Uniform Random Generation of Graphs with Graph Grammars

Jake Coxon

6th May 2013

Supervisor: Detlef Plump

Number of words = 9230, as counted by texcount. This includes the body of the report only.
Abstract

The aim of the project is to develop a graph generator that provides users with a powerful method to specify graph classes from which the generated graphs are drawn. This is achieved by adapting a method for string generation over context-free grammars to hyperedge replacement grammars, and implementing this method in Scala.
1 Introduction

Graph theory is a large topic of computer science and mathematics which studies graphs—a mathematical structure used to model relations between objects. Graph theory also crosses over with other subjects such as chemistry and physics to study complicated atomic structures, and biology to model habitats and migration paths. It is used in sociology to model social network analysis, and even architecture to model rooms and buildings. Graph theory covers many algorithms for such graphs, for example to test properties, to find routes through the graph, and to manipulate graphs into other graphs.

The use of random graph generators is mostly used for generating graphs for the testing of algorithms. There are a couple of types of testing you may want to do on graphs, for example a user may want to test the speed of two algorithms that solve the same problem and he wants to see which is quicker. In this case he would want a set of graphs which cover the available input of the algorithms. He may also want to test if these algorithms actually produce a correct answer, a random graph generator may produce an edge case that has been overlooked.

There are issues that need to be considered when designing a random graph generator. Presumably one wants a set of graphs that conform to the input of an algorithm. It is no use generating cyclic graphs when testing an algorithm on trees. Therefore, a system of designing the class of graph will be researched. The aim of the project is to research current graph generators, provide a method to generate random graphs where a user can design the ‘shape’ of graph, and finally to provide a sample computer program that applies this algorithm.
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 models a relation between them. So many different structures can be modelled using graphs that using a single formalism is needed to generalise many algorithms\[11\].

A label alphabet $L = (L_v, L_e)$ consists of a set $L_v$ of node labels and a set $L_e$ of edge labels.

A graph over $L$ is a system $G = (V_G, E_G, s_G, t_G, l_G, m_G)$ where $V_G$ and $E_G$ are finite sets of vertices and edges respectively. $s_G, t_G : E_G \rightarrow V_G$ are functions assigning a source and a target to each edge. $l_G : V_G \rightarrow L_V$ and $m_G : E_G \rightarrow L_E$ are functions assigning a label to each node and edge\[8\].

Many different ‘flavours’ of graphs exist pertaining to different problems which require a slightly different definition. The above definition is a directed graph, this is a graph where the direction of an edge matters. An undirected graph is a graph where direction doesn’t matter, in this case $E_G$ is defined as a set of unordered pairs $(x,y)$, then it implies that also $(y,x) \in E$. Motorway networks are typically undirected since a motorway has lanes going in both directions, however a street network within a city will be directed if there exists a one-way road.

Weighted or unweighted means if a graph’s edges have a weight which is a numerical value. An edge’s weight may correspond to a physical distance, a time, a speed limit etc.

Self-loops may be allowed in a graph, this means the source and target of an edge is the same vertex. A multi-graph (See Fig. 2.1) is a graph where an edge between two vertices may exist more than once. By knowing beforehand if a graph has no self-loops and is not a multi-graph then some algorithms maybe simplified, therefore graphs without these two properties are called simple graphs.

2.2 Hypergraphs

A generalised version of graphs are hypergraphs. A hypergraph has hyperedges instead of edges, a hyperedge is an edge that connects to any number of vertices, this could be zero, one or more vertices. The connections between a hyperedge and a vertex is known

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{Figure21.png}
\caption{A directed multigraph}
\end{figure}
2 Literature Review

Figure 2.2: A hypergraph with a hyperedge labeled A

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 have a type of 2. Figure 2.2 shows a hypergraph with three tentacles but hyperedges may also be drawn like flowcharts as in Figure 2.3.

An alphabet $C$ is a fixed set containing labels. A hypergraph over $C$ is a system $F = (V_F, E_F, att_F, lbl_F)$ where $V_F$ and $E_F$ are finite sets of vertices and edges the same as graphs. $att_F : E_F \rightarrow V_F^*$ is a mapping assigning a sequence of ‘attachment nodes’ to each edge $e \in E_F$, $att_F(e, i)$ can be written to denote the $i$th attachment node of $e$. $lbl_F$ 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.

If a hyperedge has a type of 2 then it is typically drawn as a directed edge from its first attachment to its second. In cases where the first and second attachment does not matter then the edge is drawn as an undirected edge for simplicity.

Two hypergraphs are said to be isomorphic if they contain the same number of vertices and are connected in the same way. In other words $G$ and $H$ are isomorphic graphs if $(x, y)$ is an edge in $G$ and there is a mapping $f$ for which $(f(x), f(y))$ is an edge in $H$ (Skiena [11, chap. 16.9]).

Figure 2.3: A hypergraph can be used to model a flow-diagram conditional

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 or seeing how often a property is true for a range of graphs.
There are two methods for random graph generation described in Skiena [11]:

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 one cannot enforce the structure of the graph, for instance it is impossible to generate strictly multi-graphs or strictly cyclic graph. For the purposes of testing algorithms, these methods aren’t good enough.

2.3.1 Stanford 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 TeX and 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 pointer $g$ 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 Arcs. 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 multi-graph.
- self—whether the graph can have self-loops.
- directed—if the graph will be directed or not.
- dist_from and dist_to—the probability distributions of arcs to vertices.
- min_len and max_len—the arc weightings 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 something like \{32,16,8,4,2,1\}

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 write the C code to do that 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 $\text{Sin}[3.4]$ will output $-0.255541$ to the screen.

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

![Mathematica function to generate three graphs](image)

Figure 2.4: Mathematica function to generate three graphs

A list of relevant graph distributions is as follows.

- $\text{BernoulliGraphDistribution}[n, p]$ generates $n$ vertices and a $p$ chance of an edge occurring between them. This is the random edge generation method described above where a fixed amount of vertices will be generated and every pair of vertices will have a probability of an edge occurring between them.

- $\text{UniformGraphDistribution}[n, m]$ generates a uniform graph distribution on $n$ vertices and $m$ edges. This distribution is equivalent 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.

- $\text{BarabasiAlbertGraphDistribution}[n, k]$ 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.

\(^1\)http://www.cs.sunysb.edu/~skiena/combinatorica/
2 Literature Review

- \texttt{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.

- \texttt{DegreeGraphDistribution[dlist]} generates a graph where vertex $i$ has degree $dlist_i$.

Additionally, the user can use the programming language to programatically generate a graph based on his own method. There exists functions \texttt{EdgeAdd} and \texttt{VertexAdd} which can be combined with some random functions. In the next example, \texttt{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 $\{0, ..., 10\}$. However once a user has added a vertex or edge then the distribution is no longer uniform.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{star_graph.png}
\caption{Mathematica function to generate a star}
\end{figure}

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.

In both programs, although various parameters can be defined, it is still not a general solution. Neither programs can generate strictly binary trees for example.

2.4 Hypergraph Languages

Graph grammars can be a good way to produce graphs that conform to a strict set of rules. In a general sense, rules are applied to part of a graph and then transformed
2 Literature Review

into something else. They are similar to string grammars where a production \( A \rightarrow aa \) applied to \( bAb \) results in \( baab \).

2.4.1 Hyperedge Replacement

Hyperedge replacement is the process of transforming a single hyperedge in a graph and producing an entirely new hypergraph. If \( H \) is a graph, \( e \) is the edge to be replace, \( R \) is the hypergraph that \( e \) is replaced with, then \( H[e/R] \) is the finally produced hypergraph.

A sequence of external nodes for \( R \) is needed in order to position \( R \) within \( H \). We call these external nodes \( ext_R \) and each node in \( ext_R \) corresponds to an attachment node of \( e \). To collect this sequence together with the hypergraph, we use the term multi-pointed hypergraph\[4\]. A multi-pointed hypergraph \( G \) is simply the combination of a hypergraph and a sequence of external nodes. It is defined as \( \langle V_G, E_G, att_G, lbl_G, ext_G \rangle \) or as an extension to a hypergraph \( \langle G, ext_G \rangle \), where \( ext_G \in V_G^* \). A useful notation is \( EXT_G \) which denotes the set of external nodes.

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 \( e \). This also means \( |att_H(e)| = |ext_R| \), i.e. the type of \( e \) is equal to the size of the external nodes in \( R \).

In some implementations, hyperedge replacement is allowed to merge two vertices together where a single attachment node points to two external nodes. In our case we will disallow this to avoid overcomplicating things.

Klempien-Hinrichs [6] gives a nice formal definition of \( H' = H[e/R] \) found below.

\[
H' = \langle V_{H'}, E_{H'}, att_{H'}, lbl_{H'}, ext_{H'} \rangle \\
V_{H'} = V_H \cup (V_R \setminus EXT_R) \\
E_{H'} = (E_H \setminus \{e\}) \cup E_R \\
type_{H'}(e') = \begin{cases} 
\text{type}_H(e') & \text{if } e' \in E_H \setminus \{e\} \\
\text{type}_R(e') & \text{if } e' \in E_R 
\end{cases} \\
att_{H'}(e', i) = \begin{cases} 
\text{att}_H(e', i) & \text{if } e' \in E_H \setminus \{e\} \\
\text{att}_H(e, j) & \text{if } e' \in E_R \text{ and } att_R(e', i) = ext_R(j) \\
\text{att}_R(e', i) & \text{if } e' \in E_R \text{ and } att_R(e', i) \in (V_R \setminus EXT_R) \\
\end{cases} \\
\text{for } i \in 1..\text{type}_{H'}(e') \\
lbl_{H'}(e') = \begin{cases} 
\text{lbl}_H(e') & \text{if } e' \in E_H \setminus \{e\} \\
\text{lbl}_R(e') & \text{if } e' \in E_R 
\end{cases} \\
ext_{H'} = ext_H
\]
A couple of properties can be observed in hyperedge replacement. Firstly, the sequentialization and parallelization property. Given two different replacements, the result will be equivalent whether the replacements happen one after another or simultaneously.

\[ H[e_1/H_1,...e_n/H_n] = H[e_1/H_1][e_n/H_n] \]  

(2.1)

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

\[ H[e_1/H_1][e_2/H_2] = H[e_2/H_2][e_1/H_1] \]  

(2.2)

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

\[ H[e_1/H_1][e_2/H_2] = H[e_1/H_1][e_2/H_2] \]  

(2.3)

These properties make hyperedge replacement simpler than other kinds of replacement and is the reason why it is used in this project.

**Hyperedge Replacement Grammars**

A structure is needed to combine hyperedge replacement rules. This structure is called a hypergraph grammar\(^\text{[10]}\) (also known as a hyperedge replacement grammar or HRG.) A rule is known as a production which is \( p = x \rightarrow R \) where \( x \) is a non-terminal symbol and \( R \) is a multi-pointed hypergraph. This production is applied to a hyperedge \( e \) in a hypergraph \( H \) if \( \text{lbl}_H(e) = x \) and \( \text{type}(e) = \text{ext}_R \) resulting in the graph \( H[e/R] = H' \). It is said that \( H \) ‘directly derives’ \( H' \) which is also written \( H \Rightarrow_p H' \).

A special kind of hypergraph is introduced called a ‘handle’ which will be used later. It consists of a single edge \( e \) that can be referred to as \( \text{edge}(H) \) where \( H \) is the handle, and its label is referred to as \( \text{label}(H) \). It must also satisfy \( \text{V}_H = \text{att}_H(e) \) meaning the only
vertices in $H$ are the ones attached to $e$. A handle with $n$ vertices is called a $n$-handle. Below is a 3-handle labelled ‘$A$’. Handles encapsulate a non-terminal along with its attachment vertices.

In the following production, the left-side of the production is drawn as a handle, which represents the production $A \rightarrow \ldots$. The handle on the left must have the the number of vertices equal to the number of external nodes on the right.

Then it follows:

A HRG is used as a system for generating a hypergraph language through a set of derivations. It is similar to other language generation concepts in computer science.

A HRG 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 a handle which specifies the initial graph. $P$ is a set of productions which are in the form of $x \rightarrow R$ where $x \in N$, $R$ is a multi-pointed hypergraph.

The language of a control flow graph can be generated with the grammar found in Rozenberg [10], $\langle N, T, P, C \rangle$ where $C$ is a 2-handle labelled $C$ and $P$ contains the productions in Figure 2.8.
Figure 2.7: Flowchart grammar
2 Literature Review

In context-free string grammars, there exists an important property which is the fact that every derivation is equivalent with a left-most derivation. This property comes in useful later so it would be good if we could define the same property for hypergraph grammars. To do this we need to put an ordering on the non-terminals in a production. When a production is applied to a graph, its ordering is substituted into the graph in the same way as a string grammar.

2.5 Ranjan’s Approach

In order to generate random graphs we can study Ranjan’s approach to graph generation. Ranjan [9] describes 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. Ranjan doesn’t use hypergraph grammars but instead uses a more general graph grammar.

Figure 2.8: Example derivation of flowchart grammar
which introduces complexity and furthermore her method does not produce a uniform distribution.

There are two separate 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 productions to be applied at specific locations, then the run stage will actually perform this sequence of productions on a graph.

- 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 \( (m, l) \) is added to the sequence of rules to be applied.
  - Assign \( n_{i+1} = n_i + \text{the number of additional locations added for rule } m \).
  - 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.

The generation stage does not actually perform any replacement productions and as such, the current state of the graph is not known. Instead a list of ‘available’ locations are recorded at each iteration. An ‘available’ location is a subgraph of the implied current state of the graph where all productions must be applicable. When a production \( (m, l) \) is applied, \( l \) 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. The down-side of this algorithm is that many subgraphs need to be found in the graph which is an expensive problem.

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 separate 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.
2 Literature Review

Figure 2.9: Binary tree grammar

Figure 2.9 shows a Binary tree grammar which the algorithm is applied to. Figure 2.10 shows an example of the output when the algorithm is run six times with a single node as the start graph and the number of iterations fixed at 5.

Figure 2.10: Result of fixed iterations

In this figure we can see that both rules will match only on leaf nodes. As such the system will need to mark just the leaf nodes as available locations. 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 a node to the available locations, so the total available locations does not increase. Similarly, $r_2$ adds two new nodes so the total available locations increases by one.

Figure 2.11 shows an example of the system applied to the same grammar with the number of vertices bound $\leq 10$.

Figure 2.11: Result of bounded vertices

2.5.1 Improvement on Ranjan’s Approach

It is likely that the reason Ranjan’s approach is constructed in two stages is a work-around to GP which has a very simple instruction set. By changing the method to use a hyperedge replacement grammar instead of a graph grammar, the system can keep a sequence of valid non-terminals instead of finding subgraphs. 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 to 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 can be performed in constant time.

- 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 productions on this edge. $r = \{ p | p = x \rightarrow R, p \in P, x = lb_F(d_l) \}$
  - 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

Unfortunately the algorithm still produces non-uniform randomness.

2.6 Hickey-Cohen Generation of Uniform Random Strings

The previous approach has a major downside that some terminal graphs have a far larger probability of occurring than others. Consider applying the algorithm to the graph equivalent to $S \rightarrow a | Sa$, we can see here that the string $a$ has the probability 0.5 of being produced and every single other terminal graph combined has the remaining probability of 0.5. The same is true whether we apply it to graph grammars or anything else. A better algorithm would be able to choose terminal graphs with uniform randomness.

The paper by Hickey and Cohen [5] describes a method of picking strings uniformly. If the method can be constructed for strings we can look into being able to use the same method for graphs.

The paper explains the meaning of producing strings uniformly - “Consider the set of all strings of length $n$ in the language. A uniform random generator is one which will produce strings from this set with equal probability.” It is noted that the probability that a production is used depends on the previous steps taken. This is different to the naive approach of choosing a derivation with equal probability and not taking into account the current state.

Firstly some notation is introduced. Let $G = (V, \Sigma, P, N_1)$ be an unambiguous cycle-free context-free grammar containing no $\epsilon$-productions where $V = N \cup \Sigma$, $N = N_1, \ldots, N_r$
is the non-terminal vocabulary, $\Sigma$ is the set of terminals, $N_1$ is the start symbol, and the set of productions is the following.

$$P = \{ \pi_{ij} : N_i \rightarrow \alpha_{ij} | i = 1, \ldots, r, j = 1, \ldots, s_i \}$$

For example a set of productions might look like this.

$$P = \pi_{11} : N_1 \rightarrow (N_1$$

$$\pi_{21} : N_2 \rightarrow N_1N_2$$

$$\pi_{22} : N_2 \rightarrow )$$

The general outline of the algorithm is to provide a weighting probability $p_{ij}$ to each production so that every terminal string has the same overall probability of being generated. For this to be possible the probability must depend on the current state of the derivation and the size of terminal string to generate. From these probabilities, we simply iteratively choose the left-most non-terminal, pick a production based on the weightings and apply it until we have a terminal string. This sequence can then be repeated until a sufficient number of terminal strings have been generated.

Choosing a string $\beta_{k+1}$ from a previous string $\beta_k$ involves rewriting the leftmost non-terminal in $\beta_k$ using one of the productions in $P$. If $N_i$ is the leftmost non-terminal in $\beta_k$, there are $s_i$ different choices for $\beta_{k+1}$.

$$S \rightarrow A|B|aa$$

$$A \rightarrow ab|ba$$

$$B \rightarrow bb$$

If $\beta_k$ is $aSa$ then there are 3 different choices for $\beta_{k+1}$.

Let $g_{\beta}(n)$ be the enumerating function which is the number of terminal strings of length $n$ that can be derived from a sentential form $\beta$. A sentential form is a string that can be derived from the start symbol. Let $\gamma$ be the result of applying $\pi_{ij}$ to $\beta$ in a leftmost derivation. The probability that must be assigned to $\pi_{ij}$ is:

$$p_{ij}(\beta, n) = \frac{g_{\gamma}(n)}{g_{\beta}(n)}$$

This is the number of strings generated after applying a certain production divided by the number of strings generated after applying any possible production. So the sum of all $p_{ij}, 1 \leq j \leq s_i$ will equal to 1.

A terminal string of length $n$ derived from the concatenation of two strings $\beta_1, \beta_2$ is the concatenation of two terminal strings whose lengths sum to $n$. The number of these terminal strings is the following.

$$g_{\beta_1\beta_2}(n) = \sum_{k=0}^{n} g_{\beta_1}(k) \cdot g_{\beta_2}(n - k)$$
This is in the form of a convolution of two functions $v$ and $w$ which is defined as:

$$(v * w)(n) = \sum_{k=0}^{n} v(k) \cdot w(n-k)$$

This convolution is a way of ‘multiplying’ a sequence of numbers, or in this case the co-domain of a function, and is described in the next section in more detail. The notation $g^{(k)}$ will be used later to denote the convolution of a function $g$ with itself $k$ times. The paper shows that this convolution has the associativity and commutativity properties. Also defined is an ‘identity’ function $\delta_0$ for which $\delta_0 * g = g$ and $g^{(0)} = \delta_0$.

$$\delta_0(n) = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases}$$

We now define a function $f(t, a)$ that calculates the number of terminal strings given a number of non-terminals to count. This function is $f(t, a)$ for $a = (a_1, \ldots, a_r)$ which counts $a_i$ number of $N_i$ non-terminals.

$$f(t, a) = \left( (g^{(a_1)}_{N_1}) * \ldots * (g^{(a_r)}_{N_r}) \right)(t)$$

For example if there was three non-terminals in our grammar, then the following would count the number of terminal strings produced from 0 $N_1$, 3 $N_2$ and 1 $N_3$:

$$f(t, (0, 3, 1)) = \left( (g^{(0)}_{N_1}) * (g^{(3)}_{N_2}) * (g^{(1)}_{N_3}) \right)(t) = (g_{N_2} * g_{N_2} * g_{N_3})(t)$$

Let $T(\beta)$ be the number of terminals in $\beta$ and let $A(\beta) = (A_1(\beta), \ldots, A_r(\beta))$ be the vector whose $i$th component is the number of occurrences of non-terminal $N_i$ in $\beta$. Then the function counting terminal strings produced from a sentential $\beta$ is the following.

$$g_{\beta}(n) = f(n - T(\beta), A(\beta))$$

Hickey and Cohen show how the function $f(t, a)$ can be rewritten as the following recurrence

$$f(t, a) = \sum_{j=1}^{n_i} f(t - T(\alpha_{ij}), a + A(\alpha_{ij}) - A(N_i)), \text{ if } a_i \neq 0$$

(The alpha character has been made bold to distinguish from the lowercase $A$.) With the initial conditions $f(t, a) = 0$ for all $a$ and all $t \leq 0$ with the single exception $f(0, (0, \ldots, 0)) = 1$.
The function $g_{N_i}$ can now be simplified to be in terms of just $g_{N_i}$:

$$g_{N_i}(t) = f(t, A(N_i))$$

$$= \sum_{j=1}^{s_i} f(t - T(\alpha_{ij}), A(\alpha_{ij}))$$

$$= \sum_{j=1}^{s_i} \left( g^{(\alpha_{ij})} \ast \ldots \ast g^{(\alpha_{rj})} \right) (t - T(\alpha_{ij}))$$

with $g_{N_i}(t) = 0$ if $t \leq 0$

This equation expresses $g_{N_i}(t)$ in terms of the values of $g_{N_i}(t')$ ($1 \leq t' < t, 1 \leq i \leq r$) so we can precompute the values $g_{N_i}(t)(1 \leq t \leq n, 1 \leq i \leq r)$ to speed up the generation of strings.

Using this function we can finally rewrite the probability that production $\pi_{ij}$ will be used given a state and a termination size. This is the number of strings generated after applying this production divided by the number of strings generated after applying any production.

$$p_{ij}(\beta, n) = \frac{f(n - T(\gamma), A(\gamma))}{f(n - T(\beta), A(\beta))}$$

$$= \frac{f(n - T(\beta) - T(\alpha_{ij}), A(\beta) - A(N_i) + A(\alpha_{ij}))}{f(n - T(\beta), A(\beta))}$$

Precomputing Values

Because the algorithm uses convolution, there are a lot of repeated calls to $g_{N_i}(t)$ for different values of $t$. This is used as an advantage because it means these function values can be precomputed as an optimization. Interestingly it is known that $g_{N_i}(t)$ only depends on function values $g_{N_i}(t')$ where $t' < t$. Therefore it is possible to compute the $g$ functions iteratively from 1 to $n$ where $n$ is the input size and where each function only depends on values already computed.

Now each of the convolution powers $g^{(r)}_{N_i}$ can be computed with at most $2 \cdot \log_2 n$ convolutions since $r \leq n$. For example $g^{(9)}_{N_i} = g^{(4)}_{N_i} \ast g^{(4)}_{N_i} \ast g_{N_i}$.

Hickey and Cohen say that this method has time complexity $O(n^2 \log n + mn^2 (\log n)^2)$ and space complexity $O(n)$ where $n$ is the size of the strings generated and $m$ is the number of terminal strings generated.

Example

Consider the following unambiguous context-free grammar $G$ that generates balances parentheses.

$$G = (V, \Sigma, P, N_1)$$
2 Literature Review

Where $V$ is the set containing the symbols: $N_1$, $N_2$, ‘(’ , ‘)’ and $\Sigma$ is the set containing the symbols: ‘(’ and ‘)’

$$P = \{\pi_{11} : N_1 \to (N_1$$
$$\pi_{21} : N_2 \to N_1N_2$$
$$\pi_{22} : N_2 \to )\}$$

The algorithm will compute the values of $g_{N_1}$ and $g_{N_2}$ for $1 \leq t \leq n$ using the formulas

$$g_{N_1}(t) = g_{N_1}(t - 1)$$
$$g_{N_2}(t) = (g_{N_1} * g_{N_2})(t) + (g_{N_1}^{(0)} * g_{N_2}^{(0)})(t - 1)$$
$$= (g_{N_1} * g_{N_2})(t) + \delta_0(t - 1)$$

![Derivations Tree with Probabilities of Strings of Length 8](image-url)

Figure 2.12: Derivations tree with probabilities of strings of length 8


2.7 Convolution

The convolution that is mentioned in the previous section is known as the Cauchy product, which is the discrete convolution of two sequences. It comes from the branch of mathematics known as combinatorics, which studies discrete structures. The most relevant aspect of combinatorics is the counting of structures of a given size and kind, this is known as enumerative combinatorics.

A main part of combinatorics is representing sequences of numbers by a generating function which is how the convolution is derived. This encodes the sequence as coefficients of a “parameter” $x$. The generating function for the sequence of numbers $a_0, a_1, a_2, \ldots$ is the following:

$$G(x) = a_0 + a_1 x + a_2 x^2 + \cdots = \sum_{n \geq 0} a_n x^n$$

It is important to note that generating functions are not normal functions that map a domain to a co-domain, and the parameter $x$ in this formula has no intrinsic meaning, it is simply used to hold information.

This generating formula has properties that we can use to our advantage. For example we can perform operators on it such as addition and multiplication.

$$\left( \sum_{k=0}^{n} a_k \right) + \left( \sum_{k=0}^{n} b_k \right) = \sum_{k=0}^{n} a_k + b_k$$

$$\left( \sum_{k=0}^{n} a_k \right) \cdot \left( \sum_{k=0}^{n} b_k \right) = \sum_{k=0}^{n} \sum_{i=0}^{k} a_i \cdot b_{k-i}$$

Using this knowledge, generating functions don’t have to be used at all. In order to enumerate a structure $X$, the generating sequence $S_X = (x_0, x_1, x_2, \ldots)$ can be found where $x_k$ is the number of $X$ objects that have size $k$. This is easy if $X$ is composed of other structures with known sequences. In order to count all objects that are either As or Bs, we simply add together the two generating functions of $A$ and $B$ (shown above to be element-wise addition). To count the structure which is the combination of As and Bs we multiplying the generating functions (Cauchy product). This has a direct correspondence with grammars, if we are given the production $A \rightarrow BC$ we wish to count the structure which combines $B$ and $C$. If we are given the productions $A \rightarrow B|C$ then we want to count all of the structure Bs plus all of the structure Cs.

So for example if we had the following sequences representing the number of objects of size $k$:

$$S_A = (0, 0, 1, 0, 1)$$
$$S_B = (0, 1, 1, 0, 0)$$
$$S_C = (0, 1, 1, 1, 1)$$
2 Literature Review

\[ S_X + S_Y = S_Z \text{ where } z_k = x_k + y_k \]
\[ S_X * S_Y = S_Z \text{ where } z_k = \sum_{i=0}^{k} x_i \cdot y_{k-i} \]

so

\[ S_A + S_B = (0, 1, 2, 0, 1) \]
\[ S_B * S_C = (0, 0, 1, 2, 2) \]

In the Hickey-Cohen section, we use convolution on functions, which is done is exactly the same way as if the function is a sequence. Here we have functions \( f(t) \) and \( g(t) \)

\[ f * g = h \]
where \( h(t) = \sum_{i=0}^{t} f(i) \cdot g(t - i) \)
3 Theory

3.1 Hickey-Cohen Generation of Hypergraphs

In this section, we will attempt to use the Hickey-Cohen approach to generating uniform random strings to generate hypergraphs from a hypergraph grammar.

In the previous chapter the functions $g_{\beta}$ are used to calculate the number of terminal strings produced from a non-terminal.

$$g_{\beta}(n) = f(n - T(\beta), A(\beta))$$

It is expressed in terms of the functions $T(\beta)$ and $A(\beta)$ where $T(\beta)$ is the number of terminals in the string $\beta$ and $A(\beta)$ is a vector whose $i$th component is the number of occurrences of $N_i$ in the string $\beta$. We must first redefine these functions for hypergraphs.

Our grammar productions are named the same as before:

$$P = \{\pi_{ij} : N_i \rightarrow \alpha_{ij} | i = 1, \ldots, r, j = 1, \ldots, s_i\}$$

But instead of strings, now $N_i \rightarrow \alpha_{ij}$ is the hypergraph production. $A(\beta)$ is now a vector whose $i$th component is the number of occurrences of the non-terminal edge $N_i$ in the graph $\beta$.

In order to define $A(\beta)$ we must decide what ‘size’ means for a hypergraph. This size must never decrease after applying a derivation and the following property must hold for derivations $d_1$ and $d_2$.

$$T(\beta) + T(\alpha_{ij}) = T(\beta \Rightarrow \pi_{ij})$$

What this means is when replacing a hyperedge with a subgraph ($\beta_1$), the overall size ($T(\beta)$) will increase by the size of the subgraph ($T(\beta_1)$).

I have chosen the following where $t_\beta$ is the number of terminal hyperedges in the hypergraph, $v_\beta$ is the number of vertices, and if $\beta$ is a multi-pointed hypergraph: $x_\beta$ is the number of external vertices in it, otherwise $x_\beta$ is 0.

$$T(\beta) = t_\beta + v_\beta - x_\beta$$

For example see below.
The method now generates a graph of size $n$. To generate a single graph the program does the following.

- Start with a graph $\beta$ which is initial graph
- For each non-terminal hyperedge in $\beta$ where the label of the hyperedge is $N_i$, the following sub-procedure must be done. This is repeated until there is no non-terminals left.

It is worth noting that it doesn’t matter which order the hyperedge is picked since it is known that hyperedge replacement can be performed in any order with the same output.

- Choose a production from the set $(\pi_{i1}, \ldots, \pi_{ik})$ by choosing a random number and using discrete cumulative probability distribution where production $\pi_{ij}$ has the probability $p_{ij}(\beta, n)$ of being chosen.

For example, if the probabilities for $(\pi_{i1}, \pi_{i2}, \pi_{i3})$ are $(\frac{1}{2}, \frac{1}{4}, \frac{1}{4})$. A random number $0 \leq r \leq 1$ is used to choose the production $p$ accordingly. Therefore

$$
p = \begin{cases}
\pi_{i1}, & r \leq 0.5 \\
\pi_{i2}, & 0.5 < r \leq 0.75 \\
\pi_{i3}, & 0.75 < r \leq 1
\end{cases}
$$

If a probability $\pi_{ij}$ has a value of zero, this means that applying this production can never produce a graph of size $n$ and therefore should not be considered in the above choice.

- The production $p$ is now applied the the graph $\beta$ which may add extra terminals and non-terminals.

- $\beta$ is now a random graph of size $n$.

If the user wants multiple graphs he can run this procedure as many times as needed. Of course if the user generates $x$ many graphs but there are less than $x$ unique terminal graphs in the grammar at the chosen size, then it is certain that at least 1 graph will be repeated.
3 Theory

3.1.1 Example

We will use the example grammar \( \langle N, T, P, A \rangle \) where\( N = \{ A \} \), A is the 1-handle labelled ‘A’ and \( P \) contains the following productions.

The program first computes values for \( g_A(t) \) for \( 1 \leq t \leq n \) using the following equation.

\[
g_{N_i}(t) = \sum_{j=1}^{s_i} \left( g_{N_{i1}}^{(\alpha_{ij})} \ast \cdots \ast g_{N_{ir}}^{(\alpha_{ij})} \right) (t - T(\alpha_{ij}))
\]

\[
g_A(t) = g_A(t - 2) + (g_A^{(2)})(t - 4) + \delta_0(t)
\]

Results for \( n = 11 \).

<table>
<thead>
<tr>
<th>( g_A(t) )</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>2</th>
<th>0</th>
<th>4</th>
<th>0</th>
<th>9</th>
<th>0</th>
<th>21</th>
<th>0</th>
</tr>
</thead>
</table>

And now the program can generate as many graphs as needed, here we have generated six. The edge labels have no been shown here for clarity.

3.2 Ambiguous Grammars

In general an ambiguous grammar is a grammar where there is a string that can be derived in more than one way. This applies to hypergraph grammars in the same way.
3 Theory

The Hickey-Cohen generation algorithm requires the input grammar to be unambiguous in order to uniformly generate terminal graphs. However, if the input grammar is ambiguous then the algorithm will generate graphs where each left-hand derivation is uniformly distributed. Therefore the probability of a graph being generated is proportional to the number of left-most derivations of the string.

An example of an ambiguous grammar is the grammar \( \langle N, T, P, A \rangle \) where \( A \) is the 1-handle labelled \( A \) and \( P \) contains the following productions.

\[
\begin{align*}
A & \to \pi_{11} \\
A & \to \pi_{12} \\
A & \to \pi_{13}
\end{align*}
\]

The ambiguity comes from the production \( \pi_{12} \) where the two hyperedges labelled \( A \) may be replaced differently. If \((a, b)\) is the replacement of the left hyperedge with \(a\) and the right hyperedge of \(b\), then the resulting hypergraph is isomorphic the replacement \((b, a)\)

The Hickey-Cohen approach using \( n = 7 \) will count 4 distinct graphs and select at random from the following.

As you can see, the first two graphs are isomorphic even though the algorithm treats them different because of the different derivation paths.

Unfortunately it is impossible to decide whether a grammar is ambiguous in the general case, and impossible to convert an ambiguous grammar to an unambiguous grammar. So it should be up to the user to be aware of the problems with ambiguous grammars and to deal with the output himself.

3.3 Cyclic Grammars

A cyclic grammar is a grammar where a non-terminal can derive exactly itself. This can be directly \((A \Rightarrow A)\) or indirectly \((A \Rightarrow B \Rightarrow A)\). If a grammar contains epsilon productions, it may be cycling also in the case where \(A \Rightarrow AB\) and \(B \Rightarrow \epsilon\)

The Hickey-Cohen approach is invalid on a cyclic grammar because the number of deriving strings from a cyclic non-terminal is infinite. We can observe this by expanding the generating function for a a simple grammar \(A \Rightarrow B|a\) and \(B \Rightarrow \epsilon\). The generating function becomes \(g_A(t) = g_B(t) + \delta_0(t-1) = g_A(t) + \delta_0(t-1)\) which is recursive and cannot be computed.

The simplest way to convert a cyclic grammar into a cycle-free grammar is to simply replacing every single production whether it cycles or not. In order to do this, we can use
3 Theory

a modified version of Algorithm 2.11 from Aho and Ullman [1]. The algorithm requires that the input grammar is $\epsilon$-free which is covered in the next section.

In the algorithm, the notation $\text{singleton}(X)$ is used to specify a multi-pointed hypergraph $H$ which consists a single edge labelled $X$ where each tentacle of the edge is attached to a distinct vertex and the only vertices in the graph are external nodes. So $E_H = \{e\}, \text{lbl}(e) = X, V_H = \text{ext}_H$ and $\text{att}_H(e, i) = \text{ext}_H(n)$.

The notation $\text{frame}$ is used later to specify a multi-pointed hypergraph $H$ which consists of just external nodes ($E_H = \{}$ and $V_H = \text{ext}_H$).

Figure 3.1: Example of a $\text{singleton}(C)$ with 2 vertices (left) and a $\text{frame}$ with 3 vertices (right)

- Given an $\epsilon$-free hypergraph grammar $G = \langle N, T, P, S \rangle$
- Construct for each $A$ in $N$ the set $N_A = \{B \mid A \Rightarrow B \}$ as follows.
  1. Let $N_0 = \{A\}$ and set $i = 1$
  2. Let $N_i = \{C \mid P$ contains $B \rightarrow \text{singleton}(C)$ and $B \in N_{i-1}\} \cup N_{i-1}$
  3. If $N_i \neq N_{i-1}$ set $i = i + 1$ and repeat step 2. Otherwise let $N_A = N_i$

Since $N_i \subseteq N$ then this procedure will terminate after a finite number of steps.

- Let $P'$ be the set of productions constructed by the following: If $B \rightarrow H$ is in $P$ and is not a single production, add $A \rightarrow H$ to $P'$ for all $A$ such that $B \in N_A$.

  'Single production' is a production $B \rightarrow \text{singleton}(X)$ where $X \in N$

3.4 Grammars with $\epsilon$-productions

An $\epsilon$-free grammar is a grammar which has no derivations to $\epsilon$ or there is exactly one $\epsilon$-production $S \rightarrow \epsilon$ and $S$ does not appear on the rightside of any production. An $\epsilon$-production has different meaning depending on the type of grammar. In string grammars an $\epsilon$ is the empty string, in hypergraph grammars $\epsilon$ is a graph containing just external nodes.

It is useful to be able to transform an $\epsilon$ grammar into a $\epsilon$-free grammar. Below I have modified Algorithm 2.10 from Aho and Ullman [1] to work with hypergraph grammars.

- Given a hypergraph grammar $G = \langle N, T, P, S \rangle$
3 Theory

- Construct \( N_\epsilon = \{ A | A \in N \text{ and } A \Rightarrow \epsilon \} \). This is the set of non-terminals which produce a \( \epsilon \) in 1 or more derivations.

1. Let \( V_0 = \{ a | a \in N \text{ and } P \text{ contains } a \rightarrow \text{frame} \} \) and set \( i = 1 \)
2. Let \( V_i = \{ a | a \in N \text{ and } P \text{ contains } a \rightarrow \text{singleton}(X) \text{ where } X \in V_{i-1} \} \cup V_{i-1} \)
3. If \( V_i \neq V_{i-1} \) set \( i = i + 1 \) and repeat step 2. Otherwise let \( N_\epsilon = V_i \)

Since \( V_i \subseteq N \) then this procedure will terminate after a finite number of steps.

- Let \( P' \) be the set of productions constructed by the following
  1. If the production \( A \rightarrow \langle V, E, att, lbl, ext \rangle \) is in \( P \), where the set of edges \( E \) is the following:
     \[
     E = \{ \alpha, B_1, B_2, \ldots, B_k \}, \ k \geq 0
     \]
     Where each \( B_i \) is in \( N_\epsilon \) but no symbols in \( \alpha \) are in \( N_\epsilon \), then add to \( P' \) all productions \( A \rightarrow \langle V, E', att, lbl, ext \rangle \), where the new set of edges is in the form of the following:
     \[
     E' = \{ \alpha, X_1, X_2, \ldots, X_k \}
     \]
     Where \( X_i \) is either \( B_i \) or \( \emptyset \), and either \( E' \neq \{ \} \) or \( V \neq ext \) (We do not want to add another \( \epsilon \)-production). This effectively constructs a new production removing every combination of the edge \( B_i \) from the hypergraph \( H \).
  2. Let \( s = \text{label}(S) \) and \( n = |\text{ext}_S| \).
     If \( P \) contains \( s \rightarrow N_\epsilon \), add to \( P' \) the productions \( s' \rightarrow H \epsilon | S \) where \( s' \) is a new symbol and \( H \epsilon \) is a multi-pointed hypergraph with just \( n \) external nodes, let \( N' = N \cup \{ s' \} \) and let \( S' \) equal a \( n \)-handle labelled \( s' \).
     Otherwise let \( N' = N \) and \( S' = S \)

- Let \( G' = \langle N', T, P', S' \rangle \)

To make an example of this algorithm, we construct a grammar \( \langle N, T, P, A \rangle \) where \( A \) is a 1-handle with the label \( A \) and \( P \) contains the following. Note the third production is an \( \epsilon \)-production.

\[
\begin{align*}
\text{After applying the algorithm the resulting grammar is } &\langle N, T, P, A' \rangle \text{ where } A' \text{ is the 1-handle with the label } A' \text{ and } P \text{ contains following.}
\end{align*}
\]
This will generate the same language as the initial grammar.
4 Implementation

4.1 Language

The language and platform that the program runs on needs to be considered. Since my main operating system is Apple Mac OS X and the university's main operating systems are Microsoft Windows and Linux then it is required that the language chosen will run on these three operating systems.

4.1.1 The Java Virtual Machine

The first obvious choice is the Java system by Sun Microsystems in 1991 and now Oracle Corporation since 2009. Java is a cross-platform runtime environment which runs on over 1.1 billion desktop computers worldwide [7].

Java achieves cross-platform by using a virtual machine called Java Virtual Machine or JVM. Java programs are written and compiled into Java bytecode which can be subsequently run on any JVM implementation. JVM implementations exist on all major operating systems and Mac OS X bundles a JVM with installation.

Other than the excellent cross-platform support, the JVM has more advantages. The Java runtime has great built-in graphical user interface library called Swing which can be used to create interfaces that will work on each operating system. This is good for the program created for this project since it provides a unified interface.

The maturity and widespread use of the JVM leads to many highly used third-party libraries. Many different graph libraries exist that will be considered later.

The fact that the Java system is used by billions of devices ensures that support for the JVM will continue into the future. This is important for users such as myself that wish to devote time into building an application and want that application to continue to work in many years to come.

The speed of the JVM has increased over the years due to many optimisation techniques such as Just-in-time compilation and HotSpot which analyses a programs performance and optimised frequently executed code paths. Another technique is garbage collection which attempts to reclaim memory that is no longer in use by the program.

Programs that involve computationally expensive procedures—such as this project—it is beneficial that the program can distribute computations across multiple cores of a computer. This is only possible if the runtime environment supports concurrency. Java has been designed to support concurrency by having built-in constructs for threading.

Java is also the name of the main programming language that is supported by the Java runtime. It has a similar style to the popular languages C/C++ but currently lacks
in certain modern language features to aid development. The Java Virtual Machine runs Java bytecode which means third-party languages are being developed for it such as Groovy, Clojure and Kotlin. In fact, any language can be compiled to Java bytecode as long as it can be internally expressed as Java bytecode. Some examples include JavaScript as Rhino, Python as Jython and Ruby as JRuby.

Scala is such a language that has been developed to run on the JVM. Scala first appeared in 2003 and has been a popular choice over Java for a number of reasons. Scala has all the positive aspects of running on the JVM as well as improved syntax, type inference, pattern matching, operator overloading. These features help to improve expressiveness.

Scala also has many functional characteristics such as anonymous functions and closures as first-class citizens, immutable data structures, map/filter/fold methods. These features are a huge benefit over Java especially when implementing mathematical concepts as this project does. For example using higher-order functions makes it easier to describe exactly what needs to be computed while abstracting implementation details away.

On the other hand, Scala is a relatively new language compared to Java and as such it is slightly 'rough around the edges'. The Integrated Development Environment support is not as good as Java and occasionally compile errors are slightly confusing but this is a steady process and these things are improving over time. There is also new syntax to learn which is a potential problem for new users of the language.

In conclusion, using Scala provides improved programming constructs while still employing the benefits of the JVM.

4.2 External Libraries

A code library is a collection of code that one or more people have brought together in a way that other people can use. The use of external libraries in an application can aid development because this code is generally proved to work by use of automated tests. This can cut time of development since it is a section of code that the developer does not need to worry about.

For this project, it would be beneficial to use a graph library where many graph modification operations are already implemented.

Java Universal Network/Graph Framework

Java Universal Network/Graph Framework or JUNG is a Java library that provides functions for the modelling, analysis and visualisation of graphs and networks. The JUNG architecture supports a variety of graph representations such as directed/undirected, multi-graphs and importantly hypergraphs.

JUNG provides a highly extensible visualisation framework which enables a developer to show the structure of a graph in a range of different ways with custom layouts, filtering mechanisms and styles.
4 Implementation

The library was designed by a software engineer working at Google and a researcher at Microsoft. It is also partially funded by the National Science Foundation which shows the library is high Unfortunately the last update was in 2011 but it isopensource and available under the Berkeley Software Distribution (BSD) license so this means anybody is free to contribute to it.

Internally, JUNG uses interfaces to refer to different types of graph. In programming an interface defines what operations the object must have but the implementation can choose how to do it. JUNG supports hypergraph interfaces but only has a concrete implementation for `set hypergraphs' which do not order the tentacles. Fortunately JUNG is designed to be extensible and implementing an sequential hypergraph is trivial. Furthermore, there is no built-in support for visualising hypergraphs but again, this can be done fairly easily.

4.3 Algorithm Implementation

I ran into a few issues while implementing the algorithms. Here I describe them.

The first issue I ran into when implementing the Hickey-Cohen algorithm was the convolution function.

\[(v * w)(n) = \sum_{k=0}^{n} v(k) \cdot w(n - k)\]

In certain cases there existed a recursive function: \(g_{N_1}(n) = (g_{N_1} * g_{N_2} * \ldots)\) and when \(k = 0\) then \(v(0) \cdot w(n)\) must be calculated and when \(k = n\) then the same for \(v(n) \cdot w(0)\). In both cases to naively compute this function would result in an infinitely recursive loop even though mathematically it is known that \(f(x) \cdot 0 = 0\) for any function.

The fix for this was to redefine the convolution and to deliberately ignore \(k = 0, n\) cases. The new convolution is the following:

\[(v * w)(n) = \sum_{k=1}^{n-1} v(k) \cdot w(n - k)\]

This is equivalent to the previous function only if it is certain that \(w(0) = 0\) and \(v(0) = 0\), which means there are no \(\epsilon\)-productions in the grammar.

The second issue I came across was the \(\epsilon\)-free conversion algorithm from Aho and Ullman \[1\] producing an unterminating grammar. The algorithm removes productions in the form of \(A \rightarrow \epsilon\) and duplicates all other productions containing the \(A\). However, if the only production deriving from \(A\) is \(A \rightarrow \epsilon\), then the production is removed but other productions may still refer to the non-terminal \(A\). This results in a grammar that has \(A\) non-terminals but no \(A\) productions and therefore cannot produce a terminal string. To work around this issue I make sure that the inputted grammar does not contain just a single \(\epsilon\)-production.
4 Implementation

4.4 Program

4.4.1 Grammar Editor

Before any algorithm can be run, there needs to be a grammar. The grammars can be written out in textual format but this is tedious. The program has a Graphical User Interface (GUI) that can be used to design the hypergraph grammar. Note that this is meant to supplement the program and is not the main focus, so it is not fully detailed here.

The program is nicknamed ‘Graphite’ and can be run by entering the command `graphite` in the terminal. Once it is open it consists of a list of production rules which the user can select and edit on the right.

The grammars can be saved to and loaded from the hard disk in order to reuse the grammar at a later time or to share the grammar with another person.

Clicking the ‘Generate’ button in the lower left corner brings up the generate dialogue window where the user can choose options for generating the graphs.

![Figure 4.1: The grammar window showing the initial graph](image)
4 Implementation

Figure 4.2: The grammar window showing a rule

Figure 4.3: Starting the generator from the GUI
4 Implementation

4.4.2 Command Options

The program can also be run in command-line mode. This is useful for repeating commands because one can copy and paste the command into the terminal. The help text can be shown by writing `graphite --help` into the terminal. It brings up the following help text.

```text
graphite
Opens the graphite gui

graphite gui filename
Opens the graphite gui with a specified file

graphite generate --size=int [--number=int] [--verbose] [--distinct] [--dontopen] filename
Generates a number of graphs with a specified size
  size : The size of graph to generate.
  number : The number of graphs to generate. Default 1
  verbose : Output detailed information. Default false
  distinct : Collect distinct graphs together. Default false
  dontopen : To stop the graphs being opened in the GUI. Default false

graphite enumerate --size=int filename
Counts the number of graphs with a specified size
  size : The size of graph to count

graphite benchmark --size=int [--increment=int] [--number=int] filename
Generates graphs with sizes iterating from 1 to a given size
  size : The maximum size graph to generate, optionally use a range eg 1..10
  number : The number of graphs to generate at each iteration. Default 1
```

The `generate` command is used to perform the generate algorithm on a specified grammar. Arguments in square brackets are optional so the only arguments that are needed is the graph size and the filename.

Some options worth noting are firstly `--dontopen` which is used to stop the graphs being opened in the GUI. This is useful to see how long the graphs took to generate and are not concerned with the actual output. Secondly the `--distinct` option is used to show only graphs which are distinct. Distinction is made by derivation paths and not isomorphism so this will give incorrect results on ambiguous grammars.

Some examples of commands are listed here.

1. `graphite generate --size=101 --number=5 data/treegrammarlr.xml`
   This will generate 5 graphs of size 501 from the treegrammarlr.xml grammar and then open the results in the GUI. The results in the terminal will look like the following.

   Loaded file: data/treegrammarlr.xml
   Converting to non-epsilon
   Added 4 and removed 1 productions
   Running algorithm...
   Generating 5 of 5
   Time to compute: 1,036 milliseconds
4 Implementation

2. `graphite generate --size=13 --number=800000 --distinct data/abgrammar.xml`
   This will generate 800,000 graphs of size 13 from the abgrammar.xml grammar and show the distinct results in the GUI. It will also display how long it took to compute. The results in the terminal will look like the following.

   Loaded file: data/abgrammar.xml
   Running algorithm...
   Precomputing took 78 ms
   Total number of terminal graphs: 8
   Generating 800,000 of 800,000
   Time to compute: 25,417 milliseconds
   Distinct graphs:
   0 -> 100,133
   1 -> 100,058
   2 -> 100,019
   3 -> 100,049
   4 -> 100,025
   5 -> 99,961
   6 -> 99,928
   7 -> 99,827

The `enumerate` command is used to simply calculate how many terminal graphs exists given a graph size. Some examples are shown below.

1. `graphite enumerate --size=11 data/flowgrammar.xml`
   This will calculate how many flow graphs of size 11 exists. It also displays the time it took to compute. Below is an example output.

   Loaded file: data/flowgrammar.xml
   Number of terminal graphs with size 11: 3,670,016
   Time to compute: 96 milliseconds

2. `graphite enumerate --size=301 data/flowgrammar.xml`
   This will do a similar thing and calculate how many flow graphs of size 301 exists. The number of terminal graphs here is very large so it is shown in mathematical standard form.

   Loaded file: data/flowgrammar.xml
   Number of terminal graphs with size 301: 1.62 * 10^-266
   Time to compute: 13,154 milliseconds
4 Implementation

Finally, the benchmark command performs the generating algorithm iteratively over a number of sizes. For example using the range 1..500 will generate a graph of size 1, then 2, then 3 etc. up to 500. This is useful for seeing how long different graphs take to generate and is used later in the report in the testing chapter.

The program first performs a ‘prerunning’ stage which runs the algorithm a few times. Without doing this it happens that the first couple of iterations will be very slow. This happens because the Java Virtual Machine takes a few iterations to grow the memory heap by a substantial amount.

The program outputs a string like ‘Time 2 ms + 0ms (2 ms)’. The first two numbers are the precalculating time and the generating time. The first number is both combined.

1. graphite benchmark --size=100 data/abgrammar.xml

This generates a graph at size 1, size 2, etc up to size 100. Below is an example of the program output.

Loading file: data/abgrammar.xml
Prerunning...
Running benchmark...
Size 3. Time 0 ms + 0 ms (0 ms)
Size 5. Time 0 ms + 0 ms (0 ms)
Size 7. Time 0 ms + 0 ms (1 ms)
...
Size 97. Time 1 ms + 2 ms (4 ms)
Size 99. Time 1 ms + 2 ms (3 ms)

4.4.3 Graph Results

When the user has selected the options he wants in the Generate window or he has run the algorithm from the command-line, the graphs are shown in the Output window as shown below.
4 Implementation

Figure 4.4: The output window

If the output graph is a tree then an option can be clicked in the menu to lay out the graph in a way that’s easy to view. Otherwise the user can use the mouse to move and manipulate the vertices to get some kind of layout that he prefers. The mouse can be used to zoom and translate the output window, and also show tentacle labels when hovering over an edge.
4 Implementation

Figure 4.5: The output window showing a tree structure

Figure 4.6: The output window
5 Testing

A number of different grammars have been tested for different attributes. The main thing to test is uniformity to verify that the program is correct. Secondly the speed of the program is tested. Although the time it takes to generate graphs is not a major concern of the project, it's useful to know for further research into the subject.

These are the grammars I have chosen to test. They are of different sizes and different number of non-terminals.

1. Binary Tree Grammar - The grammar \( \langle N, T, P, S \rangle \) where \( N = \{A\} \), \( S \) is the 1-handle labelled ‘A’ and \( P \) contains the following productions

   \[
   A \rightarrow 1 A l A r 1
   \]

   An ambiguous tree grammar can also be used in testing by removing the \( l \) and \( r \) labels on the above tree grammar.

2. AB Palindrome Grammar - The grammar representing the strings that contain palindromes of \( a \)s and \( b \)s. It is the grammar \( \langle N, T, P, S \rangle \) where \( N = \{C\} \), \( S \) is the 2-handle labelled ‘C’ and \( P \) contains the following productions

   \[
   C \rightarrow 1 a C a 2 1 b C b 2 1 a a 2 1 b b 2 1 a 2 1 b 2
   \]

3. Flow Grammar - A grammar of flow charts which can be found in Figure 2.8. This is the largest grammar that is tested but by no means the largest grammar that may be used by a user.
5 Testing

5.1 Uniformity

In order to test the program for uniformity we can group together the derivation paths of the generations. The derivation path of a grammar is a list of left-most derivations, in graph grammars obviously there is no notion of a left-most derivation but constructing a sequence of non-terminals and replacing the head of the sequence gives the same effect.

I have tested different grammars with different parameters to give a good range of results. The error indicates the maximum deviation from the assumed graph-count of a perfectly uniform distribution. What we expect that each derivation path has an equal number of graphs each with an error getting smaller with a larger sample size.

5.1.1 Binary Trees

The program was tested to generate 90,000 binary trees of size 9, resulting in 9 distinct graphs. The distribution ranged from 9,916 to 10,096, so this is an error of ±0.4%.

The generated graphs are shown below, the labels have been removed for simplicity.

The ambiguous tree grammar was also tested to show the distribution where some graphs have two left-most derivations.
5 Testing

And the generated graphs are shown below.

AB Palindrome

The program was tested to generate 800,000 graphs of size 13 from the AB Palindrome grammar, resulting in 8 distinct graphs. The distribution ranged from 99,654 to 100,414 and this is an error of ±0.96%

AAAADA, AABAAA, ABBAAA, BAAAAA, BDBBBB, BBAABB, BBABBB, BBBBBB
5.1.2 Flow Graphs

Finally, the program took 6,566 ms to generate 64,000 flow graphs of size 5, resulting in 64 distinct graphs. The distribution ranged from 921 to 1067 and thus an error of ±7%. A smaller number of graphs have been generated here due to the amount of time it takes. This makes the error percentage seem quite high but for comparison when generating a lower number of graphs, the error for AB graphs is ±6%, and for tree graphs is ±5%.

In conclusion, the program definitely seems uniform given a large enough number of graphs and therefore the algorithm seems to be a success.

5.2 Speed

The speed of the program is measured by how long it takes to generate a certain number of graphs of different sizes.
5 Testing

Time to Generate 100 Binary Trees

Time to Generate 100 AB Palindromes
As shown, the flow grammar took far longer to generate the same number of graphs with the same number of size as the other two grammars. This is because the flow grammar is more complex, but it is surprising how much of a difference it is.

Interestingly the generation time of the AB palindrome grammar looks to be linear. This is due to the derivation of AB grammars only having 1 non-terminal at any time.
6 Conclusion

The Hickey-Cohen approach has been successfully applied to hypergraph grammars to uniformly generate graphs instead of strings. This has an advantage over current solutions such as Stanford Graphbase that have little control over what kind of graphs can be generated. It also has the advantage over Ranjan’s approach that every graph produced has an equal probability of being generated. However, there are still things that can be improved.

An issue is with the program is the input of ambiguous grammars. A better program might be able to generate a uniform distribution even if an ambiguous grammar has been given. Alternatively the program could implementing an algorithm that filters out isomorphic graphs instead, but of course this will increase the generation time of the program. An algorithm that has been developed to remove duplicate isomorphisms has been detailed in Cavanagh [2].

The memory used by the program in order to generate graphs has not been optimised but since the user has to wait a very long time before this is an issue, the main point of optimising should be the speed of the program. There are a few things that can be looked into to potentially speed up the program.

1. The program is written in Scala, which is a language that sacrifices speed with ease-to-understand by implementing abstractions over the underlying executable code. This has a performance on speed and memory. Perhaps the program can be implemented in a lower level language such as C or C++ to directly control memory allocation and improve the speed of generation.

2. The convolution function in particular is the speed bottleneck of the program. Hickey and Cohen claim that convolution which has time complexity $O(n^2)$ can be written using the Fast Fourier Transform algorithm which has time complexity $O(\log n)$, this should be looked into in order to improve speed.

3. Computers nowadays have multiple cores that can run programs in parallel. An algorithm that takes advantage of this is described in Gutjahr [3] which performs parts of the algorithm is parallel.

A further improvement would be to research or develop an algorithm which generates graphs from a grammar which is not context-free. This would mean many more sets of graphs could be generated to improve testing methods.

An interesting addition to the algorithm would enable the input of a range of sizes where the graph generation is uniform across this range. I am quite certain this is straightforward but due to time constraints I could not look into it.
6 Conclusion

A final point that could be improved upon is the output of the hypergraphs. The current implementation performs very basic visualisation techniques for hyperedges that could definitely be improved upon. A preferred approach would be to output to an intermediate file format that an external program could open and be manipulate with. This was a goal of the project but again due to time constraints it wasn’t possible. Further, one could look into graph layout algorithms such as a force-directed algorithm to improve the quality of the output and make the graphs much more readable.
Bibliography


