Generating Random Graphs with Graph Grammars

Jake Coxon

January 26, 2013

Contents

1 Introduction

Introduction should go here

2 Literature Review

2.1 Graphs

A graph is a finite set of points known as vertices that are interconnected by a set of lines (edges). Edges connect exactly two vertices that shows a relation.

A label alphabet $L = \langle L_v, L_e \rangle$ consists of a set L_v of node labels and a set L_e of edge labels.

A graph over L is a system $G = \langle V_G, E_G, s_G, t_G, l_G, m_G \rangle$ where V_G and E_G are finite sets of vertices and edges respectively. $s_G, t_G : E_G \to V_G$ are functions assigning a source and a target to each edge. $l_G : V_G \to L_V$ and $m_G : E_G \to L_E$ are functions assigning a label to each node and edge[1].

Different types of graphs exist pertaining to different problems. Graph edges can be weighted and/or directed and vertices can be labelled. Graphs can allow edges to be looped where both

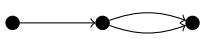


Figure 1: A directed multigraph

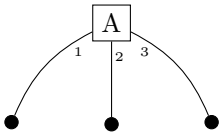


Figure 2: A hypergraph with a hyperedge labeled A

endpoints point to the same vertex. Multi-graphs are graphs where multiple edges can exist between a pair a vertices.

2.2 Hypergraphs

A generalised version of graphs are hypergraphs. Within a hypergraph, a hyperedge is similar to an edge except it is connected to any number of vertices. The connections between a hyperedge and a vertex is known as 'tentacles' and the number of the tentacles a hyperedge has is known as its type. A graph is a specialised version of a hypergraph where all hyperedges has type of 2.

An alphabet C is a fixed set containing labels. A hypergraph over C is a system $F = \langle V_F, E_F, att_F, lbl_F \rangle$ where V_F and E_F are finite sets of vertices and edges the same as graphs. $att_F : E_F \to V_F*$ is a mapping assigning a sequence of 'attachment nodes' to each edge $e \in E_F$ and lbl_F is a mapping that labels hyperedges.

The $type(E)$ of an edge is the number of vertices that a hyperedge attaches to. A tentacle is an object used to describe the attachment of a hyperedge to a vertex.

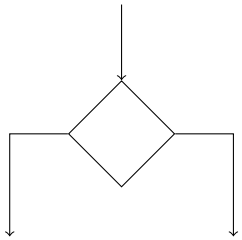


Figure 3: Hyperedges can be used to model flowcharts

2.3 Random Graph Generation

Random graph generators are systems to generate a single or set of graphs based on some non-deterministic rules. This can be used as test data for programs for testing speed and memory. It can also be used for verifying whether a property is true for a range of graphs in a range or seeing how *often* a property is true for a range of graphs.

There are two methods for random graph generation described in [3]:

Random Edge Generation—Every pair of vertices is listed and a coin is flipped to decide whether an edge is produced between them.

Random Edge Selection—For a desired number of edges m, it selects m distinct edges at random.

Both of these methods have their limitations. With these methods you cannot enforce the structure of the graph, for instance it is impossible to generate multi-graphs and it is not likely

to generate a cyclic graph. For the purposes of testing algorithms, these methods aren't good enough, you cannot test an algorithm that only accepts trees unless you can generate only trees.

2.3.1 Standford GraphBase

The Stanford GraphBase is a collection of programs written by Donald Knuth. Its main purpose is to generate and examine graphs but can also be used as a library to write ones own programs.

The programs are written in a language called CWEB which is a combination of TEXand the C programming language but a person can just as well write a program in C and include the GraphBase library.

In GraphBase, graphs are represented by the structures `Graph`, `Vertex` and `Arc`. A Graph ponter g refers to a single graph and g has multiple fields and $g \rightarrow n$ is the number of vertices in this graph. $g \rightarrow vertices$ an array of all vertices so $g \rightarrow vertices[k]$ points to the k^{th} vertex.

Directed edges between vertices are specified by `Arc`s. The head of the linked list contain all arcs for a vertex is stored in $v \rightarrow arcs$. To represent undirected edges, two arcs are required.

To generate a random graph, the function `random_graph` can be used which generates a random graph based on a number of parameters:

- `multi`—whether the graph is permitted to have duplicate arcs eg. a multigraph.
- `self`—whether the graph can have self-loops.
- `directed`—if the graph is directed or not.
- `dist_from` and `dist_to`—the probability distributions of arcs to vertices.
- `min_len` and `max_len`—the arc lengths will be uniformly distributed between these two values.

The arrays `dist_from` and `dist_to` are used to control the discrete distribution of the arcs. The probabilities are scaled so the sum of the array is 1. For example, to define the probability that v_k is twice as likely as v_{k+1} to be the source of an arc, `dist_from` should equal {`...,16,8,4,2,1`}

The function will return a `Graph` structure so in order to output, analyse or perform any validation on the graph, the user of this library must do some extra work himself.

2.3.2 Mathematica

Wolfram Mathematica is a powerful integrated environment which allows the user to input an expression language within the program. Previously, the package Combinatorica[1] was needed to provide graph theory functions however, Mathematica version 8 added most of these functions natively.

The programming language in Mathematica enables the user to build expressions and functions. user inputs a command as text and the output of the command is written below. The command `Sin[3.4]` will output `-0.255541` to the screen.

The basic function to generate a random graph is `RandomGraph[gdist, n]` which generates n graphs with the graph distribution $gdist$. The distribution function `BernoulliGraphDistribution` provides the random edge generation method. In the command `RandomGraph[BernoulliGraphDistribution[6, 0.5], 3]` we request six vertices and 50% probability that an edge occurs between them. We also request that we want three graphs.

A list of relevent graph distributions is as follows.

[1] http://www.cs.sunysb.edu/ skiena/combinatorica/

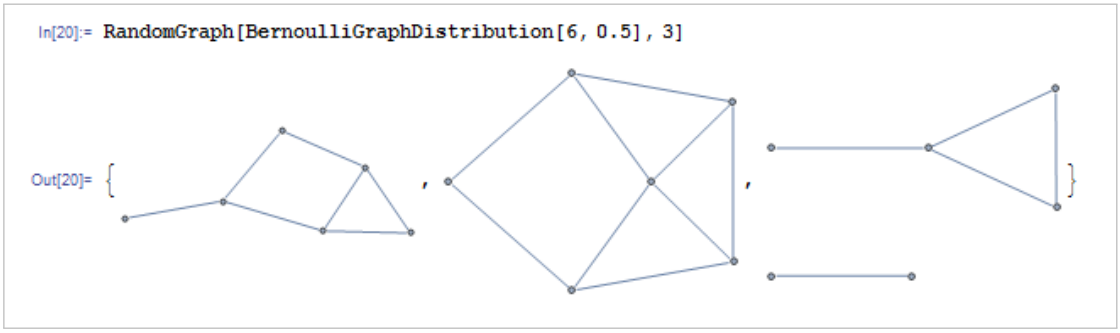


Figure 4: Mathematica function to generate three graphs

- `BernoulliGraphDistribution[n, p]` generates n vertices and a p chance of an edge occuring between them. This is the random edge generation method decribed above where a fixed amount of vertices will be generated and every pair of vertices will have a probability of an edge occuring between them.
- `UniformGraphDistribution[n, m]` generates a uniform graph distribution on n vertices and m edges. This distribution is equivilent to the random edge selection method above where there are a fixed number of vertices and edges but the positions of the edges are random generated.
- `BarabasiAlbertGraphDistribution[n, k]` generates n vertices where a new vertex with k edges is added at each step. This distribution will force each vertex to have at least k edges attached to it.
- `PriceGraphDistribution[n, k, a]` generates a graph with a de Solla Price distribution which is a distribution that gives edges a preferential attachment to vertices.
- `DegreeGraphDistribution[dlist]` generates a graph where vertex_i has degree dlist_i.

Additionally the user can use the progamming language to programatically generate a graph based on his own method. There exists functions `EdgeAdd` and `VertexAdd` which can be combined with some random functions. In the next example, `StarGraph[n]` is used which constructs a graph with n vertices with one vertex connected to all others. Next an edge is added onto this graph between v_2 and v_r where v_i is vertex i and r is a random number in the range $\{0, ..., 10\}$.

This shows that Mathematica gives the user a powerful set of tools and multiple methods of generating random graphs. However there is no general solution for generating random graphs, the user must learn the programming language and must have a specific algorithm in mind. For anything more advanced than adding specific edges or nodes, the code can get very complex.

The output graph will be immediately shown to the user and more processing can be done on it. Additionally the data of this graph can be exported in a number of largely used formats including *GraphML*, *GXL*, *Graph6*, *DOT*. Furthermore, the generated image of the graph can be saved for the user to use in documents.

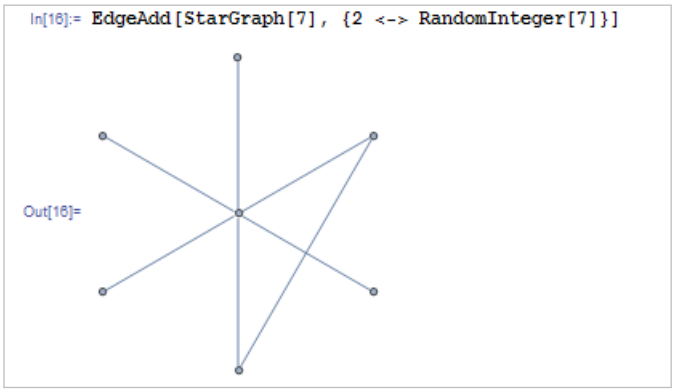


Figure 5: Mathematica function to generate a star

2.4 Hypergraph Languages

2.4.1 Hyperedge Replacement

It is possible to construct new hypergraphs by the replacement of a hyperedge by a new hypergraph to yield a new hypergraph. A hyperedge e in a hypergraph H may be replaced with another hypergraph R. An sequence of external nodes ext in R is needed where each node in ext corresponds to an attachment node of e.

A replacement happens by first removing the hyperedge e, then adding the hypergraph R excluding external nodes. Then for each hyperedge in R that points to an external node, the tentacle must now point to the corresponding attachment nodes of e.

The replacement of hyperedge e in a hypergraph H with the a hypergraph R is denoted by $H[e/R]$.

The definition of hyperedge replacement is as follows:

$$\begin{aligned} V_X &= V_H + (V_R - ext) \\ E_X &= (E_H - e) + E_R \\ lbl_X &= lbl_H \cup lbl_R \\ att_X(e) &= \{handover_e(x_1), ..., handover_e(x_n)\} \quad \text{for all } e \in E_X \text{ and where } att_H(e) = x_1...x_n \end{aligned}$$

Where $handover_e : V_X \to V_X$ is defined by

$$handover_e(x_i) = \begin{cases} a_i & \text{if } x_i \in ext \text{ and where } att_H(e) = a_1...a_n \\ x_i & \text{otherwise} \end{cases}$$

We can see an example of hyperedge replacement here.

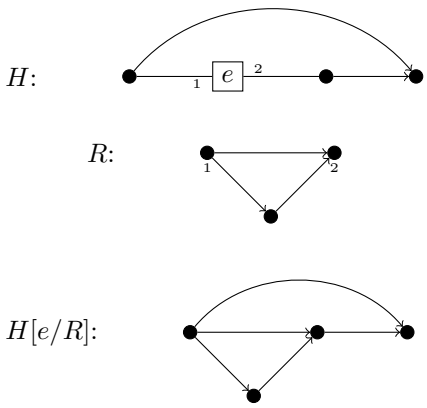


Figure 6: Hyperedge replacement

A couple of properties can be observed in hyperedge replacement. Firstly, the sequentialization and parallelization property. Given two different replacements, the result will be equivilent whether the replacements happen one after another or simultaneously.

$$H[e_1/H_1, ...e_n/H_n] = H[e_1/H_1]...[e_n/H_n] \tag{1}$$

Secondly the replacement is confluent, this means hyperedges in a hypergraph can be replaced in any order without affecting the result.

$$H[e_1/H_1][e_2/H_2] = H[e_2/H_2][e_1/H_1] \tag{2}$$

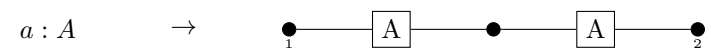
Finally we know the replacement is associative. If a hyperedge is replaced and then a part of the new hypergraph is replaced, this is equivilent to replacing the second hyperedge and then replacing the first hyperedge with the result.

$$H[e_1/H_1][e_2/H_2] = H[e_1/H_1[e_2/H_2]] \tag{3}$$

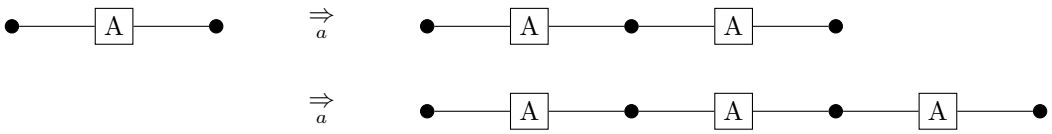
2.4.2 Hyperedge Replacement Grammars

Hypergraphs can be generated using productions based on hyperedge replacement. A production is $p = \langle x \to R, ext \rangle$ where x is a non-terminal so let H, H' be hypergraphs and let $e \in E_H$ such that $l_H(e) = x$. Then H directly derives H' by p applied to e if H' is equivilent to $H[e/R]$. This can be written $H \underset{p}{\Rightarrow} H'$.

If we are given a production:



Then it follows:



A hyperedge replacement grammar (HR grammar) is a system for generating a hypergraph language through a set of derivations. It is similar to other language generation concepts in computer science.

A HR grammar can be defined as $G = \langle N, T, P, S \rangle$ where N is a set of non-terminals, T is a set of terminals disjoint with N, S is the start graph. P is a set of productions which are in the form of $\langle x \to R, ext \rangle$ where $x \in N$, R is a hypergraph and ext is the sequence of external nodes in R.

The language of a control flow graph can be generated with the following

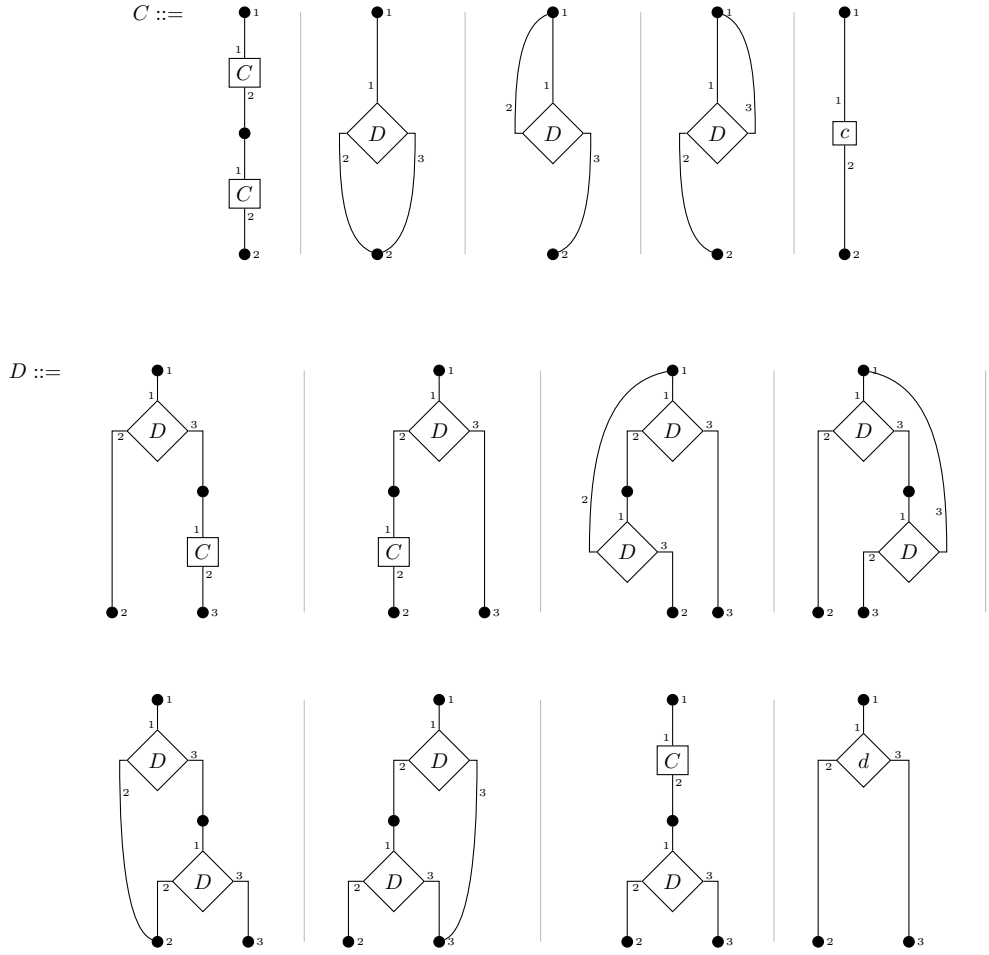


Figure 7: Flowchart grammar

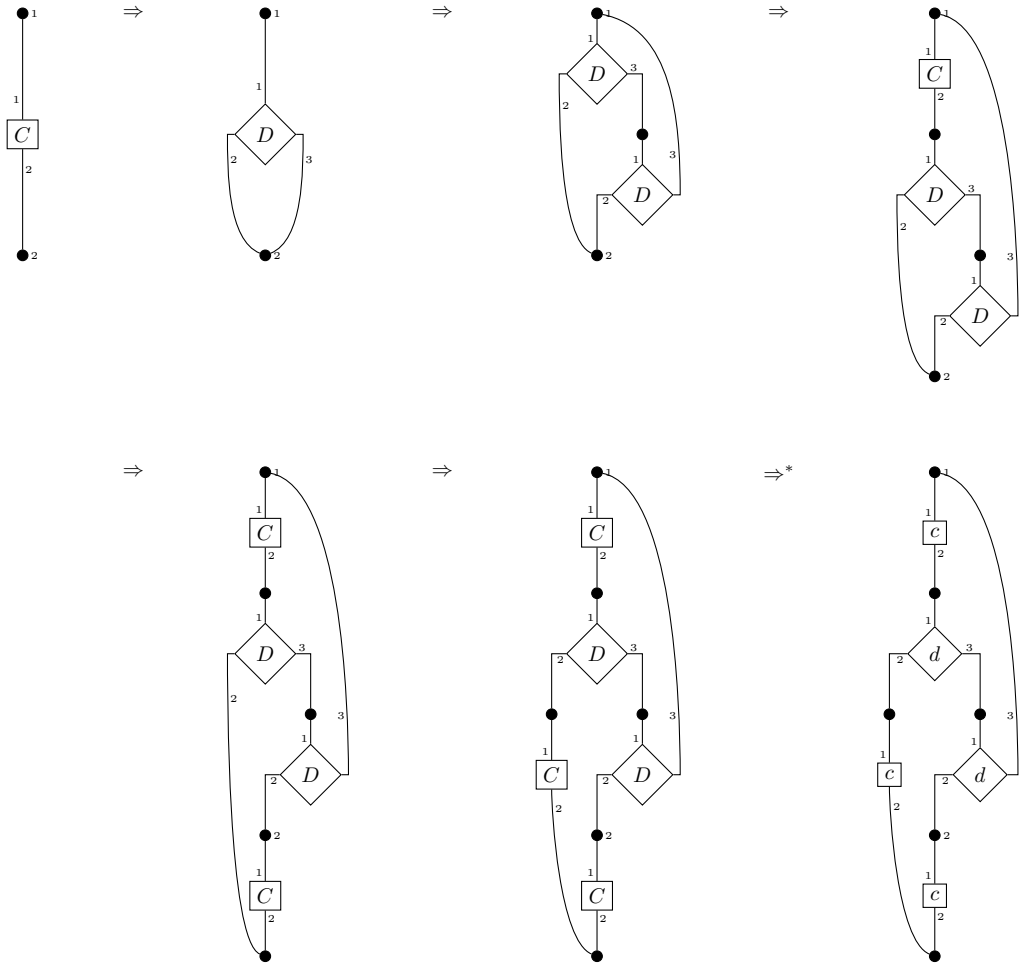


Figure 8: Example derivation of flowchart grammar

2.5 Ranjan's Approach

In order to generate random graphs we can study Ranjan's approach to graph generation. Ranjan[2] decribes a method of generating a random sequence of productions over a graph grammar G which can then be run to generate a single random graph.

There are two seperate stages in Ranjan's approach that will henceforth be known as the 'generation stage' and the 'run stage'. Firstly, the generation stage will generate a sequence of rules and locations and the run stage will actually perform this sequence of replacement rules i.e productions.

- Firstly, n_1 is initialised to contain all the available locations based on the start graph.
- For each rule in the grammar, the number of available locations that the rule adds must be calculated (See below for definition of available location.)
- Begin looping $i = 1, 2...$

- The system will generate a random integer $1 \leq m \leq \#P$ which is for identifying which rule is chosen. Another random integer $1 \leq l \leq n_i$ is chosen to identify the location that the rule is applied to. The tuple $\langle m, l \rangle$ is added to the sequence of rules to be applied.
- Assign $n_{i+1} = n_i$ plus the number of additional locations added for rule m.
- Repeat loop until some condition

The loop continues until some predetermined end condition. For example, we could restrict the number of edges/vertices, or restrict the number of iterations. If a range is given then the size is chosen randomly to make the graph even more random.

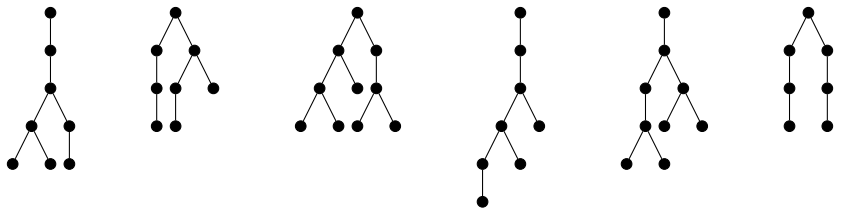
An 'available' location is a node where *all* productions can be applied. Because Ranjan's approach generates a complete list of rules and locations before actually performing any replacements, it does know the current state of the graph. When a rule $\langle m, l \rangle$ is performed, l must always refer to a currently available location and it must always be possible for production m to be performed on that location. For every graph in the language, there must be at least one available location and the number of available locations never decreases (consequently the language is infinite.) In Ranjan's examples every production adds zero or one new available location. Note that the grammar may produce nodes that one or more productions *cannot* be matched, but these must not be considered 'available'.

Ranjan uses a language called GP (Graph Programs) to perform the sequence of rules but the actual system used is irrelevant and the replacements can be performed by any means. The only requirement is that the system must keep track of the available locations. In Ranjan's implementation, the grammar itself is modified to number the available nodes $1..N$ but it is equally valid to keep a sequence of these nodes seperate to the grammar.

The result is a single graph and the process can be re-run to produce multiple random graphs. The drawbacks of this method is the strict set of rules placed on the grammar.

Figure 9: Binary tree grammar

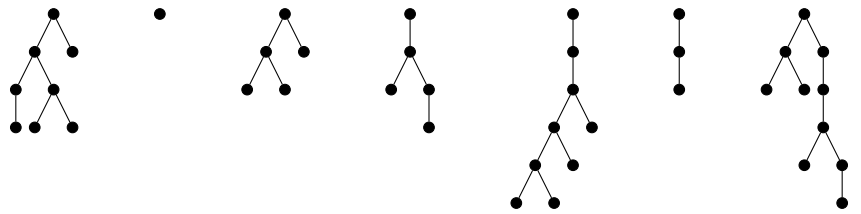
The figure below shows an example of the system applied to the binary tree grammar six times with a single node as the start graph and the number of iterations fixed at 5.



In this figure we can see that both rules will match on leaf nodes due to the dangling condition. As such the system will need to mark just the leaf nodes as available nodes. It must also unmark nodes that become parent nodes. In r_1, the rule converts a leaf node into a parent node but also

adds an available node, so the total available nodes increases by zero. On the other hand r_2 adds two new nodes so the total available nodes increases by one.

The figure below shows an example of the system applied to the same grammar with the number of vertices bound ≤ 10.



2.5.1 Improvement on Ranjan's Approach

It is likely that the reason Ranjan's approach happens in two stages is a workaround to GP which has a very simple instruction set. By changing the method keep a sequence of morphisms, we can improve it a bit.

- Let d be a sequence of all isomorphic graph morphisms to G for each rule in the grammar.
- Pick a random integer $1 \leq r \leq \#d$ and apply rule d_r to the graph G.
- Update the sequence d
- Repeat loop until some condition

This method is a more general version of Ranjan's approach and there are no restrictions of the grammar. However, the implementation of this method is very expensive to compute since matching morphisms is not trivial.

Another method is to use what we know about hypergraph languages. Below is a new method that works on hypergraphs. It keeps a sequence of available non-terminals and picks one at random. It then chooses a random rule that can be applied to this location and applies it the the graph. Using a sequence of non-terminals instead of graph morphisms gives a better computational speed since the lookup of non-terminals in a graph is a constant time operation.

- Let G be the start graph. Let d be a sequence of all the non-terminal hyperedges in G
- Begin loop
- Pick a random integer $1 \leq l \leq \#d$.
- Get the set of all valid rules on this edge. $r = \{\text{All rules } \langle x \to R \rangle \text{ where } x = lbl_F(d_l)\}$
- Pick another random integer $1 \leq m \leq \#r$ and apply the rule r_m to the graph G.
- Remove the non-terminal d_l from d and add all the non-terminals in the replaced graph R.
- Repeat loop

References

[1] Detlef Plump. Computing by graph transformation. http://www-module.cs.york.ac.uk/grat/Lectures/II.pdf, 2012.

[2] Anuja Ranjan. Grammar-based generation of random graphs. Technical report, University of York, 2011.

[3] Steven S. Skiena. *The Algorithm Design Manual*. Springer, 2008.