Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Dynamics of Social Networks

Author:
John Freeman

Supervisor:
Alex Carver

Submitted in partial fulfillment of the requirements for the MSc degree in Computing - Specialism of Imperial College London

September 2019

Abstract

Ever since the first online social networks were constructed, people have sought to understand and influence them for reasons as diverse as basic research and advertising. These early primitive social networks have rapidly evolved into the massive behemoths we know today, connecting almost 40% of the global population. As the networks grow in importance, so too does our desire to understand their characteristics. That growth in importance also correlates with a growth in intractability: the techniques of a decade ago couldn't possibly cope with the massive data we are immersed in today.

Seizing upon this challenge, we set out to create two systems to better understand online social networks: a dynamic network model and a dynamic node embedding system. Drawing from existing literature, we synthesize a model composed primarily of three others: the Apt-Markakis model of product diffusion, the Schelling model of segregation, and the Barabási–Albert model of preferential attachment. Using that model as a synthetic data generation tool, we develop a graph-convolutional autoencoder to embed graphs into a vector space that we go on to show is more effective than other existing models.

We demonstrate several emergent properties of our network model: weak scale-free structure, weak self-similarity, small-world characteristics, densification, and high modularity. To our knowledge such a study has not been done on Schelling networks in a dynamic context and as such we view the densification as especially interesting given it is a direct result of the Schelling process in our model.

We test our graph embedding system against two other embedding systems, one static and one dynamic. While our embedding quality was typically worse than retraining the static system at each time step, we consistently outperformed the reference dynamic embedding system on both clustering performance and stability. In terms of running time, our model tends to take longer on small datasets due to large overhead, but begins to outperform the others the larger the dataset.

Acknowledgments

None of the work accomplished here would have been possible without a long list of people I have known or been influenced by these past 22 years. I cannot hope to list all the giants on whose shoulders I stand but I would like to make some acknowledgments regardless.

To Alex Carver, for supervising this project and invaluable mentoring along the way.
To all the professors and teachers that have taught me almost all I know.
To all the researchers I cite for laying the groundwork for what I accomplish here.
To all my friends for keeping me sane and being great companions.
To my family, without whom I couldn't hope to be where I am today.

To all these people I extend my sincerest thanks.

Contents

Chapter 1

Introduction

The world has always been composed of networks. From the simplest intracellular protein networks to the most complex continent-spanning ecosystems, almost all real-world phenomena are results of smaller networks or components of larger ones. While these real-world networks have been and continue to be difficult to study due to the difficulty of collecting good data, the advent of the information age has led to the emergence of new classes of human designed and built networks that are inherently easier to study.

Foremost amongst this new class of human designed networks is the online social network, typified by the likes of Facebook or Twitter or Instagram. Within these networks we have a treasure trove of information at our fingertips: with the touch of a button we can collect megabytes of metadata ranging from favorite songs and foods to lists of friends. While the former may be of interest to those seeking to advertise or influence, we restrict ourselves to the latter, as indeed network topology is itself an expression of the preferences of its users, as shown by Centola [1].

No matter what a given online social network allows or encourages its users to share about themselves they have one thing in common: network structure. We can view each user in the network as a node: each connected by edges to those other users they interact with. The result of applying this to all the users of the network gives rise to the network's topology: the vast non-Euclidean substrate that users are embedded in. This topology is not static however: it is a swirling, ever-evolving thing whose structure and properties could change dramatically with as simple a change as the end of a friendship.

The scale of these networks is vast. At the time of writing Facebook alone has 2.3 billion registered users, each of which has an average of 338 friends. The underlying graph of Facebook then has 388.7 billion edges, an already a near-incomprehensible magnitude made worse by the knowledge that the network can only truly be understood as a time series, ever changing and ever-evolving on a scale that can only be fully captured at intervals of fractions of seconds. The task of analyzing the entirety of such a network is intractable for almost all useful tasks: even should NP be shown to be equivalent to P the amount of distributed computing power needed to even

load that data into memory is astronomical.

Given the scale of these networks it is prudent to ask what we can gain from their analysis. To this question there are a myriad of answers: from targeted advertising to influence maximization to broader questions such as understanding how human interactions give rise to complex phenomena and how it relates to the interactions of other systems. We live in a modern, data-driven world where knowledge of these phenomena and how to exploit them increasingly can have a major impact on the world at large. It is as such imperative to develop systems and methods that give us a deep understanding of networks to both meet the needs of those who seek to influence networks are good, and know when others are doing so for more nefarious reasons.

The only way we can hope to understand such networks is thus through models. We know basic properties shared throughout social networks in general ranging from how their degree distributions are structured to how their shortest paths are expected to behave. We can replicate these properties on a smaller scale through a plethora of models that have been proposed. Once these models have captured the dynamics and characteristics of their real-world counterparts, we can safely use them to create new methods of network analysis, in our case that of network embeddings.

Graph and network theoretic problems have an unfortunate tendency to reside in NP class, stemming in no small part from their non-Euclidean structure. This poses an obvious solution: mapping the networks to a vector space. Such a task is a highly non-convex optimization problem, an area that until the advent of neural networks was itself near-intractable. The process of solving that problem is today known as graph embedding, and is increasingly becoming a major part of social network analytics. After all, why waste time designing complex approximation algorithms for situation-specific tasks when you can generate an embedding and apply standard tools from statistics and machine learning?

The modern world is awash in data, more data than we could hope to analyze. In this project however, we will be creating two systems that should make that task ever so slightly easier: a dynamic model of social networks capturing dynamics over time, and a graph embedding system to translate graphs into a vector space.

Chapter 2

Background

Our project is based off of research primarily in graph theory, social network modeling, and neural networks. Here we discuss the antecedents of our work outside of those basics: specifically some specific network models and graph convolutional networks.

2.1 Networks and Models

We can describe a social network as a simple undirected graph G, composed of some edges E and some vertices or nodes V. In general, the vertices of a social network correspond to the individuals or entities that inhabit it. The edges have more variance in what they can represent, but in general and in our case denote a $friend$ or $follow$ relation, a trivial example being an edge exists between two people u and v if they are friends in the given network.

Many models have been put forward to simulate the behavior we see in real-world networks, social or otherwise. We will be combining three models in this paper to try to build a comprehensive model of a social network: the Apt-Markakis model of product adoption, the Schelling model of self-segregation, and the Barabási–Albert model of scale-freeness. Each of these three models simulates a different aspect of social networks we hope to capture: Apt-Markakis handling opinion spread, Schelling handling community formation, and Barabási–Albert keeping the network dynamic and scale-free.

2.1.1 Diffusion and Organization

Modeling static networks can readily be done through simple graph-theoretic and geometric techniques, and such techniques can generate representative static graphs easily, indeed many of them can produce graphs near exactly mimicking real-world network properties, as exemplified by Wu et al [2]. Such models do have a major

drawback however: the networks evolution over time is often not nearly as representative of the real-world as their final states.

We can overcome that issue by utilizing two properties seen in real-world graphs themselves: diffusion and self-organization. Diffusion refers to the process of information spreading across a network, and can be intuitively be understood as people sharing a piece of news until it reaches everyone in the network. Diffusion is a deeply complex process that is difficult to capture the entirety of, but it is likewise critical in understanding real network evolution. Again, we can intuitively imagine a case where a friend u sees a piece of news shared by their friend v originating from a third party w, we could expect u to be more likely to form a friendship with w as opposed to another random user.

Self-organization is more self-explanatory, referring to the process by which a structured network topology emerges as a consequence of the interactions of its users. This can take on many, many different forms, and the task of even discovering whether a network is organizing can be difficult in and of itself. That being said, organization again has an intuitive basis: we'd expect a group of real-world friends to have more friendship links within their group than outside, and a repetition of that structure across an entire network leads to what in graph theory we refer to as communities. One case of self-organization is self-segregation, which is what we will attempt to simulate. Self-segregation is simply the idea that people more prefer to be friends with those similar to them, and prefer not to be friends with those dissimilar, within some tolerances. Again, repeating this behavior across all users in a network can lead to complex emergent topologies in much the same manner.

2.1.2 Apt and Markakis's Model

The Apt-Markakis model [3] was designed to model the diffusion of products across a social network. In the simplest case, each node is initialized to have either adopted a product, or a threshold at which it will adopt one. The model then continuously checks nodes that have not adopted products to see if the fraction of its neighbors that have adopted a given product exceeds the node's threshold. If a node's threshold is met, it adopts the most common product of its neighbors. Apt and Markakis go on to show numerous results of computational complexity of various problems on their model, but the aspect of it we take advantage of in this project is its ability to easily model product, or in our case opinion, diffusion across a network.

2.1.3 Schelling's Model

A natural synergy with the Apt-Markakis model is Schelling's model of self-segregation [4]. In Schelling's model, nodes evaluate whether they are satisfied in their location or not by checking how many dissimilar neighbors they have. Should a node be dis-

satisfied, it will move to a neighborhood that is more similar to itself, again according to dissimilarity thresholds. Schelling's model has been extensively studied and has been shown to accurately characterize many more real-world behaviors than those put forwards in his paper. It is especially useful to us as it provides a mechanism to make our model dynamic, with connections being made and broken according to node opinions and tolerances.

2.1.4 Barabási and Albert's Model

A common if not ubiquitous characteristic of real-world networks is scale-freeness, a term used to describe graphs with log-linear degree distributions. Many models have been put forwards to create networks with this property, but arguably the most popular is the Barabási–Albert model [5]. In the model, we start with an initial graph and continuously add nodes to it. At each addition, we calculate the probability an edge exists between it and a given node in the graph by the proportion of its degree to the total degree of the graph. This simulates a "rich get richer" system where nodes with higher degree will have their degrees increase faster than those with lower degree, which in turn yields a scale-free network.

2.1.5 Properties of Real Networks

There are several emergent characteristics endemic to real-world networks that we hope to capture through our synthetic model, all of which have been seen in real-world networks ranging from the internet to protein interaction networks to social media: scale-free, small world [6], and self-similar structures [7].

2.1.6 Scale-Free Networks

A network is said to be scale-free if the degrees of its nodes follow a power law: effectively meaning that few nodes have many connections, and many nodes have few connections. While this property seems at first glance inconsequential, this property is thought to characterize many real-world networks, and as such is an important property to capture if our model is to be representative.

2.1.7 Small-World Networks

Closely related to scale-free networks, small-world networks have a less well accepted definition, but common metrics include diameter increasing at most logarithmically with network size, or higher clustering but equivalent average path length compared to random networks. Effectively, what those metrics capture is classic idea of six degrees of seperation, that all people on Earth are at most 6 friends away from

each other. As the precise definition of small-world networks varies, and measuring large real-world networks for such metrics as average path length and diameter are computationally intensive, the classification of real-world networks as small-world varies by person to person, but it is generally accepted that most social networks fall into this category [8].

2.1.8 Self-Similar Networks

A less well known and studied property of networks is self-similarity. As the name suggests, self-similarity is the network equivalent of fractal structure in geometric space. As with fractals, a network is self-similar if it is length-scale invariant, meaning as in fractals, the structure of the network is the same no matter how zoomed in you look at it.

While zooming in on a geometric structure is of course trivial, the generalization to networks is not as difficult as one might imagine, and is done through a process known as renormalization [7]. Renormalization is similar to the standard box-counting method, where the network is segmented into boxes corresponding to the neighborhoods of randomly selected nodes with a predetermined radius, and then collapsed down so that the boxes are nodes. It has been shown that for many networks, including the Internet, properties of the graphs stay the same after renormalization, most notably in the case of the Internet, the degree distribution remains scale-free.

2.2 Machine Learning on Graphs

Traditionally, graph theoretic problems have been addressed through mathematical and computer scientific approaches, with complexity classes being extensively studied and approximation algorithms being created to deal with the NP-completeness that is pervasive in graph-based problems. While such approaches are generally successful, modern data-driven methodologies have become increasingly common in the analysis of graphs, especially on large real-world datasets.

Machine learning type algorithms have arrived relatively late to graph problems, in a large part due to the dissimilarity of the data to other approaches. Most algorithms rely on input vectors of numerical data and have few ways of efficiently representing topological information of the graphs themselves, often forcing them to resort to using node attributes augmented by simple human made features. Worse still is the case of unattributed nodes where the topology of the network is all that can be learned from, where algorithms are reduced to considering adjacency information alone. While attribute information is no doubt useful, the topology of the graph often contains more useful information, especially for graph theoretic problems.

2.2.1 Graph Embedding

A common way of applying machine learning techniques to graphs is to create a graph embedding, that is, mapping the nodes of a graph into a vector space. Many techniques exist for this: initially spectral methods based off adjacency matrices and other information, and later based off random walks and eventually deep learning methods on the graph itself. The key in graph embeddings is to find a representation vector for a given node such that the structural and attribute properties of the graph are preserved in the embedding space.

The problem of graph embeddings has been extensively studied in the case of static, unchanging graphs [9]. In our project however, we hope to capture information about social networks, which are constantly evolving and poorly understood by static snapshots. As such, we need an embedding method that will work for graphs that change over time. This is a new research area, although there has been work in online spectral methods [10], as well as dynamic autoencoders [11], among others. The important difference between dynamic and static embeddings is stability: we could easily relearn embeddings at each time step but they would very probably be completely different solutions to the problem. Both the configuration space of the network and the embedding space are infinite and as such the amount of possible good embeddings is vast. Stability ensures that from one time step to the next the embeddings of nodes that haven't been significantly changed will remain more or less the same, while those that have been changed will move.

The same problem that drives the need for stability also drives the complexity of generating embeddings. Any solution will in all likelihood be highly non-trivial and non-linear with a vast, high-dimensional energy space. While spectral techniques have been used and perform passably, the problem is almost tailor-made for neural networks which address all those issues with relative ease. Neural networks have already become dominant in image processing, and graphs are little more than generalizations of images to a non-regular domain. Indeed, this analogy can be furthered: if a graph is simply a generalized image, then a dynamic network is nothing more than a video. As neural networks perform well on both, they will be what we use to generate our embeddings.

2.2.2 Neural Networks

While neural networks have been around in some form or another since the 1940s [12], difficulties in training and the conceptual issues like XOR problem led to them being understudied compared to other, more easily optimized and understood methods. A combination of research into deeper networks, convolutional layers, and GPU acceleration led to a massive increase in usage of neural networks, which are today the best performing models in most every field of data analysis. While their performance is impressive, they remain limited in their black-box design, it is in general almost impossible to understand what networks have learned, and even inspecting

their energy landscapes is a monumental task [13].

Unsupervised Networks and Autoencoders

Neural networks have a reputation of being data-hungry models, and indeed for most of their use cases like image recognition the reputation is deserved. For our use however, we will be restricting ourselves to the unsupervised domain, an area where far less data is required, although at some expense to generalizability. Unsupervised networks can be constructed in such a way that they only need to learn underlying features of the data itself and not make any predictions about it, a comparable task being clustering in the vector case. The standard architecture for unsupervised learning with neural networks is the autoencoder.

An autoencoder is a class of neural network designed around finding an encoding for a given input. Generally symmetric about an embedding layer significantly smaller than the input dimension, an autoencoder tries to optimize similarity of the reconstructed input to the true input to find a low dimensional encoding of the input from which all the inputs characteristics can be reconstructed. While numerous variations exist, most architectures follow this pattern with no more than a few changes. Autoencoders have recently become more popular with finding embeddings for graphs and are generally used to encode first and second order proximities of nodes. Thanks to the ease with which neural networks can dynamically self-scale themselves to larger inputs, autoencoders are one of the easiest methods to generalize to dynamic graphs.

Convolution on Graphs

Arguably the most significant standard operation in neural networks is the convolution operator. Originally conceived from signal analysis, the convolution operator allows a network to consider the local context of a data point, learning a filter of the data and its neighbors. The convolution operator is easy to use for images, where the neighbors of a pixel always have the same shape as an image has a standard structure. On graphs however, we face more issues stemming from the varying degrees of its nodes.

Many ways to generalize convolution to graphs have been proposed [14], in the simplest cases based on $out = \sigma(AXW)$, where σ is an arbitrary activation function, A is the graph's adjacency matrix, X is the graph's node-wise attribute matrix, and W is a learnable weight matrix. Many other generalizations exist, ranging from normalized versions of the equation above to variants using spectral information to kernel models featuring a myriad of distributions.

Luckily for us, the PyTorch Geometric [15] has recently been released and gained significant popularity. PyG implements a wide variety of graph operations while

seamlessly interfacing with PyTorch and achieving execution times we couldn't hope to match through heavy integration with CUDA. PyG will be integral to the success of our project, as graph convolution will yield significant speedup compared to existing models and should also yield significant increases in model quality.

Chebyshev Convolutions

The graph convolution we selected for the project was proposed by Defferrard et al [16], henceforth referred to as Chebyshev convolutions. Their method allows for fast, powerful, and localized graph convolutions and is one of the most used convolutional methods in existence, albeit one of the more complicated ones.

The Chebyshev convolution avoids the issue of defining neighborhoods of nodes by working in the spectral domain as opposed to spatial. This does however bring another challenge, as spectral translations are not inherently localized, and the computation of the spectrum is expensive. Chebyshev convolution overcomes both the former by introducing an extra Kronecker delta convolution against the filter, and the later by utilizing the kth Chebyshev polynomial for nodes k steps away from the current node, and taking advantage of a natural recursivity to reduce time complexity to manageable levels.

2.2.3 Random Walks

Arguably the most successful method in graph embeddings, and the one that catalyzed the field as a whole, is random walks. Drawing inspiration from language models, random walk-based methods such as DeepWalk [17] run a random walk at a given node and run it against a typical skip-gram model or other language embedding framework. The random walk yields a sequence of nodes that appear in the "context" of a given node, closely mirroring words appearing in the context of another in human language. This similarity can be exploited by applying well-studied natural language word embedding algorithms to the problem of graphs. DeepWalk was able to achieve massive improvements over existing spectral and other human-designed feature based methods, and serves as an inspiration for what we intend to use for our model.

Even outside the language analogue, random walks are a highly useful method of gathering information about graph topology while still introducing randomness. An often-used design of autoencoder is the stacked denoising autoencoder [18], which achieves better encodings by introducing noise to each layer and having them reconstruct the uncorrupted input. Random walks, through their inherent randomness, could serve as a method of introducing noise to autoencoders by using different walks per node at each training epoch, while still conveying topological information about the nodes greater neighborhoods.

2.3 Related Work

As referenced above, our main inspirations are drawn from existing systems that we hope to extend to the dynamic case. Deepwalk and other random walk based algorithms provided good baselines but were limited by not being able to capture the global structure of graphs and not being easily able to handle structure changes. Neural network architectures perform well on spectral and adjacency information and can be extended to the dynamic case, but still lack the whole context of the graph.

Graph convolutional neural networks solve most of the above problems, at the cost of complexity: they need to operate on the entire graph at once rather than individual nodes. They are however able to achieve significant performance increases over the local methods, and while they have not yet been tested significantly in dynamic graphs, we believe that they should be able to handle them as standard convolutions are ubiquitous for speech processing and other time-series data.

An embedding system similar to the one we propose has already been implemented by Ying et al [19], where they combine both random walks and graph convolutions to embed huge graphs. Although we will not have access to the computational power they had, we hope to build off it by applying a similar framework to dynamic graphs, while also exploring performance against several types of data.

We also found three systems developed for dynamic network encoding: Goyal et al's DynGEM [11], Ma et al's DGNN [20], and Trivedi et al's DyREP [21]. DynGEM uses a standard fully connected autoencoder architecture on the adjacency matrix, and is the only model we could find a standard implementation for and as such is our main comparison point for our model. DGNN uses a modified LSTM architecture that explicitly determines which nodes are directly and indirectly affected by any network change, and propagates changes as such. DyREP is interesting in that it learns a set of functions directly to transform the data and explicitly models the time a change occurs. This has another interesting application in that they can then predict with surprising accuracy how long it will be before another change occurs in the network.

On the network model side of things, we found few dynamic models that explicitly seek to model social networks rather than arbitrary properties. Skyrms and Pemantle [22] propose a model of agents interacting through games and show emergence of structure through those interactions but do not subject the resulting models to rigorous analysis. Leskovec et al [23] propose a variant of the forest fire model that they show does exhibit numerous properties of real networks, but exists in a relatively confined paradigm, not extendable beyond a few natural changes.

Chapter 3

Contribution

Our contribution is twofold. First, we design and and analyze a synthetic model of social networks that is able to capture several emergent properties at once; including network growth, scale-freeness, small-worldedness, and self-similarity. Second, we construct a graph convolutional autoencoder to embed the nodes of dynamic social networks into a vector space in both effectively and quickly.

3.0.1 Network Model Goals

Given that we will be using the model to generate synthetic data to test our embedding system on, it is critical that the model create networks as similar to real-world networks as possible. To this end, we identify three main properties we expect our networks to have: scale-freeness, small-worldness, and self-similarity. Verification of these objectives is fairly straightforward as described in the background section, and primarily comes down to inspecting degree distributions as well as some other standard graph metrics.

3.0.2 Embedding System Goals

The embedding system itself is more open-ended in construction and as such our goals are more loose. There are however a few metrics we will be checking our results against, including K-means clustering performance on embeddings compared to the Louvain algorithm [24] on the graph itself, as well as embedding stability, for example ensuring that the embeddings don't change too much from timestep $t-1$ to t.

Unfortunately we were only able to find one example of a similar dynamic embedding system with a standard implementation: Goyal et al's DynGEM [11]. As such, that will be our main source of comparison in terms of evaluating the model's performance. We will also evaluate against static embedding methods like Node2Vec [25] retrained at each timestep, which will be especially useful in measuring embedding

stability.

A final aspect of the embedding system we want to work towards is running time: especially in the real-world datasets are large and complex, and many methods can be cost-prohibitive in running time. This is one area that we believe the convolutional architecture should yield large improvements, and as such we hope to achieve comparable results to existing methods much faster.

3.1 Network Model Design

The model is a combination of the of Apt-Markakis, Schelling, and Barabási–Albert models, with a major difference being that instead of a fixed set of available products, we use a real number as a product, to try to encourage more dynamic behavior, as well as better relate to the real-world. The model is composed of two parts, an initialization and a loop over the Apt-Markakis, Schelling, and Barabási–Albert components.

3.1.1 Model Initialization

The model is initialized to a random graph with some nodes and edges, and each node is assigned two random values: an Apt-Markakis product, or lack thereof, and a threshold value, corresponding to the fraction of neighbors that must adopt products before the node itself does. There are also two important global parameters: a similarity threshold sim_thresh corresponding to how far apart two nodes opinions can be for them to be considered different, and a neighbor threshold $neighbor\_thresh$, which corresponds to the fraction of neighbors that can be different for a node to be unhappy in the Schelling step. A final parameter is the distribution used for selecting random Apt-Markakis products on new nodes, typically a uniform or beta distribution.

3.1.2 Barabási–Albert Step

The Barabási–Albert step is the simplest of the three steps, and is little changed from the original model. The step begins by computing the total number of edges in the graph and creates a probability distribution over the nodes corresponding to their degree divided by the twice total number of edges, via the handshaking lemma. A new node with a random product is created, and then a random value is sampled from a uniform distribution for each node in the graph. If that value is less than the node's share of total edges, an edge is added between it and the new node, and if not, no edge is added.

Algorithm 1 Barabási–Albert Step

function BA_STEP(G)
  $E \leftarrow G.num\_edges$
  $u \leftarrow$ a newly initialized node with no edges
  add u to G
  for v in G.nodes **do**
    $x \leftarrow$ random sample from $\mathcal{U}(0, 1)$
    $y \leftarrow v.degree$ / $(2E)$
    if $x < y$ **then**
      add edge between u and v
  return G

3.1.3 Apt-Markakis Step

The Apt-Markakis step takes a bit more liberty from the original model, primarily due to the continuous valued products, which we fix as pulls from a beta distribution and as such are in $(0, 1)$. It begins by creating a list of all nodes who have not adopted a product but have enough neighbors who have for their threshold to be met. It then selects one of those at random, and creates a list of the products of its neighbors. It then divides these into buckets, and selects a bucket randomly weighted towards buckets with more values. The sum of that bucket and a random gaussian value is then the target's product.

Algorithm 2 Apt-Markakis Step

function AM_STEP(G)
  $tar \leftarrow$ random node in G that can adopt a product
  $neighbors \leftarrow$ the products of all nodes with edges incident to tar
  $buckets \leftarrow$ list of 0s of length $n\_buckets$
  for u in $neighbors$ **do**
    $idx \leftarrow$ floor($u.product * n\_buckets$)
    $bucket[idx] += 1/len(neighbors)$
  $prod \leftarrow$ random sample from $buckets$
  $tar.product \leftarrow prod/n\_buckets + \mathcal{N}(0, 1) * jitter$
  return G

3.1.4 Schelling Step

The final part of the loop is the Schelling step. To begin the Schelling step, we select a random unhappy node, where a node is unhappy if the fraction of its neighbors whose products differ by more than sim_thresh against the total number of neighbors is greater than $neighbor\_thresh$. Next, we create the same distribution as we did in the Barabási–Albert step. We then loop until the node is happy, at each step

selecting a node according to that probability distribution and breaking the connection if it exists and the nodes are dissimilar, or creating one if it does not and they are similar. An extra check is to ensure that broken edges are not bridges, as we want to keep the graph connected. This can lead to some strange behavior especially in small graphs, but the effects seem to balance out as time goes on.

Algorithm 3 Schelling Step

```
function S_STEP(G)
    tar ← random unhappy node in G
    E ← G.num_edges
    dist ← [u.degree/(2E) for u in G.nodes]
    bs ← bridges(G)
    counter = 0
    while unhappy(tar) do
        counter += 1
        u ← random sample from dist
        if similar(tar, u) and not G.has_edge(tar, u) then
            add edge between tar and u
        if not similar(tar, u) and G.has_edge(tar, u) and (tar, u) not in bs then
            remove edge between tar and u
            bs ← bridges(G)
        if counter >= G.num_edges then
            break
    return G
```

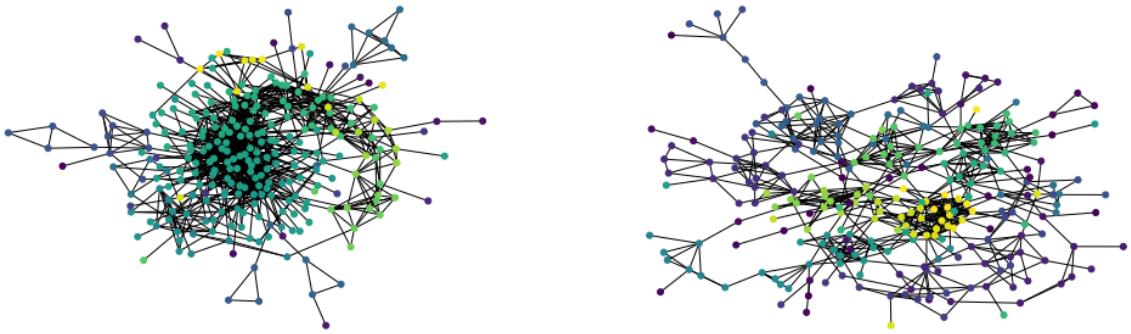
3.1.5 Implementation

We implemented our network with NetworkX, and experimented with several different methods to solve various issues in developing the network model. We had to make numerous deviations from the initial plan, most of which stemming from either prohibitive time complexities, or an issue we term saturation, wherein the amount of edges in the model increases faster than we'd expect in a real-world network.

Edge Saturation

By far the largest issue we encountered was that of edge saturation, a term we use to describe the number of edges in the model exceeding what we'd expect to see in the real world, especially within clusters. This seems to be a consequence of extending Schelling's model to graphs, as it was originally designed to work on regular Euclidean domains. The issue seems to lie in the fact that in our implementation of Schelling's model, an unhappy node will always add many more connections than

it breaks, simply by virtue of the graph being relatively sparse. This trend continues throughout the iteration cycle, and eventually causes the graph to saturate with edges, thus merging and erasing emergent communities.



(a): Saturated Graph **(b):** Fixed Graph

Figure 3.1: Generated graphs before and after addressing the saturation issue, 250 iterations each, colored by Apt-Markakis product

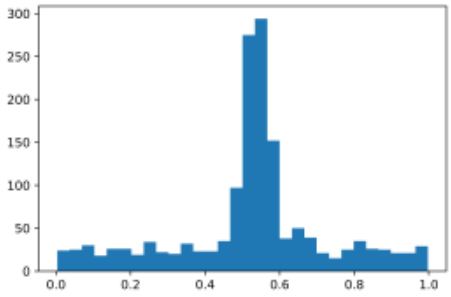
We experimented with several ways of solving this, the most obvious being putting limits on the number of edges a node can have. While that did solve the issue, it had numerous flaws, the most obvious of which being it's contrived and not representative of anything in the real world, and it also prevented the graph from being scale-free, as no nodes could exceed an arbitrary limit in degree. The next thing we tried was logarithmically decreasing the sim_thresh value, which worked quite a bit better and resulted in better community differentiation, but again felt like a contrived method and not at all natural.

We eventually settled on a combination of fixes, and while they don't perfectly solve the problem, they do have the advantage of not involving arbitrary parameter changes or caps. We changed the Apt-Markakis sampling distribution to $Beta(0.75, 0.75)$ from $\mathcal{U}(0, 1)$, introduced a bucketing system for the Apt-Markakis diffusion process, and also introduced Gaussian noise each time a node adopts a product. We settled on these changes by noticing that in our Apt-Markakis step, node products tended to skew towards the middle, resulting in many nodes with similar products and thus vastly increasing the amount of nodes that can have edges added in the Schelling step.

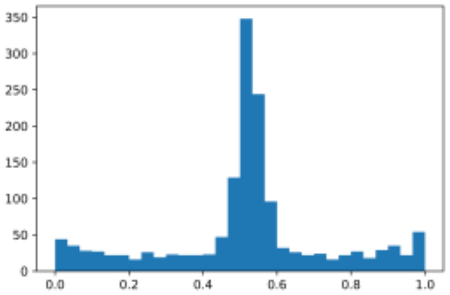
The main change we made was the introduction of a bucket system for the product adoption. We do this by working probabilistically through dividing neighbor opinions into a histogram, and then sampling a new opinion from that histogram based on the amount of nodes falling into its buckets. This introduced a degree of randomness that seems to address the saturation issue, and combined with the Beta distribution increases the average clustering while maintaining a similar modularity.

The choice of the beta distribution stemmed from the same issue that drove the bucket system, the nodes products skewing towards the center. Before we implemented the bucket system, we experimented with using a beta distribution to sample products from, and noticed that while it seemed to help a bit, it still had the same skew problem. The $Beta(0.75, 0.75)$ distribution adopts a U shape that pushes weight towards the upper and lower ends of the range, meaning new nodes are more likely to have products closer to 0 or 1, and less likely to have products in the middle of that range. While this problem was better solved by using the bucketing system, we did notice that using the beta distribution and the bucket system together led to a higher average clustering of roughly 0.05 as well as a close to uniform distribution of the final products. Histograms of products with the various solutions can be seen in figure 3.2.

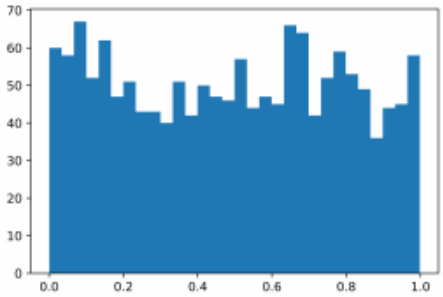
Unfortunately neither of these methods completely solved the issue, as the network still seems to saturate after more than a few thousand iterations. At only a few hundred iterations however, communities are very clearly separated as can be seen in figure 3.1. Saturation at larger amounts of iterations may well be a fundamental limit to the combination of models: at the very least no obvious solution presents itself beyond introducing still more models, which we hope to avoid.



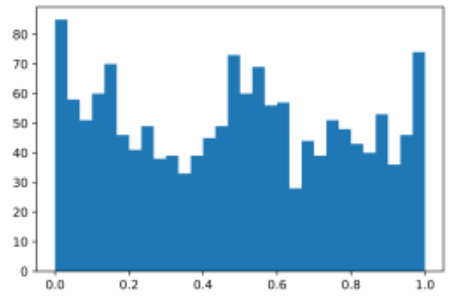
(a): Model with Uniform Distribution



(b): Model with Beta Distribution



(c): Model with Buckets



(d): Model with Buckets and Beta Distribution

Figure 3.2: Histograms of Apt-Markakis products

Time Complexity

Part of the original plan was to have the Apt-Markakis products be vectors instead of scalars, but we quickly found that extending the model to the vector case was non-trivial, and in the implementations we tried time complexity soared for minimal changes in how the model evolved, and if anything increased the number of edges far more than the scalar model.

The obvious way to extend the Apt-Markakis model to vectors of products is to assume each index of the vector corresponds to a given category of product, and move on as normal. This effectively replicates the Apt-Markakis diffusion process n times, where n is the length of the vector. This is the method we implemented, where for each node with an unset product at index i in its vector, the model checks if the neighbor threshold is met for its neighbors products at i. This then goes on to the Schelling step, where the similarity threshold becomes the L1 or L2 distance between the nodes Apt-Markakis vectors.

While this has the obvious effect of adding an additional scaling factor linear in the dimension of the Apt-Markakis vector length, but also increased the number of edges, as it seems this makes it harder for the Schelling thresholds to be met, which in turn increases model running time yet again. In all, the vector products seemed to hurt more than they helped both in terms of model quality and running time, and as such we settled on keeping the model scalar valued.

3.2 Embedding System Design

As with most neural networks, our embedding system is the result of much experimentation and trial and error. Our final autoencoder model makes use of a combination of a myriad of techniques, including Chebyshev convolutional layers, standard convolutions, batch normalization, noise layers, and more. We also augment input data with truncated random walks, and use reconstruction as well embedding losses.

3.2.1 Random Walk Augmentation

The input vector for a node i in the network is initially just the ith row of the adjacency matrix of the network. Drawing inspiration from DeepWalk [17], one of the more successful techniques we used was to augment that input with truncated random walks. Our method for this is simple enough: for each node in the graph we do a random walk with restart probability 0.1, and then treat all the nodes in the random walk as neighbors for the input. We redo the random walks at each training iteration, to both introduce noise and to get more information on context. We use a heuristic to determine walk length, $l = density * |V|$, as it seems to perform favorably compared to other methods. We define a restart probability of 0.1 to try to prevent

the walks from getting too far away from their origin, a problem that becomes more pronounced the larger and denser the graphs are.

3.2.2 Autoencoder Architecture

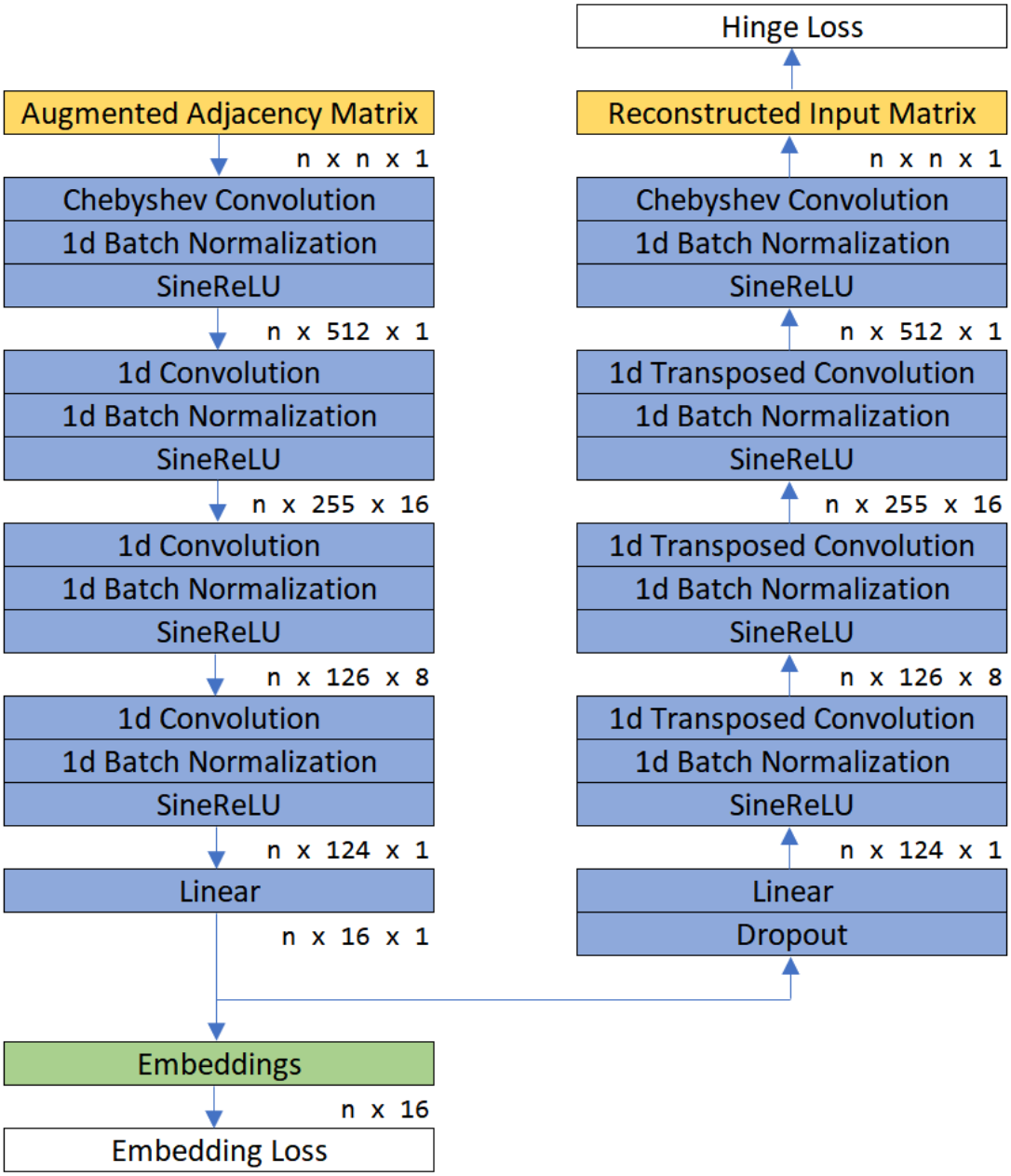


Figure 3.3: Architecture diagram of the autoencoder

Our network is a standard convolutional autoencoder architecture, consisting of a Chebyshev convolutional layer, 3 standard convolutional layers, and a linear layer, interspersed with batch normalization layers and sineReLU activations on both the

decoder and encoder sides, the only difference between the two being the decoder begins with a dropout layer. The network has two parameters: the mid layer size and hidden dimension size, set to 512 and 16 respectively in figure 3.3.

3.2.3 Network Training

Since our network is meant to work on dynamic networks, our training loop is nested, the outer looping over the networks from the first graph in the series to the last, and the inner being a standard training loop. We found that both Adam and SGD optimization with momentum worked well depending on dataset, and used two loss functions: a multi-class hinge loss for reconstruction error, and a cosine embedding loss to push adjacent nodes' embeddings closer together, and non-adjacent ones farther apart.

At each iteration of the outer loop, we load a new graph at the given time step and do some basic data processing. We then use a Net2WiderNet [26] call on the input and output layers to accommodate any increase in the size of the graph. In the inner loop, we re-run the random walk procedure to get a new augmented data matrix, and run negative sampling procedure to get input for the embedding loss, with positive examples being pushed closer together through hinge_loss($x, y, 1$) and negatives apart through hinge_loss($x, y, -1$).

Algorithm 4 Embedding System Training

function TRAIN
  for t in $timesteps$ **do**
    load graph G_t
    if $G_t.num\_nodes > G_{t-1}.num\_nodes$ **then**
      run Net2WiderNet on input and output layers
    $X \leftarrow$ random walk augmented adjacency matrix
    for $epoch$ in $epochs$ **do**
      $recon, embeddings \leftarrow model(X)$
      $hinge \leftarrow$ hinge_loss($X, recon$)
      $neighbors \leftarrow$ [random_neighbor(u, G_t) for u in G_t]
      $cosine \leftarrow$ cosine_loss($embeddings, embeddings[neighbors], 1$)
      for _ in range(25) **do**
        $negatives \leftarrow$ [random_node(G_t) for _ in G_t]
        $cosine$ += cosine_loss($embeddings, embeddings[negatives], -1$)
      $loss = hinge + cosine$
      backprop($loss$)

3.2.4 Network Evaluation

Our primary method of evaluating the quality of embeddings was using them for clustering problems. After embeddings were generated, we ran several K-means runs to cluster them, and then defined those clusters as partitions on the underlying graph and compared their modularity to that of the Louvain algorithm. We would have liked to reconstruct graphs from our embeddings to compare the reconstructed structure to the original, but unfortunately our choice of loss function as well as the augmented data prevented us from reconstructing graphs with any accuracy. Our reconstructions also prevented us from analyzing link prediction effectively, as the only method we could use was evaluating embedding proximity which provides no strong cutoffs for when there is or isn't a link. Another important metric we evaluated was embedding stability, a measure how much the embedding for a given node changed at any given time step, through simple checks of $||e_{i,t} - e_{i,t-1}||_2$, where $e_{i,t}$ is the embedding of node i at time t, or alternatively but not equivalently $||E_t - E_{t-1}||_F^2$, where E_t is the matrix of all embeddings at time t.

3.2.5 Implementation

Our autoencoder was implemented with NetworkX, PyTorch, and PyTorch Geometric, all of which were chosen due to our prior familiarity with them. We are largely happy with the results we were able to achieve, although we don't doubt there are better network architectures and training techniques waiting to be discovered, as our model was the result of much trial and error.

Designing the Augmented Data Matrix

In our initial plan, we planned to draw inspiration from artificial ant colony algorithms [27, 28] through the use of pheremones on our graph that would be preserved at each time step. In normal artificial ant colony algorithms, ants function as walkers on the graph that leave a pheromone on each edge they traverse, and the more pheromones on an edge, the higher chance an ant will transition using it. We planned to do the opposite of this, with higher pheromone levels decreasing transition probability, so as to promote further exploration of the graph. This would have been especially useful in giving preference to new edges added at any given time step.

While we did see some improvements with the ant colony method, computation time soared as we had to compute transition probabilities at each step of each walker, as the pheromone levels were dynamic. While we believe this could be addressed with further optimization, the improvements to model results were small enough compared to the increase in running time that we did not pursue the method further.

Designing the Network Architecture

The network architecture went through several distinct phases of development. Early on we had difficulty installing PyTorch Geometric and attempted to create a fully linear network, but as one would expect training times were high and results were poor. In the next iteration we attempted a fully graph-convolutional network, and as part of that experimented with numerous types of graph convolution layers. Lastly, we found that a combination of graph convolutions and standard convolutions seemed to work best, and settled on that structure.

One unfortunate limitation we ran into in testing was the inconsistent implementation of convolutional layers in PyTorch Geometric. While this wouldn't normally be an issue, we were using custom Net2Net calls on the input and output layers, which were always graph convolutional layers. These inconsistencies resulted in us needing to rewrite the Net2Net each time we tried a new layer, and as such we weren't able to test as many as we would have liked. That being said, we did try out several methods including Morris et al's k-GNN [29] and Kipf et al's GCN [30], and while they did tend to be faster, they also generated notably lower quality embeddings.

One unusual choice we made in our architecture was the use of the SineReLU activation function.

$$SineReLU(x) = \begin{cases} 0.025(sin(x) - cos(x)) & x < 0 \\ x & x \geq 0 \end{cases}$$

SineReLU has the advantage of always being differentiable, and the below zero component is best understood as being a noisy gradient to promote further exploration. We did notice a marginal increase in performance with this function, and while it's quite possible that itself is just noise we decided to keep it anyways.

Designing the Loss Functions

We initially used simple L1 and L2 loss for our reconstruction error, but quickly found that the model would just learn the degree of a node divided by the total number of nodes, a classic case of learning the average. We addressed this by making the problem a classification problem instead of a direct reconstruction, as in that context the output becomes binary: either a given node is a neighbor or it isn't. While this does have the effect of making it difficult to impossible to reconstruct a graph directly from the embeddings, performance was vastly improved and we feel that is a small price to pay. PyTorch provides two easy loss functions for the multi-label classification case: binary cross entropy and hinge loss. The latter seemed to both converge faster, and yield better embeddings in general, so we stuck with the hinge loss in our final model.

The other loss function we used was a cosine embedding loss to influence how close the embeddings are to each other. This is a standard technique in embedding con-

texts where we know pairwise which data points should be close together and which farther apart. The obvious route to take, and the one we used, is that two nodes should be embedded close together if they are neighbors, and farther apart if they are not. This was easy enough to do, and we followed a standard negative sampling heuristic of using 25 negative examples for each positive, in our case 25 non-neighbors and 1 neighbor at each iteration of the model. Increasing the number much beyond this would slow down the model more than we would like, and decreasing it tended to marginally decrease embedding quality.

Modularity and Shortest Paths

Another area we experimented in was encoding extra information into our embeddings so as to better represent the network's structure, specifically optimizing modularity of the resulting embeddings, and encoding shortest path length between two nodes. Both these methods increased computation time and decreased embedding quality, so they were ultimately abandoned despite efforts to fix them.

In encoding modularity, we attempted to force our embeddings to be linearlly reducable to assignment vectors so as to be optimized against a modularity matrix, as done by Newman [31], and in a deep learning context by Yang et al [32]. Unfortunately in the vain of Brandes et al [33] we rapidly ran into several issues. The first being the optimization space appeared extremely non-convex and it was difficult to converge to any solution, much less a good one. The other was it was difficult to constrain our assignment vectors such that they summed to one without increasing the depth of the layer translating the embeddings to assignments, which then had the effect of decorrelating the two entirely. Fixing any given issue seemed to lead to raising others, and in the end even the assignment vectors didn't get good modularity, and we decided to move on without pursuing it further.

We had more success in encoding shortest path distances, following the methods of Rizi et al [34]. For each node in the graph, we randomly choose another and compute their shortest path length, and then have a new layer of the network translate the average of their two embeddings into a shortest path length. While we were able to approximate the path lengths very well, it did not increase embedding quality by any metric, and as such we abandoned that method as well.

Visualizing the Embeddings

While loss metrics are useful in understanding how well the embeddings represent the data, it is almost always useful to inspect the results visually as well to check for issues that the loss might not capture. The obvious way to do this is to make the hidden dimension 2 so it can be plotted directly, or to introduce another layer to encode the embeddings down to 2 values as well. While the latter method has merit, since our data tended to be highly clustered we decided to use TSNE [35], a

dimensionality reduction technique that prioritizes preserving community structure in the original space.

Standard implementations of TSNE work fine for visualizing the embeddings at a given time step, but we quickly ran into issues with consistency across multiple time steps. TSNE does not in its normal form support online updating, although workarounds have been proposed [36]. We overcame this issue by simply initializing the TSNE weights at time t to the ones learned step $t-1$, and initialized any new nodes to $\mathcal{N}(\mu(E_{t-1}), I)$. While this led to more unstable visualizations than we would have liked, it was easy to implement and was still a significant step up from retraining from scratch each time.

Chapter 4

Results

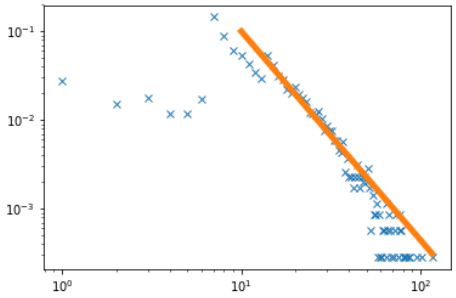
We were generally satisfied with the results of both our network model and embedding system, although both did have shortcomings that we feel are inevitable in trying to capture real-world phenomena. Our network model ended up being a jack of all trades, master of none, which was to be expected in combining models as we did. Our embedding system does seem to consistently meet DynGEM's performance when restricted in running time, and beat it when it is allowed to run longer, especially on networks with more complex topologies.

4.1 The Network Model

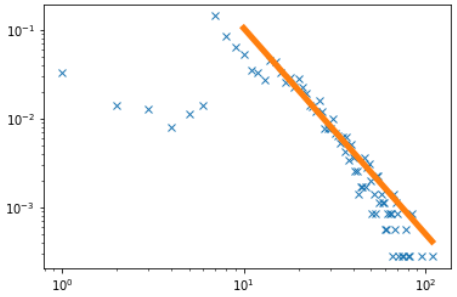
Our goal in designing the network model was for it to be able to create synthetic, dynamic graphs that are similar to real-world data. In this, we believe we were successful by most metrics. The degree distributions of our networks only followed a power-law in their upper tails, there was no discernible increase in diameter with network size but average path lengths and clustering were consistent with small-world behavior, and they seemed to be length-scale invariant but only up to a few renormalizations.

4.1.1 Scale-Freeness

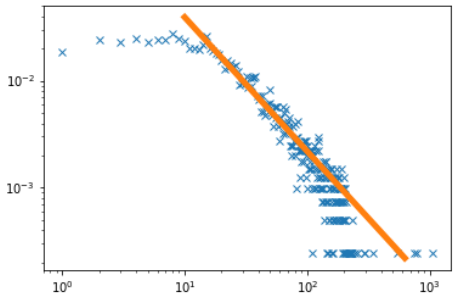
While our resulting networks were not strongly scale-free throughout, they were weakly so in the same manner as most real-world social networks, in that they are scale free for large values around the upper tail. The degree distributions of our model as well as a Facebook dataset and a run of the Barabási–Albert model can be seen in figure 4.1.



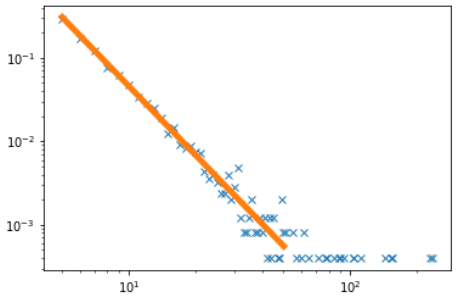
(a): 2500 Model Iterations, $\gamma \approx 2.35$



(b): 2500 Model Iterations, $\gamma \approx 2.31$



(c): SNAP's Facebook Dataset, $\gamma \approx 1.25$



(d): Barabási–Albert Model, $\gamma \approx 2.74$

Figure 4.1: Degree Plots and Power Law Fits of Graphs

We believe that the non-power law lower tail stems from the Schelling step, as in the course of model execution most nodes either become unhappy and move, or some other node moves into their neighborhood, this increasing the degree above what we'd see in a truly scale-free network. Regardless, ignoring the non-power law lower tail is common in determining whether a network is scale-free [37], and for the upper tail the fit is good and γ seems strongly within the $(2, 3)$ range.

4.1.2 Small-Worldedness

We identified two metrics to test small-worldedness with: a combination of average clustering and shortest path length compared to random graphs, and the growth in diameter over time. We are very satisfied in the former showing small-world characteristics, but in the latter our model appears to work in a densification regime rather than the expected slow increase.

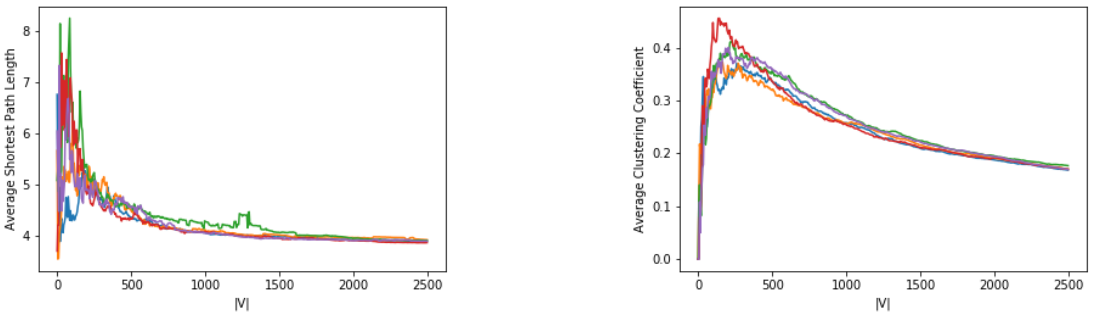
Average Clustering and Path Length

We found that over time, the average shortest path length l_G converged to around 3.9, and that the average clustering coefficient $\bar{C}$ converged to around 0.17, both

of which comfortably similar to what we'd expect, as Erdős–Rényi networks of comparable size have values of around 3 and 0.005, respectively. Although the average clustering coefficient is somewhat low, the data clearly indicates that the network has small-world properties, albeit weaker than we would like. The hard numbers can be found in table 4.1, and a visualization of $\bar{C}$ and l_G for several runs of our model can be seen in figure 4.2.

Model	$\lvert V\rvert$	$\lvert E\rvert$	l_G	$\bar{C}$
Erdős–Rényi	3500	30545	3.135	0.005
Synthetic	3525	27410	3.899	0.171
Watts-Strogatz	3500	28000	3.943	0.431

Table 4.1: Average clustering coefficients and shortest path lengths

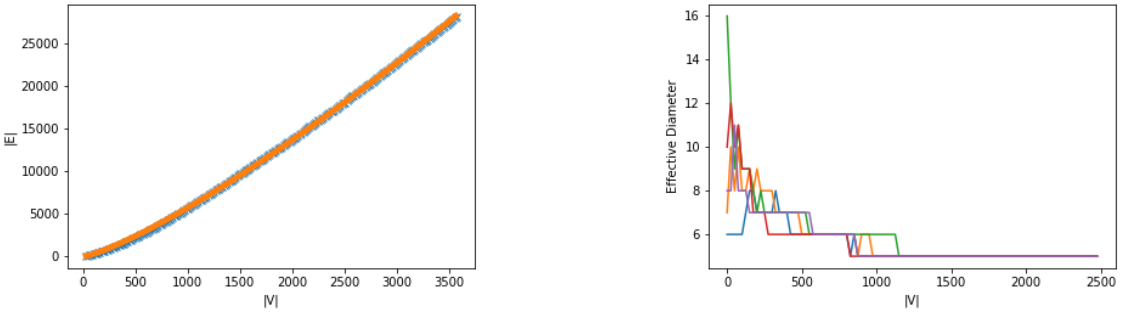


(a): Average Clustering Coefficient **(b):** Average Shortest Path Length

Figure 4.2: Metrics over time for 5 runs of the model

Diameter Dynamics

An unexpected result we found in our model was that the effective diameters seemed to converge to around 5 rather than increase logarithmically as is expected in small-worlds. This fortunately has a straightforward cause, while the number of nodes increases only linearly in time, the increase in edges appears to follow a power law. This is an especially interesting result as this is near exactly what is described by Leskovec [38], with the plot of number of nodes against number of edges following a power law with $a = 1.253$, similar to Leskovec's findings of a recommender system dataset. Figure 4.3.a shows the edge vs node growth power law, and figure 4.3.b shows the decreasing diameter over time for 5 model runs.



(a): Densification regime, $a = 1.253$ **(b):** Effective diameter over time

Figure 4.3: Densification metrics on 5 runs of the model

We view this as a confirmation of the model's relation to the real-world: this is an emergent phenomena that we did not design for, or in fact even know the existence of beforehand that nonetheless exists in our networks despite the fact that edge creation and deletion is purely governed by the Schelling process.

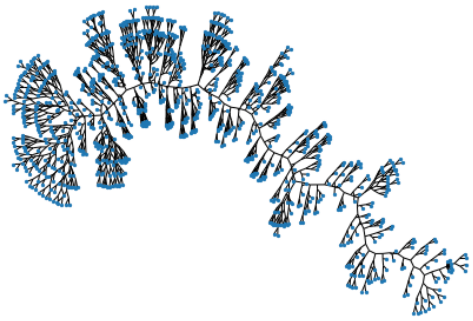
4.1.3 Self-Similarity

Another unexpected result we found was self-similarity, although as with small-worldness our model appears only weakly self-similar. Self-similarity is the least well-defined metric we evaluated, and as such there were several paradigms to consider. The first method we examine determines self-similarity by measuring characteristics of the Girvan-Newman community detection algorithm [39], and the second uses a more standard box counting and renormalization approach [40].

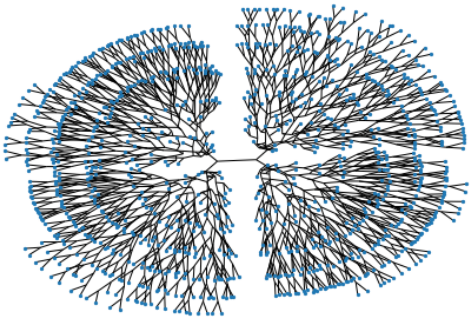
Girvan-Newman Self-Similarity

Guimera et al [39] draw inspiration from river and drainage basin literature and evaluate 3 main features of the Girvan-Newman communities: The structure of its dendrogram, the cumulative probability distribution of its community sizes, and lastly the Horton-Strahler numbers. We compare our model to an Erdős–Rényi graph, the fractal model proposed by Song [41], and Guimera's email dataset.

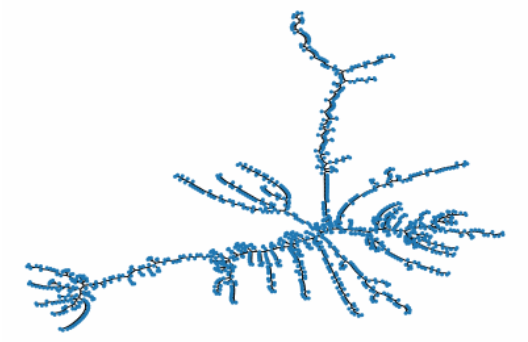
Dendrograms of the GN algorithm on random graphs tend to be straight lines, but running the algorithm on our model after 1250 iterations leads to a similar structure as Guimera's email dataset. This is the most subjective measurement as we only inspect the graphs visually. A cursory glance at the GN dendrograms in figure 4.4 confirms the similarity to the email dataset as well as such natural structures as Brownian trees.



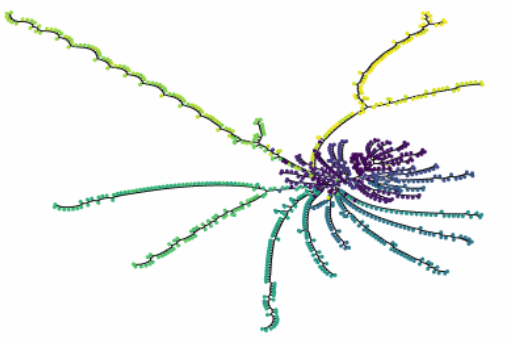
(a): Erdős–Rényi graph, $|V| = 1000$



(b): Song fractal model, $|V| = 1038$



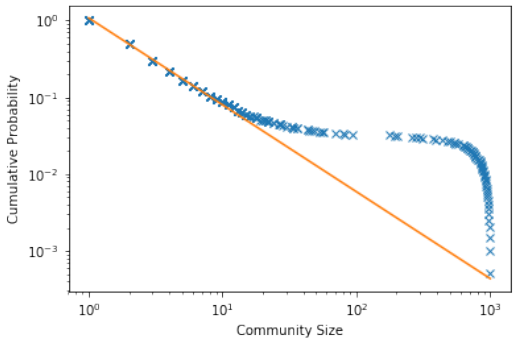
(c): Guimera's email dataset, $|V| = 1133$



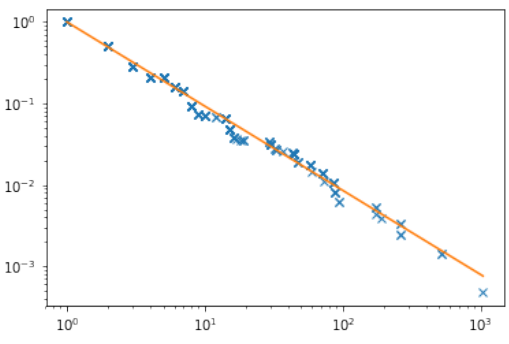
(d): Synthetic model, $|V| = 913$, colored by Apt-Markis product

Figure 4.4: GN community dendrograms

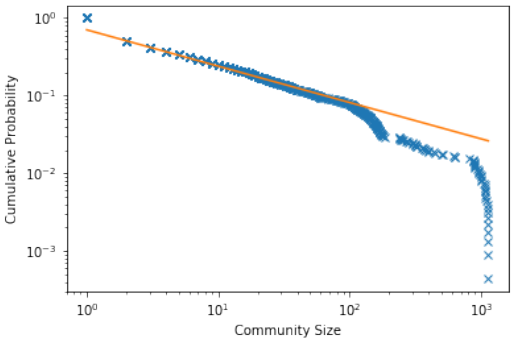
We can expand our analysis of the communities by examining the cumulative community size distribution, and again can see obvious similarities to real-world phenomena as well as Guimera's data in figure 4.5. The power law is a noticeably worse fit for the same lower tail of our data however, with a significantly lower R^2 and visually it seems another distribution might be a better fit. Regardless, the basic structure of the curves seems similar enough to warrant further investigation.



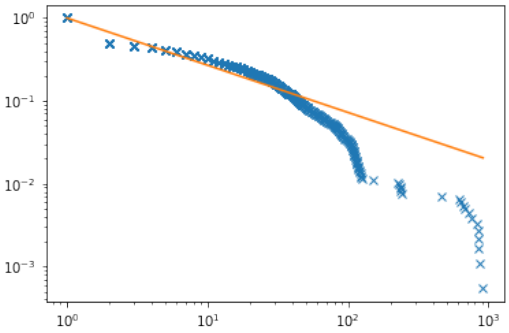
(a): Erdős–Rényi graph, exponent ≈ 1.1, , $R^2 = 0.99$



(b): Song fractal model, exponent ≈ 1.0, , $R^2 = 0.99$



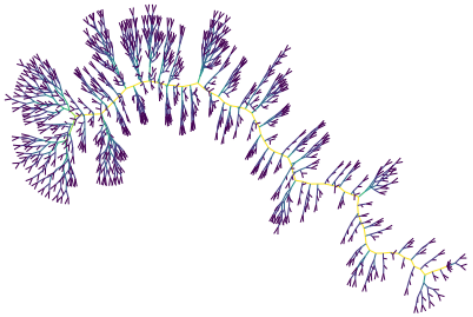
(c): Guimera's email dataset, exponent ≈ 0.48, $R^2 = 0.99$



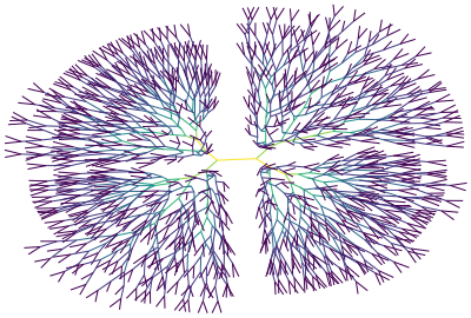
(d): Synthetic model, exponent ≈ 0.57, $R^2 = 0.83$

Figure 4.5: GN cumulative community distributions

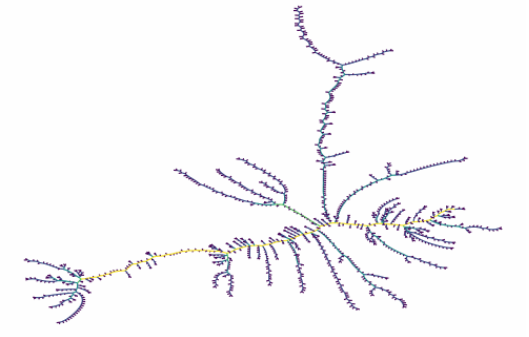
The last metric they evaluate involves the Horton-Strahler numbers [42, 43], which were initially developed to describe the size of tributaries and rivers in networks. A low number means there is small upstream flow, and a high number indicates a large amount of upstream flow. In our case, flow corresponds to community structure, where a higher number corresponds to higher community structure upstream of a given node.



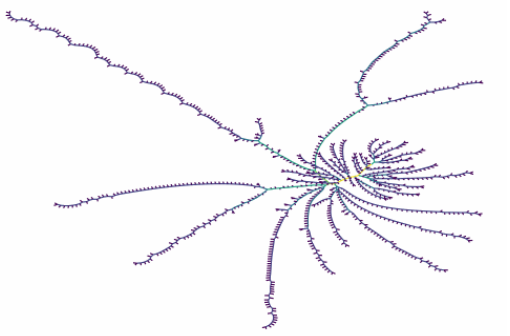
(a): Erdős–Rényi graph, $max(HS) = 7$



(b): Song fractal model, $max(HS) = 8$



(c): Guimera's email dataset, $max(HS) = 5$



(d): Synthetic model, $max(HS) = 6$

Figure 4.6: HS indices

Examining the HS indices in and of themselves as in figure 4.6 doesn't yield a great deal of information, but they can be used to calculate a useful metric of self-similarity: bifurcation ratios. The bifurcation ratio is defined for $1 < i < max(HS)$ for a given graph, and is simply $B_i = n_i/n_{i+1}$, where n_i is the number of branches with HS index i. When $\forall i B \approx B_i$, the network is said to be self-similar, as when the ratios are all similar that means that each branch's subtrees have a similar structure. We can evaluate in this by simply inspecting the mean and standard deviation of all B_i values, which yields some interesting results, shown in table 4.2.

Model	$\|V\|$	$\|E\|$	μ	σ
Erdős–Rényi	1000	2461	3.385	1.332
Song Fractal	1038	1296	2.761	0.594
Guimera Emails	1133	5452	5.872	0.889
Synthetic	913	4992	4.869	3.772

Table 4.2: Mean and Standard Deviation of B for several models

Obviously the standard deviation on our model is significantly higher than the rest. When examining the B_i values directly in table 4.3, we notice that B_1 is a major outlier, and removing it yields $\mu = 3.002$ and $\sigma = 0.592$. Unfortunately we have no explanation as to why our network has so many branches with $HS = 1$ compared to the other models, and as such we refrain from making strong conclusions with this method.

Model	B_1	B_2	B_3	B_4	B_5	B_6	B_7
Erdős–Rényi	2.6	2.6	3.1	4.1	6.0	2.0	-
Song Fractal	2.4	3.1	2.1	3.8	3.0	3.0	2.0
Guimera Emails	6.3	5.7	4.6	7.0	-	-	-
Synthetic	12.3	3.4	3.1	3.5	2.0	-	-

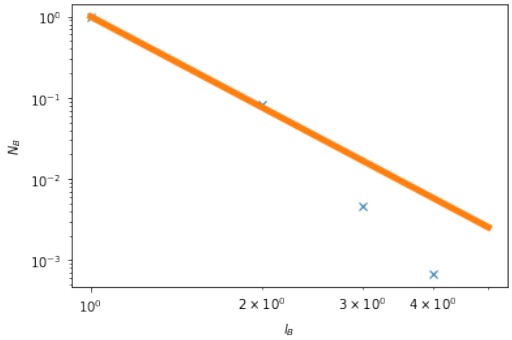
Table 4.3: B values for several models

Box Counting Self-Similarity

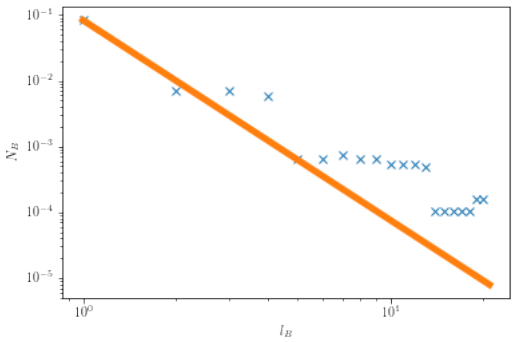
A second method of evaluating self-similarity more rooted in geometry is proposed by Song et al [7], relying on box counting. In the geometric domain, this involves iteratively tiling a shape with smaller and smaller boxes, at each time evaluating how many boxes bound some part of the shape. This of course doesn't trivially generalize to networks, but Song proposes to define boxes as neighborhood graphs whose radius corresponds to box size. Nodes in the graph are randomly sampled and box with nodes in their neighborhood until the entire graph is covered.

Song proposes two new metrics to evaluate self-similarity: the fractal dimension d_B, which is the exponent in the power law relation between box size and number of

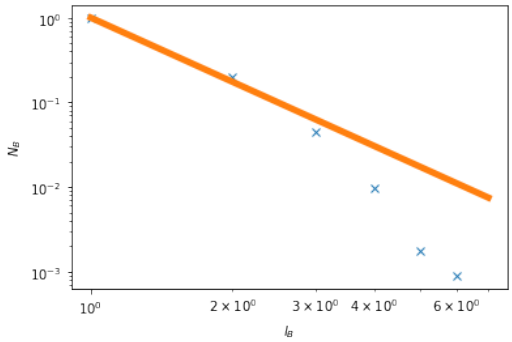
boxes needed to cover the network, and d_k, which relates to how stable a network is to renormalization. Our model seems to again be weakly self-similar under these metrics the d_B metric shown in figure 4.7, but it was difficult to calculate d_k given the relatively small size of our networks: Song calculated his against the entire Internet in 2006.



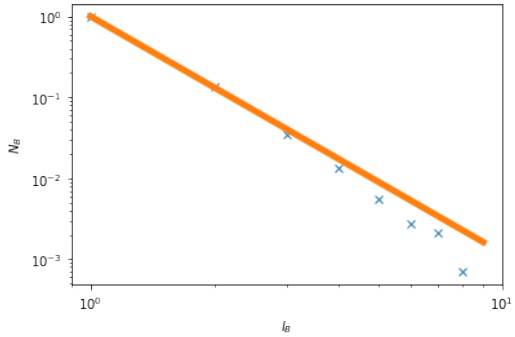
(a): Barabási–Albert graph, $d_B = 3.71$



(b): Song fractal model, $d_B = 3.04$



(c): Guimera's email dataset, $d_B = 2.52$



(d): Synthetic model, $d_B = 2.94$

Figure 4.7: Neighborhood radius against number of nodes needed to cover the networks

The main thing to notice here is how Song's fractal model skews above the fitted curve, a behavior Song also finds in his plot of the Internet. All other models seem to skew below the plot, although they all are well described by a power law. Based off the loose findings from the GN method as well as the fact that our model is closer to the power law curve than Guimera's emails dataset, we do conclude that our model is weakly self-similar.

Unfortunately in both methods evaluating self-similarity requires very large datasets to allow for examination at numerous scales, something we just were not able to both run our model for, and use the requisite algorithms on. Nonetheless, our basic findings do indicate that our networks are self-similar to some degree, and further examination could be a useful research direction.

4.1.4 Modularity

The networks generated by our model tended to be highly modular. This is clear for visual inspections of small networks of a few hundred nodes shown in figure 4.8, but as the networks grow we needed a new metric to verify that modularity was maintained. We accomplished this by using the Louvain algorithm [24] to detect communities and then check the modularity Q of those communities, shown in table 4.4.

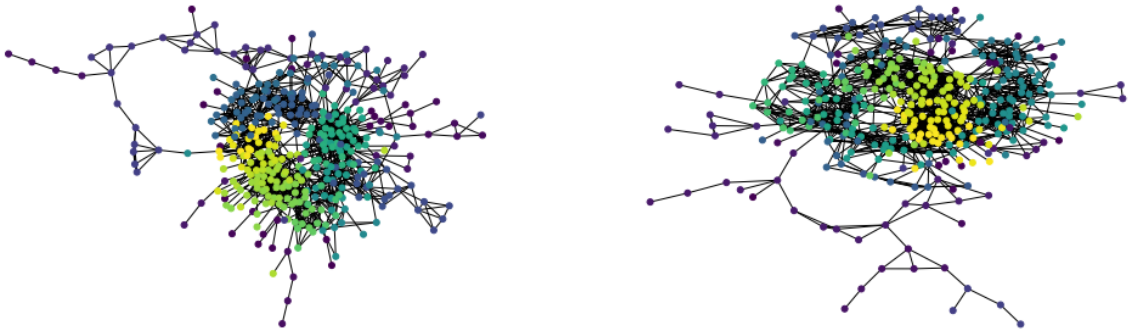


Figure 4.8: Two generated graphs after 350 iterations colored by Apt-Markakis product

	V	E	Q
Erdős–Rényi	3500	24503	0.224
Watts-Strogatz	3500	2800	0.792
Song Fractal	3512	4096	0.926
Guimera Emails	1133	5452	0.570
CORA	5278	5278	0.813
Synthetic	3525	27410	0.714

Table 4.4: Modularity of several models

The modularity scores on our network were the second highest of the models and datasets, below only CORA, and the Song Fractal and Watts-Strogatz models. Fractal networks in general are highly modular almost by definition, explaining the strong modularity of the Song networks. Our model scored close to Watts-Strogatz and CORA but was not quite comparable, which we attribute to the edge saturation issue that we were unable to completely solve: it's likely that several of the edges are superfluous even for maintaining Schelling stability.

4.2 The Embedding System

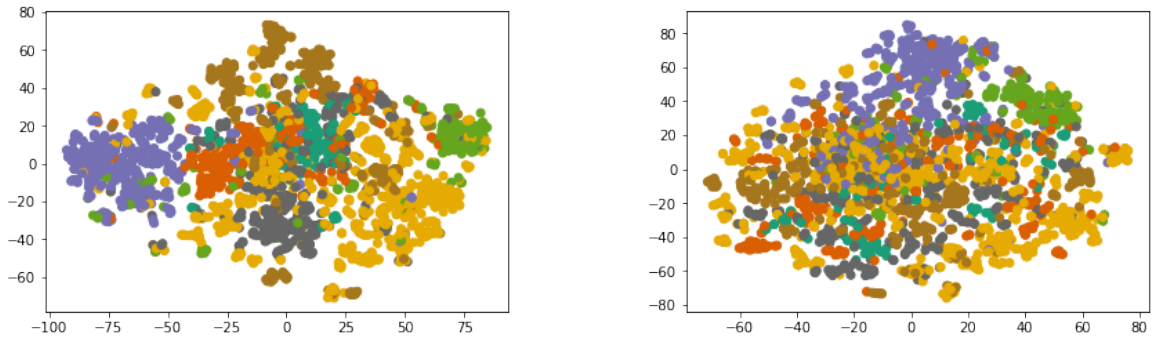
Our embedding system was able to consistently outperform DynGEM [11], the only other dynamic growing embedding model that had reference code we could find. Although several other models have been proposed [20, 21], we were unable to find reference implementations and as such restrict ourselves to DynGEM. We

compare our embeddings to DynGEM as well as Node2Vec [25] representations learned separately on each graph and show large improvements in both clustering and stability on both synthetic and real-world data. We found that Adam and SGD optimizers seem to perform well in general but on some datasets one is significantly better than the other: unless otherwise specified performances listed were trained using SGD.

4.2.1 CORA Sanity Check

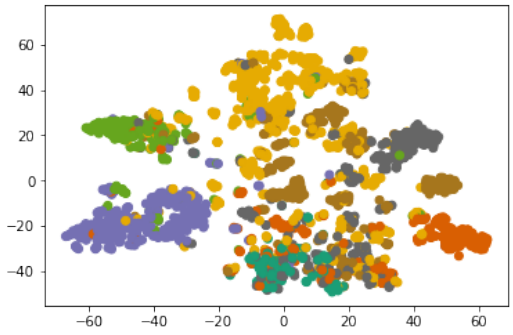
Before checking our models's performance on other metrics, we wanted to test that it was learning useful embeddings at all on a real-world dataset. We used the CORA dataset [44] for this, as it is a standard dataset for classification with ground truth labels. While the dataset is unfortunately not dynamic, it is a standard benchmark for node embedding systems and as such we view it as a natural choice for a simple sanity check. For our check, we convert it to an undirected graph and take only the giant component.

We checked network performance on it by generating embeddings and then clustering with k-means. We then match these clusters to the ground truth values and took the best of 10 performances to get an accuracy score that we compare the models on. Importantly, we do not consider any data other than adjacency information in generating the embeddings for any of our systems, as we wanted to make our system as generalizable as possible. Figure 4.9 show our model trained with Adam significantly outperforming Node2Vec and DynGEM in this metric, and as such we conclude our system is working as expected.



(a): Node2Vec, $acc = 0.599$

(b): DynGEM, $acc = 0.425$



(c): Our System, $acc = 0.692$

Figure 4.9: CORA performance by model, colored by paper classification

4.2.2 Clustering Performance

The area we saw the largest improvement over existing encoding methods was clustering performance. When trained for either the same amount of running time or epochs, our model significantly outperforms DynGEM and Node2Vec on our synthetic dataset but faces more issues with the AS [23] and Irvine [45] datasets.

We evaluate clustering performance by first running the Louvain algorithm to get a reference performance as well as an upper bound on the number of communities to check n. We define the best clustering performance as the maximum modularity among k-means clustering run 5 times each for $num\_clusters \in (2, n)$. We then compare the models using the average performance over all embeddings at all time steps. Table 4.5 contains clustering performances from the three algorithms on the three datasets. The graphs from the synthetic model correspond to a single run at 10 step intervals between 2500 and 2600, and the AS data is a subset of the first 10 graphs separated by 5 steps each. The Irvine dataset is the giant component over 10 snapshots corresponding to 10 percentiles of the timestamps.

	Node2Vec	DynGEM	Our System	*Louvain*
Synthetic Data	0.68	0.66	**0.72**	0.76
AS Dataset	**0.29**	0.13	0.17	0.65
Irvine Dataset	**0.33**	0.00	0.27	0.34

Table 4.5: K-means clustering performance of embeddings

Our model likely runs into issues on the AS dataset because of the removal of nodes at some time steps, something that we did not plan for and does not happen in our synthetic model. This throws off training as we keep the removed nodes as isolates, which appears to impede training on the embedding loss of both it and the remaining nodes. DynGEM exhibits the same issue but significantly more so, and although Node2Vec does beat our performance, our embeddings are much more stable. The Irvine dataset has a low clustering coefficient and the Louvain

algorithm struggles to find modular communities, which appears to be a real issue for DynGEM and to a lesser extent our system.

4.2.3 Embedding Stability

While our model was unable to beat Node2Vec on clustering performance, we were able to outperform both it and DynGEM on stability, which with inspiration from DynGEM we define as

$$\text{stability} = \frac{||E_t - E_{t-1}||_F^2}{||A_t - A_{t-1}||_F^2}$$

where E_t is the embedding matrix and A_t is the adjacency matrix both at time t. We used the same generated embeddings and graphs as in modularity testing, and results are shown in table 4.6.

	Node2Vec	DynGEM	Our System
Synthetic Data	20.52	58.53	**4.35**
AS Dataset	4.01	2.68	**0.56**
Irvine Dataset	9.01	22.98	**2.87**

Table 4.6: Embedding stability under Frobenius norm

One notable issue is that DynGEM seems to do worse than even Node2Vec on our synthetic model and the Irvine dataset, which we think is attributable to overfitting. We can run DynGEM for fewer iterations and get more stable embeddings than Node2Vec, although also causing significant degradation to clustering performance. Regardless, the results indicate that our embeddings are more stable than DynGEM's as well as Node2Vec, highlighting the improvements gained by graph convolutions.

4.2.4 Running Time

Running time was difficult to evaluate with our model as we had to decide arbitrary stopping criteria based off when the model had converged, which was a time-consuming process in and of itself. We settled on using the running times of the model runs we used in analyzing stability and clustering performance as they should be representative of our model performance in general.

Table 4.8 shows the runtimes of the three models. We found that Node2Vec and DynGEM tended to converge faster than our model on smaller networks, but our system began to outperform them as network size increased. Both models did significantly improve over Node2Vec's training time for the larger networks however, as shown in the chart below. We attribute this largely to overhead complexity which is negligible in Node2Vec and DynGem, but for our model is linear in the number of

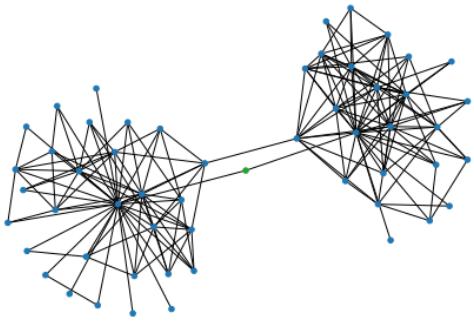
nodes for constructing the augmented input matrix.

	Node2Vec	DynGEM	Our System
Synthetic Data	197	**74**	103
AS Dataset	321	251	**197**
Irvine Dataset	**52**	138	103

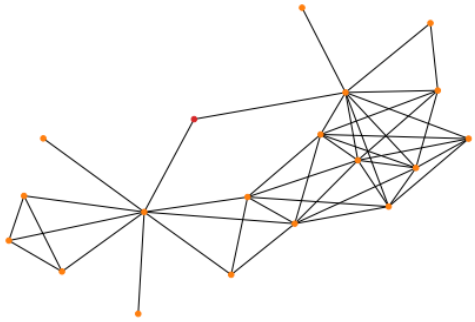
Table 4.7: Embedding system running time

4.2.5 Applications

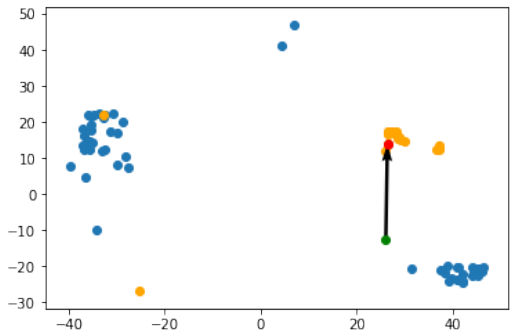
Our embedding system has many possible applications, ranging from the clustering discussed above to link prediction among many others. Two applications that fit naturally into our system and model however are anomaly detection and trajectory visualization. Anomaly detection is simple in our case, we merely need to show that our embeddings at a time step where an anomaly happens are significantly different than those at the step before. Trajectory visualization is another interesting thing, as it lets us see how a node moves in the embedding space when it is chosen as part of the Schelling step.



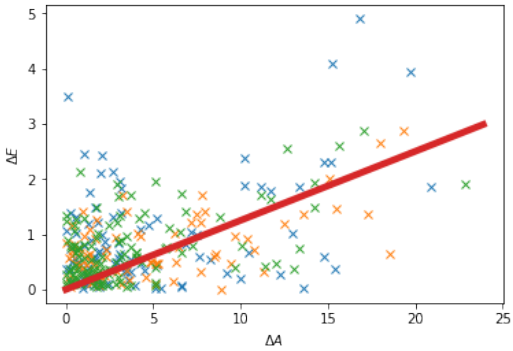
(a): Neighborhood of target at $t-1$



(b): Neighborhood of target at t



(c): Embedding space change



(d): Embedding vs adjacency stability over three runs

Figure 4.10: TSNE visualization of a Schelling step

Visualizing Trajectories

Another useful application is trajectory visualization. While we did not have the the capacity to run the model on huge annotated datasets, we could confirm that node movements in the embedding space are proportional to their change in the graph. While this is an expected result, there are a myriad of uses we could imagine in a real-world context. For example, a user could be a member of a known community, and slowly make connections in the direction of another. Our model would be able to detect this perhaps even before the user knows the existence of the other community and recommend users or products preemptively.

Figures 4.10.a - 4.11.c display the movement of one node before in green and after in red the Schelling step selects it. The blue nodes denote the neighborhood graph of the node with radius 2 before the step, and the orange the neighborhood after. As can be expected, the node clearly moves from close proximity to its old neighborhood to its new.

Anomaly Detection

One application our embeddings naturally lend themselves to is anomaly detection. In the case of known datasets, we would expect the change in embeddings between time steps to be proportional to the change in the adjacency matrix. While this isn't terribly useful as we already know of any anomalies it might encounter, in the case of real-world streaming data however examining the embeddings could be useful as they are more invariant to changes in the network than adjacency matrix.

While we leave further study of the origin of this invariance to future work, it is easy to imagine a case where we have a dense core of nodes surrounded by a sparser fringe and between time steps we have significant rearrangement in the core. This would yield a large change in the adjacency matrix, but within the embedding space the changes should be limited to the core, yielding a smaller total change. Figure 4.10.d shows the change in embeddings against the change in adjacency for three separate runs using the same metrics we used to verify stability. The changes in embeddings tend to be roughly an order of magnitude less than the changes in the adjacency matrix, highlighting the embeddings' robustness.

Chapter 5

Conclusion

Building off of prior models, we were able to construct a dynamic model of social networks that exhibits notable real-world properties, including some that to our knowledge have not been studied on Schelling networks or dynamic models in general. Using this model as a training baseline, we extend and improve existing work to construct a dynamic graph-convolutional graph embedding system that significantly outperforms most existing embedding systems.

Our dynamic network model is a synthesis of the Barabási–Albert model of preferential attachment, the Apt-Markakis model of product diffusion, and Schelling's model of segregation. We make minor changes to each to get them to work together and to adapt Schelling's model to networks. We then go on to show numerous properties of this synthetic model, including scale-free and small-world structure. In the course of examining small-world behavior we stumbled upon another emergent property of our network: densification. This is an especially interesting phenomena in that to our knowledge it has not been shown in the context of Schelling network models, which is the source of it in our synthetic model. We then go on to study self-similarity of our model, another area which to our knowledge has been little-studied in network analysis.

Building on prior work and our synthetic model, we go on to construct a node embedding system. The embedding system is a graph-convolutional autoencoder whose input is the adjacency information augmented by a random walk performed at each epoch. This walk augmentation allowed for significantly more topological information to be used in embedding a given node, and introduced a degree of randomness to encourage the encoder to generate good embeddings. We make heavy use of standard and graph convolutions in our model to keep the network size manageable and to take full advantage of the graph's topology. We then use hinge loss for reconstruction and a cosine loss to maximize distance between non-adjacent nodes and train the model via SGD or Adam, depending on the dataset. Using this method we were able to significantly outperform existing systems both on clustering performance and embedding stability.

On the CORA dataset we were able to achieve a classification accuracy of 0.692

using only simple k-means clustering on our embeddings, an accuracy that is on par with decision trees explicitly trained on the data for the classification task. On our synthetic model k-means clustering approached the modularity of partitions discovered by the Louvain algorithm, and on all tested datasets our embeddings were extremely stable over time, often by more an order of magnitude than the adjacency matrices. Our model also trains faster on larger datasets than other models, a feature we ascribe to our heavy use of convolution.

All told, both our systems exceeded our expectations in numerous ways. Our dynamic network model exhibited two emergent behaviors we did not expect: self-similarity and densification, both of which are found in real-world networks. We planned only to match performance of existing embedding systems with ours, but were able to outperform them even on static graphs, often by significant margins.

5.1 Ethics and Legality

Our work does not in and of itself have any major ethical or legal considerations and we did not see anything in the ethics checklist applicable to our project. While we did use some real-world datasets, these are all standard datasets that had been anonymized before we even downloaded them and have been widely used across the field. While an argument could be made that the entire field of social media analytics and more specifically node embedding has uncomfortable applications towards real-world users as highlighted by Facebook's recent troubles, our particular project is no more or less dubious than any other in the field. The social network model is particularly innocuous, as there is hardly any application in which it could have implications on personal privacy or any other such concern.

5.2 Future Work

There are numerous areas of future work we could imagine for our project, from extending and applying the current work to solving issues we could not find elegant solutions to.

5.2.1 The Network Model

While we were very happy with how our model behaved, the edge saturation problem remains an unsolved issue. We believe that addressing this would require a fundamentally different translation of Schelling's model to the network space, or at the very least the introduction of another model that discourages edge formation. We view the latter as the less elegant approach as our model is already a synthesis of three models: throwing another on it would hardly be desirable.

Another future direction would be closer study of self-similarity. We were unable to draw conclusive evidence of self-similarity despite strong indications of its presence. A further examination of this on the existing model would no doubt shine further light on the model properties, and could lead to studying how to tune the model to promote stronger self-similarity.

5.2.2 The Embedding System

Again, we were quite happy with our network model an don't have any major issues that remain to be addressed. That being said, two natural extensions exist in our view, both related to the network structure itself: multimodal variational autoencoders and RNNs.

A multimodal variational autoencoder such as the one proposed by Wu [46] seems like it would be a natural choice to model our social networks, as it is plausible to imagine each community as a single modality. This would intuitively map each node to a given community to and then optimize their placement within whichever community they're a member of. This approach does have numerous drawbacks however: the number of communities would need to be known in advance and it would be difficult to extend to overlapping community membership. Regardless, we view this as a promising direction if only for simple, non-overlapping networks.

RNN architectures are another natural extension due to their ability to capture time-series information. Our model preserves the previous step's information by effectively initializing the network to the same weights, but RNNs carry information about prior states and networks which could capture trends our model is unable to. We view this as a more useful direction for future research than MVAEs as they are more well-studied and more promising for more complex datasets.

An application that warrants further study is the one we touched on earlier: taking a closer look at node trajectories. We posited that examining a node's trajectory in the embedding space over time could have real predictive power in predicting future community membership or interest, and would like to study that with more detail. Doing so would require large computational power and a time-series annotated dataset, which may not be realistically achievable with our system, but the prospects are nonetheless exciting as to our knowledge no graph-based techniques have been used for this outside recommender systems.

Bibliography

[1] Damon Centola. The social origins of networks and diffusion. *American Journal of Sociology*, 120(5):1295–1338, 2015.

[2] Zhihao Wu, Giulia Menichetti, Christoph Rahmede, and Ginestra Bianconi. Emergent complex network geometry. *Scientific reports*, 5:10073, 2015.

[3] Krzysztof R Apt and Evangelos Markakis. Diffusion in social networks with competing products. In *International Symposium on Algorithmic Game Theory*, pages 212–223. Springer, 2011.

[4] Thomas C Schelling. Dynamic models of segregation. *Journal of mathematical sociology*, 1(2):143–186, 1971.

[5] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *science*, 286(5439):509–512, 1999.

[6] Xiao Fan Wang and Guanrong Chen. Complex networks: small-world, scale-free and beyond. *IEEE circuits and systems magazine*, 3(1):6–20, 2003.

[7] Chaoming Song, Shlomo Havlin, and Hernan A Makse. Self-similarity of complex networks. *Nature*, 433(7024):392, 2005.

[8] Qawi K Telesford, Karen E Joyce, Satoru Hayasaka, Jonathan H Burdette, and Paul J Laurienti. The ubiquity of small-world networks. *Brain connectivity*, 1(5):367–375, 2011.

[9] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

[10] Jundong Li, Harsh Dani, Xia Hu, Jiliang Tang, Yi Chang, and Huan Liu. Attributed network embedding for learning in a dynamic environment. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 387–396. ACM, 2017.

[11] Palash Goyal, Nitin Kamra, Xinran He, and Yan Liu. Dyngem: Deep embedding method for dynamic graphs. *arXiv preprint arXiv:1805.11273*, 2018.

[12] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

[13] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. In *Advances in Neural Information Processing Systems*, pages 6389–6399, 2018.

[14] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[15] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[16] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, pages 3844–3852, 2016.

[17] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[18] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of machine learning research*, 11(Dec):3371–3408, 2010.

[19] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 974–983. ACM, 2018.

[20] Yao Ma, Ziyi Guo, Zhaochun Ren, Eric Zhao, Jiliang Tang, and Dawei Yin. Dynamic graph neural networks. *arXiv preprint arXiv:1810.10627*, 2018.

[21] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. Representation learning over dynamic graphs. *arXiv preprint arXiv:1803.04051*, 2018.

[22] Brian Skyrms and Robin Pemantle. A dynamic model of social network formation. In *Adaptive networks*, pages 231–251. Springer, 2009.

[23] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 177–187. ACM, 2005.

[24] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[25] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[26] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.

[27] Cyrille Bertelle, Antoine Dutot, Frédéric Guinand, and Damien Olivier. Organization detection using emergent computing. *International Transactions on Systems Science and Applications*, 2(1):61–69, 2006.

[28] Xu Zhou, Yanheng Liu, Jindong Zhang, Tuming Liu, and Di Zhang. An ant colony based algorithm for overlapping community detection in complex networks. *Physica A: Statistical Mechanics and its Applications*, 427:289–301, 2015.

[29] Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4602–4609, 2019.

[30] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[31] Mark EJ Newman. Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582, 2006.

[32] Liang Yang, Xiaochun Cao, Dongxiao He, Chuan Wang, Xiao Wang, and Weixiong Zhang. Modularity based community detection with deep learning. In *IJCAI*, volume 16, pages 2252–2258, 2016.

[33] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. Maximizing modularity is hard. *arXiv preprint physics/0608255*, 2006.

[34] Fatemeh Salehi Rizi, Joerg Schloetterer, and Michael Granitzer. Shortest path distance approximation using deep learning techniques. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 1007–1014. IEEE, 2018.

[35] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[36] Laurens Van Der Maaten. Learning a parametric embedding by preserving local structure. In *Artificial Intelligence and Statistics*, pages 384–391, 2009.

[37] Anna D Broido and Aaron Clauset. Scale-free networks are rare. *Nature communications*, 10(1):1017, 2019.

[38] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):2, 2007.

[39] R Guimera, L Danon, A Diaz-Guilera, F Giralt, and A Arenas. Self-similar community structure in organizations. *arXiv preprint cond-mat/0211498*, 22, 2002.

[40] Chaoming Song, Lazaros K Gallos, Shlomo Havlin, and Hernán A Makse. How to calculate the fractal dimension of a complex network: the box covering algorithm. *Journal of Statistical Mechanics: Theory and Experiment*, 2007(03):P03006, 2007.

[41] Chaoming Song, Shlomo Havlin, and Hernán A Makse. Origins of fractality in the growth of complex networks. *Nature physics*, 2(4):275, 2006.

[42] Robert E Horton. Erosional development of streams and their drainage basins; hydrophysical approach to quantitative morphology. *Geological society of America bulletin*, 56(3):275–370, 1945.

[43] Arthur N Strahler. Dynamic basis of geomorphology. *Geological Society of America Bulletin*, 63(9):923–938, 1952.

[44] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

[45] Pietro Panzarasa, Tore Opsahl, and Kathleen M Carley. Patterns and dynamics of users' behavior and interaction: Network analysis of an online community. *Journal of the American Society for Information Science and Technology*, 60(5):911–932, 2009.

[46] Mike Wu and Noah Goodman. Multimodal generative models for scalable weakly-supervised learning. In *Advances in Neural Information Processing Systems*, pages 5575–5585, 2018.