999. However this model does not produce a small-world network, so we will just make a reference to it.

2.2.2 Tree generation

A tree can be generated through a random walk in a graph or it can be generated similar to a graph. In this subsection we will review some of these cases.
Algorithm 2.2: Random spanning tree algorithm with a root \( r \).

1. \( E = \emptyset \)
2. \( V = \{ r \} \)
3. \( T = (V, E) \)
4. \( \text{Next}[r] \leftarrow \text{NIL} \)
5. \( \text{foreach vertex } v \in V \) do
6. \( \text{savedV} \leftarrow v \)
7. \( \text{while } v \not\in T \) do
8. \( \text{Next}[v] \leftarrow \text{RandomSuccessor}() \)
9. \( v \leftarrow \text{Next}[v] \)
10. \( v \leftarrow \text{savedV} \)
11. \( \text{while } v \not\in T \) do
12. \( V \leftarrow V \cup \{ v \} \)
13. \( v \leftarrow \text{Next}[v] \)
14. return \( T \)

Rooted Tree

A Rooted Tree is a tree in which a special vertex is singled out (root). This type of tree \( T = (V, E) \) can be randomly generated through a random walk in a graph[2] (see Algorithm 2.2). The rooted tree starts with only one vertex, the root \( r \). Then, for each vertex \( v \in V \), where in the beginning there is only one (root), and taking into account the remaining vertexes \( (v_1, v_2, \ldots, v_n) \), the algorithm randomly chooses a successor from these vertexes and adds it into the tree, called random walk. While not all vertexes are in the tree, the process is repeated until the tree is completed. A feature of this generation is that cycles are avoided. Since each vertex only keeps one pointer, in a second iteration on the same vertex \( v \), it assigns the next pointer to another random vertex.

2.2.3 Complex graph generation

This subsection reviews algorithms that take into consideration some graph characteristics such as degree sequence.

Switching algorithm

The Switching algorithm[14] is used to generate graphs with single edges between a pair of nodes. Since it uses a Markov chain[18], it does not take into consideration previous nodes and only has knowledge about the present node. The probability of moving to the next node is based on this knowledge. The generation is made with a given degree sequence and it starts by giving a network (graph) and involves carrying out a series of Monte Carlo[18], a method that is based on repeating random sampling to obtain numerical results. With this Monte Carlo series, the algorithm switch steps whereby a pair of edges \( (A \rightarrow B, C \rightarrow D) \) is randomly selected and exchange the end \( (A \rightarrow D, C \rightarrow B) \). This action is performed only if multiple edges or self-edges are not generated. The process is repeated \( QE \) times.
where $E$ is the number of edges in the graph and $Q$ is chosen large enough that the Markov chain shows good mixing. The disadvantage of this switching algorithm is that it can not guaranty how much time it takes to mix properly.

**Matching algorithm**

An alternative of the previous algorithm, is the Matching algorithm [14]. In this approach, each vertex is assigned a set of "stubs", the sawn-off ends of incoming and out-going edges, according to a desired degree sequence. When the assignment is complete, in-stubs and out-stubs are picked randomly in pairs and then joined up to create edges. This process creates random directed graphs that have the desired properties, such as degree sequence. Different from switching algorithm, this alternative expect a number of edges between two vertexes often higher than one. To obviate this problem, the stub pair that creates a multiple edge is discarded and a alternative stub pair is selected at random.

**Go with the winners algorithm**

Different from the previous algorithms, the Go with the winners algorithm [14] does not use a Markov-chain and Monte Carlo approach for sampling uniformly a given distribution. This algorithm considers a colony of $M$ graphs. It starts with each node assigned with its in-stub and out-stub and randomly chooses one in-stub and one out-stub from the graph to join them together and make an edge whereby the matching algorithm. If any multiple edge or self-edge is generated then the graph is discarded from the colony. Since the graph discard results in a decay of the colony size, the algorithm periodically doubles its size by cloning each of the surviving graphs. This allows the colony to keep its original size. After the process being repeated multiple times, all the stubs are linked. Then one graph is chosen at random from the colony and assigned with a weight. The weight attribution uses the following formula $W_i = 2^{-c \frac{m}{M}}$, where the $c$ is the number of cloning steps made and $m$ is the number of surviving networks.

**Morphing**

Morphing [7] 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

This section reviews definitions related to input testing. Definitions such as input simplification, Black-box testing and its related concepts. It also introduces and analyzes Delta Debugging[26, 9, 3, 4] and Fuzzing[3, 16] as input simplification algorithms.

2.2.1 Input simplification

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. Reducing the time that a function takes to compute and facilitating the debugging process. 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.

2.2.2 Black-Box Testing

Black-box testing[24] is a method to test the functionality of an application based on its inputs and outputs. This method does not test a program internal structures and for that reason relies only on what the functionalities need to achieve and see if its doing what is supposed. Unit, integration, system and acceptance are some of the test that use this method[24]. Normally, these are the tests that a programmer shows to his clients.

Black-box testing use as feedback reference the program output. Output is the act of taking out data from a application. An example of it is a monitor where the data is presented. The main purpose of the output is to give users a feedback of what occurs in the application. This means that users can know if their functions are making what they supposed to.

Fig. 2.3 illustrates the Black-box testing[24] components and the cycle between them. Given a test battery and an initial test composed by an input $I = (i_1, i_2)$, where $i_1$ represent a person name (string type) and $i_2$ represent a person nationality. The Black-box testing receive this input and insert
it in a function that search a person object with an equal name and nationality and returns his age. As expected, the output should return the right age. Through the output observation, a program gives feedback to the user, where it reports if the test passed or failed. This process repeats until there are no more tests to run.

2.2.3 Delta Debugging

Delta Debugging[26, 9, 3, 4] 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 sub-problems, 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. 2.4[26], the minimizing delta debugging algorithm (ddmin) is described. As previously mentioned, this algorithm is based in test subset. When neither $\Delta^1$ or $\Delta^2$ 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.2.5[26] illustrates the use of the algorithm presented in Fig. 2.4 using the granularity when subsets are inconclusive. The illustration shows that a input set starts by being divided in two subsets $\Delta_1$ and $\Delta_2$, 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 is increased, $n = 4$, therefore $\Delta_1$ and $\Delta_2$ 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 $\nabla_2$ 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 $\Delta_1$ and $\Delta_2$ are divided in two subsets, where now $\Delta_1 = \{1\}$, $\Delta_2 = \{2\}$, $\Delta_3 = \{7\}$ and $\Delta_4 = \{8\}$. The complement of $\Delta_2$, $\nabla_2 = \{1, 7, 8\}$, 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 $I = \{1, 7, 8\}$.
2.2.4 Fuzzing

The term “fuzz” or “fuzzing” originates from a 1988 class project, taught by Barton Miller at the University of Wisconsin. The project developed a basic command-line fuzzer to test the reliability of Unix programs by bombarding them with random data until they crashed. So Fuzzing is a technique useful for software tests. It is an automatic tool that provides invalid, unexpected and random data as inputs of software programs. Fuzzing\[3, 16\] uses samples of data to create new input cases and test a software program. It can be random, where it receives a random bit flow in the software like an event, command lines, etc. This technique is related to Black-box testing in the way that only relies in the program input and manipulates it in order to find bugs, especially security flaws. This algorithm simplifies inputs in a perceptive that generate new case tests where it achieve a test case where a bug is more exposed than a in the previous.

As previously mentioned, Fuzzing can easily find bugs. This technique uses samples of data, and with knowledge about the input nature, it creates new input test cases. The main focus of this technique is that generate new input case tests that can isolate the failure on an application.
Chapter 3

Automatic generation of test cases for MOOCs

In this chapter we take a deep look into the architecture of our software. Starting with how the input problem was implemented, which algorithms did we use, why did we use them and in which way they work together. Next, we describe the issues that we found when implementing our software and the scenarios we created to test it.

This software integrates the course of Analysis and Synthesis of Algorithms (ASA) from the degree on information systems and computer engineering at Instituto Superior Técnico (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.3.1.

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[11], 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 3.1 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 section.
3.2 Software Components

In this section, we describe in detail each component of our software, TCSS. In particular, the algorithms we choose to solve the problem and in which way we modify them to achieve our goals.

3.2.1 Input

First of all, we implemented a class that receives certain parameters from a console application:

- Test Case (1)
- Reference Implementation (2)
- Student Implementation (3)

As illustrated in figure 3.2, the parameter 3 is a file which represents a student solution to a graph problem. The parameter 2 is also a file developed by tutors in order to provide a trustful problem solution. Finally we need a test case that fails in the student implementation, our parameter 1. After receiving this input, the parameter 1 is read so that we can get the graph information and start the input simplification algorithms. While the other two inputs, are used to test out the Delta Debugging graphs and see if they keep failing meaning that the output from student project and reference project are different.

3.2.2 Test Case Simplification Algorithms

Since the major goal of TCSS is to reduce a test case to a minimal one where the student project keeps failing and so we can provide a better feedback, there was a need to create a class specifically dedicated to the Delta Debugging procedure.
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 \(\text{Delta Deb}(G = (V, E), granularity)\)

1. if \(\frac{|V|}{granularity} \leq 2\) then
2. return \(G\)
3. else
4. \(SG \leftarrow DFS(G, granularity)\)
5. foreach \(sg \in SG\) do
6. if \(\text{checktest}(sg) = FAIL\) then
7. if \(\text{isMinimal}(sg)\) then
8. return \(sg\)
9. else
10. if \(granularity > 2\) then
11. \(granularity \leftarrow granularity - 1\)
12. return \(\text{Delta Deb}(sg, granularity)\)
13. \(granularity \leftarrow granularity + 1\)
14. return \(\text{Deltadeb}(G, granularity)\)

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 section. 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 \(\text{checktest}(sg)\) (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 \(\text{isMinimal}\) (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 \( \text{granularity} > 2 \), 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 \( \text{Visit} \) (line 6) that starts a graph transversal visiting vertexes recursively and returns a list of visited vertexes (See Algorithm 3.3).

\[
\text{Algorithm 3.2: Depth-First Search } DFS(G = (V, E), \text{granularity})
\]

1. \( SG \leftarrow \emptyset \)
2. \( \text{foreach } v \in V \text{ do} \)
3. \( \quad \text{color}[v] \leftarrow \text{white} \)
4. \( \text{foreach } v \in V \text{ do} \)
5. \( \quad \text{if } \text{color}[v] = \text{white} \text{ then} \)
6. \( \qquad \text{vertexes} \leftarrow \text{Visit}(v, (|V|/\text{granularity}), \emptyset) \)
7. \( \qquad \text{sg} \leftarrow (\text{vertexes}, \{(u, v) \in E : u \in \text{vertexes} \land v \in \text{vertexes}\}) \)
8. \( \quad SG \leftarrow SG \cup \{\text{sg}\} \)
9. \( \text{return } SG \)

\[
\text{Algorithm 3.3: Depth-First Search Auxiliary } \text{Visit}(v, \text{maxVisit}, \text{vertexes})
\]

1. \( \text{color}[v] \leftarrow \text{black} \)
2. \( \text{vertexes} \leftarrow \text{vertexes} \cup \{v\} \)
3. \( \text{if } |\text{vertexes}| = \text{maxVisit} \text{ then} \)
4. \( \quad \text{return } \text{vertexes} \)
5. \( \text{foreach } u \in \text{adj}[v] \text{ do} \)
6. \( \quad \text{if } \text{color}[u] = \text{white} \text{ then} \)
7. \( \qquad \text{vertexes} \leftarrow \text{Visit}(u, \text{maxVisit}, \text{vertexes}) \)
8. \( \quad \text{if } |\text{vertexes}| = \text{maxVisit} \text{ then} \)
9. \( \quad \text{return } \text{vertexes} \)
10. \( \text{return } \text{vertexes} \)

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. 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

\footnote{We assume each graph must be connected}
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, \textit{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 \textit{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 vertices, 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, \textit{complement}(G, sg). The condition that avoids these situations is when the \((\|V\|/\textit{granularity}) + 0.1) > |S|\), 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 \textit{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.

Testing subgraphs and their complements allows for a better testing coverage which leads to an increase of reduced test cases. However, sometimes our algorithm is unable to reduce a test case to a minimal one. In these cases, granularity keeps increasing as well as the number of subgraphs and the number of complements. It might get to a point where you only have graphs composed by two vertexes and one edge. As we will see, its useless testing innumerable graphs of this kind since if one of them pass the test, the other will pass as well. In the moment that we only have subgraphs with a minimum size, Delta Debugging algorithm stops and returns the smaller graph that it could obtain.

Testing subgraphs and their complements causes some efficiency problems. For test cases where we have a huge amount of vertexes and edges, the time that our algorithm spends increase a lot and given another problem that is the time that student project takes to give us a output, our program takes more time than it should to reduce a test case. We considered to take a graph isomorphism approach which means that if two vertexes are adjacent in a graph \(G\), they must be adjacent in a graph \(H\) and with this we could assure that with the same number of vertexes from a tested graph and one that will be tested, the edges would be the same in both graphs. In the fig.3.3, we illustrate a case where two graphs are isomorphic. As you can see, a mapping of labels was made, where \(v_1\) and \(v_2\) are still equal but \(v_3\) is mapped as \(v_4\) and \(v_4\) mapped as \(v_3\).

Testing for isomorphic graphs is not possible in our context, since this problem is \(NP - complete\) [6]. In order to work around it and find a solution to this graph isomorphism problem, we created a list with the graphs that were tested, saving time when we have a similar graph (same number of vertexes and edges) to be tested. For example, if we test a graph with three vertexes and two edges that passes, it will
be saved in our already tested list. Later, when a graph with three vertexes and two edges is generated, the program will automatically skip it and save time and CPU resources. The only disadvantage in this solution is that we cannot assure that the edges are the same in both graphs, already tested graph and the graph being tested at that moment.

In the fig.3.4, we illustrate a figure where we have two graphs. They have the same number of vertexes and edges. Since we cannot use a graph isomorphism approach in our software, with our solution this example became a problem. Given that our software already tested the first graph, we kept a registry that graphs with four vertexes and four edges will not be tested. The result is that the second graph will not be tested. This approach reduces drastically the number of tests and in the majority of the cases, the improvement is visible since that graphs with two vertexes and one edge will not be tested more than one time and saves us a lot of time. However, in certain cases like the one in Fig.3.4, this solution can be a problem in a way that we cannot assure that the second graph would pass the test like the first did.

Given all the process above, succinctly our Delta Debugging is an algorithm that thoroughly test subgraphs and their complements in order to simplify a test case so that if one of its testing subjects fails, it can reduce the original test case and keep testing until reach a point where the test case cannot further be reduced.

### 3.2.3 Test Case Generation Algorithms

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 $|V|/\text{granularity} < 2$. 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 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 Erdős–Rényi algorithm for that
purpose. Erdős–Rényi 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: \( \text{graphdensity} = \frac{|E|}{(|V| \cdot (|V| - 1)/2)} \) which give us a
number between 0 and 1. As mentioned in Chapter 2, the Erdős–Rényi 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
Erdős–Rényi 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).

Having also a random value, if the graph density is higher than this random value, then the Erdős–Rényi
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 Erdős–Rényi algorithm is that we wanted a simple and easy way to implement
random graph generator which we did not have to specify any characteristics to it work, since our graph
restriction are only about having the same number of vertexes as the original graph.

<table>
<thead>
<tr>
<th>Algorithm 3.5: Morphing ((G_o = (V_o, E_o), G_r = (V_r, E_r), p))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (V \leftarrow V_o)</td>
</tr>
<tr>
<td>2 (E \leftarrow \emptyset)</td>
</tr>
<tr>
<td>3 foreach ((u, v) \in E_o) do</td>
</tr>
<tr>
<td>4 (\text{chance} \leftarrow \text{random}(0, 1))</td>
</tr>
<tr>
<td>5 if ((u, v) \in E_r) then</td>
</tr>
<tr>
<td>6 (E = E \cup {(u, v)})</td>
</tr>
<tr>
<td>7 else</td>
</tr>
<tr>
<td>8 if (\text{chance} &gt; p) then</td>
</tr>
<tr>
<td>9 (E = E \cup {(u, v)})</td>
</tr>
<tr>
<td>10 foreach ((u, v) \in E_r) do</td>
</tr>
<tr>
<td>11 (\text{chance} \leftarrow \text{random}(0, 1))</td>
</tr>
<tr>
<td>12 if ((u, v) \notin E_o \land \text{chance} &lt; p) then</td>
</tr>
<tr>
<td>13 (E = E \cup {(u, v)})</td>
</tr>
<tr>
<td>14 return (G = (V, E))</td>
</tr>
</tbody>
</table>

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 \(\text{chance}\) (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 \(\text{chance}\), 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
be 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 \(\text{chance} < p\) (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, we add a new edge between the vertexes
being analyzed. These are minor modifications that let the morphed graph slightly similar to the \( G_0 \).

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. This will be mentioned in detail, in the next subsection.

### 3.2.4 Simplifying generated test cases

In this subsection, we see how the test case simplification, that is need to reduce test cases and provide a better feedback, can be correlated with the test case generation, that aims to create new test cases from the original test case taking into consideration some specifications.

Given that the goal is to reduce test cases and have a good effectiveness doing it, reducing test cases might not be enough because a lot of student projects only fail at the original test case or keep failing at large graphs meaning that the test case still represents a graph with a lot of vertexes and edges, which are difficult for the students to debug their projects. So the test case simplification algorithms must have a communication with the test case generation algorithms. To start with, the simplification test cases part will try to reduce a test case. TCSS has to use the Test Case Generation part in two situations: (1) If we cannot reduce a test case to a simpler test case or (2) if the graph represented in the test case remains too large to be delivered to a student. We choose that a test case is large if the original test case have more than 20 vertexes and the minimal test case that was found have more than 20 vertexes. If one of these situations happen, the test case generation part will modify test cases in a way that we can assure that more test cases will result in a minimal test case even if it is not the original one. Since the modified test cases are similar to the original, students might also be able to fix their bugs for the original test case using the modified ones. TCSS assures to put some effort into this Test Case Simplification and Test Case Generation communication in a way that it keeps creating modified test cases until we reach a minimal test case or we had created a certain number of morphed graphs where neither the original or any of the modified graphs can be reduced. In this situation, we will decide which graph we must send to the student even if the graph is still larger than it should.

### 3.2.5 Output

After combining the test case simplification and the test case generation, we obtain a simplified graph where the student’s project still fails. Given that we need to send to the user a test case, this graph is transformed in a file which contains the graph information. This file can be blank which means that our program could not find a graph reduction. Otherwise, the file’s information is composed by the number of vertexes, the root vertex, the number of edges and every single edge description which means that first appears the origin node and then the destination node. With this, the student can easily draw the graph on paper and analyze it by looking at his algorithm and check what he is doing wrong.

Even with this output, our software does not guarantee that the student project will pass the test case if they solve the bug for the simpler one that we send. The reason is that TCSS sends a test case to the student but it does not mean that the bug in that test case is the only one present in the student project. So if a student want to pass a test, he should use TCSS more than once if needed in order to find its
project bugs. The advantages of this is that we help the students solving bugs in an undiscriminated way. On the other side, we are not bug specific for a certain kind of test in a way that we do not verify if the present bug in the original test case is the same as the bug present in the graph that we send to the user.

3.3 Delivery Problem

Since our project goal is to deliver to a student a file that contains the graph information for a minimal test case so that the student can debug his project, this leads to a deliver problem which has to take in consideration some factors. The major problem is that we cannot deliver the original test case to a student. School plagiarism is a serious problem and we had to take in consideration that a student that receives a original test case, can easily use the output of this test and print it in his project as output leading to a test passed wrongly analyzed. Moreover, since our program generates at least one morphed graph in order to try to reduce it if the original one cannot be reduced to a minimal test case, that satisfy the conditions to be send to the student, and if the morphing graph cannot also be reduced to a minimal test case, we had to decide which test case should we deliver to student.

Given these problems and the relevance that they have, we started by seeing the cases where our algorithm cannot send a simpler graph to the student. This happens when neither the original test case or the morphed test cases cannot be reduced to a smaller graph and when our morphed graphs do not fail as a test case in the student project. When our algorithm keeps increasing the granularity so that it reaches a point where the subgraphs only have 2 vertexes, we try to generate a morphed graph. After this happens with our generated morphed graphs and no one of them fails, we decided that the best way to avoid sending the test case to a student was replacing that test case with a blank file that does not contain any information. This occurs when our algorithm was not able to help the student. But there is also the case where a morphed graph fails, so instead of sending a blank file, what we do is send one of the morphed graphs, since at least they have a test case that they can use to guide through in order to help them discover the bug. Since we do not discriminate bugs, does not matter if we send the bug present in the original test case because with a failing test case, the student will be able to correct at least one bug.

As previously mentioned, sometimes the original test case is reduced to a smaller graph, but not a graph in the conditions to be sent to the student. Imagining that this happens to our morphed graphs as well, we keep the smaller graphs to be compared. At this point it is important that we have some criteria on what graph should we choose to send to the student. To solve this problem we started by comparing graphs. Our criteria is based on how many vertexes and edges these graphs have. Initially we start by comparing the original graph with the first morphed graph. If it has less vertexes than a morphed graph we keep the original and discard the morphed, otherwise we keep the morphed and discard the original. Our program keeps doing this until the last morphed graph. When only one graph is kept, this is the one that we send to the student. But there are cases where the vertexes are the same in both graphs. In this situation we compared the graphs by the number of edges, since a graph is easier to analyze if have
it has fewer edges. So if the obtained graph has the same vertexes but fewer edges we should keep it to send to the user. If exceptionally, two graphs have the same number of vertexes and edges, we just keep the first one since it is irrelevant which one do we send to user because the graph analyzed will be similar in both cases. TCSS goal is to deliver a failing test case, so as said, it is irrelevant which graph we send if they both have the same size since we do not discriminate bugs and also we are sending the minimal graph that we can find that keeps failing. So with our criteria we will send a smaller graph that is easier for the student to find the bug as you can see in fig.3.5.

3.4 Segmentation Fault Problem

One issue is how we should treat segmentation fault errors when executing the student project, meaning that when we are applying your Delta Debugging algorithm to the test case or the test case subgraphs and its complements we obtain segmentation faults from the student project. Initially our thought was that we had to ignore these cases and send back to student a empty file which would inform them that his project contains a segmentation fault and could not be helped with. But while doing our program we came across with test cases that would give segmentation fault but when that happened it gave us an output, different from the reference one. To solve this problem, we started then treating a segmentation fault as a failing test case and reducing the original test case to the one that gave a segmentation fault in the student's project. Therefore, students can have a small graph or even a minimal one where they keep having the segmentation fault or a test case that gives them a bug to be treated.

3.5 Resource Limits Problem

By the time that we were developing TCSS, we faced a problem that could make TCSS waiting for a response from the student project that would not come. This happens when their project cannot process a test case and takes eternities to answer with an output or do not even produce an output. This is a
resource limits problem and we notice this by developing our Delta Debugging algorithm. Right after, we divided the test case where the student had a bug, the program would stay in an infinite loop testing a subgraph. This infinite loop was coming from the student project and was keeping us from providing an answer. To solve this, we reached out with a solution based in a timeout approach. Given this, we search for an external application that could handle this problem. We use RunSolver[20], an application that allows users to integrate it on their software and handle a certain function call. We used RunSolver in our project when the student project is called to be executed with a test case. In this way, we can be sure that the student project will not take eternities to give us an output. The time we choose was two seconds, the same timeout configured in the automatic evaluation system, Mooshak. The time is not that long but should be enough for a student project to give us an output. This is a reference timeout adequate to the project that each student was to implement. The reason for the two seconds in it, is because of the way that the faculty evaluate certain student implementation allowing that some of them pass the tests and other ones does not. With this application, we can now obtain an output from certain student project. This output will probably be blank or a repetition of the same thing (infinity loop cause this) which will be different from the reference output and will be treat as a failing test case, reducing the original test case to the one being tested. As a result, we can send to the user a smaller graph where he will still have run-time error or another bug in his project needed to be solved. Another problem related to resources, is the memory usage for each system call on the student project with a certain input. So our software uses RunSolver to limit this kind of situations. Given that a student project can be allocating unnecessary memory and have a bug where it keeps allocating it, we have to put a limit on that so that our host PC does not have problems when running it. The limit that we choose was 64MB since the automatic evaluation system, Mooshak, use this value as a memory limit for the student projects. Again, the memory limit has this value for the reason that faculties need a standard value that evaluates certain ways to implement the problem that students have in their projects. For this kind of problems, we simply just stop the test and it fails, so we can deliver to the student a minimal test case where the memory problem is present and he can debug it in order to fix the bug.

3.6 Discussion

Currently, our software only works for undirected graphs. This means that it can only reduce undirected graphs and be useful for projects that use this kind of graph. In this section, we will talk about in which way our program should change to work for different kind of graphs.

As known directed graphs are graphs that have a set of vertexes connected by edges, where the edges have a direction associated with them. Therefore, the adjacency lists should have a different insertion algorithm. Currently if we read an edge from a file and we insert the vertexes present in that edge in the other vertex adjacency list. To make our software work for directed graphs we would have to modify the inserting function in a way that only the destination vertex is inserted in the origin vertex's adjacency list. For example, if we have an input file as the one presented in the fig.3.6 and the edge 7 6, illustrated in the line 4 of the figure, we would have to inserted the vertex 7 into the adjacency list 6. So
to construct a software to reduce test cases for directed graphs, this input file would have to be analyzed in this way.

As we can see in the figure, lines are showed in the left side with a gray style so we do not confuse the readers. The first line is composed by two numbers, the first one represents the number of vertexes and the second the number of edges. In the second line, we have a single number representing the root vertex. Finally the rest of the lines represent the edges, the important part in this section. The first number represents the origin vertex and the second number represents the destination vertex.

As said before, our software would need to be modified for this test case can be transformed into a minimal test case, in directed graphs projects. Taking the example previously mentioned, we should modify the file reading function in a way that the edge 1 5 when read, had to be inserted only in the origin vertex 1 adjacency list. This would make our Depth-First Search algorithm works in a more strict way since it would have less vertexes to proceed the search. This differences are illustrate in the fig.3.7.

Since that directions matters, the subgraphs that would be tested by Delta Debugging should be different between directed graph and undirected graph in this case. And as we can see, the subgraphs are different cause by the adjacency lists differences. Even that the subgraphs are different, this does not mean that our software have to suffer a modification in Delta Debugging algorithm because it only cares about a subgraph having more than one vertex in order to test if it passes or fails.

But what if we need to generate another test case when our original cannot be reduced? For directed graphs our software would have to be modified as well. We would have to use a generation algorithm that can generate directed graphs. For example if we keep using the Erdős–Rényi and Morphing models for the test case generation, our program would have to be modified. Currently our Erdős–Rényi receives a list of vertexes and randomly create edges between them depending on a random variable and in the graph density. If the value of this variable is lower than the graph density the edge is created between two vertexes but it does not take in consideration the direction since that our software use undirected graphs. What happens is that both vertexes are add in the adjacency list of the other vertex. For a directed graph we would have to not only look for a possible edge between two vertexes but instead see
if it would be possible to have two edges between them. For this to happen, our algorithm would have to iterate all the vertexes combinations. This means that we could not ignore a vertex being destination in an edge after we evaluate a possible edge where it is the origin vertex. Because even if an edge is created where vertex 1 is a destination node and a vertex 3 is the origin node, it can also exist a edge where 1 is the origin node and 3 is the destination node. Regarding to Morphing, the algorithm only mix edges. If it exist in the original and in the random graph, so the morphed graph will have that edge. If that edge does not exist on one of them then it has a probability of being added into our Morphed graph. Given this, the only thing that should be modified in the Morphing algorithm is the creation of an edge that instead of an undirected one, would be a directed which would be present in only one adjacency list as mentioned before. The search of an edge in both graphs, original and random, would not need a modification since that it already look specifically for the edge where the origin vertex is x and destination node is y.

What if it is a directed graph with weights? For this scenario, the input file would have to be different. So our input reader function would have change to receive this kind of inputs. Depth-First Search would remain equal but as it would happen with directed graph, this one would have different subgraphs. The
bigger chances in our software for this kind of graph would be related with test case generation. Since
that a directed graph with weights already have some specifications, Erdös–Rényi would not be able to
generate a graph that we request. So for this kind of event, we have a software from another student of
my institution, that can generate a random graph giving it certain parameters or specifications that we
want in a graph. Mixing this software and Morphing model, we would created a Morphed directed graph
with weights.
Chapter 4

Evaluation

In this chapter, we show the results of our software. Using a directory which contains 618 failing student projects of a university (Instituto Superior Técnico) algorithms course. These projects were implemented by students to solve a problem given by the faculty which the goal is to find the shortest path between two edges. A failing student project meaning that it fails at least in one test case from the Mooshak test battery. These student's projects are divided into two groups. The first group $G_1$ contains 468 projects that fail in small test cases which means that the failing test case has a maximum of 500 vertexes. The second group $G_2$ contains 150 projects that fail in larger test cases that has more than 500 vertexes. We present graphs, tables and examples of test cases getting reduced, showing the efficiency of our software. The weaknesses and strengths that we have presented in this thesis and the experience that we obtained implementing this software, can be relevant for future work.

Initially, we show general results of our software, in particular how many test cases we tested and which ones were delivered to students. How much time we need to each test case reduction and show a relation of the execution time with the size of the test cases. Moreover, we will enter in details. First, we describe the Test Case Simplification details, such as the number of vertexes and edges of a reduced test case in comparison to the original one, the number of failing subgraphs and complements and number of test cases already tested that was caught by our cache and not tested by our Delta Debugging implementation. Next, we show the Test Case Generation details, such as how many minimal test cases are obtained from Morphing graphs. Finally, we do some tests where we compare two different approaches to obtain a minimal test case, which the first approach is our TCSS and the second is a software made by a student for his thesis which the goal was to generate random graphs.

4.1 Software General Results

Chapter 3 presents the details of our software (TCSS). In this chapter we show the results of it applied to student's projects of a IST algorithms course. In the beginning of the testing process, the first thing to do was to select a certain number of student projects, to be tested. We decided that our sample would contain 618 student projects, divided into two groups. Having the student projects to be tested with our
software, we had to decide the criteria that we should use to test the software.

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 4.1 we show the sample that is used to test our software, which includes the number of student 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.

<table>
<thead>
<tr>
<th>Student Projects with a failing test case</th>
<th>618</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student Projects with a minimal test case</td>
<td>570</td>
</tr>
</tbody>
</table>

Table 4.1: 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 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 4.1. 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. We cannot assure that these results will be equal for another student's projects samples, because it can always have bugs that we could not detect during the software implementation and the effectiveness can be lower or higher. Moreover, some test cases that we deliver to students are only morphing graphs of the original which means that since we could not reduce the size nether from the original or the morphed ones, we send a failing morphed graph (test case). If these morphed test cases did not fail, the effectiveness would be lower because, as previously described, we cannot send a original test case to a student.

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 were able to made a graph showing the time that every student project with a failing test case took to get its failing test case reduced into a minimal test case. In figure 4.1 we show the execution time for the submissions belonging to the first group (submissions with a failing test case that have a maximum of 500 vertexes). And in the figure 4.2 we show the execution time that we got for the submissions belonging to the second group (submissions with a failing test case that have more than 500 vertexes).

As you can see, we sort these graphics in an ascendant order so that we could easily see the differences between execution time. In the first group, the majority of the students projects had its failing test case reduced to a minimal test case in less than one second but for some exceptions the reduced time was higher than expected. The failing test case that took more time to get reduced in this group, took 68.2463 seconds. This was caused by many factor such as the time that the failing student program takes to execute each graph, the number of graphs tested and the use of our morphing algorithms as well. In the second group, the execution time is much higher. In some cases, our application reached
the 3600 seconds (one hour) time limit. Even so, some graphs as we can see took less time. These results make us conclude that our software takes a lot of time to get a minimal test case for bigger test cases and the reason is that the number of graphs tested is huge for these test cases. Therefore, 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 4.3 and 4.4, 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 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 because even though we cannot find a smaller graph that fails in the student project, the granularity is increasing by $2 \times \text{granularity}$ and reaches a point where granularity can no longer be increased, meaning that the granularity is equal to the number of vertexes.
vertexes divided by the granularity is equal or higher than two, $\|V\|/\text{granularity} \geq 2$. 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.

4.2 Test Case Simplification Detailed Results

In this section, Delta Debugging and Depth-First Search algorithms are analyzed. The main focus will be on Delta Debugging since it is our major algorithm in order to reduce test cases. The measures that we want to show are the percentage of the number of vertexes and edges that our software reduces from the original numbers, how many subgraphs and complements fail in the Delta Debugging to obtain a minimal test case and how many test cases are ignored, since they were already tested, by our Delta Debugging.

4.2.1 Delta Debugging

In this subsection, we present the measures that allows us to show the efficiency and effectiveness of the Delta Debugging algorithm. Initially we relate the reduction of vertexes with the initial number of vertexes. The aim with this relation is to understand if a specific size of a graph tend to have a certain percentage of reduction in terms of vertexes. This is relevant to conclude if our algorithm was able to obtain a better reduction for certain test cases than others.

In figures 4.5 and 4.6, we can analyze two things. First, students projects have an high variety of bugs because most of the graph sizes have different vertexes reduction, for example in the graphs with 6 to 20 vertexes we can see that it had 50% to 75% vertexes reduction for most of the situations but also a lot of 0 to 50% and 75 to 100% in another test cases. This means that student projects had different bugs which leads to different vertex reductions. A good result that we obtained was the fact that in sized graphs between 6 and 20 vertexes, our highest column has a 75% vertexes reduction. Which means that for test cases with 8 vertexes, the minimal test case is also a minimum size test, that have 2 vertexes. Since our majority of test cases have 8 vertexes, this is the reason why the yellow bar (50-75%) is the highest in the figure 4.5. Also, justifying the relevant number of test cases that could
only reduced between 0-25% the vertexes number, in the 2 and 5 vertexes test case, its easy to analyze that a graph with 2 vertexes is already minimal and have a 0% of vertexes reduction and for a test cases with 5 vertexes the information that we can take is that some cases could not be reduced or only could be reduced to a 3 vertexes graph but we also got some situations with a minimal test case equal to a minimum size test since a reduction between 25-50% means that these graphs passed from 5 vertexes to 2.

For larger test cases, the interval of values in reduction percentages is more equilibrated. Meaning that larger test cases have different vertexes reduction values and that the bugs are different between students projects for most of the cases. As you can see, in failing test cases with 1001-20000, we have a lot of student’s projects per vertexes reduction interval but the highest one is the 75-100% which is a good result since we got our graphs reduced to graphs who can be debugged through a paper and a pencil, we also have a few situations that we could not reduced at all any vertexes or just at maximum
25% of them, which means that our algorithm could not find a small failing test case which can be related to the bug present in the student project. One example of that is a wrong size allocation of a vector that could lead to a graph failing only for a certain number of vertexes but which our minimal test case will help him to understand the bug. We do not have 100% reduction because that would mean that the resulting graph would have 0 vertexes.

As a second measure to test our software, we considered a graph that relates the number of edges of the original test case and the percentage of edges reduced in the minimal test case. These graphs are important to see the effectiveness of Delta Debugging.

As you can see in the figures 4.7 and 4.8, the reduction of the number of edges (like the number of vertexes previously shown), have a large range of values. This is related to the bug present in the student project. Since the range of bugs is also large, the results through the edges reduction is different for the different test cases even with the same amount of edges. Differently of what we thought, we cannot
Figure 4.9: Relation between the number of failing subgraphs of a test case and the number of vertexes.

(G1)

see any negative values and only just some reductions equal or very close to 0% which means that our Morphing is not creating more edges than the original graph but less or equal edges. This situation happens when a test case cannot be reduced and only a morphed test case fails, since a morphed graph can have more edges than the original test case (negative value), equal number of edges (0%) or less number of edges (low edge reduction value). Our highest number of student's projects was in test cases with 10 to 49 edges. In these test cases we got good results because our highest column have more than 75% edge reduction. This situation is caused for the same reason that in the figure 4.5, since the test case with 8 vertexes has 11 edges. For the same reasons that we talked about in vertex reduction, again in edge reduction we can see that in fig.4.7, we have the highest values for 0-25% in the graphs with 1 to 9 edges.

Now that we have results on the effectiveness of Delta Debugging towards the reduction of a test case, we show the results that we obtained while the Delta Debugging is reducing a test case into a minimal test case which allow us to show behavior of our Delta Debugging implementation. Observe that the subgraphs and the complements are created by dividing the graph according to the value of the granularity. In order to find a minimal test case, they are tested to see if the student project keeps failing. To know which one fails more in the student projects, we see the number of subgraphs and complements that fail for each student submission. This allow us to conclude if is the subgraph or the complements that fail more in the student's projects. Also, helping to see a relation between these results and the vertexes reduction in each test case.

For the subgraphs, in figures 4.9 and 4.10, most of the failing test cases are not reduced to a subgraph or are only reduced one time to a subgraph. The reason that this happens is that Delta Debugging is reducing the test cases to the minimal test case with only one reduction to a subgraph or reducing to a subgraph and then using complements to reduce the test case to a minimal one. For smaller test cases, the difference between the columns is higher than in bigger test cases, this happens because the Delta Debugging does have less subgraphs to test so it means that less subgraphs will fail. Not only that
but for example graphs with 2 vertexes already are a minimum size test and don’t even use our Delta Debugging to test subgraphs and for graphs with 5 vertexes i can only reduce to a maximum of one failing subgraph since the granularity $= 2$ already divide the graph into two subgraphs with 2 vertexes and discard the remaining vertex that cannot by himself be a graph. Also having low number of failing subgraphs does not mean that we could not reduced our test cases or the reduction was low in terms of size. Quite the opposite because we saw in the vertexes reduction image that for test cases with 6-20 vertexes we obtain, for the majority, higher values than 75%. We can then affirm, that our algorithm for a lot of test cases was able to reduce to a minimal test case with only one reduction which means that our algorithm was precise in the reduction.

For the complements, in figures 4.11 and 4.12, the results are different from the subgraphs. For the small test cases, the number of failing complements is low but understandable since that with low tests being made in Delta Debugging means that there is also less number of complements failing and including the graphs with 2 and 5 vertexes situation. But for larger graphs we can check that a lot of test cases had a good number of failing complements which means that we could reduced the test cases through them. Comparing the number of times that a test case failed in a complement and the vertexes reduction, we see that for test cases with 1001-50000 vertexes we have a high percentage of vertexes reduction which lead to a high number of failing complements. Meaning that Delta Debugging found a minimal test case for a lot of student’s projects through failing complements. Also, for some student’s projects, having a low number of failing complements is not a bad result of our algorithm or it does not mean that our test case reduction was low. In some situations we can affirm that our algorithm could not reduce a test case due to a bug only present in the original test case, but for other situations our algorithm was precise and found the minimal test case with a low number of reductions. Having a low number of failing complements also means that our software feature that does not allow large sized complements at the low values of the granularity is working with effectiveness since that we are testing a lot less complements in the beginning and reducing to smaller test cases instead of test cases with
Finally, our Delta Debugging algorithm have a test case cache that prevents a test case from being tested multiple times for a student submission. This is a optimization that our algorithm have towards the original Delta Debugging which remove unnecessary testing. To show that, we have a figure representing the number of times that this optimization was used per submission.

As you can see, in figures 4.13 and 4.14, for smaller test cases we cannot take conclusions since the number of subgraphs and complements tested are low, means that our number of test cases (subgraphs and complements) already tested will be low as well. Otherwise, for student's projects with a large failing test case, the number of test cases (subgraphs and complements) already tested is high which means that this feature make our software more efficient, in a way that we take less time to reach a minimal test case and skip unnecessary testing. Also help us getting more minimal test cases in a way that without it, some of the submissions would reach our time limit.

4.3 Test Case Generation Detailed Results

In this subsection, we show some results about our Morphing algorithm. Since we implemented a set of algorithm which Morphing is the main one and aims to generate a random graph so that when our original graph cannot be reduced to a minimal test case, we can try to reduce a morphing graph and deliver a similar graph that expose the bug to a student.

As a measure to see the utility of this feature, we show a image where is represented the number of times that a morphed test case was delivered to a student as a minimal test case. This will allow us to take conclusion about the relevancy that this algorithm has in our software.

In figures 4.15 and 4.16, analyzing it we conclude that minimal test cases coming from morphing graphs are also send to the students even if the number of times that this happen is low. The reasons that relies behind these low numbers are that for submissions with a small failing test cases, there are
a lot of cases where the original graph can be reduced to a minimal test case even when the morphing is used, the original test case is being lower in terms of vertexes and edges. For example, test cases with 2 vertexes are already minimal and do not even use morphing graphs. And for submissions with a large failing test case, since we put a time limit on our application, most of the submissions are not even using morph because our software takes a lot of time creating a morphed graph and are interrupted by the time limit and ending delivering the minimal test case coming from the original graph. From our point of view, this algorithm is relevant since it can give us more minimal test cases.

4.4 Test Case Simplification VS Test Case Generation

This subsection aims to compare our software (TCSS) with another software made by a master’s student which randomly creates graphs given some specifications such as the number of edges and the number of vertexes. The results of these tests, allows us to show that our software is more appropriate and efficient to find a minimal test case. Theoretically, our software should be more efficient since we test subgraphs and complements from an original graph that fails in the student’s project while the other software randomly creates graphs with a certain number of vertexes and edges which does not guarantee the same structure in terms of edges from the failing test case.

First thing we did was select, randomly, three student’s projects from a group that got a minimal test case between 10 to 50 vertexes. A minimal test case with less than 10 vertexes would not have many differences, in terms of structure, when generating a graph with the other student software and with more than 50 vertexes would take a lot of time for that software to test all the graphs since the way that it can find a minimal test case is to generate graphs starting from a minimum size graph and incrementing the number of vertexes, $n \in \{2, ..., 2 + |V|\}$, while testing all the number of edges that a graph with a certain number of vertexes can have, $m \in \{(n - 1), ..., \frac{n(n-1)}{2}\}$. Where $n$ is the number of vertexes, $m$
the number of edges and $|V|$ the number of vertexes of the minimal test case found by our TCSS. This software testing ends when it cannot find a bug in graphs that have less or equal $2 \times |V|$, where $|V|$ is the number of vertexes of the minimal test case found by our TCSS.

Our first student project has a failing test case with 70 vertexes and 120 edges. With TCSS, we obtained a minimal test case that has 11 vertexes and 11 edges while with the other software obtained a minimal test case with 12 vertexes and 11 edges. So in this student project we were able to provide a better feedback.

But in the second student project that has a failing test case with 50000 vertexes and 49999 edges the results were different. With TCSS, we obtained a minimal test case with 19 vertexes and 18 edges while the other software obtained as well a test case with 19 vertexes and 18 edges. So the feedback provided by both softwares was equal.

Finally in our last test, a student project that has a failing test case with 10000 vertexes and 60000 edges was reduced to a minimal test case with 11 vertexes and 10 edges using TCSS. Unlikely, the other software could not find a bug in graphs that it tested. This result was very positive to us since it shows our software potential.

Concluding about these results, we can observe that if the failing test case have a higher number of edges it is more likely that TCSS finds a smaller minimal test case. The reason behind this is related to the structure that the other software generates being a lot different from the structure present in the failing test case.
Figure 4.14: Relation between a test case number of test cases already tested and number of vertexes. (G2)

Figure 4.15: Graph reduced in order to obtain the minimal test case. (G1)
Figure 4.16: Graph reduced in order to obtain the minimal test case. (G2)
Chapter 5

Conclusions

In this section, we present the conclusions that we reach through the development of this thesis and the results that we obtained, showed in the previous chapter. First, we show how our test cases were reduced and the effects that this reduction takes in the student's performance towards the present course in this thesis, Analysis and Synthesis of Algorithms. Then, we will conclude why did we use Morphing algorithm to try to achieve a minimal test case even that we had Delta Debugging as test case simplification algorithm. Finally in a more external perspective we concluded the advantages that this software, TCSS, brings to a student community in a course project environment.

As a second part of this Chapter, we propose future work that can be done to improve TCSS or to modify it to achieve certain goals in different graphs, such as directed graphs and weighted graphs.

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 Chapter 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).

As the second point of this section, we have to talk about why the Morphing algorithm was important
even that we already had a test case simplification algorithm that could work on his own to reduced
test cases (Delta Debugging). As previously mentioned, the Morphing algorithm mixes edges from two
graphs and having a variable that allows this mixing to be more similar to one of the graphs. We knew
before testing this algorithm that this would help us out in failing test cases that could not be reduced
through Delta Debugging. Given this, we tried the Morphing as a algorithm that could create test Cases
similar to the original one but with sightly edges modifications which in some cases removing some
edges could lead to a vertex remotion as well. After testing this algorithm we conclude that is needed for
this work is advantageous in several cases. For a significant number of failing test cases, the minimal test
case was obtained by the Morphing algorithm which contributes to a better feedback. Also contributing
for a good feedback is the fact that some failing test cases could not be reduced and since we cannot
send an original test case to the student, the morphing algorithm allowed us to get a similar test case
which if it fails can already be send to the student and help him out solving his project bug. Otherwise,
Morphing is an algorithm that involves a huge number of edges searches in both graphs and edges
creations in the new morphed graph for big sized test cases. This can lead to a decrease of TCSS
efficiency.

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
5.1 Future Work

In this section we refer to possible improvements in TCSS or in which future softwares can it be included. 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.

In different aspects as the ones talked above, TCSS can also be improved in future work so that its efficiency can increase as for example trying a different approach on Delta Debugging when facing the student projects problems as the time that a buggy project can take to run with a certain test case. At the moment we only have a list of already tested graphs which ignores a lot of test cases but can also
be ignoring some test cases that were not tested as we mentioned before in Chapter 3. As well, we got a time limit of 2 seconds for each time that a student project is running a test case. This improves the efficiency of TCSS but we cannot assure that there are not better ways to solve these problems and lead to a perfect intensive testing to reach a minimal test case. Since that we talked about the test case simplification process we must also talk about about the test case generation which includes our Erdős–Rényi algorithm and the Morphing algorithm. This is where our application takes more time to process since the creation of a graph and a mix of edges between two graphs can lead to a lack of efficiency. As future work we advice that this algorithm should be reviewed and seen if there are better options as algorithms so that the time spent creating a graph can be shorter. Also, our criteria about when to use the Morphing can be modified because we cannot assure that this way is the perfect one to accomplish this work goals.

In a more outside point of view, as future work TCSS should integrated with another applications so that it facilitates the use of it. For example given that the students use Mooshak as an online platform to submit their projects, TCSS could work together with this platform in a way that the first failing test case would be TCSS input and instead of only present the results of the tests, if they pass or fail, this platform should be also showing a file where a minimal test case for the student first failing test case is displayed.
Bibliography


