Interactive Random Graph Generation with Evolutionary Algorithms

Benjamin Bach[1], Andre 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 paper introduces an interactive system called GraphCuisine that lets users steer an Evolutionary Algorithm (EA) to create random graphs that match 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, all-in-all leading to a long iterative process that involves several steps of random graph generation, parameter changes, and visual inspection. Our system follows an approach based on interactive evolutionary computation. Fitting generator parameters to create graphs with pre-defined measures is an optimization problem, while assessing the quality of the resulting graphs often involves human subjective judgment. In this paper 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. An interactive demo of GraphCuisine can be found on our website `http://www.aviz.fr/Research/Graphcuisine`.

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. Existing generators, however, have limited flexibility, and may thus not be sufficient for formal evaluations of algorithms and layouts. It is very difficult, for instance, to use them to generate graphs that have a given amount of connected components, or with node degrees that are never above a certain threshold.

In this paper we introduce GraphCuisine, a novel interactive system that lets users steer an Evolutionary Algorithm (EA) to create random graphs that match user-specified properties. Our system provides three different approaches to the interactive generation of random graphs: (a) explicitly set desired graph measures (*e. g.,* density of 5%), (b) select "good" exemplars from a set of generated graphs for further evolution, and (c) extract measures from a sample graph while 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 each generator's 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 technique or algorithm, being able to evaluate its behavior on graphs with different topological properties (*e. g.,* density, diameter, clustering coefficient, etc.) is essential and almost impossible with current generators.
2. **Generate graphs that resemble a target graph,** not knowing the measures that contribute to its structure. Testing how appropriate an algorithm is can be difficult with only a few samples of real graphs available. Selecting the measures that best represent these samples, however, is not trivial and requires a combination of visualization, interaction and human judgment.
3. **Anonymize a specific graph** by generating a graph with similar or identical measures. Collaboration with other researchers and specialists can be impossible if a graph contains confidential information (*e. g.,* e-mail exchanges between employees, or telephone calls between possible suspects). As such, creating graphs that are as similar as possible to these in terms of their topological characteristics facilitates collaboration.

This 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 final section on conclusions and future work (Section 6).

2 Related Work

Random Graph Generation. Generators have been developed using different techniques to construct graphs with varying characteristics. *Preferential attachment* generators [11,23,3,18,1,19] add nodes to a network and connect each node with a particular probability to existing nodes, aiming at graphs with a power law node-degree distribution. *Rewiring* generators [22,17,10] reconnect existing edges, being mostly used to create small-world networks. Some of them initially place all nodes in a 2D space and consider spatial distances when calculating the probability of creating an edge between each pair [22,17]. *Structural generators* aim at obtaining more realistic graphs by first creating higher-level structures, such as clusters, and recursively modeling details [9,6].

Techniques that treat graph generation as a global *optimization problem* [13,7] or using local *forces* between nodes and edges by evolving the network [12,4] have also been explored. Their goal is to circumvent the problem of generators producing networks that favor a few characteristics while disregarding all others. As an example for optimization techniques, R-MAT [8] tries to model real graphs by recursively partitioning an adjacency matrix into cells and distributing edges within them with unequal probabilities. R-MAT features several graph characteristics and comes with an input parameter fitting function, but generates only one graph at a time and only matches a few properties.

Evolutionary Algorithms (EAs) are stochastic optimization heuristics that copy, in an abstract manner, the principles of natural evolution that let a population of individuals be adapted to its environment [14]. An EA treats potential solutions like a population of individuals that live, fight, and reproduce. Natural pressure from the environment is replaced by an abstract optimization pressure. Individuals representing the best solutions are reproduced and new solutions emerge by variation schemes known as *genetic operators*. In analogy to nature, variations are *mutation* to slightly change gene values, and *crossover* to combine genes from two parent solutions. A *fitness function*, which computes a *fitness value* to assess how good each solution is, 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 be optimized 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.

EAs have been used in the field of Neuroevolution to evolve the topology and edge weights of neural networks in their creation and learning phase. Neuroevolution is concerned with two main problems: (1) how to encode neural networks in genes, and (2) how to design effective crossover operators that do not destroy desired sub-structures in the networks. Two common approaches are used: *Direct Encoding*, which requires the design of specific genetic operators to ensure efficient exploration capabilities of the algorithm [20], and *Grammatical Encoding*, which represents the networks in a generative way [16,15,21]. Rather then encoding the graph topology in the chromosome, graph grammar based transformation rules are encoded and evolved. In GraphCuisine we take an approach similar to the latter, but different in that while Neuroevolution aims at optimizing and reproducing one single network that is in turn judged according to its functionality, we are interested in generating a wide diversity of solutions to give more options for human judgment.

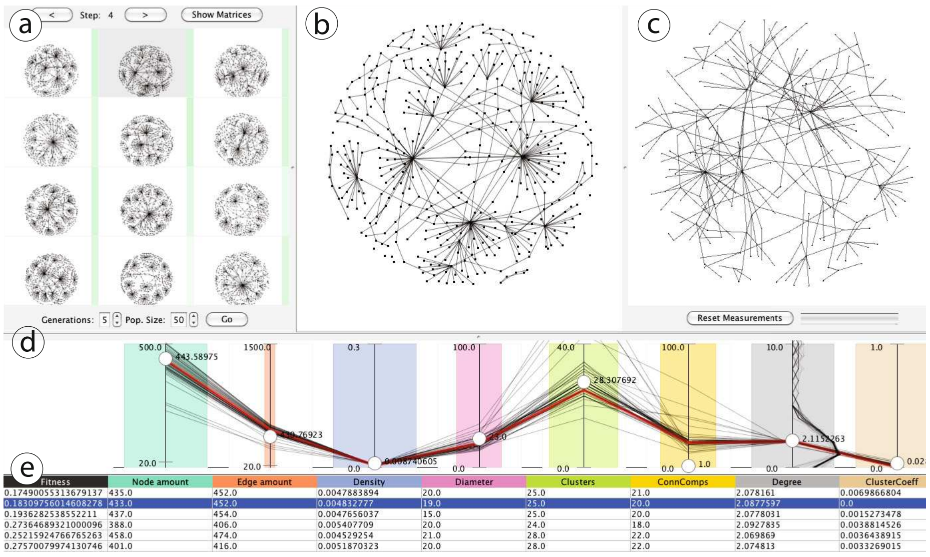
UIs for Data Generation and Analysis. Few approaches exist that tightly involve the user in the data generation process. Wong et al. [24] let users sketch graphs on adjacency matrices, adapting pixel-based drawing techniques to draw the edges. A node-link representation is provided, but only matrices are used for drawing. Except for drawing cliques, matrices make it hard to predict how a node-link representation of the final graph would look like, with its measures also being almost impossible to control. Albuquerque et al. [2] propose generators and sketching for 1D, 2D and 3D scatterplots to generate multivariate data sets. In general, such sketching interfaces do not create diverse data sets, but provide only one particular solution at a time. Biedl et al. [5] provide the user with drawing of different layouts for the same graph. The user can chose layouts he like and the system tries to interfere about the users layout preferences. In GraphCuisine the user can chose between multiple graphs, from which his preferences about graphs are interfered.

3 GraphCuisine

GraphCuisine is a random graph generator for undirected graphs by evolving a population of graphs encoded as *chromosomes*, which work as “recipes” to generate graphs. A graph is generated from a chromosome by passing a pipeline of generators, each reading its appropriate input parameters from the chromosome and modifying the graph by systematically adding or removing nodes and edges. GraphCuisine currently implements 12 graph generators, each using 2–5 input parameters and each creating basic network structures such as stars, clusters and several types of “noise”. The target graph towards which the generated graphs are evolved is encoded as a set of weighted *measures* such as node count, density and clustering coefficient. The fitness of a generated graph is calculated as the distance between the measures of the generated graph and the target measures (Section 4.2).

Encoding the genome as a set of parameters for generators has two major advantages: (1) the genome size for all graphs 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 and generators to be added easily. Having constant-size genomes facilitates crossover between chromosomes, which in turn increases the diversity of generated solutions.

The interface of GraphCuisine, seen in Fig. 1, is made up of five major parts. The *population view* (Fig. 1(a)) shows 12 representative graphs from the current population, with users being able to switch between adjacency matrix and node-link representations. Users can select graphs, with the current selection being displayed enlarged in the *detail view* (Fig. 1(b)). If users decide to extract target measures from an imported graph, the graph is shown in the *dataset view* (Fig. 1c)). The *measures view* (Fig. 1(d)) displays the distribution of measures in the whole population as an interactive parallel coordinates plot, with the polylines being the current population’s graphs, and the vertical axes representing the value range of each measure as defined by the user. Each measure axis is drawn over a color-coded rectangle whose width represents the weight of its respective measure in the fitness computation (Section 4.2). The white adjustable



Fitness	Node amount	Edge amount	Density	Diameter	Clusters	ConnComps	Degree	ClusterCoeff
0.17490055313679137	435.0	452.0	0.0047883894	20.0	25.0	21.0	2.078161	0.0069866804
0.18309756014608278	433.0	452.0	0.004832777	19.0	25.0	20.0	2.0877597	0.0
0.193628253855221	437.0	454.0	0.0047656037	15.0	25.0	20.0	2.0778031	0.0015273478
0.27364689321000096	388.0	406.0	0.005407709	20.0	24.0	18.0	2.0927835	0.0038814526
0.2521592476676563	458.0	474.0	0.004529254	21.0	28.0	22.0	2.069869	0.0036438915
0.27570079974130746	401.0	416.0	0.0051870323	20.0	28.0	22.0	2.074813	0.0033269015

Fig. 1. User Interface of GraphCuisine showing (a) the *population view*, displaying representative graphs of the current population; (b) the *detail view* of the selected graph; (c) an imported graph in the *dataset view*, (d) a parallel coordinates plot in the *measurement view* showing the distribution of measures in the population, and (e) the *measures table* with an entry for each representative graphs. In the measures view, the target measures are indicated by the position of the white markers, while black polylines line show the measures of the selected graph.

circles seen on each axis indicate the desired target values for each measure, while horizontal ticks show the desired minimal and maximal values. On the right of the gray *degree*-axis in Fig. 1(d) a vertical line chart shows the distribution of node degrees for each graph. All values can be adjusted interactively by click-and-dragging. Below the measures view, the *measures table* (Fig. 1(e)) explicitly shows the values of all measures of each representative graph. Views are coordinated with brushing-and-linking: moving the mouse over a polyline highlights the corresponding graph in the population and table views, and vice-versa.

To start generating graphs, users 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 measures can be set interactively by adapting minimal, maximal, and target values as well as the measure weights directly in the measure view.

***Graph templates*:** Templates are predefined sets of generator parameters and measures assembled to create graphs with particular characteristics such as small-sized, medium-sized or large-sized, dense or sparse, following a power law degree distribution, or being a small-world network. 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 dataset view (Fig. 1(c)). Its measures are computed, displayed in the measures view and serve as target values for the evolution. Users can modify the measures and their weights immediately or at any time during evolution; they can also be reset to the value of the loaded graph.

Generating graphs with GraphCuisine begins with the initialization and selection of the target measures, after which the evolution of generator parameters is started. An initial machine solution is created by optimizing the initial population of graphs to fit the target values. After five generations, 12 representative graphs are shown in the population view and their measures are shown in the table and the measures view. The heuristic used to select the representative graphs consists of selecting the three fittest graphs and nine at random. After this step, the measures view is updated to display the values of the entire population, and green bars appear on each graph in the population view. The saturation of the bars indicate the fitness of the graphs and is updated automatically whenever target measures are changed in the measures view so as to keep users aware of which graphs are considered similar by the system with the settings that are being used at that moment.

As an alternative to setting target measure values in the measures view manually, users can select graphs they consider *good* directly from the population view by clicking on them. Multiple graphs can be selected and the target values are updated 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 (Section 4.3). The evolution can then run for another cycle of five generations with the new target values being taken into account. The number of generations and other parameters of the EA can be adjusted by the user at any time, allowing for control of evolution behavior (population size, generations, elitism, etc.). Disabling generators prevents the graph from containing particular characteristics and setting their parameters directly gives additional freedom in specifying graphs.

Since GraphCuisine's fitness value reflects the difference between the measures of each graph and the target values, a lower value means a higher fitness. Behavior and convergence of the fitness of the whole population is monitored in the chart shown in Fig. 2(a). It shows the minimal (the best) and the average fitness in all generations using a logarithmic scale. In order to avoid previous graphs being lost during the evolution process, GraphCuisine 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.

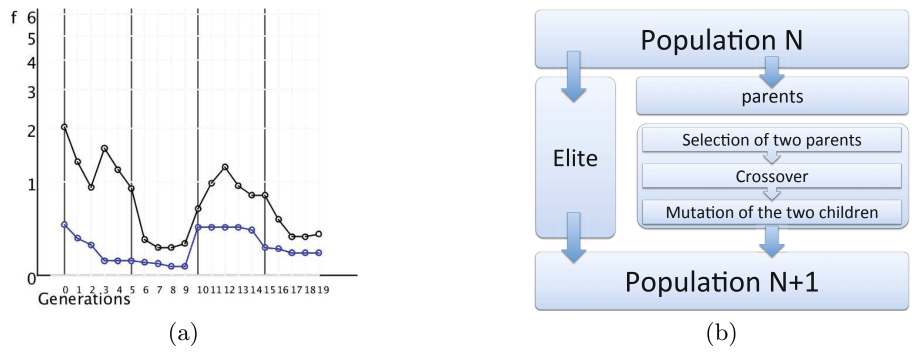
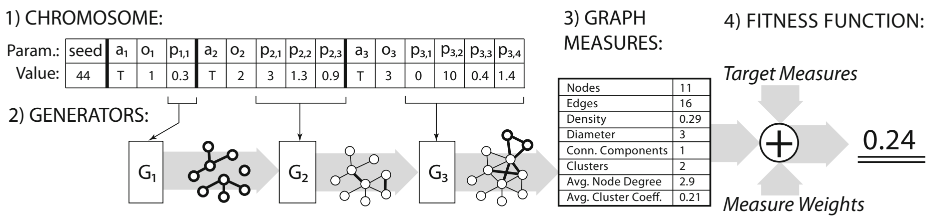


Fig. 2. (a) Fitness values on a logarithmic scale. Better fitness values are closer to 0. The blue line shows the fitness of the fittest individual, while black shows the population mean. Salient vertical lines indicate generations that have been displayed to (and possibly changed by) the user. Brusque upwards changes in the blue curve indicate that the user has changed the target measures, leading to decreased fitness of the current population. (b) Evolution process for one generation in GRAPHCUISINE.



Param.:	seed	a_1	o_1	$p_{1,1}$	a_2	o_2	$p_{2,1}$	$p_{2,2}$	$p_{2,3}$	a_3	o_3	$p_{3,1}$	$p_{3,2}$	$p_{3,3}$	$p_{3,4}$
Value:	44	T	1	0.3	T	2	3	1.3	0.9	T	3	0	10	0.4	1.4

Nodes	11
Edges	16
Density	0.29
Diameter	3
Conn. Components	1
Clusters	2
Avg. Node Degree	2.9
Avg. Cluster Coeff.	0.21

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

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 its *chromosome* 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 as illustrated in Fig. 3.

We implemented two types of generators in GRAPHCUISINE: motif generators and noise generators. *Motif generators* create topological patterns, such as clusters, stars, paths, and cycles. The parameters of these generators specify specific motif properties, such as the degree of a star's central node, and how many instances of it should be created[1]. *Noise generators* are applied to break 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

[1] For a complete list of generators and their parameters, see our `http://www.aviz.fr/Research/Graphcuisine`

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 non-connected nodes. 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, for instance, how many nodes or edges the graph should have and help control the approximate size of the graph. More specific properties exist, such as the probability of connecting two given nodes by an edge. If applied in a different order, the same noise generators with the same parameters will produce different graphs.

4.1 Chromosome Structure: Encoding of a Graph

Each chromosome is divided into three 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). The remaining genes of each generator contain their 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 amount*, *edge amount*, *density*, *graph diameter*, *number of clusters*, *number of connected components*, *average clustering coefficient*, and *average node degree*.

Some of the measures such as node amount, edge amount and density, are interdependent so that if the user restricts one measure, others are restricted automatically. It requires more investigation to provide appropriate interface components to communicate and manage such dependencies.

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 visual node link or matrix representations allow humans to compare graphs and match some of their visible structure effectively. When selecting multiple graphs S with $G_k \in S$, GraphCuisine tries to infer what graph characteristics are important to the user. The target value for each measure m_i is taken to be the average of measures of 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 $m_{i,k}$:

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

When users change the target measures or their weights, we increased the mutation rate for one generation in order to yield a larger variety of new solutions.

4.4 Implementation

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

- *A tournament selection of size five,* meaning that five individuals are randomly chosen with uniform probability, with the best one being kept as a 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 come after a randomly selected position in the genome.
- *A simple random mutation* that takes 5% of the genes of each individual and replaces them with random values.

Crossover and mutation are applied in a cascade. First new individuals are created by applying a crossover, then all new individuals undergo a mutation that alters 5% of their genes. The configuration of GraphCuisine's EA is based on the specific problem we approach and on results obtained from experimentation with the different parameters.

Performance of the generation and evaluation process is essentially limited by the computation time for the graph measures (in particular those related to clusters), with creation, mutation, and crossover times being negligible. Measure weights can be set to 0 which accelerates the process by not computing this measure.

Adding new measures or generators to GraphCuisine consists of adding the new implementations and update the user interface. Measures must be added

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

to the fitness function, while new generators require that their parameters be included in the chromosome structure. Integrating new measures into the interface amounts to adding columns to the table and dimensions to the parallel coordinates plot.

5 Example Scenario: Anonymizing a Network

Imagine an HIV contamination 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 five generations, the representative graphs are shown in the population view and the measures table. The parallel coordinates in the measures 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 *number of clusters* in the measures view. Since these two seem to be the most important measures for this evolution, the user slightly increases their weight 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 similarity between the 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, by, for example, selecting their favorite representative graphs from the population view and running another evolution cycle to obtain graphs similar to the ones they chose. 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. Fig. 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 measures view. The fitness view (Fig. 2(a)) shows the convergence of the population fitness declining, except where the target measures have been changed by the user. The pure evolution time with 50 individuals for 20 generations was 6.52 minutes.

6 Conclusion and Future Work

This article introduces GraphCuisine, a system that uses Evolutionary Algorithms to generate random graphs according to user-specified graph measures. 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 that still match the constraints.

From a practical perspective, the computation of some of the measures is expensive but cannot be avoided if they play a role in the fitness function. However, although we designed GraphCuisine for interactive manipulation, it can run for as many generations as needed without human intervention, freeing users from the trials and errors that are usually necessary to generate graphs with specific characteristics. Furthermore, simple strategies can be used to first generate smaller graphs with the required characteristics, and then grow them while assuring that their measures still match the requirements.

GraphCuisine's implementation includes a small set of well-defined generators and measures that is easily extensible, with new generators and measures being investigated and considered for future inclusion. One extension we are working on is to express and match measure distributions such as power law node degrees. The EA would require little change to support this, but the interface would need more work to effectively let 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 interaction in terms of user control and expressive power on the generation process.

References

1. Aiello, W., Chung, F., Lu, L.: A Random Graph Model for Power Law Graphs. Experimental Mathematics 10(1), 53–66 (2001)
2. Albuquerque, G., Löwe, T., Magnor, M.: Synthetic Generation of High-Dimensional Datasets. IEEE Transactions on Visualization and Computer Graphics 17(12), 2317–2324 (2011)
3. Barabási, A.L., Albert, R.: Emergence of Scaling in Random Networks. Science 286(5439), 509–512 (1999)
4. Berger, N., Borgs, C., Chayes, J.T., D'Souza, R.M., Kleinberg, R.D.: Competition-Induced Preferential Attachment. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 208–221. Springer, Heidelberg (2004)
5. Biedl, T., Marks, J., Ryall, K., Whitesides, S.: Graph Multidrawing: Finding Nice Drawings Without Defining Nice. In: Whitesides, S.H. (ed.) GD 1998. LNCS, vol. 1547, pp. 347–355. Springer, Heidelberg (1998)

6. Calvert, K., Doar, M., Zegura, E.: Modeling Internet Topology. IEEE Communications Magazine 35(6), 160–163 (1997)
7. Carlson, J.M., Doyle, J.: Highly Optimized Tolerance: A Mechanism for Power Laws in Designed Systems. Physical Review E 60, 1412–1427 (1999)
8. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: A Recursive Model for Graph Mining. In: Proc. SIAM International Conference on Data Mining (2004)
9. Doar, M.: A better Model for Generating Test Networks. In: Proc. Global Telecommunications Conference, pp. 86–93 (1996)
10. Eppstein, D., Wang, J.: A Steady State Model for Graph Power Laws. In: 2nd International Workshop on Web Dynamics (2002)
11. Erdös, P., Rényi, A.: On Random Graphs. Publicationes Mathematicae 6, 290–297 (1959)
12. Fabrikant, A., Koutsoupias, E., Papadimitriou, C.H.: Heuristically Optimized Trade-Offs: A New Paradigm for Power Laws in the Internet. In: Widmayer, P., Triguero, F., Morales, R., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) ICALP 2002. LNCS, vol. 2380, pp. 110–122. Springer, Heidelberg (2002)
13. Frank, O., Strauss, D.: Markov Graphs. Journal of the American Statistical Association 81(395), 832–842 (1986)
14. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1989)
15. Gruau, F.: Neural Network Synthesis using Cellular Encoding and Genetic Algorithms. Ph.D. thesis, Laboratoire de l'Informatique du Parallelisme, Ecole Normale Superieure de Lyon, France (1994)
16. Kitano, H.: Designing Neural Networks Using Genetic Algorithms with Graph Generation System. Complex Systems 4, 461–476 (1990)
17. Kleinberg, J.: Navigation in a Small World - It is easier to find Short Chains between Points in some Networks than Others. Nature 406(6798), 845–845 (2000)
18. Medina, A., Matta, I., Byers, J.: On the Origin of Power Laws in Internet Topologies. SIGCOMM Comput. Commun. Rev. 30(2), 18–28 (2000)
19. Pandurangan, G., Raghavan, P., Upfal, E.: Using PageRank to Characterize Web Structure. In: Ibarra, O.H., Zhang, L. (eds.) COCOON 2002. LNCS, vol. 2387, pp. 330–339. Springer, Heidelberg (2002)
20. Stanley, K., Miikkulainen, R.: Evolving Neural Networks through Augmenting Topologies. Evolutionary Computation 10(2), 99–127 (2002)
21. Suchorzewski, M.: Evolving Scalable and Modular Adaptive Networks with Developmental Symbolic Encoding. Evolutionary Intelligence 4, 145–163 (2011)
22. Watts, D., Strogatz, S.: Collective Dynamics of 'Small-World' Networks. Nature 393(6684), 440–442 (1998)
23. Waxman, B.: Routing of Multipoint Connections. Journal on Selected Areas in Communications 6(9), 1617–1622 (1988)
24. 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), 1386–1398 (2006)