Synthetic Network Generation

Dylan Kelly

BSC COMPUTER SCIENCE & INFORMATION TECHNOLOGY
NATIONAL UNIVERSITY OF IRELAND, GALWAY

MARCH, 2017

Abstract

This research proposes and evaluates a method for the generation of synthetic networks. This method allows for the generation of networks that exhibit desirable properties. The properties and dimensions required in synthetic networks are of course, application dependent. An evolutionary algorithm is employed by the method to optimize the graphs generated. An evolutionary computing framework is developed in Scala to evaluate this method over various configurations of desired network properties and measurements. An academic paper based on the results of this research, titled "Synthetic Network Generation, has been written and will be submitted for peer review to the International Conference on Evolutionary Computation and Theory (ECTA) as of April 5th, 2017.

Acknowledgments

I'd like to express my sincerest gratitude and appreciation to Dr. Colm O'Riordan for being a fantastic supervisor, mentor and friend. Completing this research project was immensely enjoyable and this is true, in no small part to Colm's knowledge, continuous encouragement and in particular, the interest and enthusiasm he showed in his role as my supervisor. It was a pleasure to work with you.

Contents

0

Introduction

In this section, a short introduction to this project is provided. The motivations, aims, and goals of this research are defined and discussed. A list of contributions to date and an overview of the structure of this report are also included.

0.1 Introduction

In this project, a method for generating synthetic networks will be designed, implemented and evaluated through a number of experiments. A synthetic network is one that is designed from the top-down, with an application in mind; this is in contrast to naturally occurring networks, which typically emerge over time, from the bottom-up. Graph theory provides the formal language in which we can measure and evaluate the generated networks. A probabilistic, rule based approach to network generation is designed and implemented: networks are constructed from a set of building blocks, called rules, and each rule has an associated probability of being fired. The choice of probabilities forms an optimization problem, of which an evolutionary algorithm is employed to solve. Experiments are carried out to evaluate the ability of the method to generate graphs across a variety of conditions and detailed analysis of this evaluation is provided.

0.2 Open Questions & Motivations

Many open questions arise on how to generate a network synthetically. This will be the primary area of investigation for my project. Network Science has seen an exploding increase in interest recently; this is due to the massive amount of network data that can now be collected and analyzed. This network data may not always be easy, or inexpensive, to collect however. Analysis of high-quality synthetic network data, which can be cheaper and easier to obtain, offers a promising alternative. Synthetic networks exhibiting certain properties, such as a high degree of robustness, or efficiency are also required in many practical applications.

0.3 Research Questions

- Can the proposed generation method produce networks that exhibit specific properties?
- Is the stochasticity of the generation process a large factor in the method's ability to generate networks with the desired properties?
- Can the method generate networks when a combination of specific properties is desired?
- What happens if the combination of specific properties desired are incongruent with each other?
- What impact does introducing a global objective, such as a desired average path length or diameter value, have on the ability of the method to find desirable networks?

0.4 Goals & Aims

The primary aim of this work is to design, implement and evaluate a method for the generation of synthetic networks. In order to achieve this objective, a substantial amount of background research will be conducted into the fields of network science, graph theory and evolutionary algorithms.

A substantial deliverable for this project will be a free-standing evolutionary algorithm (EA) framework; this system should be designed to be flexible and easily applicable to new problems. A graph generation process will be designed and implemented, and will utilize the EA framework that was developed.

A number of research questions have been formulated and proposed, and a series of experiments will be designed and conducted, in the hopes of answering these questions. Each experiment should include a detailed description of results, and a thorough analysis of these results should be carried out, with respect to the research questions proposed.

0.5 Contributions

The contributions to date include:

- A flexible, multi-purpose evolutionary algorithm framework, usable in future projects.
- A highly extensible network generation system, that could be expanded upon in future projects to perform new experiments and research.
- A series of experiments and results, which can and are being used to form the basis of academic publications and further experimental research into the generation of synthetic networks.
- A significant list of future work outlining promising possible extensions, which could form the basis of many future projects.

0.6 Report Structure

- Chapter 1 provides a brief introduction to the fields of Network Science, Graph Theory, and Evolutionary Algorithms (EAs).
- Chapter 2 details the design of the network generation process proposed. A detailed description and high level diagram of the EA used for optimization is also provided.
- Chapter 3 describes the Scala implementation of the network generation method, as well as the the EA framework that was developed including a complete diagram of the system architecture.
- Chapter 4 outlines the experiments: the aim, motivations, parameters, and the observed results, including a detailed description and analysis.
- Chapter 5 concludes this report, including discussion of results and evaluation of the project goals/aims. An extensive list of possible extensions/future work is also provided.

1

Background

This research combines ideas from multiple domains of computer science and mathematics, including network science, graph theory and evolutionary computing. This section provides a concise introduction to these domains, and imparts to the reader the background knowledge required from each.

1.1 Networks

We are surrounded entirely by vast, complex systems that are essential to life as we know it. Behind each complex system there is an intricate network that encodes the interactions between the system's components. Given the important role complex systems play in biology, science, communications and in economy, their understanding, mathematical description and prediction are of great interest. In a network, individual components are represented as nodes, and connections are called links. It is often very difficult to discern collective behavior from a knowledge of a complex system's components. When we study the underlying network of a system we are interested in the patterns of interactions between nodes; these patterns determine the behavior of the system as a whole.

Network science is a new interdisciplinary field that draws on theories and methods from computer science, mathematics, biology and sociology to study the networks that underlie these complex systems.

1.1.1 Examples of Networks

Networks are at the heart of some of the most revolutionary technologies of the 21st century, empowering everything from the Google search engine, the social networks Facebook and Twitter and the physical networks developed and maintained by CISCO. Networks permeate science, technology, business and nature to a much higher degree than may be evident upon a casual inspection - they truly are ubiquitous.

Networks can represent both physical and intangible complex systems; some examples of networks can be seen in Figure 3.1.

Network	Nodes	Links
Internet	Routers	Internet Connections
WWW	Webpage	Links
Email	Email addresses	Emails
Citation Network	Papers	Citations
Power Grid	Power Plants, Transformers	Electric Cables
Metabolism	Metabolites	Chemical Reactions
Electronic Circuit Boards	Transistors, Capacitors,..	Wires

Figure 1.1: A few examples of important networks

1.1.2 Network Attributes & Properties

Networks have certain attributes that can be calculated in order to analyze their properties & characteristics. These network properties often define network models and can be used to analyze how certain models compare to each other. The attributes influence the network structure and the observed behavior of the network as a whole.

A number of the network properties considered in this research are introduced below:

- Robustness - the ability of a network to withstand failures and perturbations, and is a critical attribute of many complex systems. In engineering, network robustness can help to evaluate the resilience of infrastructure networks such as the Internet or power grids. For biologists, network robustness can help the study of diseases and mutations, and how to recover from some mutations.
- Efficiency - A measure of how efficiently a network exchanges information. Network properties are often related i.e. the local efficiency quantifies a network's resistance to failure on a small scale.
- Clustering - Social networks tend to break up into groups of close friends. The same is true for other types of real world networks. The way a network breaks down into communities can reveal details about the organization of a network that are impossible to see without network data.

Average path length and *density* are useful network measurements when evaluating robustness and efficiency. They are introduced below, and will be defined using the formal language of graph theory in Section 1.2.5.

- Average path length - The average number of steps along the shortest paths for all possible pairs of network nodes. The average path length distinguishes an easily negotiable network from one which is complicated and inefficient.

- Density - The ratio of the number of links in a network to the maximum number of links possible for that network.

1.1.3 Network Emergence & Synthetic Networks

In most cases, networks emerge naturally and grow over-time. As a network grows over time, the structure often begins to develop in undirected and unforeseeable ways. Social networks and biological networks (via evolution) are great example of this phenomenon; These systems are unplanned: they simply emerged over time, from the bottom-up.

On the other hand, synthetic networks are networks that are designed from the top-down, having an end-goal: they are created with an optimal network structure in mind. A good example of a synthetic network is an electronic circuit board; the individual components (transistors, capacitors) have an underlying network representation which are designed from the top-down using high level technical plans.

It's important to note here that as the number of components in the network grows in a linear fashion, the number of connections between them may grow exponentially. Designing and optimizing a synthetic network is a difficult problem, due to this inherent complexity.

1.2 Graph Theory

To understand the many ways networks can affect the properties of a system, we need to become familiar with graph theory. Graph Theory is the study of graphs, which are mathematical structures used to model pairwise relations between objects. In this context, graph theory is the study of network structure; it provides us with a formal mathematical language in which to measure networks and their properties.

A graph can serve as a mathematical model of a network - representing networks as graphs allow us to utilize the measurements and concepts from graph theory to gain insight into the structure of networks. In fact, the terms network and graph are used interchangeably in the scientific literature.

1.2.1 What is a Graph?

Definition: A graph $G = (V, E)$ formally consists of a set of vertices V and a set of edges E between them. An edge $e = \{i, j\} \in E$ connects vertex v_i with vertex v_j.

In simpler terms, graphs are a way to represent a collection of objects where some pairs of these objects are connected to each other in some way. The objects are called *vertices* and the links that represent the pairwise connections are called *edges*. It's important to note here that *Vertex* and *Edge* are synonymous with *Node* and *Link*, respectively. I'll use the terms interchangeably in this report.

We denote the *vertex set* of graph G as: $V(G)$ and the *edge set* of G as: $E(G)$.

Two vertices v_i and v_j, connected by an edge $e = \{i, j\} \in E$ are said to be *adjacent*.

Figure 1.2 depicts a basic graph G=(V,E) with labeled vertices. V(G) = {A,B,C,D}. E(G) = { {A,B}, {A,C}, {B,C}, {B,D}, {C,D} }

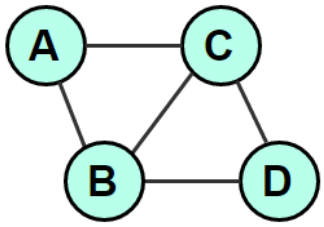

Figure 1.2: A undirected graph G with four labeled vertices.

1.2.2 Directed & Undirected Graphs

Graphs can have two types of edges: directed and undirected. Undirected edges represent a symmetric (two-way) relationship between two vertices. An undirected edge $e = \{A, B\} \in E$ is an unordered pair of two distinct vertices A, B, with $\{A, B\} \equiv \{B, A\}$. Graphs that have an edge set that strictly contains undirected edges are known as *undirected graphs*.

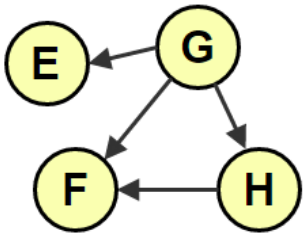

Figure 1.3: A directed graph G with four labeled vertices.

Directed edges represent an asymmetric (one-way) relationship between two vertices. An undirected edge $e = \{C, D\} \in E$ is an unordered pair of two distinct vertices C, D, with $\{C, D\} \not\equiv \{D, C\}$. Graphs with an edge set consisting of directed edges only are called *directed graphs*. Figure 1.3 is a depiction of a basic directed graph with V(G) = {E,F,G,H} and E(G) = { {G,E}, {G,F}, {G,H}, {H,F} }.

1.2.3 Weighted Graphs

It's often useful to ascribe a value or weight to edges in a graph - for example, they might represent distance or cost in a graph representation of a road network. With each edge $e \in E(G)$, let there be associated a real number $w(e)$, called its weight. Then G, together with

these weights on its edges, is called a *weighted graph*. Note: An unweighted graph G can be viewed as a weighted graph where $w(e) = 1 \; \forall e \in E(G)$.

1.2.4 Simple Graphs

All of the networks generated in this research are represented by simple graphs, which are defined below:

> Definition: A simple graph (also called a strict graph) is an unweighted, undirected graph containing no loops or multiple edges.

where *multiple edges* are defined to be two or more edges that are incident to the same vertices. In a graph, if an edge is drawn from a vertex to itself, it is called a *loop*.

Figure 1.4: A graph with multiple edges (in red) and a loop (in green).

1.2.5 Graph Measurements & Parameters

The most basic graph measurement of interest is the *order* of a graph. A graph G has order $|V|$ - ie. the number of vertices in $V(G)$. In network science, the *size of a network* is most commonly defined to be the order of its underlying graph.

There are many more attributes that can be calculated to analyze the properties and characteristics of the network, using a graph theory. The graph theory concepts of degree, density, clustering and distance measurements, are used in this research to evaluate the network generation method, and are introduced next.

1.2.6 Degree & Degree Distribution

Definition: The Neighbourhood of a vertex V_i, denoted $\mathcal{N}(V_i)$, is the set of all vertices that are adjacent to it, e.g. $\mathcal{N}(V_i) = \{V_j : e_{i,j} \in E(G)\}$.

The neighbourhood of a vertex is simply the set of all vertices that it is connected to. In Figure 1.5 the neighbourhood of the vertex B is the set {A,C,D,E}, as B is connected to each of them. The neighbourhood of every other vertex will just contain B, as each of them are connected to B only.

The degree of a vertex is the number of vertices it is adjacent to, or the size of it's neighbourhood. Alternatively, the degree of a vertex can be thought of as the number of edges it participates in. In Figure 1.5,the degree of vertex B = 4, and the degree of all other nodes is 1. The degree of a particular vertex is an important measure of the centrality, or important of that vertex. On the network level however, we're often more interested in the average degree of the network and the degree distribution.

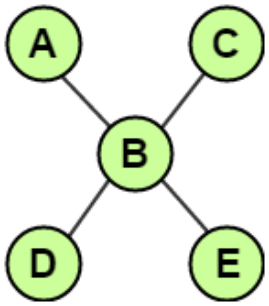

Figure 1.5: A simple graph with five vertices.

Definition: The Degree of a vertex V_i, is the number of vertices it is adjacent to, $|\mathcal{N}(V_i)|$.

The average degree of a graph is simply the average degree of all vertices in the graph, alternatively, it can also be calculated as the number of edges in G, divided by 2, since each edge contributes to the degree of two vertices. This is a good measure of how connected the components of a system are, on average.

Definition: The Average Degree of a graph G, is the sum of the degree of each vertex in G, divided by the number of vertices n, $\frac{\sum_i^n deg(v_i)}{n}$.

The degree distribution of a graph is a probability distribution of the degrees of all vertices in the graph. For each observed degree k in the graph, P(k) = nk/n, where nk is the number of vertices with degree k, and n is the total number of vertices. The degree distribution of a network provides good insight into the structure of the network. For example, a network that has many nodes is susceptible to failure, as deletion of a few nodes could easily lead to the network becoming disconnected. Figure 1.6 shows the degree distribution for the graph in Figure 1.5. The probability of a randomly selected vertex from this graph having degree 1 is 0.8, and the probability of it having a degree of 4 is 0.2.

The degree distribution P(k) of a graph G is the the fraction of vertices in G with degree k.

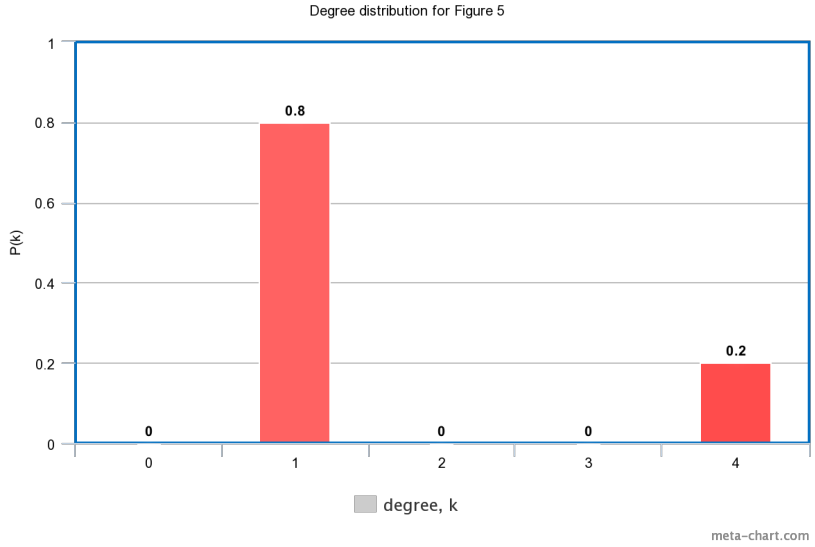

Figure 1.6: The degree distribution for Figure 5

1.2.7 Density

The density of a graph G, is the ratio of how many edges are in set E(G) to the maximum possible number of edges between vertices in set V(G).

An undirected graph can have at most $\frac{|V|*(|V|-1)}{2}$ edges, so the density of an undirected graph is: $\frac{2*|E|}{|V|*(|V|+1)}$. The number of possible edges in a graph (the denominator term) grows quadratically as the number of vertices grows linearly.

For the graph depicted in Figure 1.7, the density is calculated as $(2*4)/(4*3) = \frac{2}{3}$

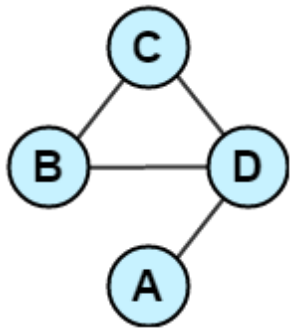

Figure 1.7: A simple graph with five vertices.

1.2.8 Clustering Coefficient

In graph theory, a clustering coefficient is a measure of the degree to which nodes in a graph tend to cluster together. The local clustering coefficient of a vertex in a graph quantifies how close its neighbourhood is to being a clique (complete graph).

Definition: The local clustering coefficient of a vertex is the proportion of links between the vertices within its neighbourhood divided by the number of edges that could possibly exist between them.

It can be calculated for a vertex V_i by $\frac{2*|\{e_{j,k}:V_j,V_k\in N(V_i),e_{j,k}\in E(G)\}}{K_i*(K_i-1)}$. This local clustering coefficient is calculated for each vertex $V_i \in V(G)$ and then averaged by dividing by the number of vertices in G. Note: a vertex with degree less than 2 is assigned a cc cc value of 0.

If we take Figure 1.7 as an example, we can calculate the network average clustering coefficient, by first calculating the local clustering coefficient of each vertex, and then dividing this number by 4. CC(A) = 0 (since A has only one neighbour). cc(B) = 1, since both of B's neighbours are connected to each other. cc(C) = 1/3, because only one out of the three possible connections exists in its neighbourhood. cc(D) is also 1/3 for the same reason.

The average CC for the entire graph is (0 + 1 + (1/3) + (1/3))/4 = 0.416

1.2.9 Distance

Definition: A path in a graph is a sequence of vertices, where any pair of consecutive vertices in the sequence is (linked by) an edge in the graph.

Definition: The shortest path, or distance, between two vertices in a graph is the path between the vertices that minimizes the sum of the weights of its constituent edges.

For an undirected, unweighted graph, we can think of each edge as having a weight of 1, which means the shortest path is simply the path with the least number of vertices in the sequence of vertices.

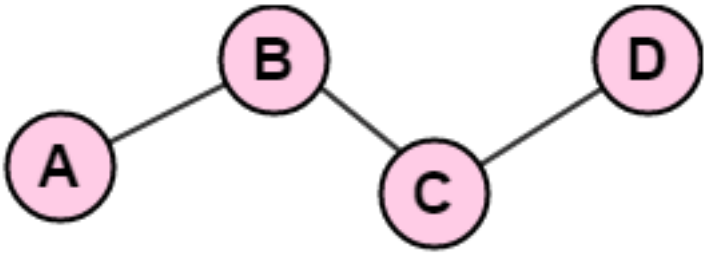

Figure 1.8: A simple graph with 4 vertices

Definition: The Diameter of a graph is defined as the longest shortest path, between any pair of vertices in the graph.

Table 1.1: The lengths of shortest paths between all pairs of vertices, for Figure 1.8

	A	B	C	D
A	0	1	2	3
B	1	0	1	2
C	2	1	0	1
D	3	2	1	0

For example, the diameter of the graph in Figure 1.8 is 3, which is the length of the longest shortest path, namely A -> D.

Definition: The Average Path Length of a graph is the average length of the shortest paths between all possible pairs of vertices in the graph.

The average path length of the same graph is calculated by first determining the length of shortest paths between pairs of vertices, summing these values, and then dividing by the n * (n-1), where n is the number of vertices in the graph.

Table 1.1 shows the length of the average paths for all pairs of vertices in this example. These values form a symmetric matrix since this is an undirected graph. In this example the shortest path length matrix values sum to 20, and we divide this number by (4*3) to get the APL, which works out to be 5/3.

1.2.10 Evolutionary Algorithms

Evolutionary computation is a family of algorithms that are typically used for solving optimization problems - In computer science and mathematics, an optimization problem is the problem of finding the best solution from the set of feasible solutions. Evolutionary algorithms (EAs) are a subset of evolutionary computation that use processes inspired by biological evolution: natural selection, reproduction and mutation, to work towards an optimal solution in a 'survival of the fittest' fashion.

EAs are particularly useful for optimization problems that are not well suited to traditional optimization algorithms, including problems in which the objective function is discontinuous, non-differentiable, stochastic, or highly nonlinear.

EAs look to replicate the way in which biological life uses evolution to find solutions to real world problems, in the following fashion:

1. EAs maintain a population of individuals (likened to chromosomes), each of which represents a point in the search space and a solution to the problem. Individuals are coded as a finite length vector of variables that are analogous to genes. Thus, a chromosome (solution) is composed of several genes (variables).

2. A fitness score is assigned to each solution (individual) using a fitness function. Fitness scores represent the abilities of an individual to 'compete'. The individual with the optimal (or generally near optimal) fitness score is sought. Figure 1.9 is a visual depiction of a two-dimensional fitness function.

3. EAs often use selective 'breeding' of the solutions to produce 'offspring' better than the parents by combining information from the chromosomes. Random mutation is also applied to further explore the search space.

4. New generations of solutions are produced containing, on average, better quality genes than a typical solution in a previous generation. Each successive generation will contain more high quality 'partial solutions' than previous generations. Eventually, once the population has converged and is not producing offspring noticeably different from those in previous generations, the algorithm itself is said to have converged to a set of solutions to the problem.

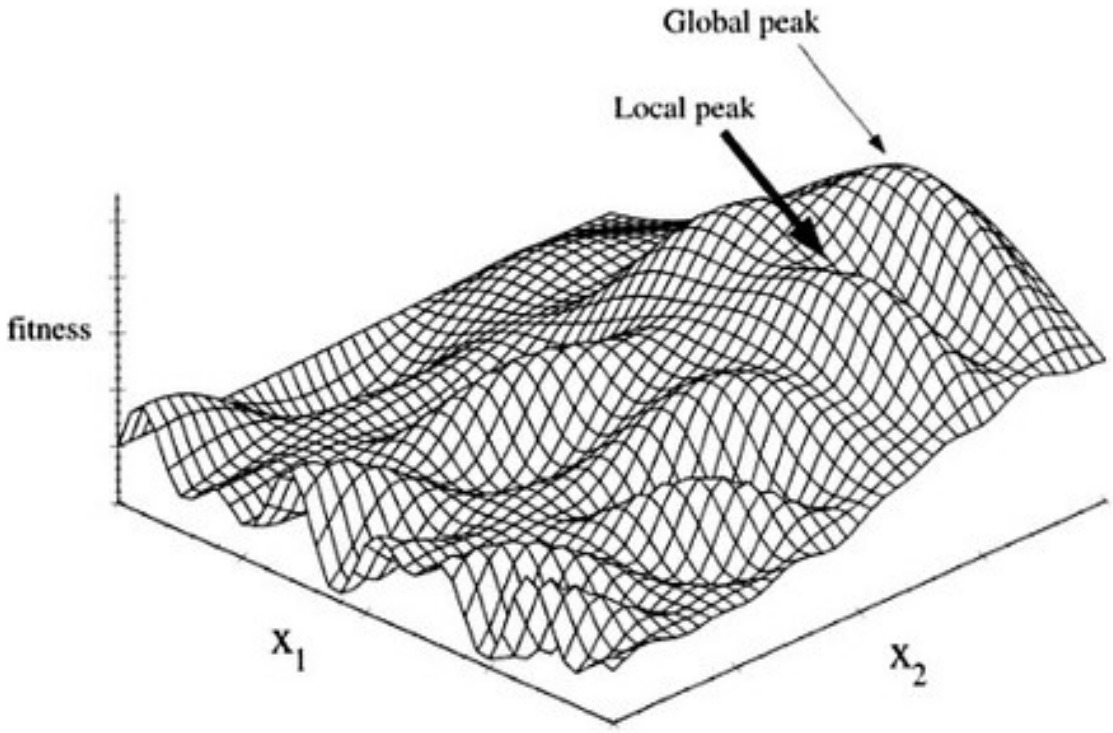

Figure 1.9: A two-dimensional fitness function visualization. [Credit: http://www.eoht.info/page/Energy+landscape]

1.2.ii Summary

In summary, this section provided a brief introduction to the area of Network Science: what a network is and why they are of interest to us. The ability to generate synthetic networks that exhibit certain properties, is of great interest and importance to us. A crash-course in Graph Theory is also supplied; Graph Theory provides us with a mathematical language in which we can measure and evaluate networks. The reader has also been introduced to evolutionary algorithms, a form of optimization algorithm inspired by biological evolution.

2

Design

This chapter provides a detailed description of the design process and considerations undertaken for this project. The graph generation method proposed is outlined: including the rules and rule selection process that were developed. The design of the evolutionary algorithm that is used to optimize the graph generation process, is also discussed in detail.

2.1 Graph Generation

At a high level, two approaches to generating graphs were considered for this project: constructive and destructive. The essence of both is as follows:

1. Choose a starting graph
2. Repeatedly apply a process that changes the starting graph, over time.

Where the two approaches differ is the nature of the process that is applied to the starting graph. In the constructive approach, the process adds vertices and edges to the starting graph in some way, whereas the destructive approach applies a process to remove vertices and edges.

This project adopts the constructive approach to graph generation, and at the heart of the research is a proposal, and evaluation, of one such process to generate graphs with desirable properties. These desirable properties are of course, highly dependent on the nature of the network we wish to create. The generation process proposed is outlined below:

1. Choose a starting graph.
2. Define some rules, that when fired can modify a given graph in some way.
3. Define a rule selection process that can select rules to be fired.

At each time-step t_i, the rule selection process is applied to select a rule to be fired. This rule is then applied to the graph from time-step t_{i-1}

The generation process can be run for a specified number of time-steps or alternatively the process can be terminated when the generated graph reaches a certain predefined state.

2.1.1 Starting Graph

The first step in designing the generation process is to choose a starting graph. In theory any graph could work for this constructive approach, however, the larger the graph chosen, the more bias/influence introduced into the process.

The starting graph chosen for this generation process is the complete graph on two vertices, denoted K_2. A complete graph is a simple, undirected graph in which every pair of distinct vertices is connected by a unique edge.

Figure 2.1: The starting graph, K_2

2.2 Rules

Rules are the at the heart of the proposed graph generation process. Rules are fired on a given graph to modify it in some way. For example, one could define a rule that, when fired on a graph, deletes a randomly chosen edge from the edge set of the graph.

Having adopted the constructive graph generation approach, this research only considers constructive rules; that is rules that add edges and/or vertices to the given graph. These constructive rules form the building blocks for the graph being generated; as such, we want these building blocks to be as simple as possible.

The generation process we have outlined will grow the graph at each time-step, as each constructive rule adds edges and/or vertices to the graph. These rules are the building blocks of the generated graph. In fact, each modification to the graph changes the properties of the graph. In this way, the properties of the generated graph are the result of the rules that were fired during the generation process. The rule set used is outlined here:

2.2.1 Rule 1: Add Node Rule

Input: graph $G_t = (V, E)$.

1. Select a random vertex $v_r \in V(G_t)$.
2. Add a new vertex, v_{n+1} to $V(G_t)$.
3. Add a new edge, $e_{r,n+1}$ to $E(G_t)$.

Output: graph G_{t+1}.

Figure 2.2 shows Rule 1 in operation. In this example, the rule is fired on graph G_t:

1. Vertex A is selected at random from the vertex set of G_t.
2. Create a new vertex E and add it to the vertex set.
3. Add an edge to the edge set of G_t, connecting the new vertex E with the vertex A.

The result of this rule being fired is the graph G_{t+1}.

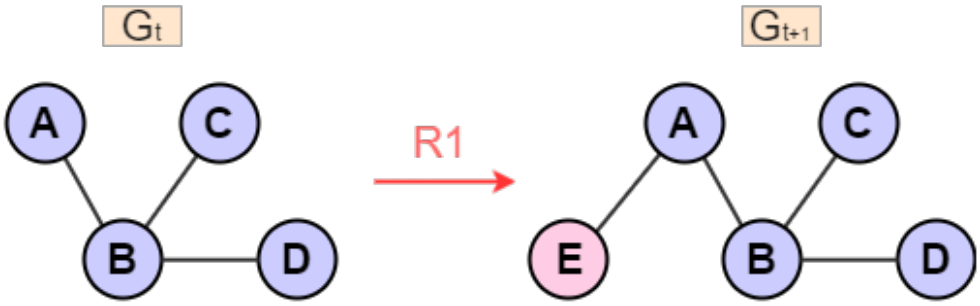

Figure 2.2: An example of the Add Node Rule firing

2.2.2 Rule 2: Add Triangle Rule

Input: graph $G_t = (V, E)$.

1. Select a random vertex $v_r \in V(G_t)$.
2. Add two new vertices, v_{n+1}, v_{n+2} to $V(G_t)$.
3. Add edges $e_{r,n+1}$, $e_{r,n+2}$, $e_{n+1,n+2}$ to $E(G_t)$.

Output: Graph G_{t+1}.

Figure 2.3 shows Rule 2 in operation. In this example, the rule is fired on graph G_t:

1. Vertex D is selected at random from the vertex set of G_t.
2. Create two new vertices, E & F and add them to the vertex set.
3. Three edges are added to the edge set of G_t, in order to complete the triangle: an edge connecting each of the new vertices to vertex D, and an edge that connects the two new vertices to each other.

The result of this rule being fired is the graph G_{t+1}.

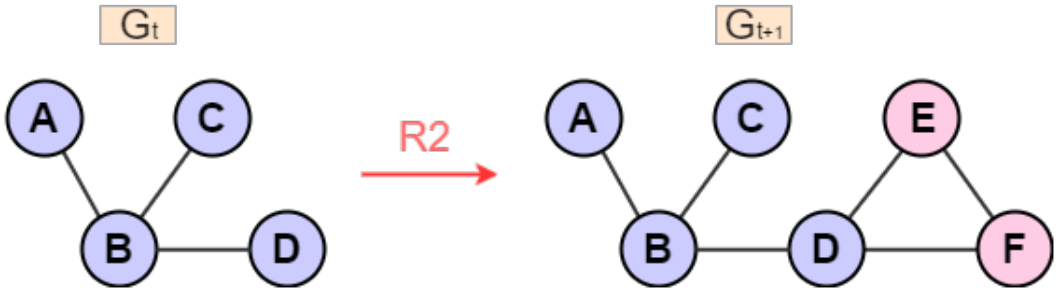

Figure 2.3: An example of the Add Triangle Rule firing

2.2.3 Rule 3: Add Edge Rule

Input: graph $G_t = (V, E)$.

if (G_t not complete):

1. Select any two distinct vertices v_i and v_j, where $e_{i,j} \notin E(G_t)$.
2. Add a new edge $e_{i,j}$, to $E(G_t)$.

else:

1. Select a random vertex $v_r \in V(G_t)$.
2. Add a new vertex, v_{n+1} to $V(G_t)$.
3. Add a new edge, $e_{r,n+1}$ to $E(G_t)$.

Output: graph G_{t+1}.

Figure 2.3 shows Rule 3 in operation. In this example, the rule is fired on graph G_t:

1. In this example G_t is not complete, and so vertex A and vertex C are chosen at random from the set of unconnected pairs of vertices.
2. An edge is added to the edge set of G_t, connecting vertex A with the vertex C.

The result of this rule being fired is the graph G_{t+1}.

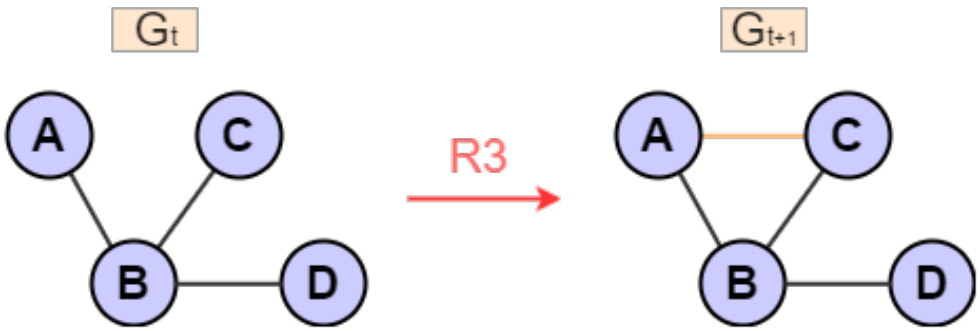

Figure 2.4: An example of the Add Edge Rule firing

2.3 Rule Selection Process

Now that the rules to build our generated graphs are defined, we need to design a rule selection process to determine, at each time-step, which rule from the rule set to fire. The proposed process employs a probabilistic approach, which is outlined below:

Given a rule set with n rules, we can define a discrete random variable R:

- $R = \{r_1, r_2, ..., r_n\}$
- $P(R)$ is the probability of R.
- $P(R = r)$ is the probability that the random variable R has a particular outcome r.

A probability mass function (pmf) on R defines the probability of each outcome, where each possible outcome r corresponds to a rule in the rule set.

Table 2.1: Am example pmf, for a random variable R with three possible outcomes.

r	r1	r2	r3
P(R = r)	0.4	0.2	0.4

It's pertinent to note that each distinct rule modifies the graph (and thus, it's properties) in a different way. As a result, when we wish to generate a specific type of graph, some rules should be favored over others. In this context, favored means they are assigned a relatively large probability in the pmf defined over the rule set.The relationship between each distinct rule, and the properties they induce, is not always intuitive or even visible. One problem remains: how do we choose the probability mass function?

Each possible probability mass function over the random variable defined on the rule set is a potential solution; a search mechanism must be employed, to explore this solution space in order to optimize our choice of pmf. In other words, the goal is to optimize the probability of each rule's corresponding outcome in $P(R)$.

2.4 Evolutionary Algorithm

The proposed graph generation process has just been distilled to an optimization problem. This research adopts an evolutionary algorithm (EA) approach, to explore the solution space. EAs were chosen as the optimization algorithm for the following reasons:

- The objective function (i.e. the fitness function) is non-linear, and stochastic.
- EAs are noise tolerant (the probabilistic approach adopted introduces noise).
- They are inherently parallel, i.e. they search different parts of the solution space simultaneously; this is extremely beneficial, due to the large search space.
- EAs combine direction, as well as chance, to the search in an effective and efficient manner; this is important due to the unpredictability of the search space.
- The evolutionary approach allows us to easily change the objective function to target different network properties, through the fitness function.

Evolutionary Algorithm Design

For the sake of correctness, this evolutionary algorithm is not a genetic algorithm: it uses population-based random variation, selection and mutation, only. The crossover operator, as used in genetic algorithms, is not utilized here. This is a direct consequence of the nature of our solution space - we are optimizing a probability mass function - which cannot be recombined with each other in any meaningful, or beneficial way.

In the EA potential solutions, will be represented as individuals, and a population of these individuals are maintained over consecutive generations. In each generation, the individuals are evaluated by testing a particular optimization problem; a real value is assigned to the individual based on how well they did at performed. This value is used as a measure of fitness, and is used to perform selection.

Elitism is employed, ensuring the best individuals from each generation are guaranteed to make it into the subsequent generation - this helps to maintain a high population fitness. Eventually, the population (and thereby it's fitness) should converge to a solution, or set of solutions. Mutation (i.e. random perturbations on individuals) is applied with some low probability to the individuals of new generations, in order to maintain diversity and prevent premature convergence to sub-optimal solutions. The process is defined below:

Evolutionary Algorithm:

1. Initialize population P_t, randomly.
2. Determine the fitness of population P_t.
3. Repeat (Until a TC is met):
   (a) Select individuals for the next generation P_{t+1}
   (b) Apply mutation on the new population P_{t+1}.
   (c) Determine the fitness of population P_{t+1}.

TC. 1: Max number of iterations reached

TC. 2: Population fitness has converged

Initialize starting population

A potential solution is any valid probability mass function of the random variable R. This makes it easy to encode our solutions as individuals - each can be represented simply as a real valued vector of length n, e.g $v = \{x_1, x_2, .., x_n\}$, where n is the number of rules in the rule set. The initial population is populated with random vector values; these values must satisfy the conditions of a pmf, i.e. all values are non-negative and sum to one.

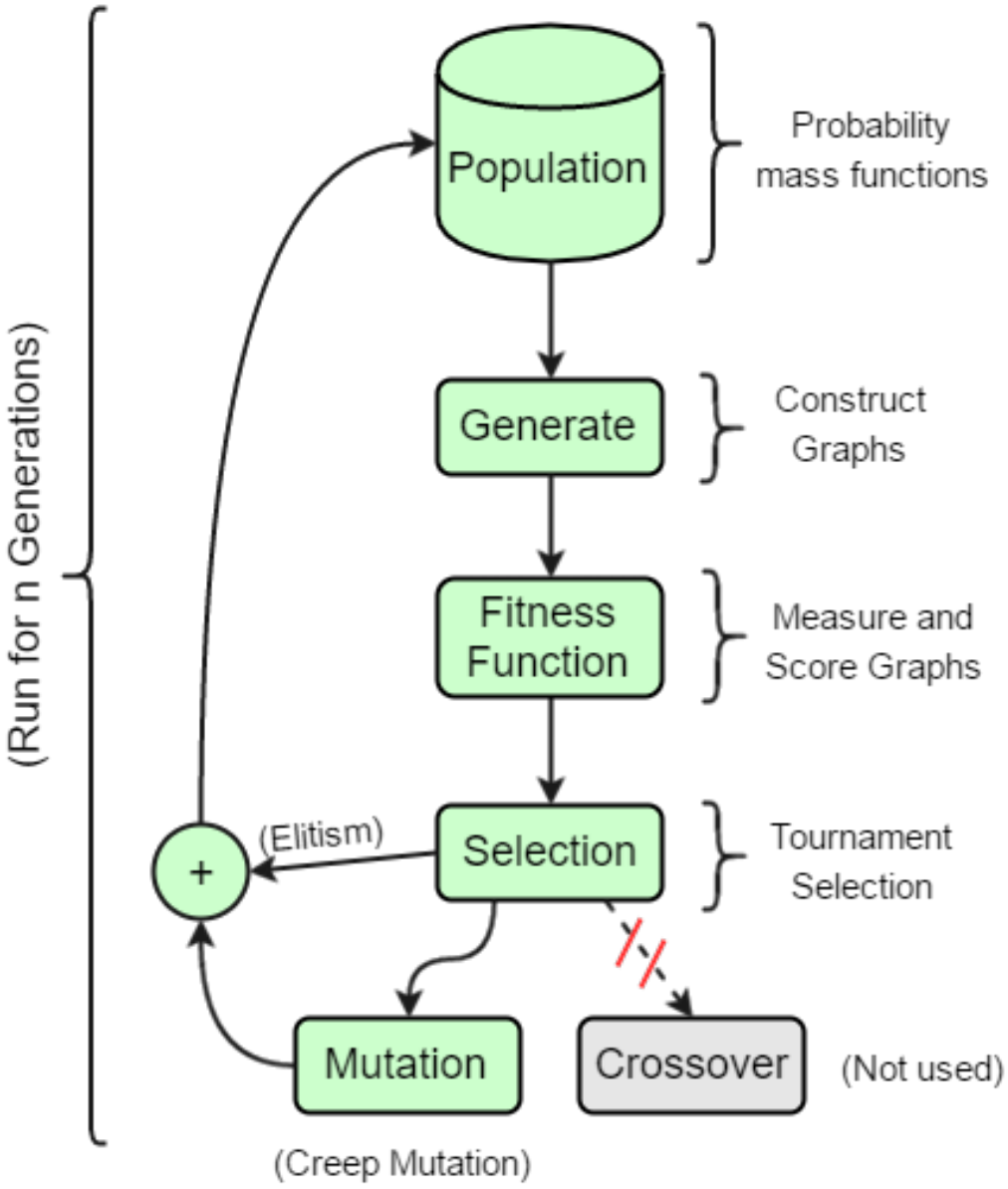

Figure 2.5: The evolutionary algorithm process

Determine Fitness

In this evolutionary approach, we need to evaluate the performance of each individual in order to guide the evolution.Each individual generates a number of graphs, by firing a certain number of rules from the rule set, using its encoded probability mass function to probabilistic select these rules. The number of rules can be varied, and as such can be used as a parameter in the experiments. A number of graphs are generated per individual due to the stochastic nature of the rules.

A fitness function is defined in some way to reward the desirable properties in the graphs generated. This fitness function returns a value in $[0, 1]$ for each individual, that is the average result of all graphs generated by the individual. For example, one such fitness function has been designed to reward graphs that exhibit a particular clustering coefficient value. The specific clustering coefficient desired, $TARGETCC$, is set as a parameter.

Clustering Coefficient Fitness Function:

- Input: [Individual I, with n generated graphs: $G_I = \{g_1, .., g_n\}$]

$$f(I) = 1 - \frac{\sum_i^n |clusteringCoefficient(g_i) - TARGETCC|}{n}$$

- Output: $f(I)$, a value $\in [0, 1]$

The output of this function, is the fitness score for the Individual. The fitness function that is used is dependent on the graph properties desired; a number of different fitness functions were explored for this project and are detailed in the relevant experiment section.

Selection

The selection process determines which individuals make it through to subsequent generations. Preference is given to the fitter individuals, as determined by fitness function. This preference can be realized in many ways, both deterministic and stochastic, of which we will use a combination:

- Elitism (deterministic) is the process of allowing a certain number of the fittest individuals to progress to the next generation, unaltered (exempt from mutation). The number of individuals that progress in this way is known as the elitism rate. Elitism ensures that the overall fitness should increase in each subsequent generation.
- Tournament Selection (stochastic) is used to select the remaining individuals for the next generation. It works as so:

    1. Select a number of individuals, sampled at random from the population.
    2. Compare the fitness values of these individuals
    3. The individual with the highest fitness declared the winner of the tournament
    4. A copy of the winner progresses to the next generation

    Individuals are sampled with replacement, so a single individual can be selected for multiple tournaments and thereby can have multiple copies of it carried over to the next generation. Each individual has an equal chance of being selected for each tournament. The number of number of individuals selected for each tournament is known as the tournament size and is an adjustable parameter of the EA.

Mutation

Mutation is the process of making small changes to the individuals "chromosome"; in this application our chromosomes are the probability mass functions, that are being evolved. Mutation can cause a solution to change entirely from the previous solution. Its purpose is to maintain diversity within the population and inhibit premature convergence by discovering new solutions in the search space.

Each individual in a new population, with the exception of the elite individuals, have a mutation operator applied with some low probability. The probability at which mutation is applied to each individual is called the mutation rate. In this evolutionary algorithm, the type of mutation applied is known as Creep Mutation, and is outlined below:

Creep Mutation:

1. Select a rule r_m, at random from the rule set.
2. Increase the probability of the outcome corresponding to rule r_m in the pmf.

   The size of the mutation is known as the mutation value.
3. Decrease the probability of all other outcomes by $\frac{mutationvalue}{numberofrules-1}$

Figure 2.6 demonstrates creep mutation applied to a solution's probability mass function.

Figure 2.6: Creep mutation applied to a pmf. P(R=r1) is increased, P(R=r2), P(R=r3) are decreased

3

Implementation

Scala is the implementation language of choice for this research project. Scala is a JVM (Java Virtual Machine) language that is both functional. Scala is an extremely expressive language and has a lot of very powerful features that make development in Scala quick, concise and in the author's opinion, enjoyable. The code for this project is available for the interested reader at the following location: https://github.com/thecodingbandit/FYP

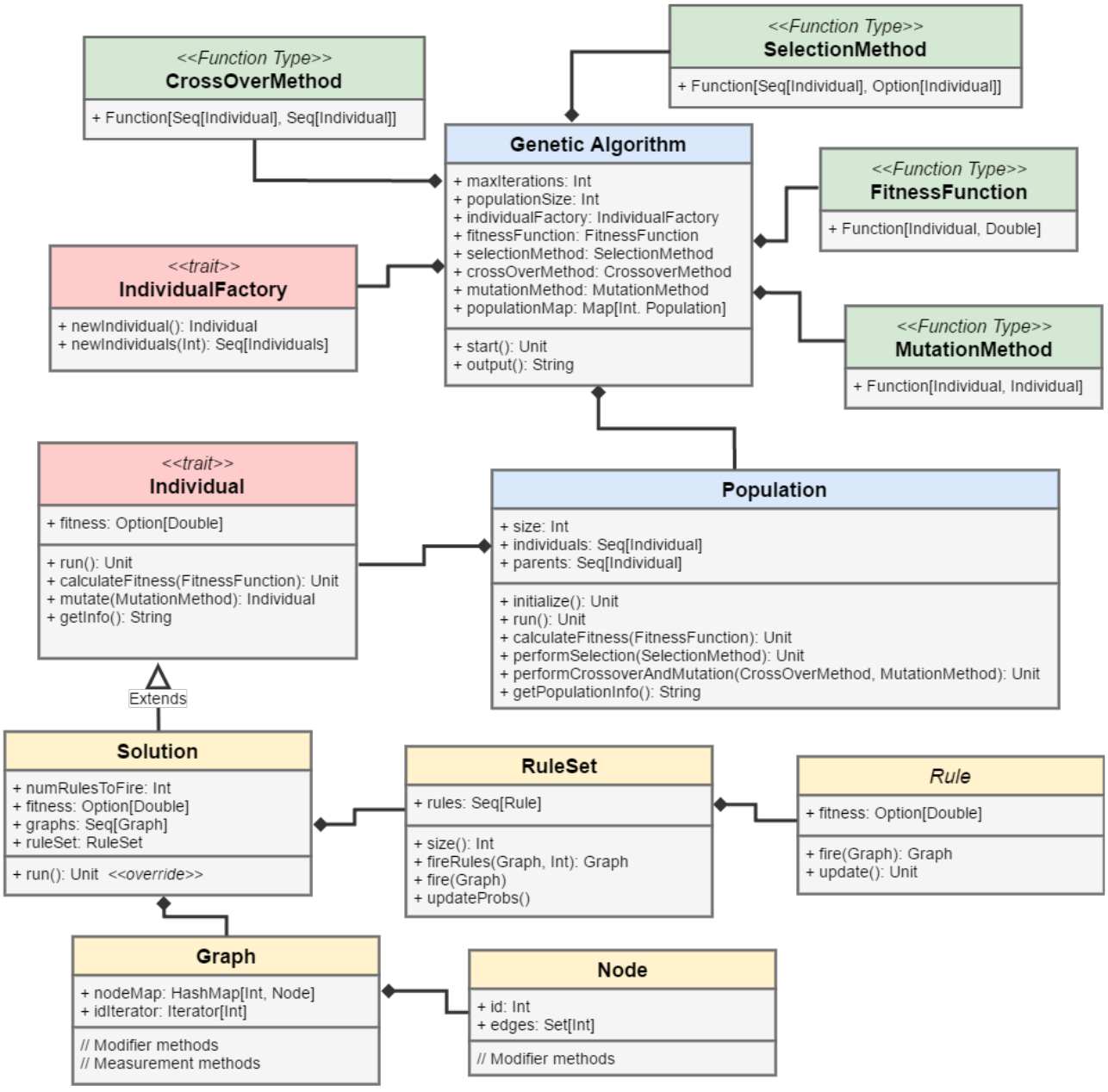

Figure 3.1: The EA framework developed. traits are in red, function types in green, and implementations are in yellow.

3.1 Evolutionary Algorithm Framework

A highly flexible and extensible evolutionary algorithm framework was developed for this research. These properties were obtained by first designing the algorithm independent to the current application, through the use of traits (similar to interfaces) and user-defined function types (like functional interfaces). This genetic algorithm could be applied to any optimization problem, by completing the following steps:

1. Implement the *Individual* trait - encapsulate information from the solution space in this class. An implementation for the *run()* method must be provided.

2. Implement the *IndividualFactory* - this implementation will be the factory object that is used by the *GeneticAlgorithm* instance to initialize the starting population.

3. Create a *FitnessFunction* type method - this is used to evaluate instances of the *Individual* implementation defined in Step 1.

4. Create a *SelectionMethod* type method - this is used to perform selection.

5. Create a *MutationMethod* type method - this is used to mutate individuals.

6. Create a *CrossoverMethod* type method - (if the crossover operator is utilized).

3.2 Graph Generation

To tackle the problem of optimizing our graph generation process, new classes and functions are developed to build upon the previously described framework:

1. An abstract *Rule* class is defined and the specific implementations of the designed rules are created. *Rule* implementations have a *fire()* method that takes a graph, modifies the graph and then returns it. Each *Rule* instance has an associated probability, which represents that rule's probability in the probability mass function.

   The *RuleSet* class is also created as a container for *Rule* instances.

2. The *Graph* and *Node* classes is created to represent the generated graphs. Modifier and measurement methods are defined. *Breadth-First Search* is also implemented, for the distance based measurements. Figure 3.2 shows the *Graph* class.

3. The *Solution* class is created, implementing the *Individual* trait. The evolutionary algorithm maintains a population of these *Solution* instances. *Solution* instances have a *RuleSet*, which is used to generate graphs when the *run()* method is invoked by the *GeneticAlgorithm*. The generated graphs are stored in a sequence.

4. *TournamentSelection* and *RouletteWheelSelection* methods are implemented, of type *SelectionMethod*, ie. they take *Seq[Individual]* and return an *Individual*.

5. The *CreepMutation* method of type *MutationMethod* is created. This method works specifically on *Solution* instances, modifying the probability mass function of the *RuleSet* with some probability.

6. A number different fitness methods (of type *FitnessFunction*), are created for the experiments carried out in this research project. Each fitness method takes a *Solution* and determines a score (a *Double* value) for that *Solution* by measuring the graphs generated by that *Solution* in some way.

```scala
class Graph(private val nodeMap: HashMap[Int, Node] = new HashMap[Int, Node],
            private val idIterator: Iterator[Int] = Iterator.from(1)) extends Serializable{

  def nodes = nodeMap.values.toList
  def nodesMap = nodeMap
  def randomNode = nodes(Probability.randomInt(nodes.size))
  def randomId = randomNode.id
  def nextId = idIterator.next
  def ids = nodeMap.keys.toList

  def addNode(n: Node) = nodeMap += n.id -> n
  def addNodes(ns: Seq[Node]) = nodeMap ++= ns.map(n => n.id -> n)
  def removeNode(i: Int) = { val n = nodeMap.remove(i); for (nb <- n.get.neighbours) nodeMap(nb).removeEdge(i)}
  def addEdge(edge: (Int, Int)) = { nodeMap(edge._1).addEdge(edge._2); nodeMap(edge._2).addEdge(edge._1) }
  def addEdges(edges: Seq[(Int, Int)]) = for (e <- edges) { nodeMap(e._1).addEdge(e._2); nodeMap(e._2).addEdge(e._1) }
  def edgeExists(id1: Int, id2: Int) = nodeMap(id1).connectedTo(id2)
  def distinctEdges = nodes.flatMap(n => n.neighbours.map(i => (n.id, i))).map{ case (a, b) =>
    if (a > b) (a, b) else (b, a)}.distinct.sortBy(_._1)

  def size = nodeMap.size
  def averageDegree = totalDegree.toDouble / size
  def totalDegree = nodes.map(n => n.degree).sum
  def isComplete = nodes.map(n => n.neighbours.size == size-1).forall(_ == true)
  def density = (2 * distinctEdges.size).toDouble / (size * (size-1))
  def degreeDist = nodes.groupBy(n => n.degree).map(t => (t._1, t._2.length)).toList.sortBy(_._1)

  def shortestPaths = nodeMap.map(n => n._1 -> BFS.shortestPaths(this, n._1))
  def diameter = shortestPaths.map(sp => sp._2.values.max).max
  def averagePathLength = shortestPaths.map(sp => sp._2).map(p => p.values.sum).sum / (size * (size -1)).toDouble

  def averageCC = nodes.map(n => localCC(n.id)).sum / size
  def localCC(n: Int) = {
    val actual = 2 *  nodeMap(n).neighbours.combinations(2).count(pair => edgeExists(pair(0), pair(1)))
    val numNeighbours =  nodeMap(n).neighbours.size
    if (numNeighbours < 2) 0.0 else actual.toDouble / (numNeighbours * numNeighbours -1)
  }
}
```

Figure 3.2: Code for the *Graph* class, showing the conciseness of Scala.

4

Experiments & Results

This section details the experiments that were conducted as part of this research. For each experiment the set of parameters and configurations, as well as the description and definition of the fitness function that was used is provided. Results are presented and discussed in detail, with respect to the research questions that are defined in Section 0.

4.1 Experiment 1

Research Question

Can the generation method produce a network with desired properties?

Aim

- To evaluate the proposed generation method, when tasked with generating synthetic networks that exhibit specific clustering coefficient and density values. Networks will be generated, and then evaluated, across a range of desired clustering coefficients and density values, from 0.0 to 1.0, in increments of 0.1.
- To gain insight into which clustering coefficient values are easy to induce in generated networks, and which values are more difficult.
- To gain insight into which density values are easy to induce in generated networks, and which values are more difficult.

Parameters & Experiment Setup

For this experiment, the evolutionary algorithm will be used to evolve solutions that generate graphs, using a probability mass function defined over the rule set.

Table 4.1 lists all of the parameters used in this experiment. The EA will run for a maximum of 15 iterations, with a population size of 150 individuals. The individuals of the starting population are initialized with a random probability mass function. In each iteration a solution fires 150 rules, using its probability mass function to select each rule to be fired.7 graphs are generated by a solution in this way, and the fitness function will report the aver-

age performance of these 7 graphs. The two best performing individuals in each generation are copied into to the subsequent generation, through elitism. A tournament size of 2 is used for selection and mutation is applied with probability 0.2.

Table 4.1: Experiment 1 parameters

Exp. Number	Ex. 1
Iterations	15
Population Size	150
Graphs per Solution	7
Rules Fired	200
Tournament Size	2
Elitism Rate	0.020
Mutation Rate	0.20
Mutation Amount	0.10

The first fitness function used is (CC_F). This fitness function rewards solutions that generate graphs that have a clustering coefficient (cc) close to the desired value. The EA will be run 11 times, starting with a target cc value of 0.0, and each time this target value is incremented by 0.1. A fitness score of 1.0 is the maximum possible.

Experiment 1 fitness function (CC_F):

$$f(I) = 1 - \frac{\sum_i^n |clusteringCoefficient(g_i) - TARGET\_CC|}{n}$$

where n is the number of graphs generated per solution

This process of 11 EA runs was mirrored, but this time using a different fitness function (DEN_F). This fitness function rewards solutions lead to graphs that have a density value that are close to the desired value. Like previously, the first EA run will have a target (density) value of 0.0, and is incremented by 0.1 each time.

Experiment 1 fitness function (DEN_F):

$$f(I) = 1 - \frac{\sum_i^n |density(g_i) - TARGET\_DENSITY|}{n}$$

where n is the number of graphs generated per solution

Results

Two sets of results, one for each of the fitness functions used, are detailed here. Each set of results contains:

1. A graph of the actual values achieved (y-axis) vs. each of the target values (x-axis).
2. A graph of the best fitness achieved (y-axis) vs each of the target values (x-axis).
3. A graph showing fitness (best/average) (y-axis) for each generation (x-axis).
4. A table containing the pmf of the best solution, for each run of the EA.

Result Set 1 (CC)

This is the result set for the EA runs that used the clustering coefficient fitness function (CC_F). The EA had a difficult time generating graphs with a cc value between 0.6 and 0.8.

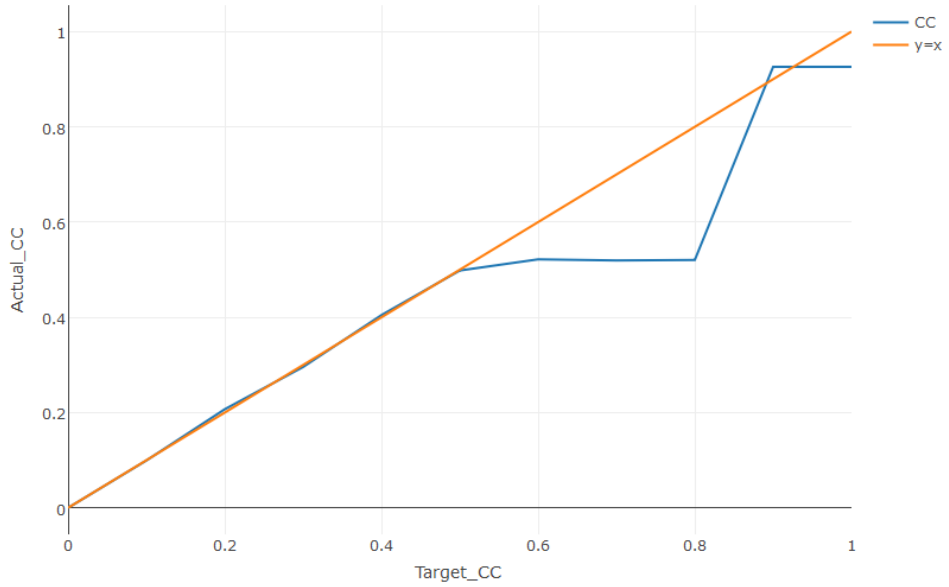

Figure 4.1: CC Result Set - Target CC value on x-axis, actual CC value achieved on y-axis.

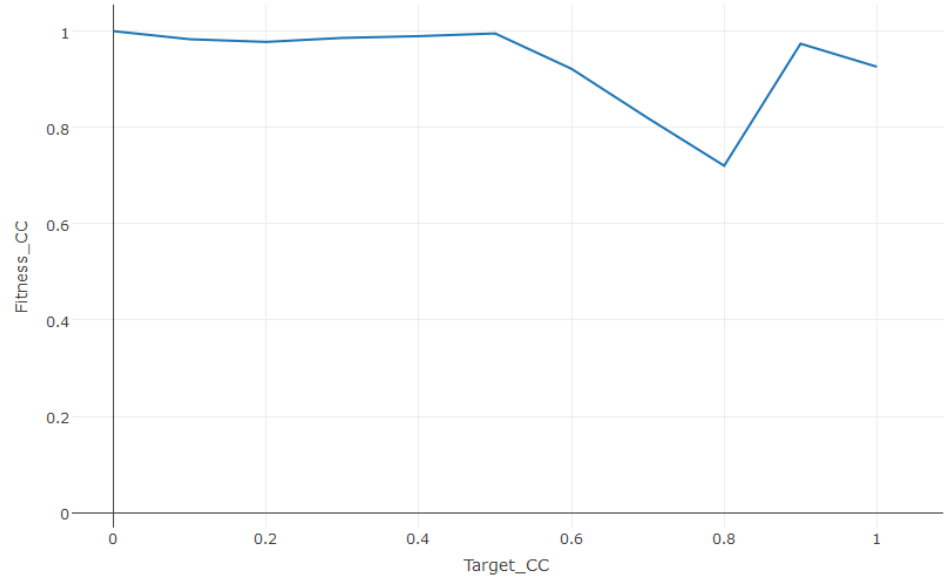

Figure 4.2: The best fitness achieved, for each desired CC value.

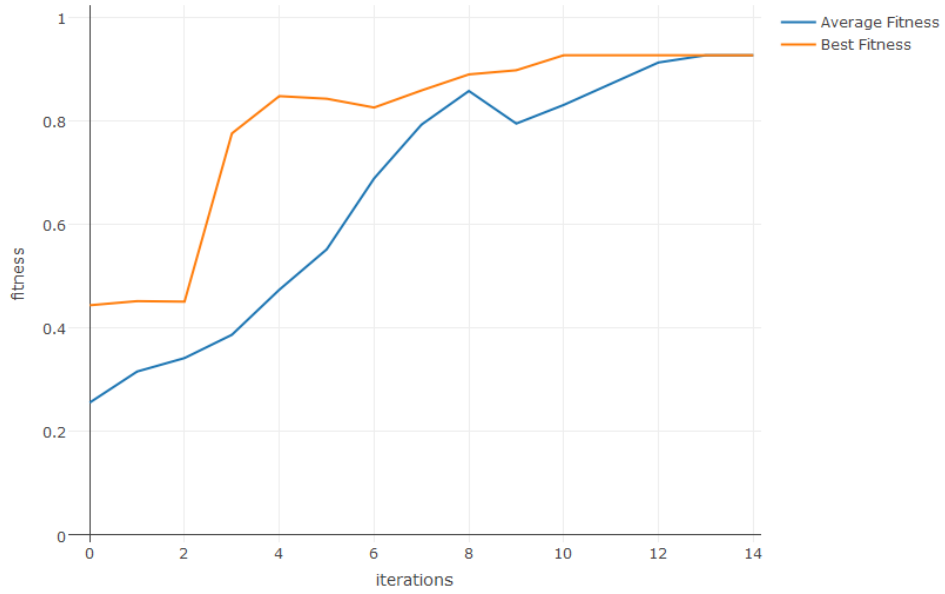

Figure 4.3: Best/Average fitness over time, for a target cc value of 1.0

Table 4.2: The probabilities for the best individual after each run, targeting cc

Target_CC	Add Edge	Add Triangle	Add Node
0.0	0.000	0.000	1.000
0.1	0.008	0.107	0.885
0.2	0.009	0.264	0.726
0.3	0.672	0.277	0.051
0.4	0.238	0.612	0.150
0.5	0.100	0.900	0.000
0.6	0.037	0.963	0.000
0.7	0.000	1.000	0.000
0.8	0.015	0.985	0.000
0.9	1.000	0.000	0.000
1.0	1.000	0.000	0.000

Result Set 2 (Density)

This is the result set for the EA runs that used the density fitness function (DEN_F). The EA had little difficultly in generating graphs across the entire range of target density values.

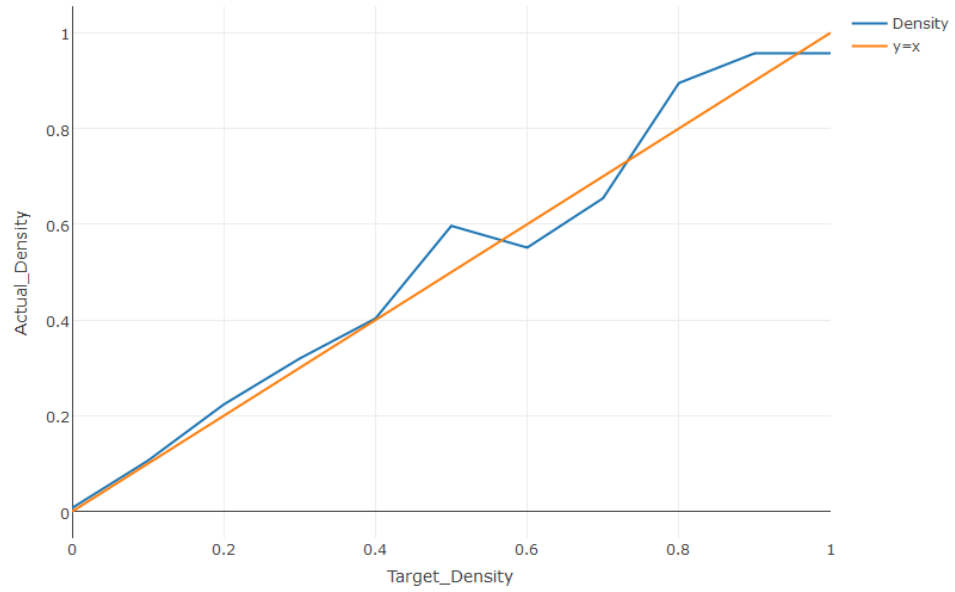

Figure 4.4: CC Result Set - Target density value on x-axis, actual density value achieved on y-axis.

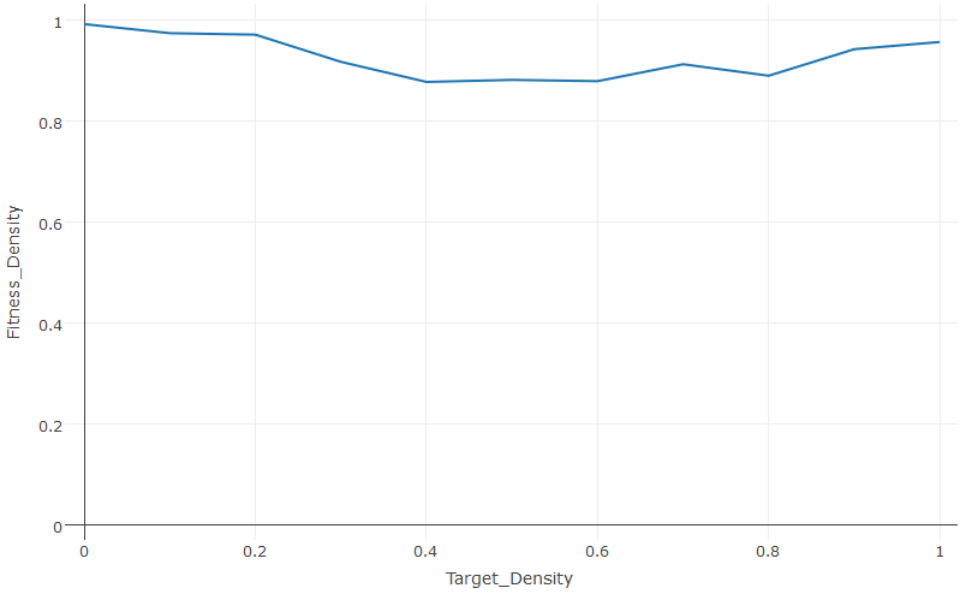

Figure 4.5: The best fitness achieved, for each desired density value.

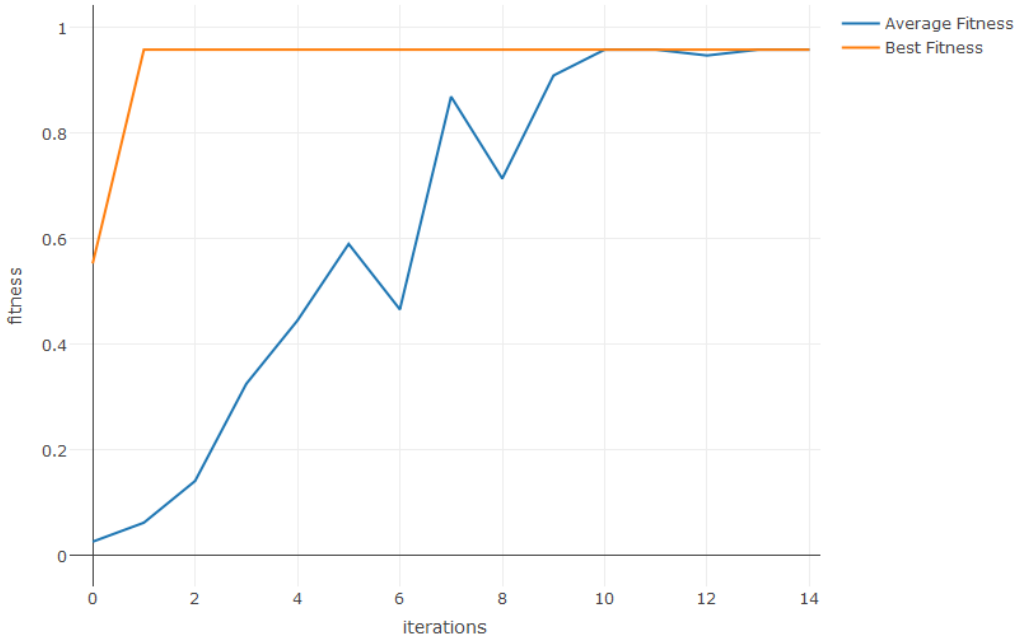

Figure 4.6: Best/Average fitness over time, for a target density value of 1.0

Table 4.3: The probabilities for the best individual after each run, targeting density

Target_Density	Add Edge	Add Triangle	Add Node
0.0	0.000	1.000	0.000
0.1	0.752	0.087	0.160
0.2	0.800	0.000	0.200
0.3	0.672	0.277	0.051
0.4	0.845	0.000	0.155
0.5	0.923	0.077	0.000
0.6	0.904	0.000	0.096
0.7	0.900	0.000	0.100
0.8	0.921	0.000	0.079
0.9	0.970	0.000	0.030
1.0	1.000	0.000	0.000

Discussion on Results

In Figure 4.1 we can see the results of the generation method when different values of cc were targeted. The EA has no difficulty in generating graphs in the range of values from 0.0 to 0.5, in fact the graphs generated are almost perfect. In Table 4.2 we can see the actual rule probabilities of the best solutions for each run, which shows that solutions that have low probabilities for the Add Triangle rule perform best, as triangles in the graph lead to a higher clustering coefficient. The EA has difficulty for the cc range of 0.6 to 0.8 however, and then is more successful for the target values of 0.9 and 1.0. This increased performance can be explained by looking at Table 4.2; solutions with a high probability for the Add Edge rule are favoured, as they will lead to a much smaller graph, and therefore it will be easier for this graph to have a high clustering coefficient. The lower extreme of target cc values leads the EA to find large graphs, and the higher extreme leads to the generation of smaller graphs, making it easier for the EA to exhibit the targeted cc values in both cases. The number of rules fired will also have an impact here, as there is a relationship between this number and the difficultly the EA will have to produce certain types of graphs - this relationship should be explored in more detail.

Figure 4.4 shows the results of the EA that used the density based fitness function. It's immediately apparent that the EA performed more successful when tasked with generating graphs with a target density value than the EA using the cc based fitness function. The generation process is successful across the entire range of desired density values; the best fitness achieved is above 0.9 for all density values that were targeted. Observing the final rule probabilities evolved by the EA in Table ?? we can see there that the Add Triangle rule is dominant at the lower extreme, this is because the Add Triangle rule leads large graphs, as

it adds more vertices than each of the other rules, and as the number of vertices increase linearly in a graph the number of possible edges (the denominator in the density calculation) increases exponentially, thus greatly decreasing the density value. When the density value desired is increased, we can see that the Add Edge rule becomes more and more favored, as this rule leads to a direct increase in density.

Figure 4.3 and Figure 4.6 show the convergence of the EA using each fitness function. When targeting a desired cc value, the EA takes 10 iterations on average to reach the optimum, whereas with the density fitness function we can see that the EA quickly found a solution that it was not able to improve upon further. In both cases we can see that the average fitness of the population eventually converges to the same fitness score as the best solution.

In summary, the EA performed quite well when tasked with generating graphs that exhibit certain clustering coefficient and density values. The EA had a little difficulty in generating graphs with a cc in the 0.6 to 0.8 range but this might be improved by adjusting the number of rules that were fired, or introducing a graph size objective into the fitness function. The EA was very successful at targeting desired density values, and had little difficulty across the entire range of values targeted.

4.2 Experiment 2

Research Question

Is the stochasticity of the generation process a large factor in the method's ability to generate networks with desired properties?

4.2.1 Aim

- To determine the impact of stochasticity on the performance of the generation method.
- To discover the number of graphs that should be generated by each solution, in order to limit this impact on performance.

Parameters & Experiment Setup

Table 4.4 lists all of the parameters used in this experiment. The set up and EA parameters are similar to that of experiment 1. This experiment will require multiple runs of the EA, and each time the number of graphs that each solution generates will be adjusted.

Table 4.4: Experiment 2 parameters

Exp. Number	Ex. 2
Iterations	15
Population Size	150
Graphs per Solution	1,3,7
Rules Fired	200
Tournament Size	2
Elitism Rate	0.020
Mutation Rate	0.20
Mutation Amount	0.10

In the first run each solution generates a single graph; in the second run, the EA will generate 3 graphs, and solutions in the final run will generate 7 graphs. In a similar fashion to experiment 1, this process will be mirrored, using a different fitness function each time, producing two sets of results to analyze.

Experiment 2 fitness function (CC_F):

$$f(I) = 1 - \frac{\sum_i^n |clusteringCoefficient(g_i) - TARGET\_CC|}{n}$$

where n is the number of graphs generated per solution

Experiment 2 fitness function (DEN_F):

$$f(I) = 1 - \frac{\sum_i^n |density(g_i) - TARGET\_DENSITY|}{n}$$

where n is the number of graphs generated per solution

Results

Figure 4.7 and Figure 4.8 show the results of the EA using the CC_F and DEN_F fitness functions, respectively.

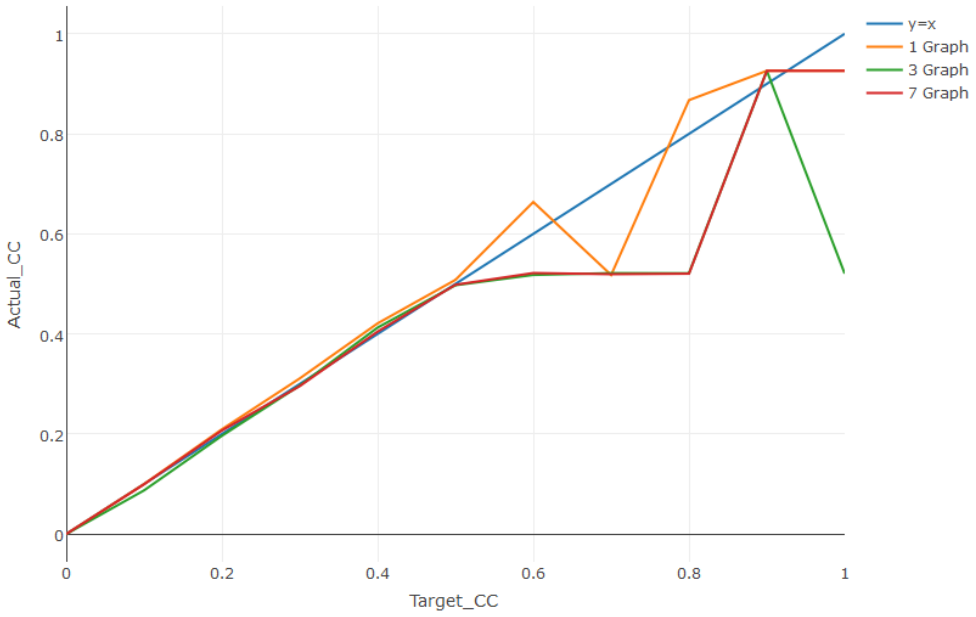

Figure 4.7: Actual cc (y-axis), Target cc (x-axis) for EA runs, with a varied number of graphs generated per solution

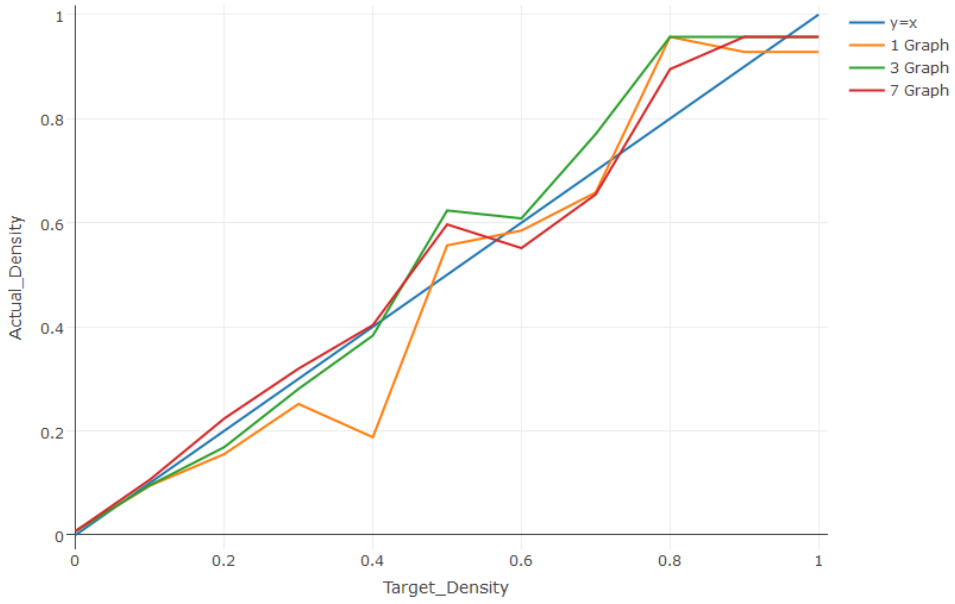

Figure 4.8: Actual density (y-axis), Target density (x-axis) for EA runs, with a varied number of graphs per solution

Discussion on Results

The network generation method as designed has two inherent sources of stochasticity, e.g. the probabilistic approach to rule selection process, as well as the randomness within each rule e.g. the fact that a random vertex or pair of vertices are chosen at random from the graph at the beginning of each rule.

In Figure 4.7, which used the clustering coefficient based fitness function, we can see that the EA performed equally as well on all values of cc up to 0.5, even when only one graph was generated per solution, and the stochasticity did not lead to any significant variance in the quality of the graphs generated. These values are very close to the ideal (the blue line).

When the target cc was increased to 0.6, and further, we start to see some variance in the performance of the EA. The 3 graph EA and 7 graph EA perform almost identically until a target cc of 0.9 is desired - here we see a very large disparity, which highlights the need to generate multiple graphs to get an accurate evaluation of the generation method, and to add validity to the results by counteracting any noise they might contain.

In this series of EA runs, the single graph EA resulted in the best performance of all three, but due to the stochasticity of the process, as evidenced by the steep drop in performance of the 3 graph EA, we can't rely on results from solutions where a single graph was generated to evaluate this generation method.

In Figure 4.8, which used the density based fitness function, we can see that the solutions evolved in all three cases were generally quite good, and are all relatively close to the targeted density values in each case. In the case of the EA evolving solutions that generate a single graph, we can see a significant drop in performance for the target density value of 0.4. The 3 and 7 graph EAs on the other hand, are consistent with each other and successfully evolved

solutions that generated satisfactory graphs.

For each fitness function, we see the the EA evolving solutions that generate 7 graphs report the most consistent performance, and provided more accurate results in which to evaluate the network generation method.

Generally speaking, the more graphs that were generated per solution, the more confidence we should have in the reported results. However, each time we double the number of graphs generated, we also double the run time complexity of the EA. In Figure 4.8, we can see that there is only marginal difference in the results reported by the 3 graph and 7 graph EAs, and this implies that increasing the number of graphs generated any more than 7 would be a case of diminishing returns.

In summary, we have observed some variance in the results of the EA, as a result of the stochasticity of the rule based generation. The variance introduced by this stochasticity was not as pronounced as was originally hypothesized, however. We can conclude that generating 7 graphs per solution is sufficient to reduce this variance to acceptable levels.

4.3 Experiment 3

Research Questions

- Can the method generate networks when a combination of specific properties is desired?
- What happens if the combination of specific properties desired are incongruent with each other?

4.3.1 Aim

- To determine the impact of stochasticity on the performance of the generation method.
- To discover the number of graphs that should be generated by each solution, in order to limit this impact on performance.

Parameters & Experiment Setup

Table 4.5 lists all of the parameters used in this experiment. The set up and EA parameters are similar to that of experiment 1, however the number of rules fired per graph, and the number of iterations have been increased. These values have been increased as this experiment has additional complexity, due to the objective function having two graph measurements, and we want to ensure the EA has enough time-steps to converge to an optimum solution, in this more difficult solution space.

Table 4.5: Ex3 Parameters

Exp. Number 3	Ex. 3
Iterations	20
Population Size	150
Graphs per Solution	7
Rules Fired	200
Tournament Size	2
Elitism Rate	0.020
Mutation Rate	0.20
Mutation Amount	0.10

Experiment 3 fitness function (CC-DEN_F):

$$f(I) = 1 - \frac{\sum_i^n |clusteringCoefficient(g_i) - TARGET\_CC| + |density(g_i) - TARGET\_DENSITY|}{2n}$$

where n is the number of graphs generated per solution

The fitness function used for this experiment (CC-DEN_F) measures both clustering coefficient and density, with each contributing equally when the fitness score of a solution is calculated. The maximum fitness score possible is 1.0. For each target density value in the range (0.0 -> 1.0, increments of 0.1), the EA was run with each target cc value in the same range. This was 11 x 11 runs of the EA, for a total of 121 runs.

For example, if the EA was run with a target cc value of 1.0, and a target density value of 0.5, and the graphs generated for a particular solution had an average cc value of 0.8, and average density value of 0.4, the fitness score would be calculated as so:

$$fitness = 1 - (((1.0 - 0.8) + (0.5 - 0.4))/2) = 1 - 0.15 = 0.85$$

Results

The results for a number of the EA runs, with different combinations of target cc/density values are shown in Figure 4.9. Each line on the graph represents the performance of the EA with a fixed density target, with the cc target (x-axis) being varied. For example, if we observe the brown line (target density = 1.0) we can see the the best fitness scores achieved increased, almost linearly, with the increasing target cc value.

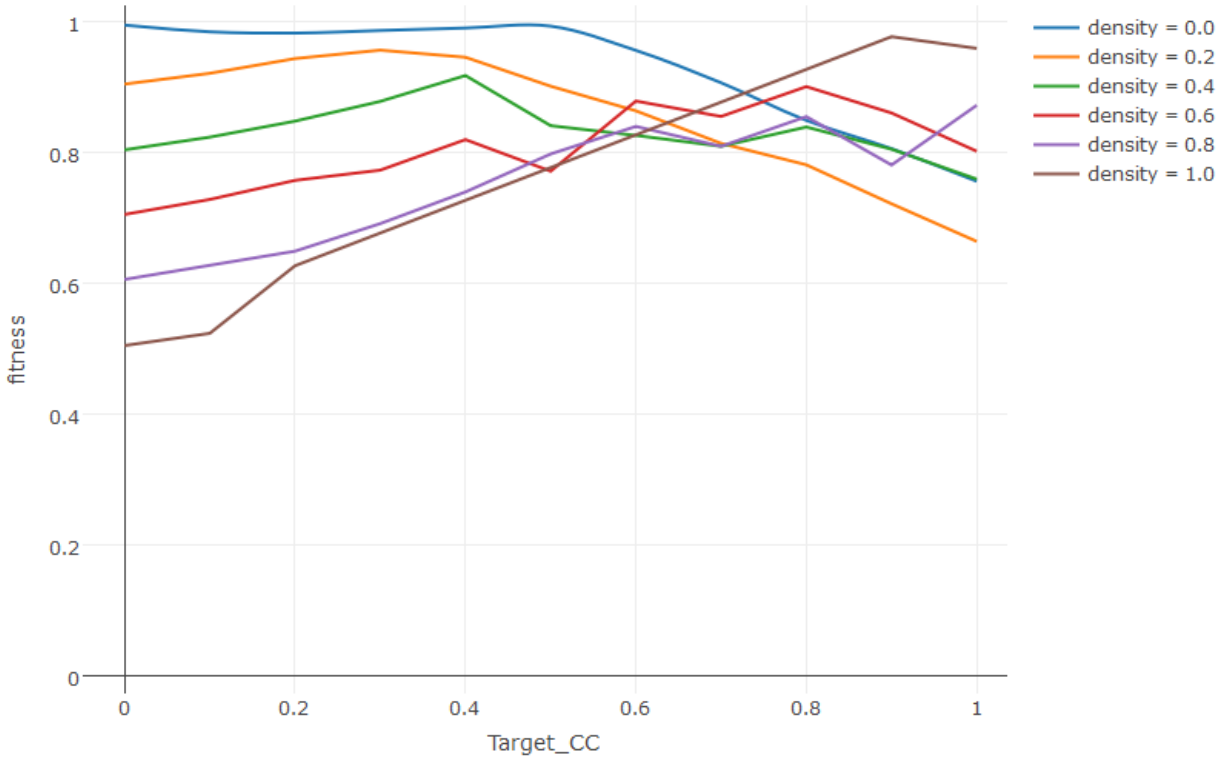

Figure 4.9: The fitness achieved (y-axis) for certain target density values across the range of target cc values (x-axis)

Table 4.6: Results of the EA run with a fixed target density of 0.0, across the range of target cc values

Target Density	Target CC	Add Edge	Add Triangle	Add Node	Best Fitness Achieved
0.0	0.0	0.000	0.000	1.000	0.995
0.0	0.1	0.240	0.100	0.660	0.984
0.0	0.2	0.089	0.259	0.651	0.982
0.0	0.3	0.185	0.393	0.422	0.986
0.0	0.4	0.041	0.645	0.314	0.990
0.0	0.5	0.050	0.915	0.035	0.993
0.0	0.6	0.000	1.000	0.000	0.956
0.0	0.7	0.000	1.000	0.000	0.906
0.0	0.8	0.000	0.950	0.050	0.849
0.0	0.9	0.000	1.000	0.000	0.806
0.0	1.0	0.000	1.000	0.000	0.756

Table 4.7: Results of the EA run with a fixed target density of 0.6, across the range of target cc values.

Target Density	Target CC	Add Edge	Add Triangle	Add Node	Best Fitness Achieved
0.6	0.0	0.044	0.000	0.956	0.705
0.6	0.1	0.640	0.000	0.360	0.728
0.6	0.2	0.757	0.000	0.243	0.757
0.6	0.3	0.879	0.041	0.080	0.773
0.6	0.4	0.839	0.050	0.100	0.820
0.6	0.5	0.850	0.050	0.100	0.771
0.6	0.6	0.870	0.000	0.130	0.879
0.6	0.7	0.912	0.000	0.088	0.855
0.6	0.8	0.897	0.000	0.103	0.901
0.6	0.9	0.900	0.000	0.100	0.860
0.6	1.0	0.900	0.000	0.100	0.802

Discussion on Results

Figure 4.9 shows the results for the generation method when a combination of two desired properties are used in the fitness function. When the target density is fixed to 0.0, we see very good performance when low cc values are targeted. The generation method creates large sparse graphs by greatly favoring the Add Node rule. As the desired cc value is increased, we see an increase in difficultly - this is because the Add Triangle rule probability must be increased to introduce some clustering to the graph, this also introduces more edges, which makes it difficult to keep an extremely low density in the graph.

The reverse is true for a density target fixed at 1.0 - we see the best performance on the other extreme of the target cc range, and as we decrease this target cc the performance suffers slightly. In Table 4.7, we can see that with a fixed target density of 0.6, as we increase the target cc value, there is an almost inverse relationship between the Add Edge and the Add Node Rules - as the target cc is increased, the solutions that favor the Add Edge rule are more successful, and the solutions that favor the Add Node rule become less and less so.

The most difficult combination of these target values, e.g. the combinations that were targeted when the solutions evolved by the EA had the lowest final fitness scores, can be seen when a high cc is targeted at the same time as a low density. The EA, when tasked with generating graphs that had a combination of 0.0 density and 1.0 cc, could only achieve a best fitness score of 0.5. This is a difficult combination because highly clustered graphs tend to be quite dense, making these two particular targets incongruent with each other.

4.4 Experiment 4

Research Question

What impact does introducing a global objective, such as a specific diameter value, have on the ability of the method to find desirable networks?

4.4.1 Aim

- To investigate the affect of introducing a global objective into the fitness function, namely diameter and average path length. These measures are fragile in that they have potential to change drastically, in just a few time steps of generation.
- To investigate the relationship between these fragile global objectives and less fragile measures, such as clustering coefficient and density.

Parameters & Experiment Setup

In a similar fashion to the previous experiments, two different fitness functions will be used. The first fitness function (APL-CC_F) comprises of two objectives - solutions that generate graphs with a specific average path length, as well as a specific clustering coefficient are rewarded favorably, whereas solutions that generate graphs that are far away from these target values are less favored. The second fitness function works in a similar manner, but this time the fitness function will direct the search towards the solutions that generate the best graphs, based on the target diameter and density values that are input. Table 4.8 lists all of the EA parameters used for this experiment. The EA will be run 33 times for each fitness function. For the first fitness function, 11 runs will be conducted, fixing a target average

path length, and incrementing the target cc values in the same range as the previous experiments. This will be repeated for two more target average path lengths, for a total of 33 runs. This process will then be mirrored for the second fitness function, this time updating the diameter target, and iterating across the range of density values.

Table 4.8: Ex 4 Parameters

Exp. Number 4	Ex. 4
Iterations	20
Population Size	150
Graphs per Solution	7
Rules Fired	200
Tournament Size	2
Elitism Rate	0.020
Mutation Rate	0.20
Mutation Amount	0.10

Experiment 4 fitness function (APL-CC_F):

$$f(I) = 1 - \frac{\sum_i^n log(|avgPathlength(g_i) - TARGET\_APL| + 1) + |clustCoeff(g_i) - TARGET\_CC|}{2n}$$

where n is the number of graphs generated per solution

Experiment 4 fitness function (DIAM-DEN_F):

$$f(I) = 1 - \frac{\sum_i^n log(|diameter(g_i) - TARGET\_DIAM| + 1) + |density(g_i) - TARGET\_DENSITY|}{2n}$$

where n is the number of graphs generated per solution

Results

The results for the EA runs using each of the fitness functions are included below. Figure 4.10 and Figure 4.11 show the results of the first (APL-CC_F) and second (DIAM-DEN_F) fitness functions, respectively. The fitness of the best individual for each of the EA runs is on the y-axis, while the x-axis shows which value of cc/density was input to the fitness function on that particular run:

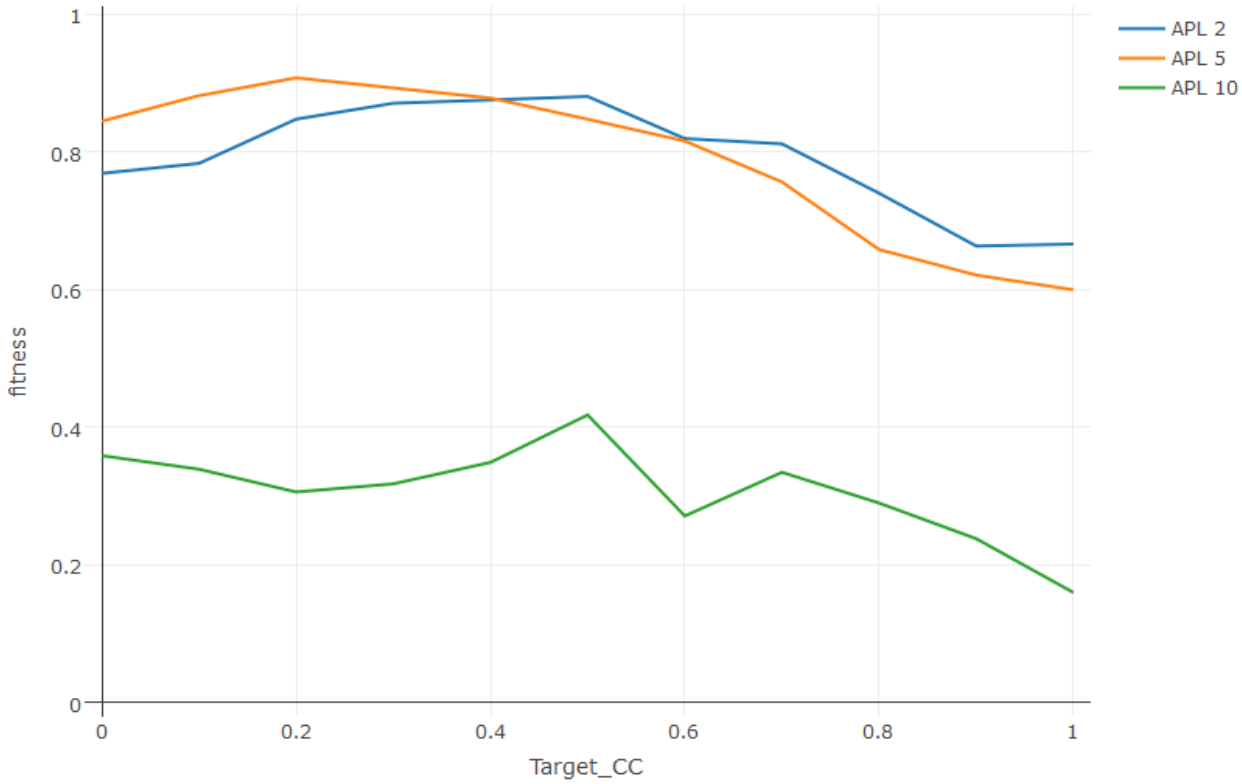

Figure 4.10: Fitness on the (y-axis) vs the target CC (x-axis), for three different average path length targets

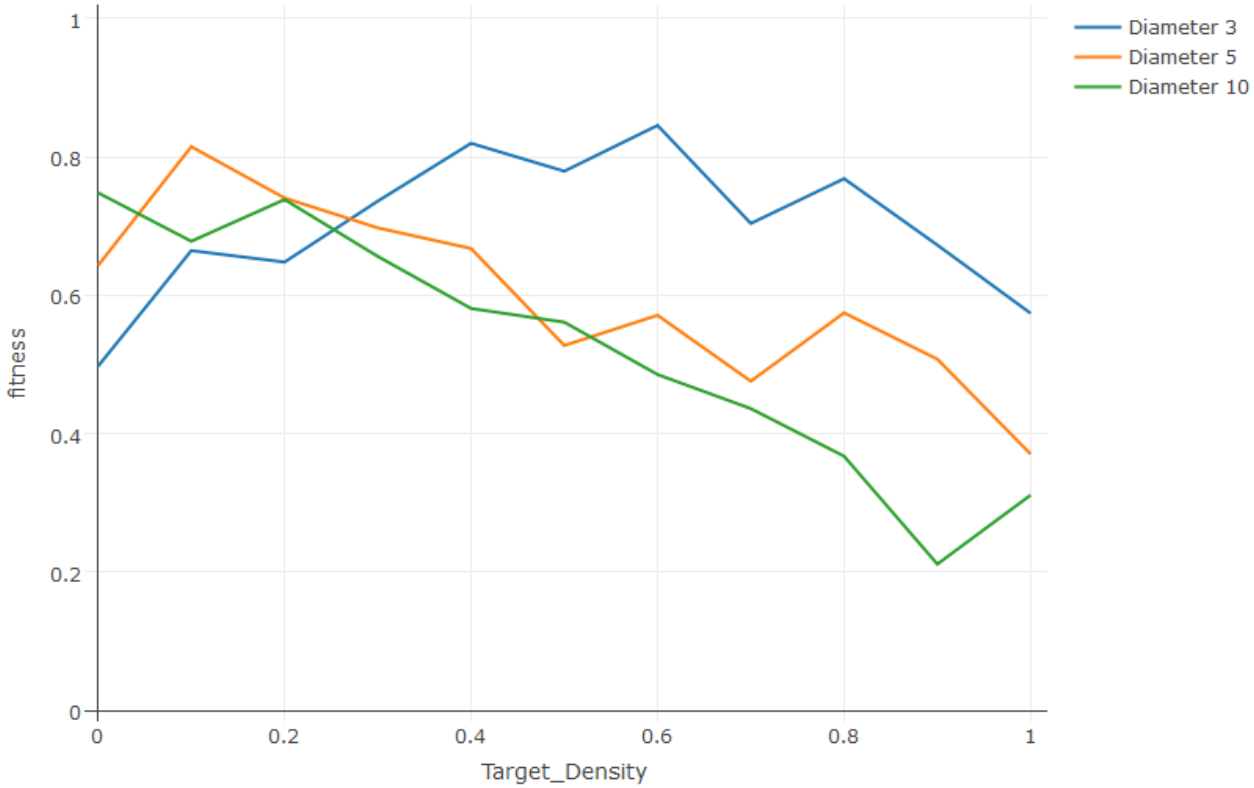

Figure 4.11: Fitness on the (y-axis) vs the target Density (x-axis), for three different diameter targets

Discussion on Results

In this experiment, we introduced new network properties: average path length and diameter. In Figure 4.10 we see the results of the EA when the clustering coefficient and average path length values were targeted in the fitness function. The EA was generally successful across the range of target cc values, when an average path lengths of 2 and 5 were targeted, with the higher cc targets being more difficult. This is because as the clustering coefficient target is increased, the generation process tends to favor smaller graphs, which are created predominantly with a combination of the Add Triangle and Add Edge rules. When we task

the EA with evolving solutions that generate graphs exhibiting a much larger average path length, we see a drastic decrease in the EAs ability to find quality solutions. This is expected however, as most networks are ”small-world” networks, e.g. that all of the vertices are connected to each other through a relatively small path. The small world concept is the idea behind the popular ”Six degrees of Kevin Bacon” game. Considering the relatively small number of rules being fired to generated graphs in the current configuration, and thereby the maximum size of the generated graphs, this would likely preclude the generation of a graph with an average path length of 10.

Similar results were observed for the diameter-density based fitness functions. As the target density value is increased it becomes increasingly difficult to maintain a high diameter value, as the solutions that generated graphs with a high diameter utilized the Add Node rule almost exclusively. A target of diameter 10, when combined with a desired density value, becomes extremely difficult to induce in a generated graph.

5

Conclusion

This section is comprised of a summary of the work that has been completed, as well as conclusions and discussions on the project, and the results obtained. Also included is a substantial list of potential expansions to this project, which can form the basis for further research in the future.

5.1 Summary

A method for generating synthetic networks that exhibit certain desirable properties has been designed, implemented and evaluated through a series of experiments. A substantial amount of background research and thought went into the design of this network generation method. Networks are generated by applying rules to an initial starting graph, each rule adds edges and/or vertices, and the graph grows in this way. A probabilistic selection process is used to select which rule to fire at each time of the generation process, and an evolutionary algorithm was chosen to optimize this probabilistic selection process. A highly flexible and reusable evolutionary algorithm framework was implemented in Scala, to carry out this optimization.

A series of research questions were formulated, of which experiments were designed and conducted to investigate. In each run of the EA, the specific properties deemed desirable are configured through the fitness function, which directs the search towards the probability mass function (rule probabilities) that generate the graphs that most closely match these desirable properties.

In experiment 1 the generation method performed quite successfully, and had little difficulty in targeting the vast majority of desired values that were input. In Experiment 2, the stochasticity of the rule-based approach was investigated. The variance of results introduced as a result of this stochasticity was not as profound as originally hypothesized, and the ideal number of graphs each solution should generate was found to be 7. This minimizes the variance and improves our confidence in the results of the EA without sacrificing too much in terms of run-time complexity. Experiment 3 introduced a multi-attribute fit-

ness function and the performance of the EA was evaluated in this more difficult search space. In experiment 4 we tasked the EA with generating networks across a large range of combinations of two network attributes, density and clustering coefficient. The EA performed quite well and evolved high quality solutions for the majority of combinations. Certain combinations, such as an extremely high clustering coefficient paired with an extremely low density value were much more difficult to generate, as these two targets are generally incongruent with each other. In experiment 4 we introduced the diameter and average path length measures to our generation process. These hyper-global measures introduced a lot of difficulty to the search space, and performance was generally much lower than when targeting clustering coefficient and density values. It was shown that a high diameter and average path length values are very difficult to induce in small/medium sized graphs.

5.2 Conclusion

The network generation method designed has been fully realized in the Scala based implementation. The research questions formulated have been fully examined and answered, through the experiments that were conducted. The EA framework developed was used extensively and proved to be very flexible and easy to modify, fulfilling the original requirements. The generation method has very little difficulty in generating graphs with specific values of properties such as density and clustering coefficient, however some difficulty can arise when certain values of fragile, global objectives are introduced in combination with these desired measurements. A series of promising potential expansions and improvements have been proposed. These extensions should do much to increase the accuracy and quality of this network generation method. In all, the EA was run hundreds of times, and millions of synthetic networks were generated and measured. In conclusion this project has succeeded in designing and evaluating a promising method for generating synthetic networks, as well as a flexible and reusable evolutionary algorithm framework.

5.3 Future Work

The network generation method designed and developed in this project has great potential to be continued and expanded, and can form the basis for many future experiments and further research into methods of generating synthetic networks. Many ideas for further experiments on, and expansions to, this research, occurred to both myself and my supervisor but unfortunately due to time constraints and the already large scope of this project were unable to be explored further. A brief outline of some of these ideas are provided here. The author will happily discuss, and expand upon the ideas formulated here, with the interested reader. The author is available to contact at: dylankelly@live.ie

5.3.1 Expanded Rule Set

The rule set used in this work consists of three constructive rules. This rule set could be easily extended and then evaluated, with this work providing a baseline for comparison. It should be relatively easy to expand the rule set due to the flexible implementation that was developed, i.e. the abstract 'Rule' class can be easily extended to include additional rules. For example, a rule could be added that selects a triplet of existing nodes that aren't adjacent to each other, and adds three new edges, thus completing a new triangle.

Another substantial extension to this work would be the design and implementation of a network generation method that adopts the destructive approach to graph generation. A new set of destructive rules would be designed, with a similar rule selection process as used in this work. Similar experiments could be carried out, the results of which could be compared with the results achieved in this project.

After the destructive approach has been designed, implemented and evaluated, one could

also evaluate a network generation process that uses both constructive and destructive rules in the rule set and compare the results to the previous experiments in which strictly one genus of rule was used.

In the current implementation, each solution uses an identical rule set; instead, each solution could use a rule set that is made up of rules sampled randomly from the list of rules that are available. This approach would allow us to gain better insight into the relationship of the rules with each other.

5.3.2 Modified Rule Selection Process

Each of the rules designed and implemented in this work added to the graph in some way. The first step of each rule involved selecting a node or pair of nodes at random as the part of the graph that would be expanded; each node has an equal chance of being selected in this process. An interesting concept that could be explored is the idea of preferential attachment. Preferential attachment works in a "rich get richer" fashion - instead of each node having an equal chance of being selected, the probability of a node being selected is proportional to the number of connections it already has. Employing this process would direct graph growth from the areas of the graph that are dense. Reversing this relationship could also be investigated, i.e. one could instead make the chance of being selected in a rule inversely proportional to the number of connections that a node has. This would encourage growth of the graph in the sparser areas.

Another possible modification would be to introduce random sampling to the makeup of the rule set. In the current implementation, each solution uses an identical rule set; instead, each solution could use a rule set that is made up of rules sampled randomly from the

list of rules that are available. This approach would allow us to gain better insight into the relationship of the rules with each other.

5.3.3 Self-Adaptive Evolutionary Algorithm

The control parameters of the evolutionary algorithm (EA) used in this research can be considered, at least in some part, arbitrary. EA parameters such as the rate of mutation, tournament size and the mutation value affect the performance of the EA, and it is very difficult to find the optimal choice of these values. Self-adaptive EAs incorporate these control parameter values into genes of the population, and have been shown to produce results that compare favorably with standard EAs. The graph generation method designed and evaluated in this work would be an interesting problem to test this self-adaptive approach, and the results presented here would form a baseline for comparison.

5.3.4 Introduce Context to the Rules

The rules designed and utilized in this research are context free rules i.e. the probability of a rule being fired is independent of the current state of the graph. If the possible graph states were discretized into categories, the current state of the graph could be used to influence the probability that a particular rule is selected. This idea has the potential to greatly increase the ability of the graph generation method to generate graphs with the desired properties.

For example, when a node is selected to have a rule fired on it, we might adjust the probabilities based on which rule was fired when the selected node was created. Another possible context that could be added is the current size of the graph - a rule fired on a very small graph has a different effect on the properties of the graph than when fired on a very large

graph . This would introduce a notion of time into the rule selection process, as the further along in the generation process we are the larger the graph is. These context based rules would be optimized using the evolutionary algorithm.

5.3.5 Evolved Rules

An interesting approach that could be explored is the idea of evolving the actual rules that are used in the generation process. Each rule adds edges and/or vertices to the graph; these edges/vertices can be viewed as a subgraph or substructure. The particular substructure that each rule adds is defined a priori when a rule is designed. For example, the Add Triangle rule adds a simple triangle structure to the graph. Each of these substructures has local properties and measures independent of the graph as a whole. In fact, some global measures such as clustering coefficient are simply the average of the local clustering coefficient across all nodes in the network.

Small graphs, with 10 vertices for example, could be generated and measured, and then these small graphs, or substructures, can form the right hand side of a rule. In fact, the basis for this approach has already been implemented by the author. Small graphs are created randomly and measured, using the evolutionary algorithm framework, with the density and fitness functions implemented for the current work guiding the search towards suitable substructures. In this way, new rules would be evolved, and these rules in turn could be introduced to the rule set, to generate the larger networks. The idea is that we can build networks exhibiting desired global properties by gluing together many smaller graphs, which have been evolved by the EA.

References

[1] Bäck, T., Fogel, D. B., & Michalewicz, Z. (1997). Handbook of evolutionary computation. *New York: Oxford*.

[2] Barabási, A.-L. (2013). Network science. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 371(1987), 20120375.

[3] Barabási, A.-L. & Albert, R. (1999). Emergence of scaling in random networks. *Science*, 286(5439), 509–512.

[4] Cherkassky, B., Goldberg, A. V., & Radzik, T. (1993). Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73, 129–174.

[5] Cormen, T. H., Stein, C., Rivest, R. L., & Leiserson, C. E. (2001). *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition.

[6] Dobson, I., Carreras, B. A., Lynch, V. E., & Newman, D. E. (2007). Complex systems analysis of series of blackouts: Cascading failure, critical points, and self-organization. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 17(2), 026103.

[7] Easley, D. A. & Kleinberg, J. M. (2010). *Networks, Crowds, and Markets - Reasoning About a Highly Connected World.* Cambridge University Press.

[8] Watts, D. & Strogatz, S. (1998). Collective dynamics of 'small-world' networks. *Nature*, 393(6684), 440–442.

[9] West, D. B. et al. (2001). *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River.

[10] Wikipedia (2017). Network science — wikipedia, the free encyclopedia. [Online; accessed 3-April-2017].