Interactive Random Graph Generation with Evolutionary Algorithms

Benjamin Bach[1], André Spritzer[2], Evelyne Lutton[1], and Jean-Daniel Fekete[1]

[1] INRIA, France,
{firstname.lastname}@inria.fr,
[2] Universidade Federal do Rio Grande do Sul, Brazil,
spritzer@inf.ufrgs.br

Abstract. This article introduces an interactive system called GraphCuisine that lets users steer an Evolutionary Algorithm (EA) to create random graphs matching a set of user-specified measures. Generating random graphs with particular characteristics is crucial for evaluating graph algorithms, layouts and visualization techniques. Current random graph generators provide limited control of the final characteristics of the graphs they generate. The situation is even harder when one wants to generate random graphs similar to a given one. This is due to the fact that the similarity of graphs is often based on unknown parameters leading to a long and painful iterative process including steps of random graph generation, parameter changes, and visual inspection. Our system is based on an approach of interactive evolutionary computation. Fitting generator parameters to create graphs with defined measures is an optimization problem, while judging the quality of the resulting graphs often involves human subjective judgment. We describe the graph generation process from a user's perspective, provide details about our evolutionary algorithm and demonstrate how GraphCuisine is employed to generate graphs that mimic a given real world network.

1 Introduction

Conducting formal evaluations of graph algorithms, layouts, and visualization techniques requires the availability of a large number of comparable graphs with specific controllable characteristics. Real world graphs are always desirable when ecological validity is needed in formal evaluations, but finding real graphs with a specific set of characteristics is usually hard or impossible. Even when they do exist, they might not be available in sufficient quantity or variety, or may contain sensitive information requiring more than anonymization to be disclosed. An alternative to employing real-world graphs is the generation of graphs with particular properties using random graph generators. However, existing generators have limited flexibility, which may not be sufficient for formal evaluations of algorithms and layouts, for example if one wants to evaluate graphs with an upper bound on the node degrees or a given number of connected components. whereas the algorithms and layouts that are to be tested may need more control—*e. g.* one connected component or an upper bound on the node degrees.

In this article we introduce GraphCuisine: a novel interactive system that that lets users steer an Evolutionary Algorithm (EA) to create random graphs matching user-specified properties. Our system provides three different approaches to the interactive generation of random graphs: (a) directly setting desired graph measures (*e. g.* density between 5% and 10%), (b) selecting "good" examples from a set of generated graphs for further evolution, and (c) extract properties from an imported sample graph, still including the possibility of user control. To allow for these different modes of interactive random graph generation, GraphCuisine consists of an interactive interface and a graph generation model that combines different generators and optimizes their parameters with an EA. Graphs are created by traversing a pipeline of generators, each acting on the output of the previous one by adding or removing nodes and edges in very specific ways according to the generators respective input parameters. The EA then attempts to find sets of parameters that produce graphs that best match the user-defined properties.

GraphCuisine has been designed with three main scenarios in mind:

1. **Generate graphs with specific graph measures**. When designing a new layout, routing, or edge bundling algorithm, being able to test its behavior with varying topological properties, described by graph measures (*e. g.* density, diameter or clustering-coefficient) is essential and almost impossible with current generators.
2. **Generate graphs that "look like" a target graph** not knowing in advance the measures that contribute to the graph's structure. When testing the appropriateness of an algorithm for a specific application domain (*e. g.* network traffic) with only a few samples of real graphs available, generating graphs similar to the samples is important. In that case, selecting the measures that best characterize the sample graphs requires visualization, human judgment and interaction.
3. **Anonymize a specific graph** by generating a graph with similar or identical measures. When a company has confidential graphs to analyze (*e. g.* e-mail exchanges between employees or telephone calls between possible suspects), collaborating with researchers or specialists can be impossible. Generating graphs as similar as possible as to the interesting confidential graphs makes collaborations much simpler.

The paper is structured as follows: after reviewing related work (Section 2), with a focus on graph generation and network evolution, we present an overview of GraphCuisine from a user's perspective (Section 3). We then describe the graph generation and evaluation processes in detail (Section 4), followed by a usage scenario (Section 5) and a selection of generated graphs (Appendix B.)

2 Related Work

Random Graph Generation. Generators have been developed using different techniques to construct graphs with varying characteristics. *Preferential attachment* generators [1–6] add nodes to the network by connecting them to existing

nodes with a particular probability, while *rewiring* generators [7–9], reconnect existing edges. Preferential attachment aims to yield networks with power law degree, rewiring is used to create mostly small world structures. Some rewiring generators initially place all graph nodes onto a 2D space and consider spatial distances between nodes when calculating the probability of an edge between them [7, 8]. Other techniques have been proposed to overcome drawbacks of preferential attachment and rewiring techniques, mainly to create networks with more realistic structures. *Structural generators* first create higher level structures of the graph, such as clusters and recursively model details [10, 11]. A problem common to most generators is that they focus on producing networks with one or two particular characteristics, disregarding all the others. Consequently, techniques have been proposed that treat graph generation as a global *optimization problem* [12, 13] or using local *forces* between nodes and edges by evolving the network [14, 15]. R-Mat tries to model as many characteristics as possible in a flexible way by partitioning an adjacency matrix into cells, similar to BRITE [4] and creates edges with different probabilities. The model features several graph characteristics and comes with an input parameter fitting function. However, R-Mat generates one graph at a time and only matches few features.

Choosing the appropriate generator and modelling the desired graph characteristics is crucial for several contexts; the lack of expressiveness of existing generators impacts the generation quality or requires a long sequence of trials and errors to obtain graphs with a desired structure.

Evolutionary Algorithms are stochastic optimization heuristics that copy, in a very abstract manner, the principles of natural evolution that let a population of individuals be adapted to its environment [16]: they have the major advantage of making only few assumptions on the function to be optimized. In short, an EA considers populations of potential solutions exactly like a natural population of individuals that live, fight, and reproduce, but the natural environment pressure is replaced by an "optimization" pressure. In this way, individuals that reproduce are the best ones with respect to the problem to be solved. Reproduction consists in generating new solutions via variation schemes (the genetic operators), that, by analogy with nature, are called mutation if it involves one individual, or crossover if it involves two parent solutions. A *fitness function*, computed on the individuals, is optimized by the EA. Evolutionary optimization techniques are particularly well suited to complex problems, where classical methods fail due to the irregularity of the function to optimize or to the complexity of the search space. In this article, we deal with an interactive evolutionary algorithm (IEA) as it is applied to the optimization of a quantity that is partially specified by the user via an interactive interface.

In the field of Neuroevolution, EAs have been used to evolve the topology and edge weights of neural networks in their creation and learning phase. Neuroevolution faces two main problems: (1) How to encode neural networks in genes, and (2) how to design effective cross-over operators that do not destroy desired sub-structures in the networks. There are mainly two approaches: *Direct encoding* that requires the design of specific genetic operators to ensure efficient

exploration capabilities of the algorithm [17] or *Grammatical Encoding*, that represents the networks in a generative way [18–20], where an individual represents transformation rules (i. e. how the network is generated), not the network itself.

Grammar-based approaches are similar to ours but differ in that Neuroevolution aims at optimizing and reproducing one single network that is in turn judged according to its functionality. We are interested by generating a wide diversity of solutions to give more options to human judgment.

UIs for Data Generation and Analysis. Few approaches exist that tightly involve the user in the data generation process. Wong et al. [21] generate graphs through sketching by using adjacency matrices and adapt pixel-based drawing techniques to draw edges inside the matrix. Although a node-link representation is provided in parallel with the matrix, only the matrix is used for drawing. Except when drawing cliques, this makes it hard to predict the final layout of the resulting graph in the form of a node link diagram, and, except for node count and edge count, it is almost impossible to control measures of the resulting graph. For the general creation of multivariate data, Albuquerque et al. [22] present interactive approaches using generators and sketching methods for 1D, 2D and 3D scatterplots. Still, sketching interfaces do not create diverse data sets, but instead only provide one particular solution.

3 GraphCuisine

GraphCuisine is a random graph generator for undirected graphs that evolves a population of individuals encoded in *chromosomes*, each of them representing a "recipe" to generate a graph. A graph is generated from a chromosome by running a pipeline of generators, each reading its input parameters from the chromosome and modifying the graph by systematically adding or removing nodes and edges. GraphCuisine implements 12 graph generators, each using 2–5 input parameters and creating basic network structures such as stars, clusters and several types of noise (see Table 1). The target graph towards which the graphs (as representatives of individuals) are evolved is encoded as a set of *target measures*— such as node count, density and clustering coefficient. The fitness of a generated graph is calculated as the distance between actual graph measures and the target values (see Section 4.2).

Encoding the genome as a set of parameters to apply to simple generators has several advantages: (1) the genome size remains constant and independent of the graph size, and (2) the measures used to optimize the graphs are independent from the number and parameters of the simple generators, allowing new measures to be added easily, as well as new generators. Having constant-size genomes also facilitates crossovers to obtain a wider diversity of generated solutions.

The interface of GraphCuisine is made up of five major parts (see Fig. 1): (a) a visualization of 12 representative graphs from the current population (*population view*), (b) an enlarged view of a selected graph (*detail view*), (c) if desired, an imported graph from which target measures are extracted (*data set view*), (d) an interactive parallel coordinates plot showing the distribution of measures

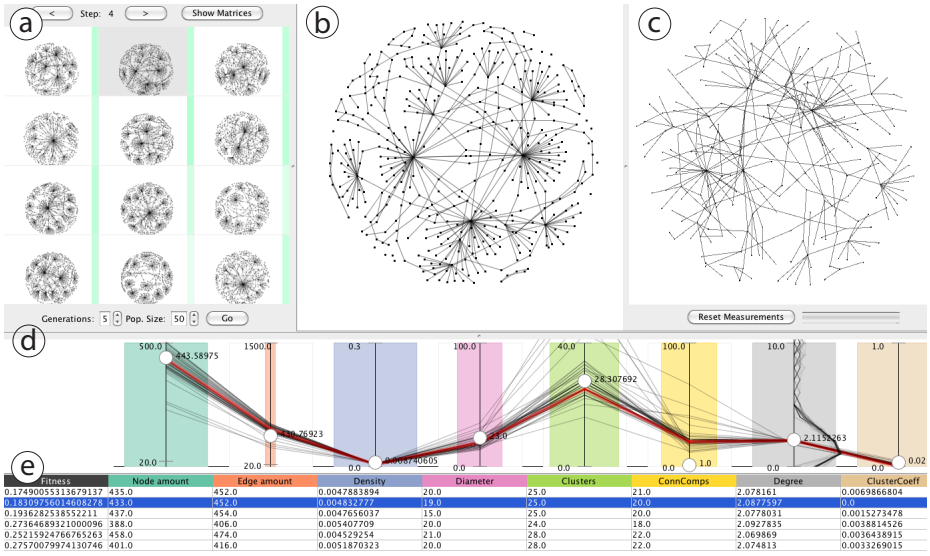
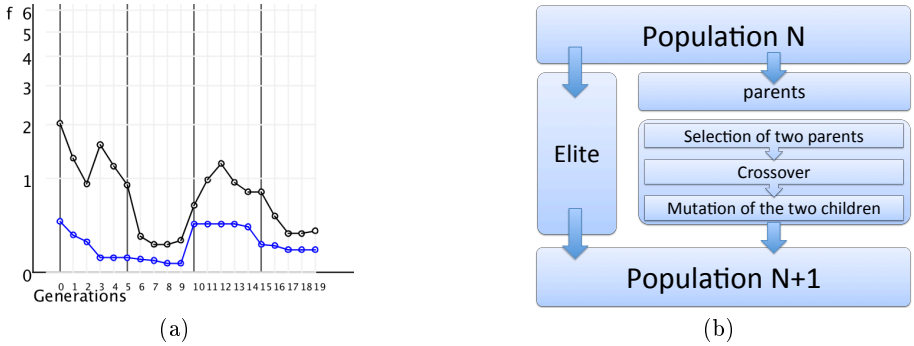


Fig. 1. User Interface of GraphCuisine showing (a) the population view displaying representative graphs of the current population (*population view*), (b) a detail view of selected graph (*detail view*), (c) an imported graph (*data set view*), (d) the parallel coordinates plot showing the distribution of measures in the population (*measure view*) and (e) the table with the graph measures of the representative graphs (*measures table*). The position of the white markers in the measure view indicate the target measurements and the red line the measurements of the selected graph.



(a) (b)

Fig. 2. (a) Fitness curve showing the evolution of the population fitness in a logarithmic scale. Better fitness values are closer to 0. The blue line shows the value of the fittest individual in the population and the black shows the mean fitness. The stronger black vertical lines indicate the generations that have been presented to the user and upward peaks in the blue curve show that the user has changed the target measures, leading to decreased fitness the current population. (b) Evolution process for one generation in GraphCuisine.

in the whole population (*measure view*), and (e) a table showing all measures of the representative graph (*measure table*).

The user can switch between adjacency matrix and node-link representations for all the graphs on the population view (a) since some structural characteristics are more visible on one or the other. The measure view (d) shows a parallel coordinate visualization where each measure is an axis over a color coded rectangle; the width indicating the weight of the measure in the fitness computation. As in standard parallel coordinates, each poly-line represents a graph in the current population and connects all the graph's measures. Vertical axes show the value range of each measure as set by the user.

A vertical line plot draws the distribution of node degrees for each graph, right of gray *degree* measure axis. It shows how much the graphs converge or diverge in this measure. Tick marks on the axis lines indicate the desired minimal and maximal values, while the white circle indicates the desired target value for the population All the values can be adjusted interactively by dragging the marks. Views in GraphCuisine are coordinated with brushing&linking: moving the mouse over a poly-line highlights the corresponding graph in the population and table views, and vice-versa. To initialize the graph generation, the user can choose among the following options:

***Random initialization*:** The population is created from a random set of generator parameters for each chromosome and random target measures, providing a base for a more exploratory generation.

***Set target measures directly*:** The values for the desired graph measures can be set interactively by adapting the minimal, maximal and target values as well as the measure weight directly in the measure view.

***Graph templates*:** Templates are predefined sets of generator parameters and measures selected to create graphs with particular characteristics such as small-sized, medium-sized or large-sized, or that are dense or sparse, with a power law degree distribution, or small world. Templates are chosen from a menu where complementary templates can be selected at a time. New templates can be defined by the user for later reuse. Based on a template, an initial population is created.

***Load an existing graph*:** To mimic an existing graph, the graph is imported and shown in the data set view (Fig. 1(c)). Its measures are computed, displayed in the measure view, and serve as target values for the evolution. The measures and their weights can immediately be modified by the user or at any time during the evolution; they can also be reset to the value of the loaded graph.

Generating graphs with GraphCuisine consists of two alternating phases. After the initialization and selection of the target measures, the evolution of the generator parameters is started and a machine solution is created by optimizing the initial population of graphs to fit the target values. After a few generations, we select 12 representative graphs, the three fittest and nine at random for purposes of diversity; they are displayed in the population view and the measure table. The measure view is then updated to display the values from the whole

population. The saturation of the green bars in the population view indicates the fitness of each graph and adapts immediately if the user changes the target measures in the measure view. This feedback is required for the user to know which graphs are considered similar by the system with the current settings.

Alternatively to changing measures in the respective view, users can select graphs they judge *good* directly from the population view by clicking on them. Multiple graphs can be selected and the target measures are set to reflect the chosen graphs. Measure weights are updated according to the variance of their respective values in the selected graphs—the more a particular measure differs, the lower it is weighted and vice-versa (see Section 4.3). The evolution can then run for a further step with the new measures being taken into account. The number of generations as well as other parameters of the EA can be adjusted by the user at any time, allowing for control of the evolution behavior (populations size, generations, elitism, etc.). Disabling generators prevents the graph from containing particular characteristics and setting their parameters directly gives additional freedom in specifying graphs.

Behavior and convergence of the fitness of the whole population is monitored in the chart shown in Fig. 2(a). It shows minimal (the best), maximal (the worst) and the average fitness through all generations using a logarithmic scale. In order to avoid previous graphs being lost during the evolution process, GraphCuisine keeps a The system can be reset to any generation to try alternative evolutions. Additionally, the chromosome of any generated graph can be saved as template to initialize new populations later on.

4 Generation and Evolution

As explained in the previous section, each individual in GraphCuisine's EA corresponds to a graph. It is encoded in a generative way, that is as *chromosome* each of which fully specifies a sequence of generators and their parameters. The creation of a graph from a chromosome and the evaluation of its fitness is done in four steps, illustrated in Fig. 3.

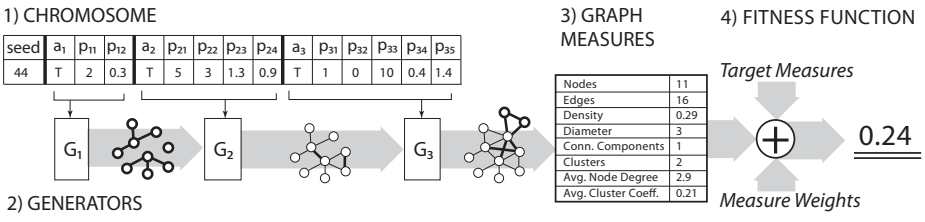


Fig. 3. Graph creation process: (1) Parameters encoded in genes of a chromosome, (2) employ graph generators, (3) extract graph measures, and (4) calculate fitness value.

We implemented two specific types of generators in GraphCuisine: Motif generators and noise generators. They are applied in two steps: Motif generators start by creating topological patterns, such as clusters, stars, paths, and cycles. The generators parameters specify subgraph specific properties (*e. g.* the degree of a star's central node) and how many instances of it should be created (see Table 1). Noise generators are applied in a second step to break these regularities, making structures more varied and less predictable by adding or removing nodes and edges. Nodes and edges can be inserted randomly with different node degree distributions: uniform, power-law, logarithmic or exponential. Nodes and edges to be removed are either chosen randomly or according to some criterion, such as eliminating isolated nodes (degree 0). These two steps are conceptually similar to some structural generators but the motifs and noise we produce are more varied and generalized.

Parameters for noise generators define how many nodes or edges the graph should have (and thus how many should be added or removed) as a means to control the approximate size of the graph. More specific properties exist, such as the probability of connecting two given nodes by an edge. The count of nodes and edges to be inserted and which nodes should be connected depend on what is already in the graph. As such, if applied in a different order, the same noise generators with the same parameters will produce completely different graphs.

4.1 Chromosome Structure: Internal Encoding of an Individual

Each chromosome is divided into two blocks. The first is a single gene with an integer value, which serves as a seed for the random number generator that ensures consistent random values for the generation. The second and third blocks stand for the motif and noise generators, respectively. They are both represented in the same way: the first gene contains a boolean value indicating whether the corresponding generator is active or not and the second gene contains a value specifying the execution order of the generator in the creation pipeline (Fig. 3). Finally, the other genes of each generator contain its input parameters.

4.2 Fitness Function: Quality Assessment of an Individual

The computation of the fitness of each chromosome/graph is based on the set of target graph measures $M = \{m_i | m_i \in \mathbb{R}, i \in [0, n]\}$, with tolerance bounds $m_i \in [\min_i, \max_i]$, and importance weights $w_i \in [0, 1]$. For a graph G_k each measure in M is computed as: $M_k = (m_{0,k}, \ldots, m_{n,k})$ where $m_{i,k}$ is the i^{th} graph measure. The $\mathit{fitness}(G_k) \in \mathbb{R}^+$ of the graph G_k is its distance to the optimal solution, *i. e.* the following weighted sum:

$$\mathit{fitness}(G_k) = \sum_{i \in [0,n]} w_i * \left| \frac{m_i - m_{i,k}}{\max_i - \min_i} \right|$$

Normalization is necessary to weight and sum up measures equally. Currently, GraphCuisine supports the following measures: *node count*, *edge count*, *density*,

graph diameter, *number of clusters*, *number of connected components*, *average clustering coefficient*, and *average node degree*.

4.3 Interactive Evolution

As mentioned in Section 3, the population view allows selecting *good* solutions by visual inspection. This feature is based on the assumption that node link diagrams or matrix representations allow humans to quickly compare graphs and match some of their visible structure effectively although with limited accuracy. When selecting multiple graphs $G_s = \{G_0, \ldots, G_s\}$, GraphCuisine tries to infer what characteristics are important to the user. The target value for each measure m_i is taken to be the average of measures from all selected graphs: $m_i = \overline{m_{i,k}}$. The weights w_i are adjusted to reflect the diversity or similarity of the single graph measures using the standard deviation of the normalized graph measures $m_{i,k}$:

$$w_i = stddev(m'_{i,0}, \ldots, m'_{i,s}) \text{ with } m'_{i,k} = \frac{m_i - m_{i,k}}{\max_i - \min_i}$$

Each time the user has changed the target measures or weights, the mutation rate is increased for one generation in order to explore new solutions.

4.4 Implementation

GraphCuisine is implemented in Java using the JGAP library[3]. By default, the graph generation process begins with a randomly generated population of 50 individuals, which is evolved for 5 generations in each evolution cycle. These values were chosen to keep a balance between population diversity and acceptable running time for each cycle, an important issue for GraphCuisine due to its interactive nature. In order not to lose good solutions over time, we guarantee that each generation keeps 30% of the individuals from the previous generation (*elitism*). Fig. 2(b) illustrates how one generation is evolved; a set of very fit graphs is selected (*parents*) to create offspring individuals by application of the genetic operators. The components of the evolutionary algorithm are the following:

- *A tournament selection of size 5* that randomly chooses 5 individuals from the population with uniform probability, and keeps the best one as parent. The process is repeated twice to select two parents for a crossover.
- *A one-point crossover* that creates two offsprings by swapping all the genes of the parents that comes after a randomly selected position in the genome.
- *A simple random mutation* that selects 5% of the genes of each individual and replaces them by random values.

Crossover and mutation are applied in a cascade, that means that 70% of the new individuals are created by applying a crossover, and then all new individuals undergo a mutation that alters 5% of its genes.

[3] `http://jgap.sourceforge.net`

The generator's performance is essentially limited by the measure computation; creation, mutation and crossover times are negligible. When a measure's weight is set to 0, the measure is not computed, allowing for tradeoffs between generation speed and expressive power. Number of clusters and clustering coefficient are currently the most expensive measures.

Extending the set of measures and generators consists in adding new implementations of generators and measures, adding the generators' parameters to the chromosome, adding the measures to the fitness function and updating the interface. Including measures into the interface boils down to adding columns to the table and dimensions to the parallel coordinates plot, respectively.

5 Example Scenario: Anonymizing a Network

Imagine, an HIV spreading network must be anonymized while keeping the characteristic structures present so that it can be given to students in epidemiology for analyses. By importing the network in GraphCuisine, its measures are calculated and automatically set as target measures. The graph population is randomly initialized to guarantee diversity. After a first evolution step of 5 generations, the representative graphs are shown in the population view and the statistics table. The parallel coordinates in the measure view shows that the distribution of measure values for all the 50 graphs in the current population has already converged well towards the target values.

The user can then decide to increase the graph size and, proportionally, the number of clusters. This is done by setting the desired *size* and the target *number of clusters* in the measure panel. Since these two seem to be the most important measures for this evolution, the user slightly increases the weight of these measures by varying the width of the colored rectangles in the parallel coordinates plot. After another evolution cycle, the graphs have grown and show a proportionally higher number of clusters. The degree of optimization and the similarity between generated graphs and the original one can be controlled by adjusting the number of generations in the evolution step.

If desired, the user can continue to refine the graphs for example by selecting favorite representative graphs from the population view and running another evolution cycle for obtaining graphs similar to the selected ones. Since selecting graphs causes the target measures to reflect the measures of the selection, they might slightly diverge from those of the imported graph. When a desired result has been reached, generated graphs can then be exported. Figure 1 shows the imported HIV network (right), the generated graphs (left), as well as a particular selected graph (center) with its measures highlighted in red in the measure view. The fitness view (Fig. 2(a)) shows the convergence of the population fitness declining, except when the target measures have been changed by the user. The pure evolution time (20 generations) was 6.52 minutes, for a population size of 50. Further examples of graphs generated with GraphCuisine are shown in Appendix B.

6 Conclusion and Future Work

This article describes GraphCuisine, a system that generates random graphs according to user specified graph measures, using Evolutionary Algorithms. We encode graphs using a generative model where several generators are run in sequence, each taking as input a set of parameters. We store the parameters and execution order in the genome of the EA engine and evolve it to produce graphs matching the set of target measures.

GraphCuisine's parameter encoding has several advantages for our specific goal compared to directly encoding graphs in the genome: it is efficient, both in space and computation time. One genome can potentially generate an unlimited number of similar random graphs. Changing the parameters through mutation generates more diversity than direct encoding. Diversity in turn leads to faster convergence of the evolution and can introduce interesting unexpected results still matching the constraints.

From a practical perspective, computing some of the measures is expensive in time but cannot be avoided if it plays a role in the fitness function. However, although we designed GraphCuisine for interactive manipulation, it can run as many generations as needed without human intervention, relieving the human from trials and errors to generate graphs with specific measures. Furthermore, simple strategies can be used to first generate smaller graphs with the required measures and then growing the size while controlling that the graph measures still match the requirements.

Currently, GraphCuisine implements a small set of well defined generators and measures which is easily extensible and we are in the process of investigating other generators as well as measures. One useful extension we are working on, is to consider measure distributions instead of mean values, *e. g.* for node degree. The EA would support to match distributions with little change, but the interface would require more work to effectively letting the user specify the distributions.

In our experience GraphCuisine's EA converges rapidly. However, we want to conduct more formal studies to better understand the behavior of our generators compared to existing ones in terms of convergence and expressive power. Such a formal study would also help to balance speed of convergence of the evolution against diversity in the population. In addition, we will conduct usability evaluations to assess and potentially improve the interactions in term of user control and expressive power on the generation process.

References

1. Erdös, P., Rényi, A.: On random graphs. Publicationes Mathematicae **6** (1959) 290–297
2. Waxman, B.: Routing of multipoint connections. Selected Areas in Communications, IEEE Journal on **6**(9) (1988) 1617–1622
3. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439) (1999) 509–512

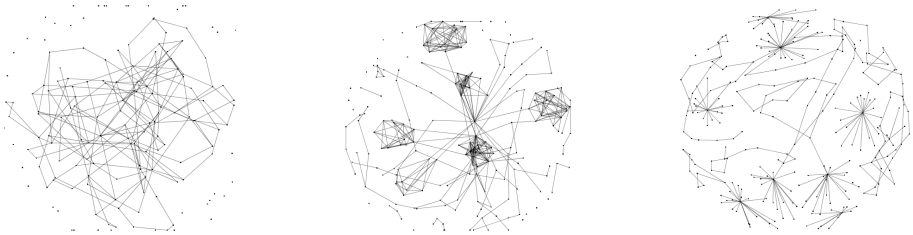
4. Medina, A., Matta, I., Byers, J.: On the origin of power laws in internet topologies. SIGCOMM Comput. Commun. Rev. **30**(2) (April 2000) 18–28
5. Aiello, W., Chung, F., Lu, L.: A random graph model for power law graphs. Experimental Mathematics **10**(1) (2001) 53–66
6. Pandurangan, G., Raghavan, P., Upfal, E.: Using pagerank to characterize web structure. In: Proceedings of the 8th Annual International Conference on Computing and Combinatorics. COCOON '02, London, UK, Springer-Verlag (2002) 330–339
7. Watts, D., Strogatz, S.: Collective dynamics of 'small-world' networks. Nature **393**(6684) (1998) 440–442
8. Kleinberg, J.: Navigation in a small world - It is easier to find short chains between points in some networks than others. Nature **406**(6798) (2000) 845–845
9. Eppstein, D., Wang, J.: A steady state model for graph power laws. 2nd International Workshop on Web Dynamics (2002)
10. Doar, M.: A better model for generating test networks. In: Global Telecommunications Conference, 1996. GLOBECOM '96. 'Communications: The Key to Global Prosperity. (1996) 86–93
11. Calvert, K., Doar, M., Zegura, E.: Modeling Internet topology. Communications Magazine, IEEE **35**(6) (1997) 160–163
12. Frank, O., Strauss, D.: Markov Graphs. Journal of the American Statistical Association **81**(395) (1986) 832–842
13. Carlson, J.M., Doyle, J.: Highly optimized tolerance: a mechanism for power laws in designed systems. Phys. Rev. E **60** (Aug 1999) 1412–1427
14. Fabrikant, A., Koutsoupias, E., Papadimitriou, C.H.: Heuristically optimized trade-offs: A new paradigm for power laws in the internet. In: Proceedings of the 29th International Colloquium on Automata, Languages and Programming. ICALP '02, London, UK, Springer-Verlag (2002) 110–122
15. Berger, N., Borgs, C., Chayes, J., D'Souza, R., Kleinberg, R.: Competition-induced preferential attachment. In: Automata, Languages and Programming, Microsoft Corp, Redmond, WA 98052 USA (2004) 208–221
16. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1989)
17. Stanley, K., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary Computation **10**(2) (2002) 99–127
18. Kitano, H.: Designing Neural Networks Using Genetic Algorithms with Graph Generation System. Complex Systems **4** (1990) 461–476
19. Gruau, F.: Neural Network Synthesis using Cellular Encoding and Genetic Algorithms. PhD thesis, Laboratoire de l'Informatique du Parallelisme, Ecole Normale Superieure de Lyon, France (1994)
20. Suchorzewski, M.: Evolving scalable and modular adaptive networks with developmental symbolic encoding. Evolutionary Intelligence **4** (2011) 145–163 10.1007/s12065-011-0057-0.
21. Wong, P.C., Foote, H., Mackey, P., Perrine, K., Chin Jr., G.: Generating graphs for visual analytics through interactive sketching. IEEE Transactions on Visualization and Computer Graphics **12**(6) (November 2006) 1386–1398
22. Albuquerque, G., Löwe, T., Magnor, M.: Synthetic generation of high-dimensional datasets. IEEE Transactions on Visualization and Computer Graphics **17**(12) (2011) 2317–2324

A Appendix: Graph Generators in GraphCuisine

Generator	Description	Parameters
Cluster	Create dense subgraphs.	Number of clusters (integer), ideal cluster size (integer), size variability (double), density (double).
Path	Sequence of nodes in which there is always an edge from one node to the next.	Number of paths (integer), ideal path length (integer), length variability (double). Star and end nodes are chosen randomly from the graph.
Cycle	Closed path. The start/end node is an existing nodes.	Number of cycles (integer), ideal cycle size (integer), size variability (double).
Star	Tree with one central node and a given number of leaves.	Number of stars (integer), ideal central node degree (integer), central node degree variability (double).
Edge Noise	Adds edges linking randomly chosen nodes.	Ideal number of edges the graph should have (integer).
Eppstein Wang Power Law Noise	Adds or removes edges according to the model by Eppstein & Wang [9].	Number of iterations (integer).
Exponential Edge Noise	Adds edges while trying to achieve an exponential degree distribution.	Ideal number of edges the graph should have (integer).
Logarithmic Edge Noise	Adds edges while trying to achieve a logarithmic degree distribution.	Ideal number of edges the graph should have (integer).
Node Noise	Adds nodes until the target count for the graph is reached.	Ideal number of nodes the graph should have (integer).
Orphan Cleanup Noise	Removes nodes with degree 0.	Percentage of orphans to be removed (double).
Random Edge Cleanup Noise	Removes randomly chosen edges.	Percentage of edges to be removed (double).
Random Node Cleanup Noise	Removes randomly chosen nodes.	Percentage of nodes to be removed (double).

Table 1. GraphCuisine's Motif (above) and noise generators (below).

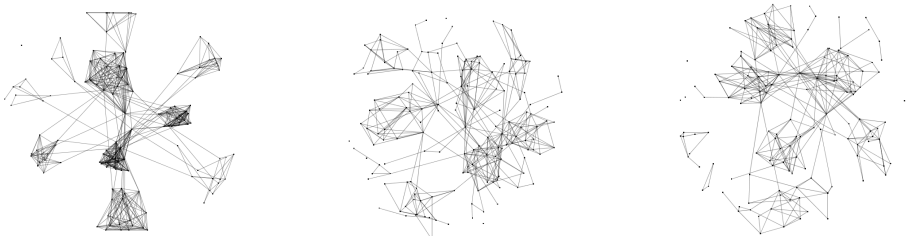
B Appendix: Examples of Generated Graphs



(a) (b) (c)

Graph	Node Count	Edge Count	Density	Diameter	Clusters	Avg Degree	Avg Clust. coefficient
(a)	161	191	0.014	9	49	2.37	0.018
(b)	213	457	0.020	15	43	4.30	0.230
(c)	283	273	0.007	11	34	1.93	0.213

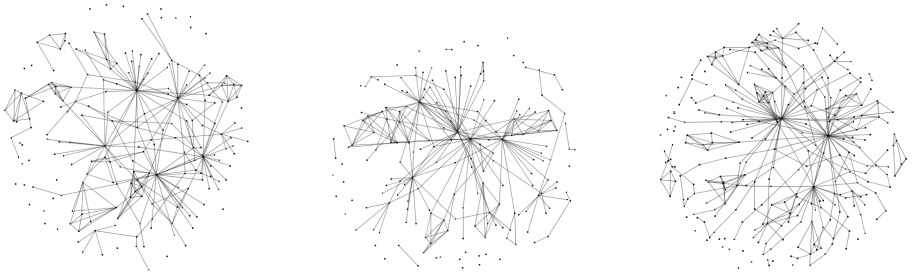
Fig. 4. Examples of Graphs from a Random Chromosome Initialization.



(a) (b) (c)

Graph	Node Count	Edge Count	Density	Diameter	Clusters	Avg Degree	Avg Clust. coefficient
(a)	140	733	0.075	7.0	11	10.471	0.621
(b)	161	379	0.029	10.0	12	4.708	0.251
(b)	146	420	0.039	7.0	10	5.753	0.273

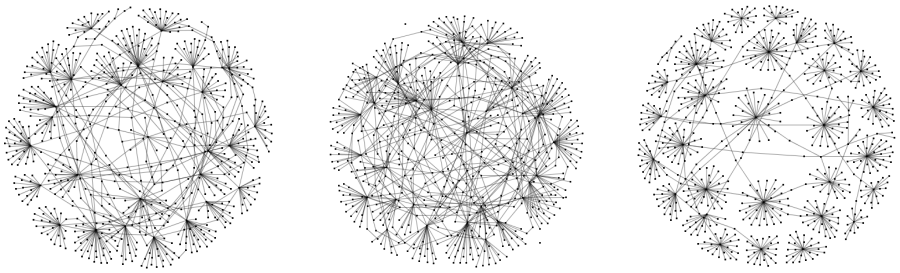
Fig. 5. Examples of small world Networks created with GraphCuisine.



(a) 212, 298, 0.133 (b) 108, 176, 0.031 (c) 266, 205, 0.006

Graph	Node Count	Edge Count	Density	Diameter	Clusters	Avg Degree	Avg Clust. coefficient
(a)	208	285	0.013	14	31	2.740	0.209
(b)	190	243	0.013	9	35	2.557	0.167
(c)	296	396	0.009	14	35	2.675	0.163

Fig. 6. Scale free Networks using, among others,the Eppstein-Wang generator.



(a) (b) (c)

Graph	Node Count	Edge Count	Density	Diameter	Clusters	Avg Degree	Avg Clust. coefficient
(a)	673	751	0.003	17.0	–	2.231	–
(b)	600	750	0.004	9.0	–	2.5	–
(c)	745	752	0.002	22.0	–	2.018	–

Fig. 7. 3 randomly picked graphs from the same population showing convergence across almost all measures after 5 generations (population size=50). Measures showing "–" have been disabled in order to increase evolution speed.