Automatic Generation of Exercises for Massive Open Online Courses (MOOCs)

Fernando César Silva Teixeira dos Santos
fernando.cesar@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2015

Abstract

There has been a lot of research made on random graph generation, however most existing generators only offer a probabilistic expectation that a certain property will exist in the final graph. Therefore, this graphs are not adequate to be used as exercises in introductory algorithms courses since their properties may not relate to the algorithms being studied. We propose new algorithms that generate graphs parameterized by specific properties, specially selected for classes' exercises. Our algorithms generate unweighted, undirected, connected graphs and weighted, directed graphs. We evaluated our algorithms by implementing a set of tools which were used in an introductory algorithms' course lectured at our University. Our tools were used to evaluate the current automatic grading process, to generate the graphs used to grade students' solutions for curricular exercises and to find problems in those solutions with small inputs. The results were highly positive.
Keywords: algorithm, graph, graph generator, education, MOOC

1. Introduction

Never was the goal of wide spread knowledge so available to the common user as today, with massive open online courses (MOOC) taken by thousands of students worldwide. However, the world classroom also faces new dilemmas. Tutors must provide exercises for thousands of students, and students sometimes need additional practice.

Graphs are an important data structure in an introductory algorithm's course. From this perspective, the generation of thousands of different graphs for students' exercises is a daunting task, since existing tools are aimed at very large networks and not at classroom exercises.

We start this paper with a terminology section. Then, we review the state-of-the-art. In the following section we propose our new algorithms. Afterwards, we describe the evaluation or our work. The paper finishes with the conclusions and some topics for future work.

2. Preliminaries

In this section we explain the most important terminology used in this paper. The definitions mentioned here are generally accepted in the field of study of algorithms and can be found in any academic manual of reference (see, for instance, [12]).

A *graph* G is a pair (V, E), where V denotes a finite set of vertices and E a set of relations on V, each one referred to as an edge. In an undirected graph, the edge set E holds unordered pairs, while in a directed graph each pair is ordered.

A *path* p from vertex u to vertex x is a sequence of vertices $\langle v_0, v_1, \ldots, v_k \rangle$, such that $u = v_0, x = v_k$ and $(v_{i-1}, v_i) \in E$, for $i = 1, 2, \ldots, k$. If there exists a path p from u to x, we say that x is *reachable* from u, via p. This is represented $u \rightsquigarrow x$. The *length* of a path is the number of edges in the path. The *weight of an edge* is a function that maps edges to real-values. The *weight of a path* $w(p)$ or $w(v_0, v_k)$ is the sum of the weights of its edges $\sum_{i=1}^{k} w(v_{i-1}, v_i)$. In an unweighted graph we assume every edge has weight 1.

A path $\langle v_0, v_1, \ldots, v_k \rangle$ forms a *cycle* if $v_0 = v_k$ and $k \geq 2$ (for a directed graph), or $k \geq 3$ (for an undirected graph). A graph with no cycles is called *acyclic*. We call a cycle with a length of 1 a *self-loop*. A cycle c with a negative weight $w(c) < 0$ is called a *negative-weight cycle*.

A *shortest path* from u to v is a path p with minimum weight $w(p)$. Let $\delta(u, v)$ denote the weight of the shortest path from vertex u to vertex v. When there is no path from u to v, we have $\delta(u, v) = \infty$.

An undirected graph is *connected* if every vertex is reachable from all other vertices. A *tree* is a connected, acyclic, undirected graph. A *rooted tree* is a tree where one of the vertices is distinguished from

the others by calling it the *root* of the tree.

A *bipartite graph* is an undirected graph in which V can be partitioned into two sets V_1 and V_2 and each edge $(u, v) \in E$ connects a vertex in V_1 to a vertex in V_2.

3. State-of-the-art

In this section we review the state-of-the-art. Since the graphs our algorithms generate will be used as exercises in introductory algorithms courses, we first review the work done in automatic generation of exercises.

Although much research has been done in this area, the topic of algorithms has not yet been focused. McArthur et al. describe an Intelligent Algebra Tutor, which creates new problems based on a provided grammar [29]. In the field of Theory of Computation, Tscherter creates the Exorciser System, which automatically generates exercises and also comments for students, regarding the solving process [39]. Singh et al. describe a new approach for generating random algebra problems similar to a given one [36]. For generating problems for natural deduction, Ahmed et al. describe two alternatives: producing problems similar to a template and producing parameterized problems [1].

One recurring problem in automating the generation of exercises is how to adapt the level of difficulty of such problems to students expertise [29, 5]. Andersen et al. describe a trace-based framework for analyzing the complexity of problems, exemplifying their work with early mathematics education and level generation for an interactive puzzle game [5].

A second field of interest is the automatic assessment of programing exercises, since our goal is precisely the generation of graphs to be used in programming exercises. Automated systems for assessing programming exercises seem to be, at least, as old as exercise assessment itself. For a survey with an historical perspective between 1960 and 2005, see Douce et al. [13]. For a more feature focused perspective see Ala-Mutka [3], till 2005 and Ihantola et al. [21] from 2005 to 2010.

One of the first references on this subject is from Hollingsworth [20]. The 1960 Grader System produces a simple outcome: "wrong answer" or "program complete". Later that same decade, Hext and Winings [18] address some new topics, such as students' plagiarism, stored test data and summaries of execution results with their Basser Automatic Grading Scheme. The TRY System appears in the late eighties [34], focusing on making tests also timely available to students and avoiding data corruption by (innocent or malicious) students' programs. In 1997, Jackson and Usher present the ASsessment SYSTem, which allows weights to be assigned to particular aspects of tests, thus using a scaling approach in awarding marks. Finally, in 2013 Alur et al. also focus on assigning partial grades for incorrect answers [4], but their final aim is to provide meaningful feedback to students. From the same year, Pex4Fun is an automatic white-box test generation tool for .NET [38]. This system tries to create inputs to cross every execution path, effectively tackling the problem of scalability in grading and providing personalized feedback for programming assignments.

Besides the creation of assessment system, research also concentrate on the problems facing these systems. Glassman et al. investigate the predisposition of teaching staff and automatic assistants to direct students to one specific solution [16], instead of pursuing the solution intended by the student. Singh et al. propose a new method to provide detailed feedback to MOOC Students [37], based on a reference implementation provided by the instructor and on an error model, which describes potential corrections to errors that students might make. Finally, gamification (the use of game design elements in non-game contexts) has been used to teach programming [15] to under college students [9] and other broader audiences [19]. In this case, all the interaction with the student is automatic, from the presentation of the problem, to the solution assessment.

Lastly, we go over random graph generators. The research done on the random generation of graphs focuses primarily on very large networks, such as social networks or the World Wide Web. The study of this type of graphs is out of scope in this paper, since our goal is not to replicate real world scenarios. Additionally, these models usually focus on a set of properties [2] different from the one relevant to this work. Nevertheless, the generators for this kind of models should be mentioned, since they also have the goal of generating random graphs and might, therefore, present good leads to our work.

Erdős and Rényi are generally pointed as pioneers in this field [10]. Their seminal work dates back to 1959, when they created a model for generating graphs that takes their names [32]. This model is still considered the simplest way to generate a random graph: given n vertices, an edge (u, v) is added to the graph with probability p.

There has been a lot of research done on the transition phase of p [7, 8, 23, 26], i.e., the value of p for which the probability of certain property to exist, in every generated graph, approaches 1 as $N \to \infty$ (with N referring to the number of edges). Some of the properties which have been studied are connectedness [35], the size of the largest component [6, 28, 25, 30], or the bounded average degree [14, 31].

One recent discovery in graph theory is that many networks with complex topology have a common property: the vertex connectivities follows a scale-free power-law distribution. Although several algorithms have been used to generate graphs associated with the power-law model, we can distinguish between two main approaches: the curve fitting family and the preferential attachment family [17]. In the first case, the algorithms receives as input a sequence of a scale-free degree distribution $D = \langle d_1, d_2, \ldots, d_n \rangle$. Each vertex is assigned a value from D, representing its degree. Next, random edges are generated, but they are only added to the graph if their addition does not exceed the pre-established degree value [33].

In the second approach (the preferential attachment family), the algorithms start with a small, fully connected graph. Then, an incremental process follows, where at each time-step one vertex is added to the network, and connected to m random vertices $R = \langle v_1, \ldots, v_k \rangle$, with probability proportional to v degree [11].

Besides the preciously mentioned, some algorithms do exist to generate graphs not necessarily related to very large networks.

To generate a random bipartite graph, Kannan et al. [24] suggest applying alterations to an existing bipartite graph G. The main idea is to select two random edges (t, u) and (v, x) and swap both vertices from one partition, so maintaining the bipartite graph property and two new edges: (t, x) and (v, u). This is called "switching".

Algorithm 1: Obtain a random spanning tree $T = (V, E)$ with prescribed root r

Input: Root vertex r, number of vertices n
Output: Random spanning tree (V, E) with root r plus $n - 1$ vertices

```
1  V ← {r}
2  E ← ∅
3  foreach v_i ∈ {v_1, ..., v_{n-1}} do
4      savedV ← v_i
5      while v_i ∉ V do
6          Next[v_i] ← randomVertex()
7          v_i ← Next[v_i]
8      v_i ← savedV
9      while v_i ∉ V do
10         V ← V ∪ {v_i}
11         E ← E ∪ {(v_i, Next[v_i])}
12         v_i ← Next[v_i]
13 return (V, E)
```

In this paper, we use a known algorithm to generate a random rooted tree. A random rooted tree $T = (V, E)$, with n vertices, can be created with a random walk, as described by Wilson [40] and represented in pseudocode in algorithm 1. The tree $T = (V, E)$ starts with only one vertex, called root, r (line 1). All the remaining vertices $(v_1, v_2, \ldots, v_{n-1})$ are then cycled through (lines 3–12), and for each $\{v_i : i = (1, 2, \ldots, n-1)\}$, two steps are taken: a random walk is performed (lines 5–7) and the resulting path is added to the tree (lines 9–12).

Both steps need to start at the same vertex and so, v_i is saved (line 4) for later use (line 8). The random walk is performed by, first, randomly selecting a vertex v_j with the `randomVertex()` function. Then, a pointer is added from the previous vertex to the latter: $Next[v_i] \leftarrow v_j$ (line 6). This random step is carried out until a vertex in the tree is found $v_j \in V$, as shown in lines 5 and 9.

Cycles are ingeniously avoided. Since each vertex only keeps one pointer, a second pass on the same vertex assigns the *Next* pointer to another random vertex, thus avoiding any cycles (including self-loops).

In the second step of the algorithm (lines 9–12), the random walk is again visited to assign the new path to the tree (in lines 10 and 11 vertices and edges are added).

4. Parameterized Graph Generation

We propose the random generation of graphs parameterized by specific properties, thus enabling their use for specific academic goals.

4.1. Unweighted Undirected Graphs

In order to generate an unweighted undirected connected graph, we start by using the algorithm described by Wilson [40] to generate a random rooted tree. Further edges are then added randomly without breaking the desired properties. As a result, every generated graph will be a connected graph.

4.1.1 Parameterizable Properties

Our algorithm generates a random graph (V, E) with the following parameterizable properties:

1. Number of vertices, n

2. Number of edges, m
   This property allows the user to determine the number of edges of the graph. These two first properties allow the user to control the size and density of the graph, however, the number of vertices is interpreted as mandatory while the number of edges is not. The algorithm will return a graph with a different number of edges only if it is not able to comply with the user's request, either because no such graph exists, or because one of the other properties would be violated.

3. Existence of a shortest path p with a minimum length, *minSp*

This property ensures that there exists a shortest path between a given vertex v_{root} and a vertex v, of length $\delta(v_{root}, v)$ at least *minSp*.

$$\exists v \in V : \delta(v_{root}, v) \geq minSp \quad (1)$$

4. Maximum length *maxSp* of all the shortest paths from a single source v_{root}
   In this case, we bound all the shortest paths from a single source (the root v_{root}) to a maximum length *maxSp*.

$$\forall v \in V : \delta(v_{root}, v) \leq maxSp \quad (2)$$

It should be stressed that not all combinations of properties and their values are able to generate a graph. A simple example is an undirected graph with 2 vertices and 2 edges.

4.1.2 Algorithm Description

From a high level perspective, this algorithm creates a graph G that respects the previously mentioned properties, and then adds additional edges, meanwhile checking at each iteration if every property is still correctly enforced.

The algorithm consists of three stages: first, it creates a path $sp = \langle v_{root}, v_1, \dots v_{minSp} \rangle$, with *minSp* size: $\delta(v_{root}, v_{minSp}) = minSp$. Then, algorithm 2 includes sp in a tree G connecting all the vertices V. Finally, algorithm 3 adds the remaining edges to G.

Safe-Path. The algorithm first creates a path $sp = \langle v_{root}, v_1 \dots v_{minSp} \rangle$ with *minSp* length: $\delta(v_{root}, v_{minSp}) = minSp$. This path is safeguarded to maintain its length: in order to respect property 3, $\delta(v_{root}, v_{minSp})$ can not become smaller. As the algorithm ensures this path maintains its properties, we call it the "safe-path". To create a safe-path $sp = \langle v_{root}, v_1 \dots v_{minSp} \rangle$, the algorithm begins by adding vertex v_{root} to sp. The remaining $minSp - 1$ vertices are then randomly selected and also added to sp, making sure that no duplicated vertices are added. This algorithm, establishes property 3, generating a *minSp* length path.

Tree. After the safe-path sp has been created, algorithm 2 includes sp in a tree T containing all the vertices $v_i \in V$ of G. Algorithm 2 is adapted from algorithm 1, described in section 3. We choose this algorithm because it is a known way to generate a random spanning tree.

In algorithm 2 we use different data structures from the original algorithm description. T represents every vertex v_i that is already reachable from v_{root}: $\{v_i \in V : v_{root} \rightsquigarrow v_i\}$. On the contrary, T' groups every vertex which is not reachable from v_{root}.

Algorithm 2: Create a tree with a safe-path (adapted from Wilson [40]).

Input: Graph (V, E) where $\exists sp = \langle v_{root}, v_1, \dots, v_{minSp} \rangle : \delta(v_{root}, v_{minSp}) \geq minSp$, root vertex v_{root}, maximum length $maxSp$

Output: Tree (V, E) where $\exists sp = \langle v_{root}, v_1, \dots, v_{minSp} \rangle : \delta(v_{root}, v_{minSp}) \geq minSp$, and $\forall v_i \in V : \delta(v_{root}, v_i) \leq maxSp$

```
1  T ← {v_i : v_i ∈ sp}
2  T' ← V \ T
3  while T' ≠ ∅ do
4      v_i ← randomSelect(T')
5      savedV ← v_i
6      while v_i ∉ T do
7          Next[v_i] ← randomVertex()
8          v_i ← Next[v_i]
9      v_i ← savedV
10     newPath ← list()
11     while v_i ∉ T do
12         newPath.push-front(v_i)
13         v_i ← Next[v_i]
14     newPath.push-front(v_i)
15     addableVertices
       ← min(newPath.length(), maxSp − δ(v_root, v_i) + 1)
16     foreach j in range [0, addableVertices − 1] do
17         T ← T ∪ {newPath[j + 1]}
18         T' ← T' \ {newPath[j + 1]}
19         E ← E ∪ {(newPath[j], newPath[j + 1])}
20 return (V, E)
```

The algorithm has four cycles. The outer cycle (lines 3–19) simply iterates over every vertex $v_i \in T'$. To do this, we random select $v_i \in T'$ (line 4) while T' is not empty (line 3). The algorithm ends when every vertex $v_i \in V$ is reachable from v_{root}: $\forall v_i \in V : v_i \in T$. The three inner cycles implement the algorithm main loop.

In the first inner cycle (lines 6–8), a random walk creates a new acyclic path. Function `randomVertex()`, in line 7, can randomly select any vertex, since the algorithm takes care of avoiding cycles and self-loops. The vertices are temporary kept in the *Next*[ ] vector. This is a vector of pointers, each pointing to the next vertex on the random walk. Eventually, a $v_i \in T$ is added to the random walk. This vertex v_i connects the tree to this the new path.

In the second inner cycle (lines 11–13), the random walk path is again traverse. We begin in the same initial vertex as before, which was saved (line 5) and follow the vector of *Next*[ ] pointers. The vertices of the random walk are added to the front of a new list (line 12), called *newPath* (line 10). Again, the cycle ends when $v_i \in T$ is found. This vertex is also added to *newPath* (line 14) and is now

the first vertex of this list *newPath*[0].

In the last cycle (lines 16–19), we again traverse the random walk path, but this time in the reverse order, since *newPath* was constructed this way. We start at $v_i : v_i = newPath[j] \wedge j \leftarrow 0 \wedge v_i \in T$, and finish either when *newPath* ends, or when $\delta(v_{root}, newPath[j]) > maxSp$. To determine which one happens first (variable *addableVertices*), we calculate the maximum number of vertices from *newPath* that can be added to T, respecting property item 4. This can either be all the vertices from *newPath* (*newPath*.`length()`, line 15) or the difference between *maxSp* and $\delta(v_{root}, v_i)$ ($v_i \in T$ is the vertex that connect the *newPath* to the tree; line 15). The +1 addition compensates for the fact that v_i, which is already in T, is also in *newPath*.

An advantage of using a tree is that determining the length of every shortest path from v_{root} is simpler, since there is only one path from v_{root} to any $v_i \in T$. At each *newPath*[j], we update $\delta(v_{root}, newPath[j])$, add the vertex to T (line 17), remove it from T' (line 18) and the edge to E (line 17). The -1 in line 16 and +1 in line 17 and line 19 serve to ignore the first vertex $v_i : v_i = newPath[0] \wedge v_i \in T$.

In order to include *sp* in the tree, *sp* is given to the algorithm as an existing path (line 1), similarly to any other random walk.

Ensuring property 3 is simpler, since adding vertices and edges to a tree does not change already existent shortest paths: every new path added to the tree will add new shortest paths from v_{root} to new vertices without changing the shortest paths to vertices already in T. This way, property 3 is ensured by construction and does not need to be verified in this stage.

Remaining Edges. Following the previous described algorithms, we generate a graph that complies with most of the properties described before (subsubsection 4.1.1), with the possible exception of the number of edges. The generated graph is a connected tree and, as such, it has exactly $|V| - 1$ edges. Since our algorithm always starts with a connected tree, if the user requested a lower number of edges, the graph is finished. In our implementation, the user is informed that the graph has a different number of edges from the one requested.

However, the user may have requested a higher number of edges. In that case, algorithm 3 is responsible for adding the remaining requested edges, while respecting all the other properties the user selected. This can be problematic, since some sets of properties are not possible to translate to a graph.

In order to ensure termination, and still comply as close as possible with the user's parameters, we impose a limit t (for tries), on the number of failed attempts to add a new edge to the graph.

Algorithm 3: Add edges to graph preserving at least one shortest path of length *minSp*.

Input: Graph $G = (V, E)$ where $\exists v \in V : \delta(v_{root}, v) \geq minSp$, root vertex v_{root}, number of edges m, minimum shortest path length *minSp*

Output: Graph $G = (V, E)$ where $\exists v \in V : \delta(v_{root}, v) \geq minSp$ and $|E| = m$

```
1 while |E| ≤ m do
2     (u, v) ← randomEdge()
3     if (u, v) ∉ E then
4         E ← E ∪ {(u, v)}
5         BFS(G, v_root)
6         if ∀v ∈ V : δ(v_root, v) < minSp then
7             E ← E \ {(u, v)}
8 return G
```

Since the maximum number of edges is also determined by previous decisions in the graph generation process, if the requested number of edges is not achieved, the algorithm is run from the beginning *ta* times, creating a completely new random generated graph. To summarize, in the worst case, $ta \times t$ failed attempts to add a new edge are made. In the end, the application returns the graph with the number of edges closest to the number of edges requested by the user and informs the final number of edges of the graph.

The main concern, when adding new edges to the graph, is with property 3. In fact, since we generate a connected tree, no existing shortest path from root $\delta_{original}(v_{root}, v_i)$ can be increased by adding new edges. All new shortest paths $\delta_{new}(v_{root}, v_i)$, resulting from new added edges, have the same or a lower length: $\forall v_i : \delta_{new}(v_{root}, v_i) \leq \delta_{original}(v_{root}, v_i)$. This is a consequence of having a graph where every vertex is already reachable from v_{root}. For this reason, the algorithm does not need to actively enforce property 4, as it will naturally be respected.

Hence, the goal of algorithm 3 is to extend a graph $G = (V, E)$ with additional edges, such that it maintains an existing shortest path of length *minSp*, starting in the root vertex v_{root}.

In algorithm 3, for each potential new edge (u, v) (line 2), one applies a BFS traversal (line 5) in order to guarantee that at least one shortest path of length *minSp* is maintained (line 6). If the shortest path length is maintained, edge (u, v) is added to the graph (line 4). Otherwise, (u, v) is discarded (line 7). The algorithm terminates when the number of edges in G reaches the specified value (line 1). To keep the pseudocode simpler, we omit the representation of testing for t and ta.

4.2. Weighted Directed Graphs

In this section we describe a new set of algorithms to generate random weighted directed graphs.

These algorithms always start by creating a graph based on the algorithm described by Wilson [40] to generate a random (undirected) tree. Our algorithms are then responsible for direction, weight, and additional edges.

4.2.1 Parameterizable Properties

1. Number of vertices, n
   See subsubsection 4.1.1, item 1.

2. Number of edges, m
   See subsubsection 4.1.1, item 2.

3. Range of values for weights of the edges, $[minw, maxw]$
   This property determines the range of possible values for the weights of the edges, which will be randomly chosen.

4. Existence of negative-weight cycles, nc
   This property determines the existence (or nonexistence) of negative-weight cycles reachable from vertex v_{root}. If the user requests a negative-cycle but the graph only has 1 vertex reachable from the v_{root} no cycle is added (our implementation warns the user of such situation).

5. Probability of paths not reachable from the root vertex, *reach*
   This property determines the probability of the generated paths being reachable from the source vertex, v_{root}. The user has always the possibility to determine that all or none of the vertices are reachable from v_{root}.

4.2.2 Algorithm Description

Our algorithm follows three steps to generate a weighted directed random graph. It begins by creating what would be a tree in an undirected graph. However, this time the edges will be directed. In the next step, remaining edges are added. Finally, weights are associated with every edge.

Creating a tree with direction To create the initial version of the graph, algorithm 4 again makes use of an adaptation of the algorithm described in section 3, authored by Wilson [40]. In our adaptation, each new path created will have a direction (lines 16 and 18).

The direction of the edges is used to enforce the reachability property (property 5), making a new path reachable, or not reachable, from v_{root}. For the vertices to be reachable, the edges are given a direction from v_{root} to the leafs (line 16), otherwise, the direction will be from the leafs to v_{root} (line 18). The decision whether the new path is reachable or not is taken randomly, according to a parameter passed by the user between 0 and 1, determining how likely a path is to be reachable from v_{root} (used in line 4).

Algorithm 4: Create a directed tree, adapted from Wilson [40].

Input: Root vertex v_{root}, number of vertices n, probability *reach* of new paths being reachable from v_{root}
Output: Directed graph $(V_r + V_u, E)$ where $\forall v \in V_r : v_{root} \rightsquigarrow v$ and $\forall v \in V_u : v_{root} \not\rightsquigarrow v$

```
1  V ← {v_root}
2  E ← ∅
3  foreach v_i : 1 ≤ i < n do
4      reachability ← randomBool(reach)
5      savedV ← v_i
6      while v_i ∉ V do
7          if reachability == true then
8              Next[v_i] ← randomReachableVertex()
9          else
10             Next[v_i] ← randomVertex()
11         v_i ← Next[v_i]
12     v_i ← savedV
13     while v_i ∉ V do
14         V ← V ∪ {v_i}
15         if reachability == true then
16             E ← E ∪ {(Next[v_i], v_i)}
17         else
18             E ← E ∪ {v_i, (Next[v_i])}
19         v_i ← Next[v_i]
20 return (V, E)
```

An important facet of the algorithm is that a new reachable path can only be connected to an already reachable vertex. If the vertex connecting the new path to the graph is itself not reachable from v_{root}, every vertex from the new path is, also, unreachable from v_{root}. In algorithm 4 this is represented by calling function `randomReachableVertex()` (line 8) which returns a random reachable vertex or a vertex not yet connected by an edge, while function `randomVertex()` (line 10) returns any random vertex.

Adding Remaining Edges If the parameter given for the number of edges is still not reached, more edges are randomly added, preserving the existing condition of each vertex: unreachable vertices remain unreachable. Two random vertices v_0 and v_1 are selected, and a new edge is added between the two. Table 1 shows how the direction of the new edge is selected to accomplish the described reachability goal.

Table 1: New edges direction.

v_0	v_1	new edge direction
reachable	reachable	indifferent
reachable	unreachable	$v_1 \rightarrow v_0$
unreachable	reachable	$v_0 \rightarrow v_1$
unreachable	unreachable	indifferent

Weights Attribution Since the final graph is a weighted graph, weights must be associated with each edge. The user selects the weight range that is used (a consecutive interval of integers, from a minimum value, *minw*, to a maximum value, *maxw*). The difficulty in assigning weights to edges is related to another option available to the user: the existence or nonexistence of negative-weight cycles (notice that only cycles reachable from v_{root} are considered). Regarding this last property (existence of negative-weight cycles), three options are available to the user:

1. *random-cycles*
   The weights are attributed randomly and the generation of negative-weight cycles is not controlled.

2. *no-cycles*
   The algorithm must produce a graph without negative-weight cycles.

3. *has-cycles*
   The algorithm must produce a graph with, at least, one negative-weight cycle.

If the user decides for *no-cycles*, algorithm 6 may be used to achieve that end. Otherwise, a random weight, in the selected range, is associated with each edge. If the user selected the *random-cycles* option regarding negative cycles, the algorithm finishes. When the user decides for the *has-cycles*, additional steps are needed. First, Bellman-Ford's algorithm is run on the graph to search for existing negative-weight cycles. Bellman-Ford algorithm returns a pair, consisting of a boolean *status* result, and, case the status is true, a negative-weight cycle (represented by the variable *cycle*). If such a cycle is found, the graph complies with the user request and the algorithm finishes. If no negative-weight cycle is found, a depth first search (DFS) is performed to find an existing random generated non-negative-weight cycle (this function returns a pair, once again, consisting of a boolean status result and a possible *cycle*). If found, it is transformed to a negative-weight cycle (see paragraph "Transforming a Cycle" below). Finally, if no cycle is found, a new cycle is generated (algorithm 5) using only reachable vertices. In case there are less than two reachable vertices our implementation exits and warns the user that it is not possible to generate a negative-weight cycle.

Generating a Negative-Weight Cycle. To generate a negative-weight cycle, in a graph where none exists, we use algorithm 5. First, `randomReachableVertex()` randomly selects two different vertices v_0 and v_1 (lines 1–4), reachable from v_{root}, and adds them to a path p (line 5). A third vertex is necessary to create a cycle, however it must be different from v_1, or we would create a self-loop (line 6). Now we check if v_j is already in p, for that would mean we created a cycle. Repeatedly, random vertices v_j are added to p (lines 11–10) until a repeated vertex is selected ($v_j \in p$), creating a cycle (line 9). We add the last vertex to p to close the cycle (line 12) and assign random weights to every edge in the new path (lines 13–14). Finally, the cycle is weighted $w(p)$ (line 15). If $w(p) < 0$, the algorithm finishes, otherwise, the cycle is transformed (line 16, further detailed below in Transforming a Cycle).

Algorithm 5: Create a reachable negative-weight cycle.

Input: Graph (V, E), range of admissible weights $[minw, maxw]$
Output: Graph (V, E) with at least one negative-weight cycle p

```
1  repeat
2  |  v0 ← randomReachableVertex()
3  |  v1 ← randomReachableVertex()
4  until v0 ≠ v1
5  p ← ⟨v0, v1⟩
6  repeat
7  |  vj ← randomReachableVertex()
8  until vj ≠ v1
9  while vj ∉ p do
10 |  p ← p ∪ {vj}
11 |  vj ← randomReachableVertex()
12 p ← p ∪ {vj}
13 for vi ∈ p do
14 |  (vi, vi+1) ← randomWeight(minw, maxw)
15 if w(p) ≥ 0 then
16 |  transformCycle(p)
17 return (V, E)
```

Transforming a Cycle. To transform a negative-weight cycle into a non-negative one (and *vice versa*), we randomly select an edge and give it a new weight. If the new weight helps the intended goal, it is kept, otherwise the previous value is restored. So, if the goal is to transform a non-negative-weight cycle into a negative one, every weight which is lower than the existent is kept, and the rest are ignored. This algorithm keeps iterating until a non-negative-weight cycle is reached.

Weights Attribution Algorithm. To attribute weights, algorithm 6 begins by attributing only positive weights to every edge (line 2). Then, a percentage of edges, proportional to the negative segment of the weight range requested by the user (line 4), is potentially turned negative (lines 5–11). For instance, considering the range $[-5, 15]$, a quarter of the edges are potentially turned negative. A negative value is associated to each one of these edges (line 8), and a Bellman-Ford algorithm is run on the graph (line 9). If no negative-weight cycle is found, the new value is attributed to the selected edge, otherwise the previous value is restored (line 11).

Algorithm 6: Attribute weights without negative-weight cycles.

Input: Graph $G = (V, E)$ without weights, range of admissible weights $[minw, maxw]$
Output: Weighted graph $G = (V, E)$, without negative-weight cycles

```
1  for e ∈ E do
2      e ← randomWeight(0, maxw)
3  i ← 0
4  j ∝ [minw, 0[
5  for i < j do
6      (u, v) ← randomEdge()
7      w' ← w((u, v))
8      w(u, v) ← negativeValue()
       // if a negative-weight cycles is found
9      (status, cycle) ← Bellman-Ford(G, w, v_root)
10     if status == true then
11         w((u, v)) ← w'
12 return G
```

5. Evaluation

5.1. Evaluation Data

The evaluation of our algorithms and implemented tools was accomplished in a real scenario, using data from the Analysis and Synthesis of Algorithms course, in the second semester of 2014/2015. This course is taught in the second year of the Information Systems and Computer Engineering Bachelor's Degree at Instituto Superior Técnico. The students' evaluation method includes two programming projects, besides tests and other exercises. Each project consist in a problem, provided by the faculty, that the students must solve with an application, using the knowledge acquired in classes.

The hardware used to run the evaluation was a 64-bit AMD Opteron Processor 6172, running at 2.1 GHz, with 24 CPU's, and 64 Gb of RAM, running Linux (Fedora release 13, Goddard). We used the GNU C/C++ compiler version 4.4.5.

5.2. Evaluation Model

5.2.1 Code Coverage

Towards evaluating the contributions to tutors' work, we analyze how the graphs generated by our tools can replace those created by faculty members as input for students' exercises and evaluation.

To ascertain the value of the input provided by our tool, a code coverage tool was used. The purpose of the code coverage tool is to measure how much of the students code is executed using the generated graphs as input. The goal is that our tool generates graphs that will execute the maximum amount of students' source code.

The tool selected for ascertain code coverage was GCC gcov [27], the GNU coverage tool. The percentage of code coverage we present measures all the cumulative code that was executed by running students' solutions with the input of the 16 graphs used for the grading process of each project.

5.2.2 Smallest Graph to Find a Problem

In order to evaluate the contribution of our tool to help students, we analyze how it can be used to debug their applications.

Our random graphs generators were made available to the students to provide them with a way to generate input examples during the implementation phase of their solutions.

To evaluate the value of our tools when used by the students, we determine what is the smallest graph the tools generate that makes the student's solution fail. Each graph is used as input to the students' solutions and behaves as a test. If a student's solution fails this test it means it has a problem, and can be improved. The goal is to create the smallest graph able to identify a problem in students' solutions, meaning, a graph small enough that students could use in a paper and pencil scenario to manually retrace their algorithms. This was implemented by successively testing bigger graphs, starting at the smallest one possible.

First project In order to search for the minimum graph to find a problem, we created a tuple symbolizing all the varying properties that parameterize our tool to generate an unweighted undirected graph: $G = \langle n, m, minSp, maxSp, seed \rangle$. G represents the final graph, and the properties are, respectively, the number of vertices, the number of edges, the minimum shortest path, the maximum shortest path and the seed.

These properties have all been described before (subsubsection 4.1.1) except for the seed value. The seed is an integer number, used to initialize the random number generator in order to make the results of the random generation repeatable.

In short, we began testing each student's application with graph $\langle 2, 1, 1, 1, 1 \rangle$ and ended when a problem was found, or when we reached 20 vertices. In order to keep the graphs small, this was the limit

set (20 vertices). This also determines the limit of the other parameters. The maximum number of edges is based on the number of vertices $\frac{n(n-1)}{2}$. The minimum shortest path is bounded by 1 and by the number of vertices $(n-1)$. The maximum shortest path has as minimum and maximum bounds, respectively, the minimum shortest path value ($minSp$) and $n-1$.

A pseudocode representation of the search algorithm is shown in algorithm 7 (specially lines 1 to 5). The search is performed in lexicographic order, by generating every possible graph from $\langle 2,1,1,1,1\rangle$ to $\langle 20, \frac{n(n-1)}{2}, n-1, n-1, s)\rangle$, considering the given parameter order and excluding impossible options. A simple search algorithm was used, were in each iteration only one parameter is changed. As an example, the first graphs to be generated were $\langle 2,1,1,1,1\rangle$, $\langle 2,1,1,1,97\rangle$, $\langle 2,1,1,1,1000\rangle$ and $\langle 2,1,1,2,1\rangle$.

As shown in algorithm 7, after the new graph is generated (line 6), the correct solution is obtained by running the reference implementation with that graph (line 7). Then, every students' solutions is tested with the same graph (lines 8–11). If it produces the correct output, the algorithm continues to the next student's solution. Otherwise, the failing student's solution and the graph that found the problem are registered (line 10), and the student solution is removed from the set of students' solutions (line 11). The algorithm then continues to the next student solution. When all students' solutions are tested with this graph, the algorithm proceeds to generate a new graph.

This search pattern creates several biases that must be taken into account in the analysis of the results. However, at least one of the bias has a positive outcome. This is the case of the bias towards smaller graphs. The concept of smaller is used here in conjunction with the lexical order that was chosen to perform the search, and not as general definition of a small graph.

Second project In the second project we follow the same approach. The varying properties are different, although there are some that remain the same: $G = \langle n, m, minw, maxw, nc, reach, seed\rangle$. G represents the final graph, and the properties are, respectively, the number of vertices, the number of edges, the minimum weight of an edge, the maximum weight of an edge, the existence of negative-weight cycles, the existence of unreachable vertices and, finally, the seed. These properties have all been described before (subsubsection 4.2.1) except for the seed value (explained in the previous topic about the first project).

The weight ranges selected for testing were $[-10, 10]$, $[-1000, 1000]$ (both with negative-weight cycles and without), $[0, 200]$ and $[-200, -1]$ (both without imposition regarding negative-weight cycles).

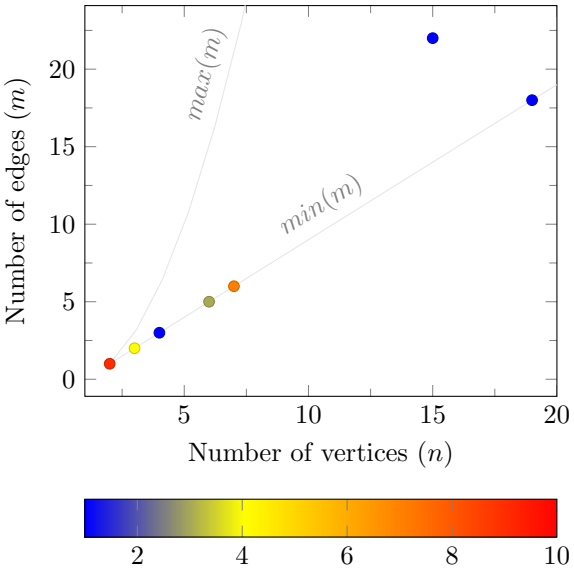

Figure 1: Frequency of problems found by graphs with n, m for project 1.

5.3. Results

5.3.1 First project

We analyzed 227 students' solutions pertaining the first project.

Code Coverage The coverage tool reported an average value of 92.15% coverage of students' solutions code, for the first project, after running the 16 graphs used for the students' grading process. There were no values below 65%. These results indicate that our tool generates graphs that tutors can use to evaluate students code.

Smallest graph In graphs with a maximum of 20 vertices, we found 26 solutions with problems, representing 11.40% of solutions, from a total of 227 solutions delivered by the students.

Although this seems a very low figure, we must remember that we can not pinpoint exactly how many students' solutions indeed have problems. Comparatively to the automatic grading process, where 48 students' solution failed at least one test, our search found 54.11% of these problematic solutions.

We notice that a total of 96.17% of students' solutions which were found to have a problem, failed with graphs that are trees. In fact only one of the listed graphs is not a tree. As such, we can hypothesize that varying the number of vertices is the single most important factor to finding problems in this project.

Algorithm 7: Lexicographic search of smallest graph to find a problem for project 1.

```
Input: Set of all students' solutions S
Output: List of graphs L with n ≤ 20 that find problems in S
for n ∈ {2, ..., 20} do
    for m ∈ {(n − 1), ..., n(n−1)/2} do
        for minSp ∈ {1, ..., n − 1} do
            for maxSp ∈ {minSp, ..., n − 1} do
                for seed ∈ {1, 97, 1000} do
                    graph ← GraphGenerator(n,m,minSp,maxSp,seed)
                    correct output ← ReferenceSolution(graph)
                    for StudentsSolution() ∈ S do
                        if StudentsSolution(graph)! = correct output then
                            L ← L ∪ register(StudentsSolution(),graph)
                            S ← S \ {StudentsSolution()}
return L
```

We take note of this trend in Figure 1. In this chart, every graph that found a problem is represented by a dot. The number of vertices n and the number of edges m are represented by the x and y axis, respectively, and the color of the dot represents the quantity of problems found by graphs with n vertices and m edges. The limit line identified with $min(m)$ signals dots with $(m = n - 1)$. It is noticeable that, with one exception, all the dots are aligned along the "tree" line.

5.3.2 Second project

There were 221 students' solutions analyzed for the second project.

Code coverage In the second project, the coverage tool reported a 96.17% coverage for students' solution, after running the 16 tests used for grading. Notice that there are no values below 55%.

Smallest graph In this second project, we found 141 solutions with at least one problem, representing 63.80% of a total of 221 analyzed students' solutions.

Comparatively to the automatic grading process, where 91 students' solutions failed to produce the correct output for at least one of the inputs, this represents a 154.95% increase. Note however, that the automatic grading system was not bounded to a limit of 20 vertices.

6. Conclusions

The automatic generation of exercises for students is a topic that has received some attention in the last decades. However, the area of graph generation for introductory algorithm courses has not yet been addressed. Graph generation on itself has received a lot of attention, but most research is done in relation to the Erdős-Rényi model and very large networks, such as social networks or the Internet. Hence, the algorithms proposed thus far are not adequate to exercises for algorithms course.

We propose new algorithms to generate unweighted undirected connected graphs and weighted directed ones. These algorithms allow tutors and students to generate graphs with a selected size, helping to create new exercises, test applications and finding bugs with small inputs.

Our evaluation showed that our tools achieve their goals. They were able to generate 32 graphs used as tests for grading students' solutions. When using this graphs as input, students' solutions executed most of their code, proving the value of our tools. We also found small graphs that would detect problems in students' solutions, thus showing the usefulness of out tools for debugging.

7. Future Work

Some interesting questions result from the evaluation data. An interesting idea that becomes clearer during the evaluation process is that some graphs seem to find a lot more problems than all the rest, with apparently similar properties.

It would be interesting to investigate these graphs and understand which properties make them useful for finding problems. The results could, for instance, be used as teaching examples of students common mistakes.

Finally, an obvious future work is the research about algorithms to generate graphs with different properties from those present in this paper.

Acknowledgements

This work was partially supported by national funds through FCT with reference UID/CEC/50021/2013 and FCT grant AMOS (CMUP-EPB/TIC/0049/2013).

References

[1] U. Z. Ahmed, S. Gulwani, and A. Karkare. Automatically generating problems and solutions for natural deduction. In *Proceedings of the Twenty-Third International Joint Con-*

ference on Artificial Intelligence, IJCAI '13, pages 1968–1975. AAAI Press, 2013. ISBN 978-1-57735-633-2.

[2] W. Aiello, F. C. Graham, and L. Lu. A random graph model for power law graphs. *Experimental Mathematics*, 10(1):53–66, 2001. doi: 10.1080/10586458.2001.10504428.

[3] K. M. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005. doi: 10.1080/08993400500150747.

[4] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of dfa constructions. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, pages 1976–1982. AAAI Press, 2013. ISBN 978-1-57735-633-2.

[5] E. Andersen, S. Gulwani, and Z. Popovic. A trace-based framework for analyzing and synthesizing educational progressions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 773–782, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1899-0. doi: 10.1145/2470654.2470764.

[6] B. Bollobás. The evolution of random graphs. *Transactions of the American Mathematical Society*, 286(1):257–274, 1984.

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

[8] B. Bollobás and O. Riordan. The phase transition in the erdős-rényi random graph process. In L. Lovász, I. Ruzsa, and V. Sós, editors, *Erdős Centennial*, volume 25 of *Bolyai Society Mathematical Studies*, pages 59–110. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-39285-6. doi: 10.1007/978-3-642-39286-3_3.

[9] A. Buisman, M. van Eekelen, and E. Hubbers. Gamification in educational software development. Master's thesis, Radboud University Nijmegen, 2014.

[10] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.*, 38(1), 2006. doi: 10.1145/1132952.1132954.

[11] Q. Chen, H. Chang, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. The origin of power-laws in internet topologies revisited. In *INFOCOM*, 2002.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009. ISBN 978-0-262-03384-8.

[13] C. Douce, D. Livingstone, and J. Orwell. Automatic test-based assessment of programming: A review. *ACM Journal of Educational Resources in Computing*, 5(3), 2005. doi: 10.1145/1163405.1163409.

[14] A. Dudek and A. Frieze. Tight hamilton cycles in random uniform hypergraphs. *Random Structures & Algorithms*, 42(3):374–385, 2013.

[15] S. Esper, S. R. Foster, W. G. Griswold, C. Herrera, and W. Snyder. Codespells: Bridging educational language features with industry-standard languages. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, Koli Calling '14, pages 05–14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3065-7. doi: 10.1145/2674683.2674684.

[16] E. L. Glassman, N. Gulley, and R. C. Miller. Toward facilitating assistance to students attempting engineering design problems. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13, pages 41–46, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2243-0. doi: 10.1145/2493394.2493400.

[17] J. M. Hernandez, T. Kleiberg, H. Wang, and P. Van Mieghem. A qualitative comparison of power law generators. *Delft University of Technology, report20061115*, 2006.

[18] J. B. Hext and J. W. Winings. An automatic grading scheme for simple programming exercises. *Communications of the ACM*, 12(5):272–275, May 1969. ISSN 0001-0782. doi: 10.1145/362946.362981.

[19] J. Hidalgo-Céspedes, G. Marín-Raventós, and V. Lara-Villagrán. Playing with metaphors: A methodology to design video games for learning abstract programming concepts. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ITiCSE '14, pages 348–348, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2833-3. doi: 10.1145/2591708.2602661.

[20] J. Hollingsworth. Automatic graders for programming classes. *Commun. ACM*, 3(10):528–529, Oct. 1960. ISSN 0001-0782. doi: 10.1145/367415.367422.

[21] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0520-4. doi: 10.1145/1930464.1930480.

[22] D. Jackson and M. Usher. Grading student programs using assyst. *ACM SIGCSE Bulletin*, 29(1):335–339, Mar. 1997. ISSN 0097-8418. doi: 10.1145/268085.268210.

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

[24] R. Kannan, P. Tetali, and S. Vempala. Simple markov-chain algorithms for generating bipartite graphs and tournaments (extended abstract). In M. E. Saks, editor, *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 5-7 January 1997, New Orleans, Louisiana.*, pages 193–200. ACM/SIAM, 1997. ISBN 0-89871-390-0.

[25] M. Karoński and T. Łuczak. The phase transition in a random hypergraph. *Journal of Computational and Applied Mathematics*, 142(1):125–135, 2002.

[26] M. Karoński and A. Ruciński. The origins of the theory of random graphs. In *The Mathematics of Paul Erdös I*, pages 311–336. Springer, 1997.

[27] G. License. Gcov: Gnu coverage tool.

[28] T. Łuczak. Component behavior near the critical point of the random graph process. *Random Structures & Algorithms*, 1(3):287–310, 1990.

[29] D. McArthur, C. Stasz, J. Hotta, O. Peter, and C. Burdorf. Skill-oriented task sequencing in an intelligent tutor for basic algebra. *Instructional Science*, 17(4):281–307, 1988.

[30] A. Nachmias and Y. Peres. Component sizes of the random graph outside the scaling window. *ArXiv Mathematics e-prints*, Oct. 2006.

[31] A. Nachmias and Y. Peres. Critical percolation on random regular graphs. *Random Structures & Algorithms*, 36(2):111–148, 2010.

[32] A. R. P. Erdős. On random graphs, i. *Publicationes Mathematicae*, 6:290–297, 1959.

[33] C. Palmer and J. Steffan. Generating network topologies that obey power laws. In *Global Telecommunications Conference, 2000. GLOBECOM '00. IEEE*, volume 1, pages 434–438 vol.1, 2000.

[34] K. A. Reek. The try system -or- how to avoid testing student programs. *ACM SIGCSE Bulletin*, 21(1):112–116, Feb. 1989. ISSN 0097-8418. doi: 10.1145/65294.71198.

[35] A. Ruciński and A. Vince. The solution to an extremal problem on balanced extensions of graphs. *Journal of graph theory*, 17(3):417–431, 1993.

[36] R. Singh, S. Gulwani, and S. K. Rajamani. Automatically generating algebra problems. 2012.

[37] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. *SIGPLAN Not.*, 48(6):15–26, June 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462195.

[38] N. Tillmann, J. De Halleux, T. Xie, S. Gulwani, and J. Bishop. Teaching and learning programming and software engineering via interactive gaming. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 1117–1126. IEEE, 2013.

[39] V. Tscherter. *Exorciser: Automatic Generation and Interactive Grading of Structured Exercises in the Theory of Computation*. PhD thesis, Swiss Federal Institute of Technology Zurich, 2004.

[40] D. B. Wilson. Generating random spanning trees more quickly than the cover time. In G. L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24*, pages 296–303. ACM, 1996. ISBN 0-89791-785-5. doi: 10.1145/237814.237880.