Automated Design of Random Dynamic Graph Models

Aaron Scott Pope
Department of Computer Science
Missouri University of Science and Technology
Rolla, Missouri
Los Alamos National Laboratory
Los Alamos, New Mexico
aaron.pope@mst.edu

Daniel R. Tauritz
Department of Computer Science
Missouri University of Science and Technology
Rolla, Missouri
dtauritz@acm.org

Chris Rawlings
Los Alamos National Laboratory
Los Alamos, New Mexico
crawlings@lanl.gov

ABSTRACT

Dynamic graphs are an essential tool for representing a wide variety of concepts that change over time. Examples include modeling the evolution of relationships and communities in a social network or tracking the activity of users within an enterprise computer network. In the case of static graph representations, random graph models are often useful for analyzing and predicting the characteristics of a given network. Even though random dynamic graph models are a trending research topic, the field is still relatively unexplored. The selection of available models is limited and manually developing a model for a new application can be difficult and time-consuming. This work leverages hyper-heuristic techniques to automate the design of novel random dynamic graph models. A genetic programming approach is used to evolve custom heuristics that emulate the behavior of a variety of target models with high accuracy. Results are presented that illustrate the potential for the automated design of custom random dynamic graph models.

CCS CONCEPTS

• **Mathematics of computing → Random graphs**; • **Theory of computation → Dynamic graph algorithms**; • **Software and its engineering → Genetic programming**;

KEYWORDS

Random graph models, dynamic graphs, genetic programming

ACM Reference Format:
Aaron Scott Pope, Daniel R. Tauritz, and Chris Rawlings. 2019. Automated Design of Random Dynamic Graph Models. In *Genetic and Evolutionary Computation Conference Companion (GECCO '19 Companion), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3319619.3326859

1 INTRODUCTION

Graphs are a powerful and flexible method of representing a wide variety of concepts where the relationships between objects are a critical element [16]. Common applications include utility and transportation grids as well as computer and social networks. Because graphs are such a versatile way to represent and store data, countless algorithms exist that operate directly on graph structures to solve problems. Graph partitioning algorithms are used to efficiently distribute parallel computation jobs [14]. Social networks use graph-based community detection approaches to improve automated recommendations [3].

When developing new graph algorithms, researchers often turn to random graph models to test and demonstrate the flexibility and scalability of their solutions. Random graph models are also an invaluable tool for anticipating the development of a network, such as predicting the spread of a communicable disease [7]. Regardless of the specific application, the proper selection of a random graph model is critical. An inappropriate model will produce graphs that can differ dramatically from the intended target and provide an unrealistic representation. For example, a model that produces graphs that resemble transmission grids will probably be unsuitable for representing social networks.

A variety of graph similarity metrics exist that can be used to select the most appropriate model [8]. However, this approach only works if a good set of models is available a priori. A new model can be developed to suit a specific application, but manual development can be difficult and time-consuming [17]. Hyper-heuristic search techniques [4] have been used to automate the design of generative random graph models [2, 18].

Random dynamic graph models are a trending research topic, but the field is still relatively new [10, 21]. Previous methods of automating the design of random graph models are limited to static graphs by design [2, 9, 18]. This work investigates extending the use of hyper-heuristics for automated graph model generation to dynamic graph applications. Genetic programing (GP) [11] is used to evolve heuristics that capture the behavior of a variety of random dynamic graph models. Results show that evolution is able to capture the characteristics of the target models with a high degree of accuracy.

2 BACKGROUND

This section reviews a fundamental random graph model as well as a variation of this model capable of producing dynamic graphs. The applications targeted in this work build upon this extended model. Also covered is some background on automated algorithm design using a hyper-heuristic search.

GECCO '19 Companion, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6748-6/19/07…$15.00
https://doi.org/10.1145/3319619.3326859

2.1 Erdös-Rényi Model

The Erdös-Rényi graph model is one of the most basic random graph models, but it is also the most studied [5, 6]. This simple model takes two parameters: the number of vertices n and a probability p. Any possible edge between two vertices in the graph will exist with an independent probability p. The model is typically denoted as $G(n, p)$.

2.2 Dynamic Erdös-Rényi Model

Previous work introduced an extension of the Erdös-Rényi graph model that can be used to create dynamic graphs [21]. This extension adds two additional model parameters: α and β. The initial graph is created according to the static Erdös-Rényi model. At each time step, missing edges are added with probability α and existing edges are removed with probability β. If α and β are constant, the number of edges in the graph will tend towards $\binom{n}{2} * \frac{\alpha}{\alpha+\beta}$. See Zhang et. al. [21] for more detail on the characteristics of this model.

2.3 Hyper-Heuristics

Instead of attempting to solve a specific problem instance, a hyper-heuristic search aims to find a heuristic solution that can produce high-quality solutions to a class of problems [4]. This work leverages genetic programming (GP), a common hyper-heuristic technique, to evolve a population of programs that modify an input graph in a way that resembles a target random dynamic graph model. Solutions are represented using traditional Koza-style parse trees [11] with strongly-typed versions of tree construction and variation [15].

3 RELATED WORK

Automating the design of static random graph models is well studied. Bailey et. al. demonstrated that GP could be used to automate the inference of graph models for complex networks [2]. This approach was extended with increased primitive granularity to achieve more flexibility in random graph generation [18]. Harrison et. al. investigated the impact of objective selection when using GP to evolve random graph models [9]. The evolution of graph models has also been extended to directed graph applications [12]. Menezes et. al. employed a symbolic regression GP approach to select edges to add to incrementally build a network [13]. Methods other than GP have been used to automate the design of graph models, such as using simulated annealing to optimize an action-based approach to construct complex network models [1].

A key limitation of these approaches is that they are designed to generate static graph models. Many of these methods start with an empty initial graph and aren't capable of modifying an existing graph. Since most of these methods are focused on iteratively building graphs, they don't incorporate functionality to remove existing edges. This work aims to extend automated graph model design to create graph update heuristics that accurately capture dynamic graph behavior.

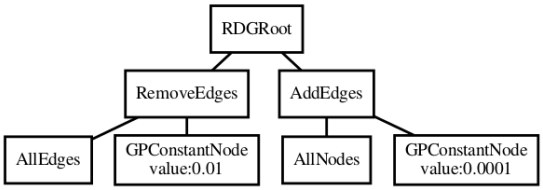


Figure 1: Example random graph heuristic parse tree that first removes 1% of existing edges, then adds new edges with probability 0.01%.

4 METHODOLOGY

This section describes the approach used in this work to evolve heuristics for the generation of dynamic random graphs.

4.1 Representation

Solutions are represented using strongly typed parse trees [15]. These trees are constructed from primitive graph-based operations that are described in Section 4.4. Solutions in the initial parent pool are randomly generated from the available operations using a ramped half-and-half approach. See Figure 1 for an example parse tree representation of a basic random graph heuristic.

4.2 Evaluation

To evaluate the quality of an evolved solution, its behavior is compared to that of a target dynamic graph model. An initial graph is constructed according to the target model and duplicated for comparison. The target model is used to update the initial graph by adding and/or removing edges. Similarly, the evolved solution is used to update the duplicate graph. The number of edges added and removed from both graphs is tracked along with the total number of edges. This process is repeated for a configurable number of time steps to produce a final graph for both the target model and the evolved solution.

The distribution of vertex degrees of the final output graphs are compared to measure the similarity between them. A two-sample Kolmogrov-Smirnov (KS) distribution comparison test is used to compare the sample distributions for both graphs. This test returns a p-value in the range $[0, 1]$ that is maximized when the samples are similar and likely to have come from the same distribution. **DC** (degree centrality) is used to refer to the p-values from this KS test comparison.

The distributions of the number of edges added and removed at each time step are compared in a similar fashion to calculate the terms **EA** (edges added) and **ER** (edges removed). The final fitness component measures how well the evolved heuristic mimics the target model with respect to the number of edges in the graph at each time step. This metric, refered to as **SD** (size difference), is defined as

$$\text{SD} = max\left(1 - \frac{1}{T}\sum_{t=1}^{T}\frac{|size(G_t) - size(H_t)|}{size(G_t)}, 0\right) \quad (1)$$

where $size(G)$ is the number of edges in graph G, T is the configurable number of time steps per evaluation, G_t is the graph produced by the target model at time step t, and H_t is the graph

produced by the evolved heuristic at time step t. The absolute difference between the size of the two graphs is normalized by the size of the target and averaged over all time steps. This value is subtracted from one to convert it to a maximization objective. Values for this objective below zero are set to zero to keep each objective on the same $[0, 1]$ scale.

To make it easier to compare evolved objective scores across applications, each objective score is scaled using the objective value achieved by comparing a model against itself using the formula

$$\Theta = 1 - \frac{|\Theta_T - \Theta_E|}{\Theta_T} \tag{2}$$

where Θ is an objective in $\{\mathbf{DC}, \mathbf{EA}, \mathbf{ER}, \mathbf{SD}\}$, Θ_T is the value for that objective achieved by the target model evaluated against itself, and Θ_E is the value for that objective achieved by the evolved model. This has the added benefit of penalizing overfit solutions that mimic the evaluation test cases better than the model fits itself.

The entire evaluation process is repeated for a configurable number of test cases to measure the robustness of the evolved graph model. Final solution fitness is defined as

$$fitness = \frac{1}{C} \sum_{i=1}^{C} \frac{\mathbf{DC} + \mathbf{EA} + \mathbf{ER} + \mathbf{SD}}{4} \tag{3}$$

where C is the configurable number of test cases per evaluation.

The four metrics previously described are used to produce a single fitness value. Alternatively, they could be used as separate objectives in a multi-objective approach. In this proof-of-concept, the single fitness value is used to assist with interpretability of the results and selecting exemplar solutions without problem-specific knowledge. Future work will leverage a multi-objective optimization approach.

During evaluation, fitness is calculated incrementally after each test case. If a solution's fitness is in the bottom quartile compared to the population after a configurable minimum number of evaluation test cases, the evaluation process is terminated early. This is done to avoid wasting expensive evaluation time on obviously inferior solutions.

4.3 Evolution

To better leverage parallel computation resources, this work employs an asynchronous evolutionary approach. In the initial phase, parent solutions are generated randomly until enough of them have been evaluated to form a starting population. Subsequent solutions are added to the population one at a time as they complete evaluation. After adding a newly evaluated solution to the population, an inverted k-tournament (selecting the lowest fitness) removes a solution from the population. Then, a new offspring is generated either by sub-tree crossover with two parents or sub-tree mutation from a single parent. Parent solutions are chosen using traditional k-tournament selection. The new offspring is then added to the asynchronous queue for evaluation. This process continues until a configurable number of evaluations have been completed.

4.4 Primitive Operations

As this work employs a strongly typed GP approach, each instance of an operation has an associated type to enforce compatibility. The available primitive types are as follows:

Boolean: returns a boolean value (true or false)
Integer: returns a whole number
Float: returns a floating point number
Probability: returns a floating point number bound to the range $[0, 1]$
Numeric: pseudo-type that refers to operations that can handle Integer, Float, or Probability types (e.g., Add)
NodeList: a collection of nodes in the input graph
EdgeList: a collection of node pairs from the input graph
List: pseudo-type that refers to operations that can handle both NodeList or EdgeList types
GraphOp: an operation that, instead of being used for a return value, alters the input graph
NodeOp: an operation that takes a node input when called and alters the input graph
Op: pseudo-type that refers to operations that can handle GraphOp, NodeOp, or EdgeOp types
Root: a special primitive type only used for the root node

Note that all references to a pseudo-type (Numeric or List, or Op) must match for an instance of a primitive operation. For example, all the List types must match for an instance of the ListIntersection operation; this primitive cannot find the intersection of a NodeList and an EdgeList.

All evolved solutions begin with a special root node primitive. This primitive has one, two, or three GraphOp children that it calls sequentially. In addition to altering the input graph through the actions of its children, this primitive also tracks and returns the edges added and removed from the input graph during execution of the parse tree. See Table 1 for a description of the rest of the primitive operation set.

The primitive set used was initially inspired by previous work evolving static random graph models [2, 9, 18]. Some operations were added specifically to ensure the primitive set was capable of capturing the behavior of the application models targeted in this work. This primitive set is fairly large, mostly due to the strongly-typed genetic programming approach used. Future work will investigate how well the heuristic search makes use of each primitive in an attempt to prune unnecessary operations.

4.5 Parameters

Table 2 lists the values of the configurable parameters used in this work. These parameter values were initially inspired by previous work evolving random graph models, but they have been hand-tuned to improve performance for this application.

5 EXPERIMENT

The Dynamic Erdös-Rényi model described in Section 2 is used to create a variety of target application models. For each target model, a population of heuristics is evolved to mimic the model's behavior. The target models are created by manipulating the α parameter. All target models use the same values for the other model parameters: $n = 1000$, $p = 0.01$, $\beta = 0.03$. The model parameter values used in these test cases are chosen to create noticeably different dynamic graph behavior while keeping the size of the graphs manageable computationally. With the exception of the final application, each of the following models was manually constructed using the available

Table 1: Primitive Operations

Primitive	Type	Inputs	Description
SequentialOp	Op	**O1**,**O2**[,**O3**]:Op	sequentially executes two or three subtrees
NoOp	Op	None	does nothing
ForNodeLoop	GraphOp	l:NodeList, **N**:NodeOp	for each node n in l, execute **N**(n)
ForIndexRange	GraphOp	i:Integer, **G**:GraphOp	execute **G** i times
CreatePath	GraphOp	l:NodeList, b:Boolean	connect subsequent nodes in l to create a path; if b, connect first and last nodes in l to create a cycle
ConnectToNodes	NodeOp	u:Node, l:NodeList, p:Probability	for every node v in l, connect u and v with chance p
AddEdges	GraphOp	l:NodeList, p:Probability	for every pairing of nodes in l, connect with chance p
RemoveEdges	GraphOp	l:EdgeList, p:Probability	for each edge in l, remove with chance p
RewireEdges	GraphOp	l:EdgeList, p:Probability	for each edge in l, rewire with chance p
AddPairwiseEdges	GraphOp	$l1, l2$:NodeList	connect node pairs at each index in lists $l1, l2$ with chance p
CreateTriangles	GraphOp	$l1, l2, l3$:NodeList, p:Probability	add edges to create triangle with nodes at each index in lists $l1, l2, l3$ with chance p
IfOp	GraphOp	b:Boolean, **G**:GraphOp	if b, execute **G**
IfElseOp	GraphOp	b:Boolean, **G**,**H**:GraphOp	if b, execute **G**, else execute **H**
Add	Numeric	x, y:Numeric	returns $x + y$
Subtract	Numeric	x, y:Numeric	returns $x - y$
Multiply	Numeric	x, y:Numeric	returns $x * y$
SafeDivide	Numeric	x, y:Numeric	returns 1 if $y = 0$, else x/y
Modulus	Numeric	x, y:Numeric	returns $x\%y$
Not	Boolean	x:Boolean	returns $\neg x$
And	Boolean	x, y:Boolean	returns $x \cap y$
Or	Boolean	x, y:Boolean	returns $x \cup y$
LessThan	Boolean	x, y:Numeric	returns $x < y$
LessThanOrEqual	Boolean	x, y:Numeric	returns $x <= y$
FloatFromInt	Float	i:Integer	returns i as a Float
ProbFromFloat	Probability	f:Float	convert f to a probability in the range $[0, 1]$
GraphAverageDegree	Float	None	returns graph average degree
AverageDegree	Float	l:NodeList	returns average degree of nodes in l
GraphMaxDegree	Integer	None	returns maximum degree of graph
MaxDegree	Integer	l:NodeList	returns maximum degree of nodes in l
GraphOrder	Integer	None	returns number of nodes in graph
GraphSize	Integer	None	returns number of edges in graph
TrueWithProb	Boolean	p:Probability	returns true with chance p, else false
NearestNeighbors	NodeList	d:Integer, l:NodeList	returns list of all nodes within d hops from nodes in l
IncidentEdges	EdgeList	l:NodeList	returns list of all edges with at least one endpoint in l
IncidentNodes	NodeList	l:EdgeList	returns a list of unique endpoints from edges in l
AllNodes	NodeList	None	returns list of all nodes in graph
AllEdges	EdgeList	None	returns list of all edges in graph
NodeListIntersection	NodeList	$l1, l2$:NodeList	returns list intersection of NodeLists $l1$ and $l2$
EdgeListIntersection	EdgeList	$l1, l2$:EdgeList	returns list intersection of EdgeLists $l1$ and $l2$
NodeListUnion	NodeList	$l1, l2$:NodeList	returns list union of NodeLists $l1$ and $l2$
EdgeListUnion	EdgeList	$l1, l2$:EdgeList	returns list union of EdgeLists $l1$ and $l2$
ListFilterWithProb	List	l:List, p:Probability	returns sublist of l randomly filtered with chance p
ListPortion	List	l:List, p:Probability	returns first $floor(p * length(l))$ elements of l
ListShuffle	List	l:List	returns elements of l in randomized order
GPConstantNode	List	None	returns an empty list
	Numeric	None	returns randomly initialized number
	Boolean	None	returns randomly initialized boolean

Table 2: Parameters

Parameter	Value
Population size	50
Evaluation limit	10000
Crossover chance	0.5
Mutation chance	0.5
Initial tree depth	2-7
Mutation tree depth	1-3
Evaluation test cases	10-30
Time steps per test case	100

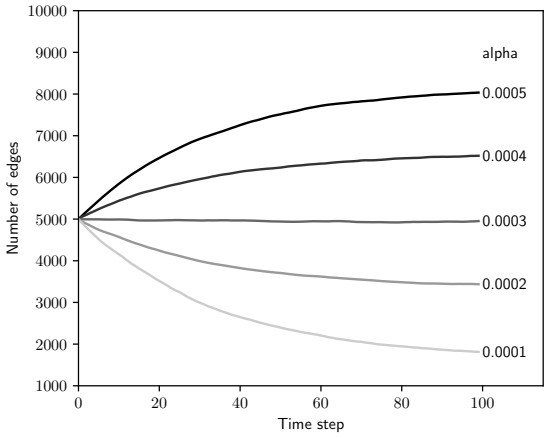


Figure 2: Graph size (number of edges) over time for various α settings. For each model, $n = 1000$, $p = 0.01$, and $\beta = 0.03$.

primitive set to ensure that the language was expressive enough to achieve the desired behavior. The final application investigates the use of this approach to model the dynamic behavior of an enterprise computer network.

5.1 Stable, Shrink, and Grow Models

The first three models use static values for the α parameter. The **stable** model uses an α of 0.0003 to create a model that adds and removes approximately the same number of edges at each time step as can be seen in the middle line in Figure 2. α values of 0.0001 and 0.0005 are used to define the **shrink** and **grow** models, respectively. These models correspond to the bottom and top lines in Figure 2.

5.2 Parameterized Model

The fourth application targets a **parameterized** version of the model during evolution. During evaluation, the evolved heuristics are tested against each of the models shown in Figure 2. To make this an achievable target, the model parameter values p, α, and β are made available to the evolved heuristics through additional terminal primitives: PInput, AlphaInput, and BetaInput, respectively. The evolutionary process must discover how to properly leverage this additional information.

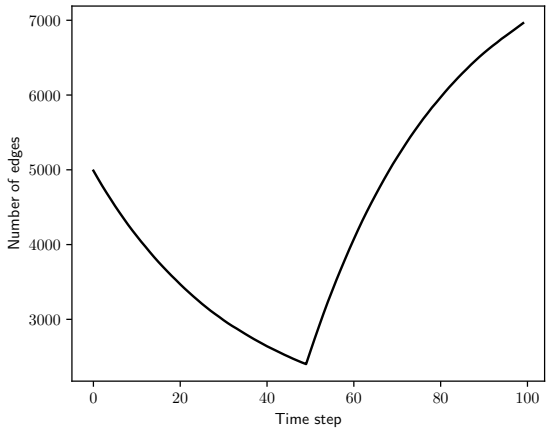


Figure 3: Average graph size over time for model with a parameter changepoint at time step 50.

5.3 Changepoint Model

The fifth application model incorporates a sudden change in the model parameters. Initially, this **changepoint** model has an α value of 0.0001. At the halfway point of each evaluation, the α parameter is set to 0.0005 and the behavior of the dynamic graph changes noticeably. This trend is illustrated in Figure 3. Five additional primitives are added to provide the flexibility to handle this application. The TimeInput and TimeInputPercentage primitives return the time step and percentage of time steps passed, respectively. ChangepointInput returns the threshold time step (50) at which the model parameters change. In this work, the changepoint is simply provided to the evolved heuristics, but automated methods of detecting this transition exist [20]. ChangepointSwitch and ChangepointSwitchElse are conditional branching primitives that determine which branches to execute based on whether or not the changepoint has been reached.

5.4 Time-dependent Model

The final manually constructed application model includes a **time-dependent** model parameter. At each time step t, the α parameter's value is updated to $0.00001 * t$. The impact this has on the size of the graph can be seen in Figure 4. This application also leverages the TimeInput and TimeInputPercentage primitives described in Section 5.3.

5.5 Modeling Enterprise Network Traffic

The final application investigates the potential of this approach to model real-world phenomenon. NetFlow event logs are taken from the computer network at Los Alamos National Laboratory (LANL) [19] that contain information about communication sessions between pairs of computers on the network, such as the ports used or the amount of data transferred. A static graph is generated for six minute increments during normal business hours (7am to 5pm) that contains an edge between two computer vertices if traffic is observed between those computers during that time window. To keep the evaluation time manageable for this proof-of-concept, the

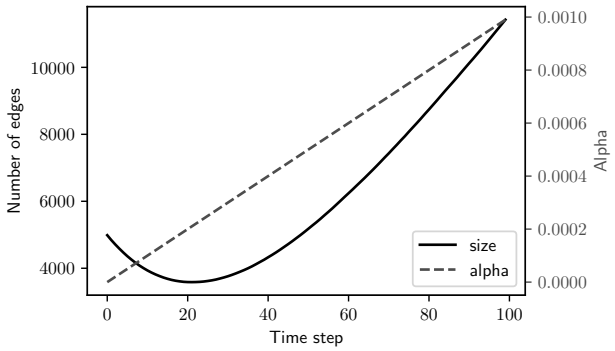


Figure 4: Average graph size over time for model with a time-dependent model parameter ($n = 1000$, $p = 0.01$, $\alpha(t) = 0.00001 * t$, $\beta = 0.03$). The dashed line indicates the value of the alpha parameter as it changes over time.

resulting graphs are reduced to activity between the most active 1000 computers. These static graphs are combined to produce a dynamic graph with 100 time steps for each of the 50 highest activity days. The down-selection in terms of days is done to remove non-business days, such as weekends and holidays, and provide a more consistent target for the evolutionary process to model. During solution evaluation, a subset of these days is chosen randomly without replacement to generate test cases. This application also leverages the TimeInput and TimeInputPercentage primitives described in Section 5.3.

6 RESULTS AND DISCUSSION

Figure 5 shows the progression of fitness values over time during an evolutionary run targeting the NetFlow application. The top gray line indicates the best fitness seen so far during a run. The shaded region shows the interquartile range of population fitness values, with the black line indicating the median fitness. Although this is an example, it exhibits features seen in the results from other runs and applications. The large vertical jumps in the best fitness make it obvious where evolution has discovered a key functionality. These are typically followed by several smaller increases. Manual inspection reveals that the large jumps typically correspond to the introduction of entirely new subtrees whereas fine tuning of constant values lead to the smaller successive increases. Figure 6 shows an example summarization of the fitness trends for 30 experimental runs, this time for the parameterized application model.

Figure 8 summarizes the breakdown of the fitness scores of the highest-fitness solutions from 30 evolutionary runs for each application. The light shaded region shows the average objective values of the heuristics evolved for that application. Since these values are scaled to the objective values achieved by evaluating the model against itself, objective values closer to one indicate more accurate models.

The objective value results suggest that the size difference (SD) objective is the easiest to optimize. Note that due to the way objective values are scaled, a score under one can be the result of the model overfiting on the test cases during evaluation. Manual

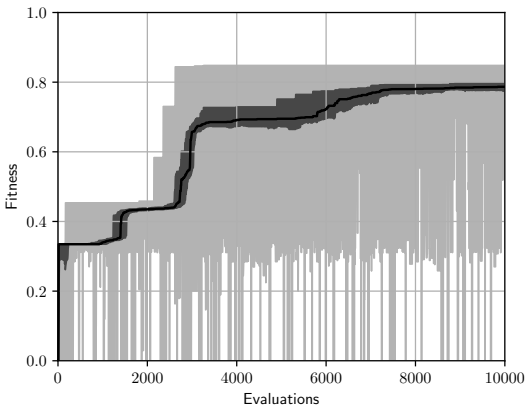


Figure 5: Fitness value over time for an example evolutionary run targeting the NetFlow application. The dark shaded region indicates the interquartile range of fitness values with the black line showing the median. The lighter shaded region shows the full range of population fitness values (minimum to maximum).

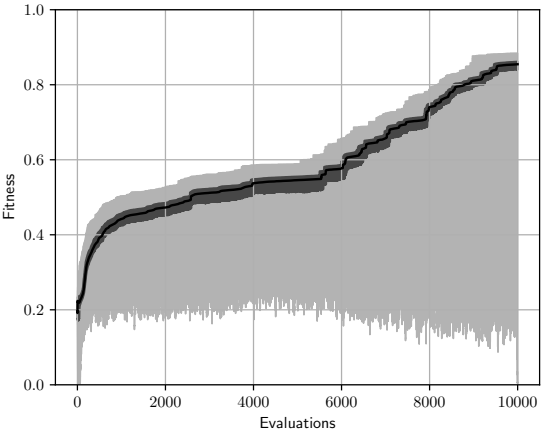


Figure 6: Population fitness values over time for the parameterized model application averaged over 30 evolutionary runs. The dark shaded region indicates the average interquartile range of fitness values with the black line showing the median. The lighter shaded region shows the average minimum to maximum range of population fitness values.

inspection reveals that overfitting occurs just as often as underfitting for each of the manually constructed target models. The NetFlow application, on the other hand, is more consistently underfit. This is likely the result of this application exhibiting far more unpredictable edge activity than the manually constructed models.

To illustrate the effect of the targeted evolution, the stable application model is also compared against each target model (except itself). The black region indicates the objective values achieved by

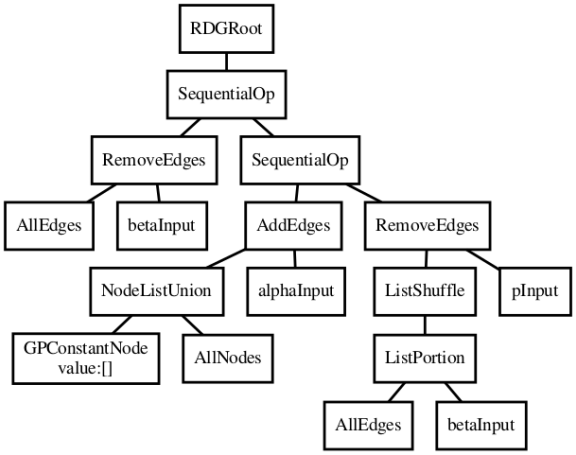


Figure 7: Example evolved parse tree targeting the parameterized application model. This parse tree has been simplified from its evolved form for clarity.

this off-target model. Unsurprisingly, the stable model does poorly at mimicking the behavior of the other application models. However, this comparison demonstrates the need for measuring the multiple objective values. If, for instance, evaluation only considered the **SD** metric, this model would still perform relatively well on multiple applications despite obviously different behavior.

See Figure 7 for an example evolved parse tree targeting the parameterized application model. To visually analyze evolved parse trees, some rudimentary tree simplification techniques were automatically performed to reduce the size and complexity of the tree without changing the functionality. Examples include replacing arithmetic subtrees that produce constants with a single constant node or pruning operation subtrees that never actually change the graph. Note that this simplification is only done to help understand the behavior of the parse tree and is never used to modify the genotype of a solution during evolution. The example tree in Figure 7 was chosen for its relative simplicity. Many of the evolved solutions are still too large to include here even after simplification. To summarize this heuristic's behavior, it removes random edges with probability β, then adds random edges with probability α, and finally removes more random edges with probability $p * \beta$.

7 FUTURE WORK

This work demonstrates a proof-of-concept for utilizing hyper-heuristics to automate the development of random dynamic graph models. A variety of application models are considered, but they are relatively simple and have some key similarities. Edges in these models are added and removed independently without consideration of larger graph characteristics, such as community structure. More complex models are available that produce graph with specific properties (e.g., scale-free networks). The system detailed in this work can be applied to target more complex models. However, the evaluation method will need to be adjusted to ensure that the evolved solutions produce the desired graph characteristics. For example, community detection can be incorporated into evaluation when targeting models that produce specific community structures.

The primitive operation set will likely also need to be expanded to address more complex models. Operations that can operate on a higher scale than single edges would be required to achieve some desired behaviors. Continuing with the community structure example, primitives could be added that group vertices into communities by examining their connectivity patterns.

There are numerous additional real-world applications this work could target. Some of these applications will likely require that this technique be converted to a multi-objective approach. When targeting a clearly defined mathematical graph model, it is easy enough to ensure that the operation set is flexible enough to perfectly recreate the model's behavior. In this case, a single objective search is sufficient because the optimal solution will simultaneously optimize all objectives. For real-world applications, on the other hand, there is no such guarantee that a single solution is best in terms of all objectives. A more flexible multi-objective search can produce a set of Pareto optimal candidate solutions from which the end-user can select the most appropriate model.

The application of this approach to certain types of real-world data could have impact on a number of research domains. Data sets modeling information that could exploited, such as activity on an internal computer network, often must be protected from release and are limited in availability. The system designed in this work could be applied to private data sets to produce accurate generative models that can be used for open research without releasing protected data.

8 CONCLUSION

Random graph models are an invaluable tool in a variety of scientific domains. However, research in the field of random dynamic graph models is still relatively undeveloped. When modeling a dynamic concept with a random graph, the appropriate model must be selected for an accurate representation. Automated model selection techniques can be leveraged to identify the best choice from a pool of candidates, but this requires a versatile set of available models which is often limited when it comes to dynamic applications. Accurate models for new applications can be manually developed, but this process can be difficult and time-consuming.

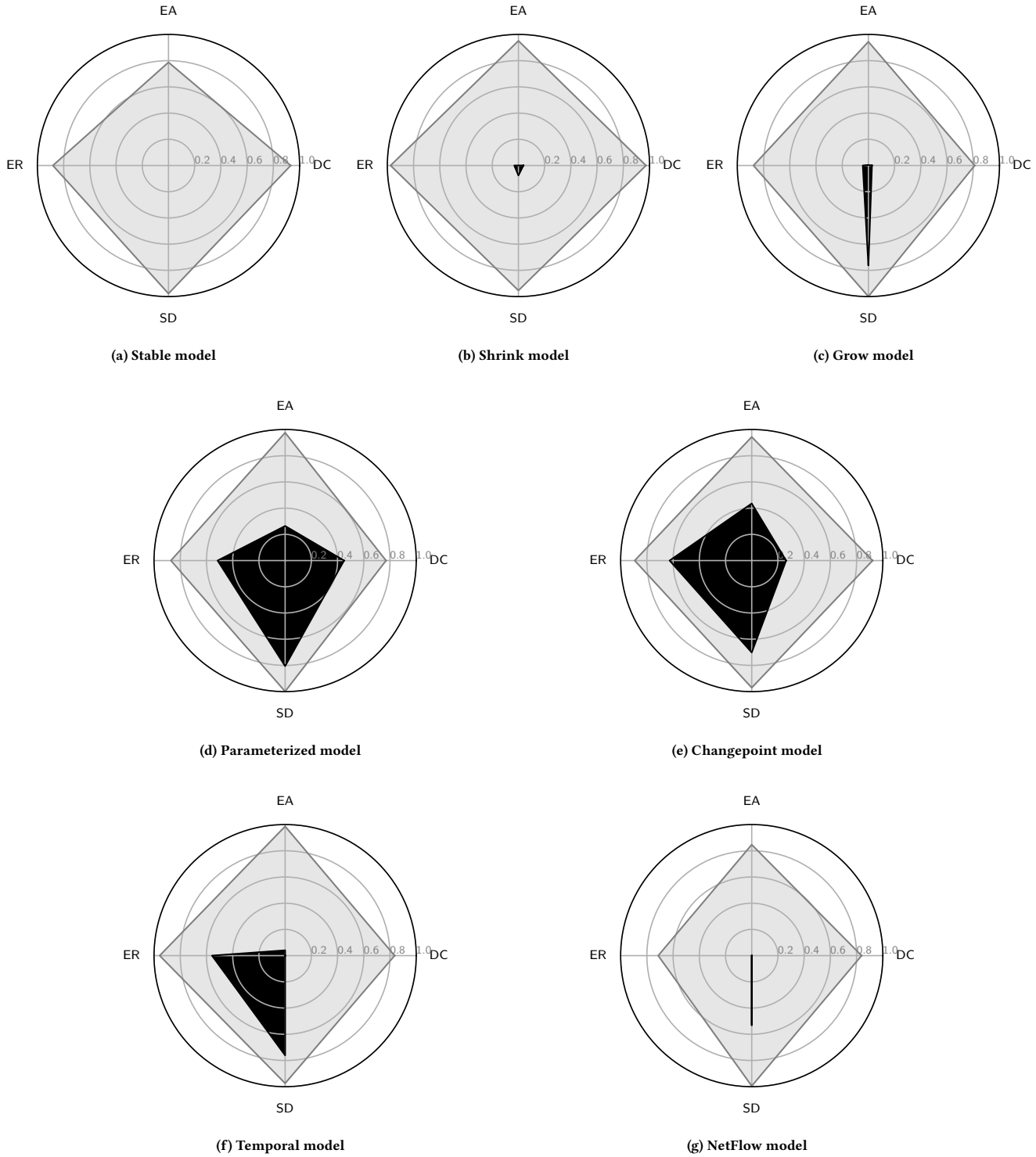
This work investigated the potential of hyper-heuristics for automating the design of generative models for random dynamic graphs. Results demonstrate that the genetic programming approach has the capability to produce algorithms that accurately recreate the behavior of a number of random dynamic test models. Also, a preliminary proof-of-concept using enterprise network traffic data demonstrates the potential for leveraging this approach to model a variety of real-world concepts.

ACKNOWLEDGMENTS

This work was supported by Los Alamos National Laboratory via the Cyber Security Sciences Institute under subcontract 259565.

REFERENCES

[1] Viplove Arora and Mario Ventresca. 2017. Action-based Modeling of Complex Networks. *Scientific Reports* 7, 1 (2017), 66–73.



(a) Stable model **(b) Shrink model** **(c) Grow model**

(d) Parameterized model **(e) Changepoint model**

(f) Temporal model **(g) NetFlow model**

Figure 8: Objective values for each application model. The light shaded region shows the objective values achieved by the heuristic evolved for that application. For comparison, the black shaded region indicates the objective values measured for the stable model against each alternate model.

[2] Alexander Bailey, Mario Ventresca, and Beatrice Ombuki-Berman. 2014. Genetic Programming for the Automatic Inference of Graph Models for Complex Networks. *IEEE Transactions on Evolutionary Computation* 18, 3 (2014), 405–419.

[3] L. Boratto, S. Carta, A. Chessa, M. Agelli, and M. L. Clemente. 2009. Group Recommendation with Automatic Identification of Users Communities. In *2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*, Vol. 3. 547–550. https://doi.org/10.1109/WI-IAT.2009.346

[4] Edmund K. Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. 2013. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society* 64, 12 (2013), 1695–1724.

[5] Paul Erdős and Alfréd Rényi. 1959. On random graphs I. *Publ. Math. Debrecen* 6 (1959), 290–297.

[6] Paul Erdős and Alfréd Rényi. 1960. On the evolution of random graphs. *Magyar Tud. Akad. Mat. Kutató Int. Közl* 5 (1960), 17–61.

[7] Stephen Eubank, Hasan Guclu, VS Anil Kumar, Madhav V Marathe, Aravind Srinivasan, Zoltan Toroczkai, and Nan Wang. 2004. Modelling Disease Outbreaks in Realistic Urban Social Networks. *Nature* 429, 6988 (2004), 180.

[8] Kyle Robert Harrison. 2014. *Network Similarity Measures and Automatic Construction of Graph Models using Genetic Programming.* Master's thesis. Brock University.

[9] Kyle Robert Harrison, Mario Ventresca, and Beatrice M. Ombuki-Berman. 2016. A meta-analysis of centrality measures for comparing and generating complex network models. *Journal of Computational Science* 17 (2016), 205 – 215. https://doi.org/10.1016/j.jocs.2015.09.011

[10] Petter Holme and Jari SaramÃďki. 2012. Temporal networks. *Physics Reports* 519, 3 (2012), 97–125. https://doi.org/10.1016/j.physrep.2012.03.001

[11] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA.

[12] M. R. Medland, K. R. Harrison, and B. M. Ombuki-Berman. 2016. Automatic Inference of Graph Models for Directed Complex Networks using Genetic Programming. In *2016 IEEE Congress on Evolutionary Computation (CEC)*. 2337–2344. https://doi.org/10.1109/CEC.2016.7744077

[13] Telmo Menezes and Camille Roth. 2014. Symbolic Regression of Generative Network Models. *Scientific reports* 4 (2014), 6284.

[14] Henning Meyerhenke. 2013. Shape optimizing load balancing for MPI-parallel adaptive numerical simulations. *Proceedings of the 10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering* (2013), 67–82.

[15] David J. Montana. 1995. Strongly Typed Genetic Programming. *Evolutionary Computation* 3, 2 (June 1995), 199–230. https://doi.org/10.1162/evco.1995.3.2.199

[16] Mark Newman. 2010. *Networks: An Introduction.* Oxford Univ. Press, New York, NY, USA.

[17] M. E. J. Newman, S. H. Strogatz, and D. J. Watts. 2001. Random graphs with arbitrary degree distributions and their applications. *Physical Review E* 64 (Aug. 2001), 026118. Issue 2. https://doi.org/10.1103/PhysRevE.64.026118

[18] Aaron S. Pope, Daniel R. Tauritz, and Alexander D. Kent. 2016. Evolving Random Graph Generators: A Case for Increased Algorithmic Primitive Granularity. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 1–8. https://doi.org/10.1109/SSCI.2016.7849929

[19] Melissa J. M. Turcotte, Alexander D. Kent, and Curtis Hash. 2018. *Unified Host and Network Data Set.* World Scientific, Chapter Chapter 1, 1–22. https://doi.org/10.1142/9781786345646_001 arXiv:https://www.worldscientific.com/doi/pdf/10.1142/9781786345646_001

[20] Yu Wang, Aniket Chakrabarti, David Sivakoff, and Srinivasan Parthasarathy. 2017. Fast Change Point Detection on Dynamic Social Networks. *CoRR* abs/1705.07325 (2017), 1–8. arXiv:1705.07325 http://arxiv.org/abs/1705.07325

[21] Xiao Zhang, Cristopher Moore, and Mark E. J. Newman. 2017. Random Graph Models for Dynamic Networks. *The European Physical Journal B* 90, 10 (18 Oct 2017), 200. https://doi.org/10.1140/epjb/e2017-80122-8