Emergence and Collapse in Dissipative Dynamical Systems

Spring 2019 Network Science Project

S. Migirditch, TF. Varley

1 Introduction

The second law of thermodynamics captures how local decreases in entropy can facilitate global increases. This provides a mechanism for the spontaneous emergence of structure in systems. Far from equilibrium dynamics are mathematically difficult to study, partially because the second law gives no deterministic insight into the dynamics of transient emergent behavior beyond motivating its existence. The few other theoretical tools available like Crooks theorem [1] place only weak restrictions on possible transient behavior. However the complex systems found in nature ranging from whirlpools to evolved life manage to develop increasing complexity with relative stability. In the absence of theoretical fire power, some conceptual framework can give limited experimental guidance. Diffusion driven emergence is caused by the dissipation and equilibriation of a resource. If there is no resistance to diffusion no structure will emerge. Therefore it is the structure and dynamics of the system which the resource is diffusing through that exclusively determines the properties of emergent phenomena[2, 3]. This concept has been used with varying degrees of acknowledgment in several experimental designs. Of particular interest to this project is the computational network ecology work based on the niche model [4]. The primary purpose of this work is to explore the properties inherent to the niche model rather than the parameters of the broader systems it is incorporated into. A secondary focus is discovering the set of dynamics that can be produced by a very general model of competitive resource consumption in a network structured by the niche model.

2 The Model

The model and all analyses where programmed in Python 3.6, using the NetworkX package, version 2.2, as provided by Anaconda. The complete code needed for reproduction is included in Appendix 1. The most recent version and any features added since publication can be found on Github

2.1 Elements of the Model

The model takes the form of a time evolving directed graph without self-loops. All time evolution is done with a simple Euler step method and updates are random asynchronous. There are two classes of nodes: resource nodes and producer nodes. While the code is highly configurable, by default each producer is initialized by computing the following attributes:

Tab. 1: Individual Producer Attributes:

Attribute Name:	**Attribute Effect:**
Index	A unique identifier for accessing a producer node. Nodes are sequentially created and labeled so the index also informs on the creation order of nodes. Indexes are integers starting from 1.
Volume	The amount of rescource currently in the node. This is updated on each time step. If this value drops below a set threshold, the node and its links are removed from the graph. Producers are initialized with a volume of 1.0
Niche Score	A double precision number that determines which nodes may consume from this individual
Niche Lower/Upper Bound	A given node i may consume from another node j if the niche score of j is between the lower and upper niche bounds of i.

Additionally, all producers share some global properties defined in table 2:

Tab. 2: Global Producer Attributes:

Attribute Name:	**Attribute Effect:**
Metabolic Ratio = 0.005	A value on the interval $[0, 1)$ which is multiplied by a nodes volume and then removed from its volume on each time step.
Consumption Ratio = 0.01	A value on the interval $[0, 1)$ which is multiplied by a nodes volume. The node will attempt to consume the resulting amount of resource on the next time step.
Death Limit = 0.01	If any producer's volume falls below this value, it is removed from the graph.

Resource nodes have only the property of a niche score of zero, an index and a volume which was initialized to 10^8.

2.2 Initialization

In a warm up phase before the Euler integration begins, a single resource node is initiated with an index of zero. Then a seed population of nine producers are initialized. They are different from other producers only in that their niche low bound is set to zero and they are forcibly connected to the resource node.

2.3 Node Creation and Targeting

On each time step a check is first performed to see if the resource node still exists. If there is still a resource node, a number of new producers are created according to the function

$$ceil(population * growth\_rate)$$

where *population* is the number of producers currently in the graph. This ensures at least one node is created on each time step and after $\frac{population}{growth\_rate} > 1$ an exponential growth phase is entered. It is between this time step and the point at which the resource node is exhausted that the first set of interesting dynamics occurs. When a node i is initialized, a search is run over all nodes within the niche range of i. The node that exists within this range and has the largest value of the target score

$$target\_score_j = \frac{volume_j}{in\_degree_j}$$

is selected and a link is created from node i to node j. Parallel edges are not allowed, but anti-parallel edges are. On each time step, if there are less than $seed\_number = 9$ producer nodes attached to the resource (and the resource still exists), randomly selected producer nodes are forcibly connected to the resource node. This stabilizing feature tends to be activated infrequently and is only observed early on in the constant growth phase of warm up. It does not seem to effect long term dynamics significantly.

2.4 Producer Metabolism, Consumption and Re-Targeting

The second event to occur on each time step centers on resource consumption. The existing producers are randomly iterated through. For each selected node i the metabolic cost is computed as

$$metabolic\_cost_i = metabolic\_ratio * volume_i.$$

The metabolic cost is subtracted from the volume. If the resulting volume is less than the death limit the node and its edges are removed from the graph and a new node is selected. If the node survives the metabolic step, its consumption quota is computed as

$$consumption\_quota_i = consumption\_ratio * volume_i.$$

Node i attempts to remove $\frac{consumption\_quota}{target\_number}$ units of resource from the volume of each of the $target\_number$ nodes in its outgoing edge list. If a target has less than this amount of volume, however much resource the target has is moved into node i and the target is removed from the graph. If node i fails to consume its full resource quota it attempts to add a new outgoing edge according to the same targeting procedure described in Node Creation and Targeting.

3 Analyses

We ran a series of analyses on the developing network at every moment in time. The most basic where analyses of the degree distribution: the total number of nodes at each moment, the mean, variance, and standard deviations of the degree distribution at every moment, and the normalized Shannon entropy, which was calculated as:

$$H(G) = \frac{1}{N}[-\sum_{0}^{i} p_i log_2(p_i)]$$

where p_i is the probability of each degree and N is the number of nodes in the network. We also calculated graph properties of the network at each timestep, including the average clustering coefficient, the global and average local efficiencies, the algebraic connectivity (defined as the eigenvalue of the second eigenvector the graph Laplacian), and the average harmonic centrality.

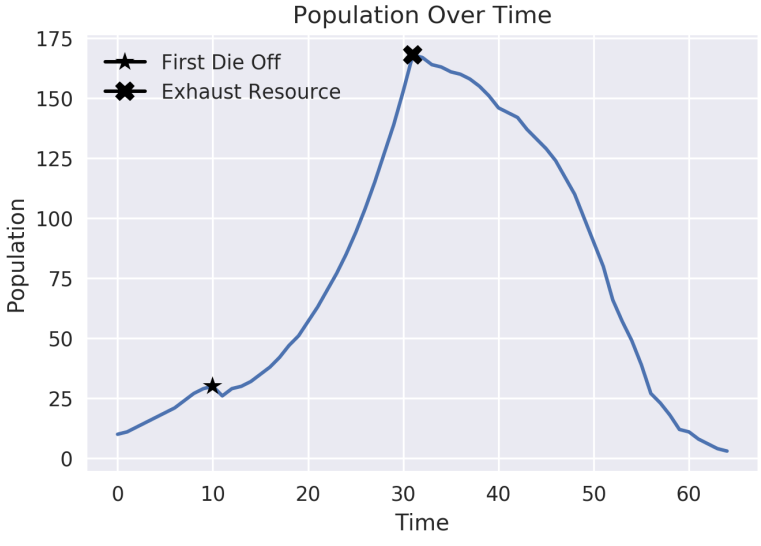
Finally, for each node, over the course of it's life, we tracked the average volume it consumed, the average local efficiency, the average harmonic centrality, and the average in- and out-degrees. This allowed us to estimate what sorts of relationships between nodes most allowed individuals to live the longest.

The simulation ran until either one of two conditions was met: there were fewer than 3 nodes left in the network, or the system had run for 100 timesteps. Typically, the network had collapsed down to <3 nodes by about 50 timesteps.

4 Results

4.1 Whole-Network Measures

The most basic network measure we tested was the population over time. For a visualization, see Figure 1 We found that the network displayed a very stereotyped evolution through time. There was typically a period of initial growth, which ended with a small die-off, or extinction event, which decreased the population. Following that, the population showed a smooth, exponential growth until the resource node was exhausted. After the resource node was exhausted, the network quickly began to collapse, showing an increasingly steep population collapse.



[H]

Fig. 1: The number of nodes in the network through time. Notice that there are two local maxima: an early die-off, indicated with a ⋆, and the global maxima, indicated with an **x**

.

To characterize the degree distribution, we calculated the mean and standard deviation of the degree distribution over time, as well as the variance, and the normalized Shannon's entropy. The mean and standard deviations, and the variance (Figure 2) showed similar changes over time to the population: a spike in mean degree and standard deviation near the early die-off, followed by a much shallower recovery that peaked when the network exhausted it's resource node. The variance showed a similar pattern, with a dramatic peak right before the die-off. Both the mean degree and the variance of the distribution began to become erratic as the network decayed.

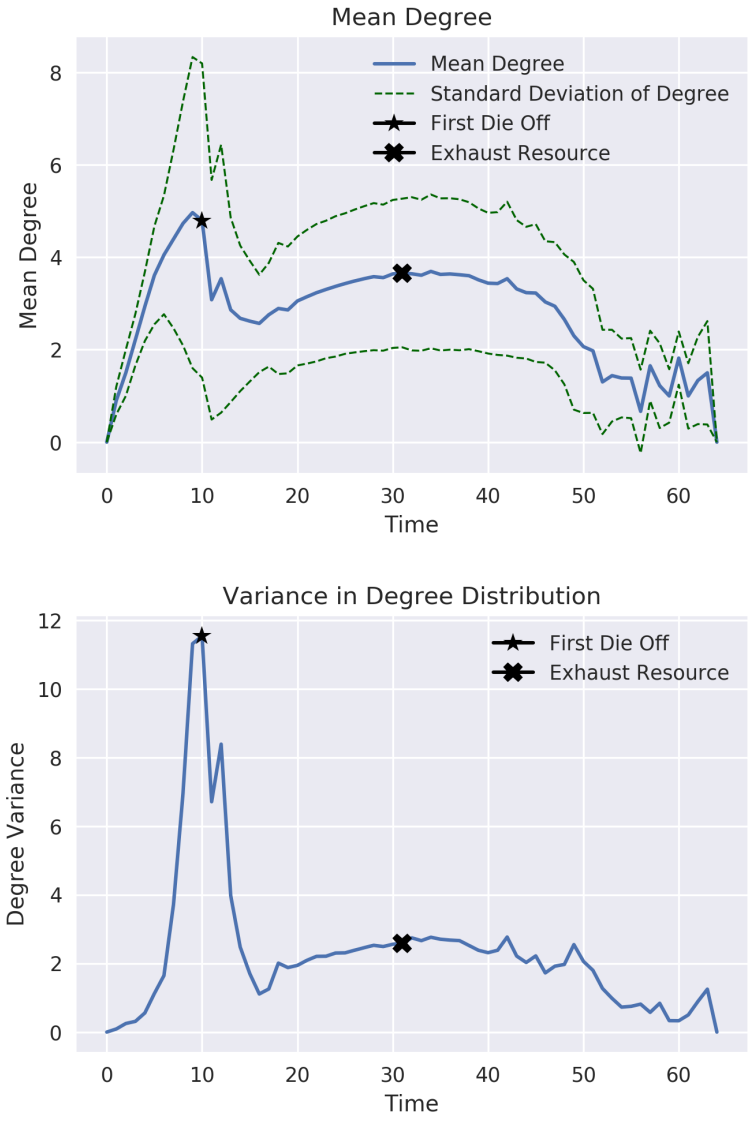


Fig. 2: The mean degree of the network at each timestep ± one standard deviation and the variance of the distrubiton at each moment. As with the population, two distinct local maxima are visible: both the mean degree and the variance in the degree distribution peak immediately before the early die-off and then collapse, before recovering to a smaller, broader peak at the moment of resource exhaustion.

The normalized Shannon entropy of the degree distribution showed a very similar pattern (see Figure

3), although with intriguing differences. Instead of climbing, as variance and mean degree do as the network develops, entropy falls, reaching a local maxima near the die-off, and a nadir when the resource node is exhausted. This is somewhat unexpected: we might interpret this to indicate that, near the peak of network development, the structure has a very 'lattice-like' topology, which most nodes connecting to similar numbers of other nodes. It is also possible that, as the normalization term increases with network size, that comes to dominate, dragging down the overall value. This finding strongly suggests that the dissipative process is not simply building a 'random' network through time: the dynamics of the process seems to constrain the structure of the network to a more globally ordered state. This is an excellent validation of the dissipative structure concept and implementationn.

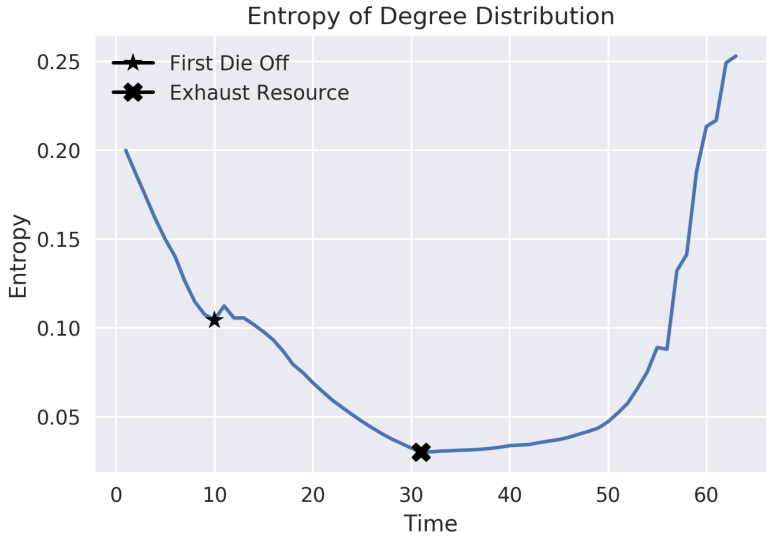


Fig. 3: The normalized Shannon entropy of the degree distribution over time. In contrast to other measures like variance or mean degree, normalized entropy falls as the network develops.

Over the course of the lifetime of the network, particularly after the resource node was exhausted, we were interested in how the network decayed. We had two competing hypothesis about how the network could collapse: Hypothesis 1 was that the network would fragment along weak ties leading to multiple, high-connected modules that would in turn fragment into nothing. Hypothesis 2 was that the network would collapse from the fringes in, maintaining a single connected component, which would loose periphery nodes over time. In Figure 4, we can see that, for most of the networks lifetime, the network is dominated by a single, giant component, which persists for quite a bit of time after the network has exhausted the resource node. There seems to be a critical point, however, where the network rapidly collapses into many disconnected components, that quickly decay into small chains of individual nodes or pairs. This suggests a mixture of both Hypothesis 1 and Hypothesis 2: for as long as the network is able to maintain a certain level of flux through it, a giant component can be maintained, however, once that level is lost, the network very quickly fragments.

As the network evolved and fragmented, we calculated the edge density, which was defined as the number of existing edges, divided by the number of possible edges. Interestingly, we found that the edge density was highest before the initial die-off, and towards the very end of the systems life (see Figure 5). At no point does the network become particularly dense, like most real-world networks, it remains

spares throughout. The finding that it was maximally sparse near the peak of network development is interesting, but not entirely unexpected, since nodes do not necessarily connect to more neighbours than is necessary to keep them alive. In the context of a natural system, this could be considered a form of specialization: most animals don't evolve to be able to consume more different species than they need to to survive. Similarly, in economic systems, firms will buy the smallest amount of product necessary to turn a profit on whatever they are producing. Sparsity is efficient.

We also looked at algebraic connectivity, local efficiency, global efficiency, and average harmonic centrality.

4.1.1 Correlating Measures Through Time

While many of these measures showed similar patterns (a maxima/minima near the initial die-off, a less dramatic peak near the moment of resource exhaustion), it was not immediately apparent how these various measures related to each-other. To explore this relational space, we created trajectories of the networks through phase-space, by plotting metrics at each moment against each-other. In some cases, clear, well-defined trajectories were visible, typically when comparing the normalized entropy to some other value. For instance, the relationship between the normalized entropy and mean of the degree distribution show a highly non-trivial pattern (Figure 8).

A very similar pattern was visible when plotting the variance of the degree distribution against the normalized entropy (Figure 9).

While entropy, variance, and mean degree are all direct functions of the degree distribution, and consequently might be expected to be more tightly related, entropy also showed clean trajectories when compared to some graph theoretic measures, such as algebraic connectivity (Figure 10). The exact nature of the relationship between entropy of the degree distribution and algebraic connectivity is unclear, although algebraic connectivity has been used as a measure of system complexity in fields such as computational neuroscience [5]. To test the relationship in a toy model, we created ensembles of Erdos-Renyi random graphs, of various sizes, and plotted entropy against algebraic connectivity, mean degree, and variance (see Figure 11). While in the null models similar looking trajectories could be seen, none were obvious parallels of what we saw in our evolving system, suggesting that, at least for these few measures, the observed behaviours are truly non-trivial.

4.2 Node Characteristics & Lifetimes

In addition to characterizing the global behaviour of the system, we were also interested in what local structures where conducive to nodes living longer or shorter lifespans relative to their neighbours. The simplest measure to test is node creation order: are the first nodes likely to live longer, or die earlier than nodes that come into being later? For visualization, see Figure 12. Interestingly, there seem to be two distinct 'groups' when comparing creation order to lifetime. There is a small set of nodes that are created very early (the seed population, and then a few subsequent nodes), that die very quickly. However, after the initial die-back has occurred, the network seems to find a stable structure, and subsequently, node creation order does not have an obvious effect on node lifetime (although the variance seems to climb as time goes on). This suggests that the die-back is part of a process by which the network reconfigures itself to achieve a more stable structure and dynamics.

Other measures that we explored where the average harmonic centrality of a node (which is a function

of how close it is to every other node, Figure 13) and lifetime, as well as the relationship between average in- and out-degrees and lifetime (Figure 14), and the volume that the node consumed (Figure 15). In all the explored relationships, a broad similar pattern was observed: there was a small group of nodes, created early in time, that had strikingly different average scores on whatever metric we were testing, separate from the vast majority of other nodes.

As we discussed above, we interpret this as indicating that the growth phase of the network can be divided into two epochs (recall that no new nodes are created after the resource node is exhausted): the first epoch is characterized by short-lived nodes that are configured in a relatively inefficient structure, possibly due to the initialization of the system. This unstable structure collapses during the initial die-off, and is subsequently replaced by a much more stable structure, which grows until the resource node is exhausted. Within this regime, the average behaviour of the node does not show significant correlation with life-time.

4.3 A Note on Null Models

Ideally, this kind of analysis would include some kind of null model, to ensure that the structures and behaviours we've observed are driven by the dissipative niche model itself and are not simply a general case of random nodes attaching themselves to the network in random orders. We struggled, however, to come up with a plausible null model that would fully capture our alternative hypothesis. The first null model we explored was, for each graph, creating an ensemble of Erdos-Renyi graphs,where the number of nodes is the same as the size of the 'real' network, and the connection probability is the edge density of the same. However, that does not capture the degree distribution, which we could not assume was binomial, or Poissonian in character. We then tried an ensemble of configuration models, running the same structural analysis. The primary problem with this model is that each moment in time is dependent on the structure of the graph at the moment before, so the connectivity and degree distributions seen in the null model are still a function of the dissipative process playing out at all previous time-steps (the ER null model shares the problem).

In future work on this project, we would want to create some kind of model that brings nodes into being randomly, attaches them to the graph randomly, and sever edges randomly, however, constructing such a model on top of our dissipative niche model proved beyond the scope of what we could accomplish here.

5 Conclusions

The lifetime of the network seems to be separable into approximately three distinct phases, characterized by distinct distributions of network properties and trajectories.

1. *Initial Emergence*:
2. *Stable Growth*: The phase begins following the end of the initial die-off. In this phase, the network size begins to grow exponentially
3. *Collapse*:

References

[1] Gavin E. Crooks. The Entropy Production Fluctuation Theorem and the Nonequilibrium Work Relation for Free Energy Differences. *Physical Review E*, 60(3):2721–2726, September 1999. arXiv: cond-mat/9901352.

[2] Steven H. Strogatz. Exploring complex networks. *Nature*, 410(6825):268–276, March 2001.

[3] Alexander J. Gates and Luis M. Rocha. Control of complex networks requires both structure and dynamics. *Scientific Reports*, 6(1), July 2016.

[4] Richard J. Williams and Neo D. Martinez. Simple rules yield complex food webs. *Nature*, 404(6774):180–183, 2000.

[5] Danielle S. Bassett, Andreas Meyer-Lindenberg, Sophie Achard, Thomas Duke, and Edward Bullmore. Adaptive reconfiguration of fractal small-world human brain functional networks. *Proceedings of the National Academy of Sciences*, 103(51):19518–19523, December 2006.

6 Appendix 1: Code

```
# -*- coding: utf-8 -*-
"""
End of the world simulator V.1.0
Python 3.7
By Sam Migirditch and Thomas Varley
samigir@iu.edu
04/10/2019
"""

# Imports
import statistics as stats
import matplotlib.pyplot as plt
import seaborn as sns
import networkx as nx
import numpy as np
import math as m
import scipy.stats as st
import scipy.signal as ss
import os
from datetime import datetime

#Misc path stuff
path = os.getcwd()
time = datetime.now()
target_dir = "run_" + str(time.hour) + ":" + str(time.minute) + "." + str(time.second) +
```

```
os.mkdir(path + "/" + target_dir)
os.mkdir(path + "/" + target_dir + "plots")

imgdir = path + "/" + target_dir + "plots/"

# Globals
## Safety triggers
popultion_limit = 10**8
time_limit = 10**2
## Experimental parameters

#rescource volume regeneration and metabolism controls

# Regen and metabolism are seperated because we'll probably want to look at non
#-linear metabolic/regen scaling effects at some future point.
source_initial_volume = 10**8
producer_initial_volume = 1
source_regen_ratio = 0
produce_regen_ratio = 0.0
consumer_regen_ratio = 0.0
source_metabolic_rate = 0.0
producer_metabolic_ratio = 0.005
#consumer_metabolic_ratio = 0.1
producer_consumption_ratio = 0.01
DEATH_LIMIT = 0.01

#niche controls
niche_creep_rate = 0.01 # rate of increase in niche dist. mean per # of nodes

#Network Growth Controls
consumer_delay_time=100#How many time steps to wait until consumers appear
producer_spawn_ratio= 0.1 # Comment out #spawn section of grind and set to cnst if you
producer_seed_number = 10

#Plotting controls
scale_factor = 200.0 # scales node volumes
plot_frequency = 1
savefigures = True #Toggle to not save figures.


#Global indicies and trakcers
max_niche_score = 0.0
niche_list = []
```

```
kill_list = []

# SVM 01/27: All node objects must be hashable for nice networkx features.

### Useful Functions
def set_niche_score_2():
    global max_niche_score
    moment = []
    total_volume = 0
    for node_index in G.nodes:
        role = G.node[node_index]["role"]
        #if role != "Producer": continue
        try:
            niche_score = G.node[node_index]["niche_score"]
            volume = G.node[node_index]["volume"]
            moment.append( volume * niche_score )
            total_volume += volume
        except: continue
    if len(moment)<2: return( niche_creep_rate,0,niche_creep_rate)
    sd = stats.stdev( moment )
    moment = stats.mean(moment)
    lb = moment - (niche_creep_rate * sd)
    lb = max(0, lb)
    ub = moment + ( niche_creep_rate * sd)
    ub = min( max_niche_score, ub)
    return(moment,lb,ub)

def set_niche_score(node_index):
    global max_niche_score
    niche_score = first_moment() + niche_creep_rate
    #print("SETTING NICHE:", niche_score)
    update_niche_stats
    niche_list.append(niche_score)
    return( niche_score )

def update_niche_list():
    global niche_list, max_niche_score
    niche_list = []
    for node_index in G.nodes:
        niche_score = G.node[node_index]["niche_score"]
        niche_list.append(niche_score)
        if (niche_score > max_niche_score):
            max_niche_score = niche_score
```

```
# For updating individual w/0 updating all
def update_niche_stats(node_index):
    global max_niche_score, niche_list
    niche_score = G.node[node_index]["niche_score"]
    if niche_score > max_niche_score: max_niche_score = niche_score


def create_source(node_index):
    #init node
    G.add_node(node_index)
    #set properties
    G.nodes[node_index]["node_index"]=node_index
    G.nodes[node_index]['role']="Source"
    G.nodes[node_index]["volume"]= source_initial_volume
    G.nodes[node_index]["niche_score"]= 0
    G.nodes[node_index]["niche_lb"]= 0
    G.nodes[node_index]["niche_ub"]= 0
    G.nodes[node_index]["metabolic_ratio"]= 0
    G.nodes[node_index]["consumption_ratio"]= 0
    G.nodes[node_index]["regen_ratio"]= 0.0
    update_niche_list()

def create_producer(node_index):
    #init node
    G.add_node(node_index)
    #set properties
    G.nodes[node_index]["node_index"]=node_index
    G.nodes[node_index]["role"]="Producer"
    G.nodes[node_index]["volume"]= producer_initial_volume
    niche, lb, ub = set_niche_score_2()
    G.nodes[node_index]["niche_score"]= niche
    G.nodes[node_index]["niche_ub"]= ub
    G.nodes[node_index]["niche_lb"]= 0.01*ub
    G.nodes[node_index]["metabolic_ratio"]= producer_metabolic_ratio
    G.nodes[node_index]["consumption_ratio"]= producer_consumption_ratio/(ub)
    G.nodes[node_index]["regen_ratio"]= 0.0
    #update trackers
    update_niche_list()
    #find targets
    find_target( node_index )

def find_target( node_index ):
```

```
    ub = G.node[node_index]["niche_ub"]
    lb = G.node[node_index]["niche_lb"]
    best_score = 0.0
    best = -1
    possible_targets = list(G.nodes())
    np.random.shuffle(possible_targets)
    for target in possible_targets:
        # don't make parallel or self loops
        blacklist = [node_index]
        edges = G.out_edges(node_index)
        blacklist.extend(edges)
        if target not in blacklist:
            target_niche_score = G.node[target]["niche_score"]
            if ( lb<=target_niche_score<=ub ):
                target_volume = G.node[target]["volume"]
                target_degree = G.degree(target)
                target_score = target_volume / max(1,target_degree)
                if ( target_score > best_score ):
                    best_score = target_score
                    best = target
    if (best_score > 0):
        G.add_edge(node_index,best)
    #else: print( "EDGE:",node_index," failed to find target")

def force_connect_source(target):
    if G.node[target]["role"]=="Source": return
    if 0 not in G.nodes: return
    G.add_edge(target,0)

def kill_node( node_index ):
    #global niche_array
    #niche_score = G.node[node_index]["niche_score"]
    G.remove_node( node_index )
    #niche_array.remove(niche_score)

def remove_isolates():
    global kill_list
    isolates = nx.isolates(G)
    kill_list.extend(isolates)

def run_kill_list():
    global kill_list
    #remove duplicates
```

```
    kill_list = list(set(kill_list))
    for k in kill_list:
        kill_node(k)
    kill_list = []

def do_producer_step(node):
    global kill_list
    node_index = G.node[node]["node_index"]
    #iche_score = G.node[node]["niche_score"]

    targets = [ t for t in G.neighbors(node) ]
    target_number = len( targets )

    node_volume = G.node[node]["volume"]
    node_quota = G.node[node]["consumption_ratio"] * node_volume
    node_metabolism = G.node[node]["metabolic_ratio"] * node_volume


    # Die if starved
    if (node_volume<=DEATH_LIMIT):
        #print( "t:",t,"node:", node,"fate: 0 starved\n")
        kill_list.append(node_index)
        return()

    # Hunger
    if (target_number>0):
        per_capita_quota = node_volume / target_number
        node_volume-=node_metabolism
        intake = 0.0
        for target in targets:
            target_index = G.node[target]["node_index"]
            target_volume = G.node[target]["volume"]
            if (per_capita_quota >= target_volume):
                intake += target_volume
                kill_list.append( target_index )
                #print( "t:",t,"node:", node,"killed",target,"\n")
            else:
                intake += per_capita_quota
                target_volume += -per_capita_quota
                G.node[target]["volume"] = target_volume
        # eat gathered rescource
        G.node[node]["volume"] += intake
        # serch for new targets if under quota
```

```
        if (intake < node_quota):
            find_target( node_index )
    else: find_target(node_index)
    G.node[node_index]["volume"] -= node_metabolism


def change_niche_score(node_index, new_score):
    global max_niche_score, niche_list
    #swap
    old_score = G.node[node_index]["niche_score"]
    G.node[node_index]["niche_score"] = new_score
    #update stats
    niche_list.remove(old_score)
    niche_list.append(new_score)
    if (new_score > max_niche_score):
        max_niche_score = new_score


def plotter(target_dir, show = True):
    pos=nx.spring_layout(G) # positions for all nodes
    labels={}
    max_volume = 0.0
    for node in G.nodes:
        role = G.node[node]["role"]
        if(role=="Source"): continue
        volume = G.node[node]["volume"]
        if volume>max_volume:
            max_volume = volume

# Draw nodes
    for node_index in G.nodes:
        # Type-> color
        role = G.node[node_index]["role"]
        if (role=="Source"):
            color = "green"
        elif (role=="Producer"):
            color = "cornflowerblue"
        elif (role=="consumer"):
            color = "red"
        else:
            color = "black"
        # volume -> size
        volume = scale_factor*min(1,G.node[node_index]["volume"] / max_volume )
```

```
        #Draw node
        nx.draw_networkx_nodes(G,pos,
                               nodelist=[node_index],
                               node_color= color,
                               node_size=volume,
                           alpha=0.8)
        # draw edges
        edges = G.out_edges(node_index)
        nx.draw_networkx_edges(G,pos,
                               edgelist=edges,
                               width=1,
                               alpha=0.5,
                               edge_color=color)

        # some math labels
        labels[node_index]=node_index

    nx.draw_networkx_labels(G,pos,labels,font_size=16)

    plt.axis('off')
    local_now = datetime.now()
    #plt.savefig(target_dir + str(local_now) + "_graph.png") # save as png

    plt.show() # display
    #nx.write_gexf(G, target_dir + str(local_now) + "_graph.gexf")

### Script
# Init the world
G = nx.DiGraph()

# Create source node & some seed producers
create_source(0)

for t in range(1, producer_seed_number):
    create_producer(t)
    #force into source niche range
    G.node[t]["niche_lb"]=0.0
    G.node[t]["niche_ub"]=niche_creep_rate
    change_niche_score(t,0.1*niche_creep_rate)

### GRIND
```

```
run_condition=1
t=0
index_max = producer_seed_number

#Various graph property time-series
num_producers = []
num_nodes = []
size_source = []
total_volumes = []
min_niche_lb = []
max_niche_ub = []
num_components = []
global_eff = []
local_eff = []
mean_degree = []
std_degree = []
var_degree = []
ent_degree = []
density = []
largest_comp = []
avg_clustering_coeff = []
avg_harmonic_centrality = []
alg_conn = []
alg_conn_norm = []

node_lifetime = {}
avg_node_clustering_coeff = {}
avg_node_harmonic_centrality = {}
avg_in_degree_centrality = {}
avg_out_degree_centrality = {}
avg_page_rank = {}
avg_volume = {}

ticker = 0

while (run_condition):

    # Update State Variables
    num_producers.append(len([x for x in G.nodes() if G.node[x]["role"] == "Producer"]))
    num_nodes.append(len(G.nodes()))

    undir_G = nx.to_undirected(G)
```

```
if 0 in G.nodes:
    size_source.append(G.node[0]["volume"])


#Density of the graph
prob = len(G.edges) / ((len(G.nodes)*(len(G.nodes)-1))/2)
density.append(prob)

#Useful for finding peaks near 10 and when the resource runs out
peaks = ss.find_peaks(num_nodes)[0]

#Get volumes of each node as dict
volumes = nx.get_node_attributes(G, "volume")

#Dictionaries of graph properties
clustering_coeff = nx.clustering(G)
harmonic_centrality = nx.harmonic_centrality(G)
degs = nx.degree(G)
in_deg_cent = nx.in_degree_centrality(G)
out_deg_cent = nx.out_degree_centrality(G)

#In and out degree seqs
in_deg_sequence = list(x[1] for x in G.in_degree())
out_deg_sequence = list(x[1] for x in G.out_degree())

config_global_eff_list, config_local_eff_list, config_avg_clustering_coeff_list, =
config_avg_harmonic_centrality_list, config_alg_conn_list = [], []

#Making configuraiton models
"""
for i in range(1):
    config_G = nx.directed_configuration_model(in_deg_sequence, out_deg_sequence)
    undir_config_G = nx.Graph(nx.to_undirected(config_G))

    config_global_eff_list.append(nx.global_efficiency(undir_config_G))
    config_local_eff_list.append(nx.local_efficiency(undir_config_G))
    config_avg_clustering_coeff_list.append(nx.average_clustering(undir_config_G))
    config_avg_harmonic_centrality_list.append(config_G)
    config_alg_conn_list.append(undir_config_G)
"""
#Calculating average config statistics for summary stats.
config_global_eff = 1#np.mean(config_global_eff_list)
config_local_eff = 1#np.mean(config_local_eff_list)
```

```
config_avg_clustering_coeff = 1#np.mean(config_avg_clustering_coeff_list)
config_avg_harmonic_centrality = 1#np.mean(config_avg_harmonic_centrality_list)
config_alg_conn = 1#np.mean(config_alg_conn_list)

#Calculating general model descriptors
total_volumes.append(np.sum([G.node[i]["volume"] for i in G.nodes() if i != 0]))
min_niche_lb.append(min([G.node[i]["niche_lb"] for i in G.nodes() ]))
max_niche_ub.append(max([G.node[i]["niche_ub"] for i in G.nodes() ]))

#Degree-distrubtion-specific graph measures.
mean_degree.append(np.mean([x[1] for x in degs if x[0] != 0]))
std_degree.append(np.std([x[1] for x in degs if x[0] != 0]))
var_degree.append(np.var([x[1] for x in degs if x[0] != 0]))
ent_degree.append(st.entropy([x[1] for x in degs if x[0] != 0])/len(G))
num_components.append(len(list(nx.connected_component_subgraphs(undir_G))))
largest_comp.append(len(max(nx.connected_component_subgraphs(undir_G), key=len)) / l

#Measures normalized by config_models.
global_eff.append(nx.global_efficiency(undir_G) / config_global_eff)
local_eff.append(nx.local_efficiency(undir_G) / config_local_eff)
avg_clustering_coeff.append(nx.average_clustering(G) / config_avg_clustering_coeff)
avg_harmonic_centrality.append(np.mean([harmonic_centrality[i] for i in G.nodes()]))
alg_conn_norm.append(nx.algebraic_connectivity(undir_G) / config_alg_conn)
alg_conn.append(nx.algebraic_connectivity(undir_G))

for i in G.nodes():
    if i != 0:

        if i in node_lifetime:
            node_lifetime[i] += 1
        elif i not in node_lifetime:
            node_lifetime[i] = 1

        if i in avg_node_clustering_coeff:
            avg_node_clustering_coeff[i] = avg_node_clustering_coeff[i] + (clusterin
        elif i not in avg_node_clustering_coeff:
            avg_node_clustering_coeff[i] = clustering_coeff[i]

        if i in avg_node_harmonic_centrality:
            avg_node_harmonic_centrality[i] = avg_node_harmonic_centrality[i] + (ha
        elif i not in avg_node_harmonic_centrality:
            avg_node_harmonic_centrality[i] = harmonic_centrality[i]
```

```
            if i in avg_in_degree_centrality:
                avg_in_degree_centrality[i] = avg_in_degree_centrality[i] + (in_deg_cent
            elif i not in avg_in_degree_centrality:
                avg_in_degree_centrality[i] = in_deg_cent[i]

            if i in avg_out_degree_centrality:
                avg_out_degree_centrality[i] = avg_out_degree_centrality[i] + (out_deg_c
            elif i not in avg_out_degree_centrality:
                avg_out_degree_centrality[i] = out_deg_cent[i]

            if i in avg_volume:
                avg_volume[i] = avg_volume[i] + (volumes[i] - avg_volume[i])/ticker
            elif i not in avg_volume:
                avg_volume[i] = volumes[i]

    ticker += 1

    t+=1
    population = list( G.nodes() )
    population_size = len(population)

    # run through and-listed run conditions
    run_condition *= t<time_limit
    run_condition *= population_size < popultion_limit
    run_condition *= population_size > 3
    #print( "TIME:", t,"| POPULATION:", population_size,"|",population)


    update_list = list(G.nodes())
    np.random.shuffle(update_list)
    for node in update_list:
        if(G.node[node]["role"] == "Producer"):
            update_niche_stats(node)
            do_producer_step(node)

    # Reap nodes
    #remove_isolates()
    run_kill_list()
    #plot
    if t%plot_frequency ==0:
        plotter(target_dir)
    # spawn Nodes
```

```
    #skip spawn if rescource has died
    if 0 not in G.nodes: continue
    # force connect random node to seed node if disconnected
    random_nodes = list(G.nodes())
    np.random.shuffle(random_nodes)
    condition = nx.degree(G)[0] <= producer_seed_number
    while condition > 0:
        target = random_nodes.pop(0)
        force_connect_source(target)
        cond_1 = nx.degree(G)[0] <= producer_seed_number
        cond_2 = len(random_nodes) > 0
        condition = cond_1 * cond_2


    spawn_number = m.ceil(population_size*producer_spawn_ratio)
    for spawn in range(0,spawn_number):
        create_producer(index_max)
        find_target(index_max)
        index_max +=1


timeline = [x for x in range(ticker)]

sns.set(style = "darkgrid")


plt.subplots()
plt.plot(size_source)
plt.xlabel("Time")
plt.ylabel("Source Volume")
plt.title("Source Volume Over Time")
if savefigures == True:
    plt.savefig(imgdir + "volume_time.png", dpi = 250)

plt.subplots()
plt.plot(min_niche_lb, label = "Min Niche Score")
plt.plot(max_niche_ub, label = "Max Niche Score")
plt.xlabel("Time")
plt.ylabel("Min/Max Niche Score")
plt.yscale("log")
plt.title("Niche Range")
plt.legend()
if savefigures == True:
```

```
    plt.savefig(imgdir + "niche.png", dpi = 250)

plt.subplots()
plt.plot(num_nodes)
plt.plot(timeline[peaks[0]], num_nodes[peaks[0]],
         marker="*",
         color="black",
         markersize="10",
         label="First Die Off")
plt.plot(timeline[peaks[1]], num_nodes[peaks[1]],
         marker="X",
         color="black",
         markersize="10",
         label="Exhaust Resource")
plt.xlabel("Time")
plt.ylabel("Population")
plt.title("Population Over Time")
plt.legend()
if savefigures == True:
    plt.savefig(imgdir + "population.png", dpi = 250)


plt.subplots()
plt.plot(num_components)
plt.plot(timeline[peaks[0]], num_components[peaks[0]],
         marker="*",
         color="black",
         markersize="10",
         label="First Die Off")
plt.plot(timeline[peaks[1]], num_components[peaks[1]],
         marker="X",
         color="black",
         markersize="10",
         label="Exhaust Resource")
plt.xlabel("Time")
plt.ylabel("Count")
plt.title("Number of Connected Components")
if savefigures == True:
    plt.savefig(imgdir + "num_components.png", dpi = 250)

plt.subplots()
plt.plot(largest_comp)
plt.plot(timeline[peaks[0]], largest_comp[peaks[0]],
```

```
            marker="*",
            color="black",
            markersize="10",
            label="First Die Off")
plt.plot(timeline[peaks[1]], largest_comp[peaks[1]],
            marker="X",
            color="black",
            markersize="10",
            label="Exhaust Resource")
plt.xlabel("Time")
plt.ylabel("Largest Component Percentage")
plt.title("Largest Component")
if savefigures == True:
    plt.savefig(imgdir + "largest_component.png", dpi = 250)

plt.subplots()
plt.plot(global_eff)
plt.plot(timeline[peaks[0]], global_eff[peaks[0]],
            marker="*",
            color="black",
            markersize="10",
            label="First Die Off")
plt.plot(timeline[peaks[1]], global_eff[peaks[1]],
            marker="X",
            color="black",
            markersize="10",
            label="Exhaust Resource")
plt.xlabel("Time")
plt.ylabel("Global Efficiency")
plt.title("Global Efficiency")# (Normalized w/ Config Model)")
plt.legend()
if savefigures == True:
    plt.savefig(imgdir + "global_efficiency.png", dpi = 250)

plt.subplots()
plt.plot(local_eff)
plt.plot(timeline[peaks[0]], local_eff[peaks[0]],
            marker="*",
            color="black",
            markersize="10",
            label="First Die Off")
plt.plot(timeline[peaks[1]], local_eff[peaks[1]],
            marker="X",
```

```
            color="black",
            markersize="10",
            label="Exhaust Resource")
plt.xlabel("Time")
plt.ylabel("Local Efficiency")
plt.title("Local Efficiency")# (Normalized w/ Config Model)")
plt.legend()
if savefigures == True:
    plt.savefig(imgdir + "local_efficiency.png", dpi = 250)

plt.subplots()
plt.plot(avg_harmonic_centrality)
plt.plot(timeline[peaks[0]], avg_harmonic_centrality[peaks[0]],
            marker="*",
            color="black",
            markersize="10",
            label="First Die Off")
plt.plot(timeline[peaks[1]], avg_harmonic_centrality[peaks[1]],
            marker="X",
            color="black",
            markersize="10",
            label="Exhaust Resource")
plt.xlabel("Time")
plt.ylabel("Average Harmonic Centrality")
plt.title("Average Harmonic Centrality")# (Normalized w/ Config Model)")
plt.legend()
if savefigures == True:
    plt.savefig(imgdir + "harmonic_centrality.png", dpi = 250)

plt.subplots()
plt.plot(avg_clustering_coeff)
plt.plot(timeline[peaks[0]], avg_clustering_coeff[peaks[0]],
            marker="*",
            color="black",
            markersize="10",
            label="First Die Off")
plt.plot(timeline[peaks[1]], avg_clustering_coeff[peaks[1]],
            marker="X",
            color="black",
            markersize="10",
            label="Exhaust Resource")
plt.xlabel("Time")
plt.ylabel("Average Clustering Coefficint")
```

```
plt.title("Average Clustering Coefficient")# (Normalized w/ Config Model)")
plt.legend()
if savefigures == True:
    plt.savefig(imgdir + "clustering_coeff.png", dpi = 250)

plt.subplots()
plt.plot(mean_degree, label = "Mean Degree")
plt.plot([mean_degree[x] + std_degree[x] for x in range(len(mean_degree))],
          color = "darkgreen",
          linestyle = "--",
          linewidth = 1,
          label = "Standard Deviation of Degree")
plt.plot([mean_degree[x] - std_degree[x] for x in range(len(mean_degree))],
          color = "darkgreen",
          linestyle = "--",
          linewidth = 1)
plt.plot(timeline[peaks[0]], mean_degree[peaks[0]],
         marker="*",
         color="black",
         markersize="10",
         label="First Die Off")
plt.plot(timeline[peaks[1]], mean_degree[peaks[1]],
         marker="X",
         color="black",
         markersize="10",
         label="Exhaust Resource")
plt.xlabel("Time")
plt.ylabel("Mean Degree")
plt.title("Mean Degree")
plt.legend()
if savefigures == True:
    plt.savefig(imgdir + "mean_std_degree.png", dpi = 250)

plt.subplots()
plt.plot(var_degree)
plt.plot(timeline[peaks[0]], var_degree[peaks[0]],
         marker="*",
         color="black",
         markersize="10",
         label="First Die Off")
plt.plot(timeline[peaks[1]], var_degree[peaks[1]],
         marker="X",
         color="black",
```

```
            markersize="10",
            label="Exhaust Resource")
plt.xlabel("Time")
plt.ylabel("Degree Variance")
plt.title("Variance in Degree Distribution")
plt.legend()
if savefigures == True:
    plt.savefig(imgdir + "variance_degree.png", dpi = 250)

plt.subplots()
plt.plot(ent_degree)
plt.plot(timeline[peaks[0]], ent_degree[peaks[0]],
            marker="*",
            color="black",
            markersize="10",
            label="First Die Off")
plt.plot(timeline[peaks[1]], ent_degree[peaks[1]],
            marker="X",
            color="black",
            markersize="10",
            label="Exhaust Resource")
plt.xlabel("Time")
plt.ylabel("Entropy")
plt.title("Entropy of Degree Distribution")
plt.legend()
if savefigures == True:
    plt.savefig(imgdir + "entropy_degree.png", dpi = 250)

plt.subplots()
plt.plot(density)
plt.plot(timeline[peaks[0]], density[peaks[0]],
            marker="*",
            color="black",
            markersize="10",
            label="First Die Off")
plt.plot(timeline[peaks[1]], density[peaks[1]],
            marker="X",
            color="black",
            markersize="10",
            label="Exhaust Resource")
plt.xlabel("Time")
plt.ylabel("Density")
plt.title("Graph Density Over Time")
```

```
plt.legend()
if savefigures == True:
    plt.savefig(imgdir + "density.png", dpi = 250)

plt.subplots()
plt.scatter(list(node_lifetime.keys()), list(node_lifetime.values()),
            c = list(node_lifetime.keys()), cmap = "winter", label = "Creation Order")
plt.xlabel("Node Creation Order")
plt.ylabel("Node Lifetime")
plt.title("Creation Order vs. Node Lifetime")
plt.colorbar(label = "Creation Order")
if savefigures == True:
    plt.savefig(imgdir + "creation_order_lifetimes.png", dpi = 250)

plt.subplots()
plt.scatter(list(avg_node_clustering_coeff.values()), list(node_lifetime.values()),
            c = list(node_lifetime.keys()), cmap = "winter", label = "Creation Order")
plt.xlabel("Average Clustering Coefficient")
plt.ylabel("Node Lifetime")
plt.title("Clustering Coefficient vs. Node Lifetime")
plt.colorbar(label = "Creation Order")
if savefigures == True:
    plt.savefig(imgdir + "clustering_lifetimes.png", dpi = 250)

plt.subplots()
plt.scatter(list(avg_node_harmonic_centrality.values()), list(node_lifetime.values()),
            c = list(node_lifetime.keys()), cmap = "winter", label = "Creation Order")
plt.xlabel("Average Harmonic Centrality")
plt.ylabel("Node Lifetime")
plt.title("Harmonic Centrality vs. Node Lifetime")
plt.colorbar(label = "Creation Order")
if savefigures == True:
    plt.savefig(imgdir + "harmonic_lifetimes.png", dpi = 250)

plt.subplots()
plt.scatter(list(avg_in_degree_centrality.values()), list(node_lifetime.values()),
            c = list(node_lifetime.keys()), cmap = "winter", label = "Creation Order")
plt.xlabel("Average In-Degree Centrality")
plt.ylabel("Node Lifetime")
plt.title("In-Degree Centrality vs. Node Lifetime")
plt.colorbar(label = "Creation Order")
if savefigures == True:
    plt.savefig(imgdir + "in_degree_lifetimes.png", dpi = 250)
```

```
plt.subplots()
plt.scatter(list(avg_out_degree_centrality.values()), list(node_lifetime.values()),
            c = list(node_lifetime.keys()), cmap = "winter", label = "Creation Order")
plt.xlabel("Average Out-Degree Centrality")
plt.ylabel("Node Lifetime")
plt.title("Out-Degree Centrality vs. Node Lifetime")
plt.colorbar(label = "Creation Order")
if savefigures == True:
    plt.savefig(imgdir + "out_degree_lifetimes.png", dpi = 250)

plt.subplots()
plt.scatter(list(avg_volume.values()), list(node_lifetime.values()),
            c = list(node_lifetime.keys()), cmap = "winter", label = "Creation Order")
plt.xlabel("Average Node Volume")
plt.ylabel("Node Lifetime")
plt.colorbar(label = "Creation Order")
plt.title("Average Node Volume vs. Node Lifetime")
if savefigures == True:
    plt.savefig(imgdir + "volume_lifetimes.png", dpi = 250)

plt.subplots()
plt.plot(alg_conn)
plt.plot(timeline[peaks[0]], alg_conn_norm[peaks[0]],
         marker="*",
         color="black",
         markersize="10",
         label="First Die Off")
plt.plot(timeline[peaks[1]], alg_conn_norm[peaks[1]],
         marker="X",
         color="black",
         markersize="10",
         label="Exhaust Resource")
plt.xlabel("Time")
plt.ylabel("Algebraic Connectivity")
plt.title("Algebraic Connectivity")#(Normalized w/ Config Model)")
plt.legend()
#plt.ylim([-0.01,0.25])
if savefigures == True:
    plt.savefig(imgdir + "alg_conn.png", dpi = 250)

plt.subplots()
plt.scatter(ent_degree, var_degree, c = timeline, cmap = "autumn_r")
```

```
plt.plot(ent_degree[peaks[0]], var_degree[peaks[0]],
         marker="*",
         color = "black",
         markersize = 10, label = "First Die-Off")
plt.plot(ent_degree[peaks[1]], var_degree[peaks[1]],
         marker="X",
         color = "black",
         markersize = 10, label = "Exhaust Resource")
plt.xlabel("Entropy of Degree Distribution")
plt.ylabel("Variance in Degree Distribution")
plt.title("Entropy vs. Variance Over Time")
plt.colorbar(label = "Time" )
plt.legend()
if savefigures == True:
    plt.savefig(imgdir + "ent_var_degree.png", dpi = 250)

plt.subplots()
plt.plot(ent_degree[peaks[0]], alg_conn[peaks[0]],
         marker="*",
         color = "black",
         markersize = 10, label = "First Die-Off")
plt.plot(ent_degree[peaks[1]], alg_conn[peaks[1]],
         marker="X",
         color = "black",
         markersize = 10, label = "Exhaust Resource")
plt.scatter(ent_degree, alg_conn, c = timeline, cmap = "autumn_r")
plt.xlabel("Entropy of Degree Distribution")
plt.ylabel("Algebraic Connectivity")
plt.title("Entropy vs. Algebraic Connectivity Over Time")
plt.colorbar(label = "Time" )
plt.legend()
if savefigures == True:
    plt.savefig(imgdir + "ent_alg_conn.png", dpi = 250)

plt.subplots()
plt.plot(var_degree[peaks[0]], alg_conn[peaks[0]],
         marker="*",
         color = "black",
         markersize = 10, label = "First Die-Off")
plt.plot(var_degree[peaks[1]], alg_conn[peaks[1]],
         marker="X",
         color = "black",
         markersize = 10, label = "Exhaust Resource")
```

```
plt.scatter(var_degree, alg_conn, c = timeline, cmap = "autumn_r")
plt.xlabel("Variance of Degree Distribution")
plt.ylabel("Algebraic Connectivity")
plt.title("Variance vs. Algebraic Connectivity Over Time")
plt.colorbar(label = "Time" )
plt.legend()
if savefigures == True:
    plt.savefig(imgdir + "var_alg_conn.png", dpi = 250)

plt.subplots()
plt.plot(ent_degree[peaks[0]], mean_degree[peaks[0]],
         marker="*",
         color = "black",
         markersize = 10, label = "First Die-Off")
plt.plot(ent_degree[peaks[1]], mean_degree[peaks[1]],
         marker="X",
         color = "black",
         markersize = 10, label = "Exhaust Resource")
plt.scatter(ent_degree, mean_degree, c = timeline, cmap = "autumn_r")
plt.xlabel("Entropy of Degree Distribution")
plt.ylabel("Mean Degree")
plt.title("Entropy vs. Mean Degree Over Time")
plt.colorbar(label = "Time" )
plt.legend()
if savefigures == True:
    plt.savefig(imgdir + "ent_mean_degree.png", dpi = 250)

plt.subplots()
plt.plot(alg_conn[peaks[0]], mean_degree[peaks[0]],
         marker="*",
         color = "black",
         markersize = 10, label = "First Die-Off")
plt.plot(alg_conn[peaks[1]], mean_degree[peaks[1]],
         marker="X",
         color = "black",
         markersize = 10, label = "Exhaust Resource")
plt.scatter(alg_conn, mean_degree, c = timeline, cmap = "autumn_r")
plt.xlabel("Algebraic Connectivity")
plt.ylabel("Mean Degree")
plt.title("Algebraic Connectivity vs. Mean Degree Over Time")
plt.colorbar(label = "Time" )
plt.legend()
if savefigures == True:
```

```
plt.savefig(imgdir + "alg_conn_mean_degree.png", dpi = 250)
```

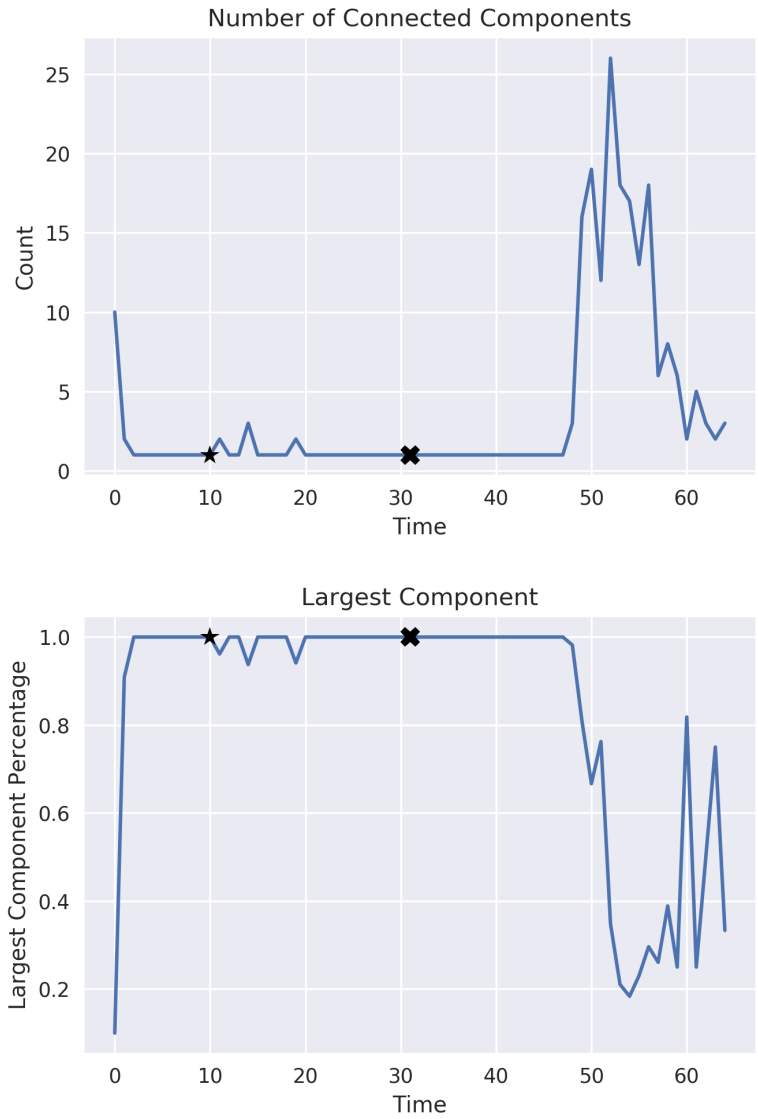


Fig. 4: The number of connected components in the network over time, and the fraction of the total number of nodes contained in the largest component over time. In the upper plot, we can see that the network maintains a single, core component throughout most of it's life (the small bumps between time 10 and 20 are new nodes that have been added, but not yet connected to the main core). The network remains a single, connected component until what seems to be a critical point near time 50, at which point it quickly fragments into many smaller components, few of which dominate the set the available nodes.

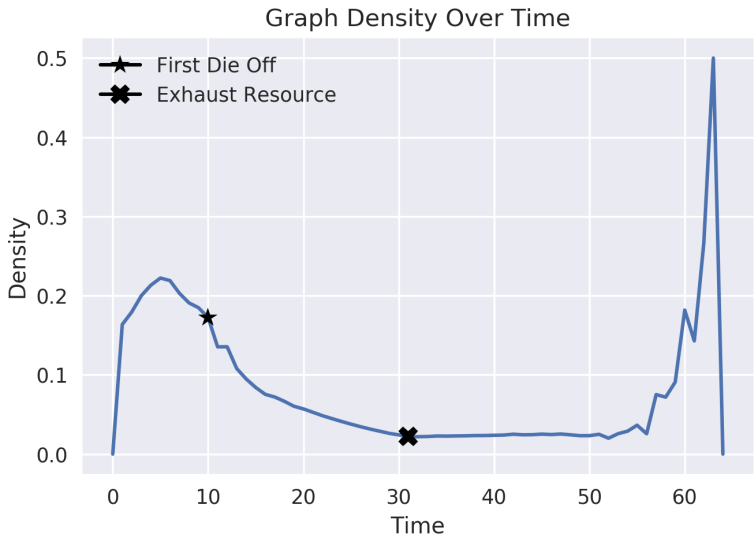


Fig. 5: The density of the graph, defined as the number of existing edges, normalized by the total number of possible edges. The spike in density before the initial die-off was unexpected, although the later spike is more easily explained, as the number of nodes in the network shrinks to nothing, so the number of possible edges does as well.

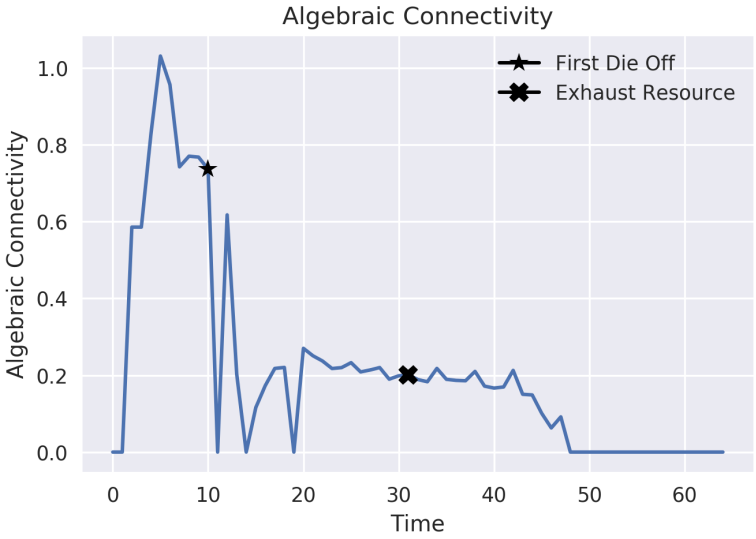


Fig. 6: The algebraic connectivity of the graph over time. Algebraic connectivity is greater than 0 only when the graph is fully connected, so any moment where the graph is not, the plot drops to zero. The moments between time 10 and 20 where the plot drops match the brief moments of disconnection that can be seen in Figure 4. Despite these brief drops, an overall pattern is clearly visible: algebraic connectivity spikes just prior to the initial die-off, and then falls, slowing as the graph reaches it's maximal size. Most interestingly, algebraic connectivity begins to drop more precipitously right before the graph becomes fully disconnected, suggesting that it may predict total collapse slightly before it begins.

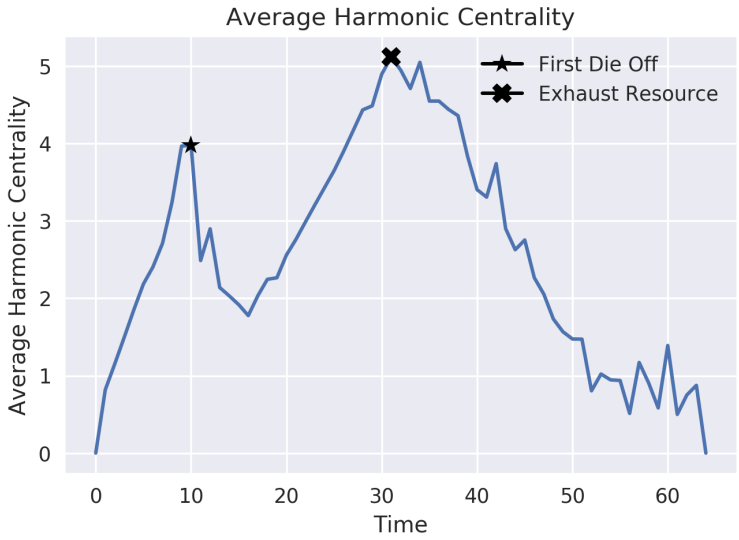
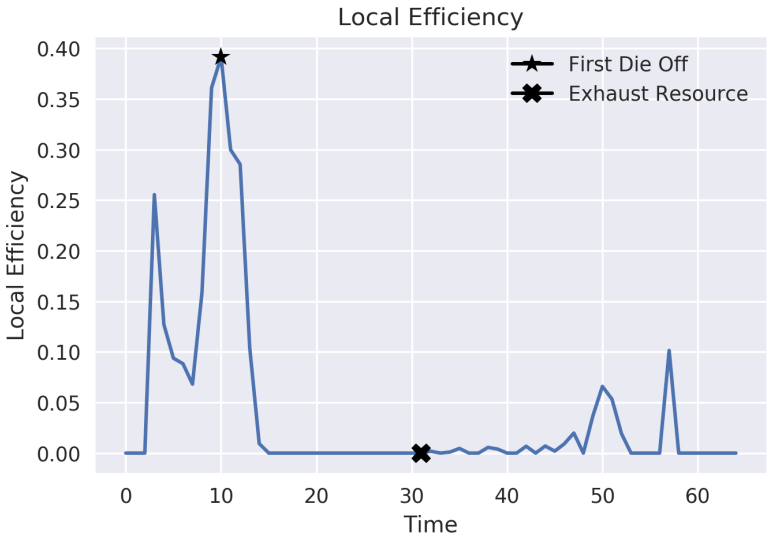
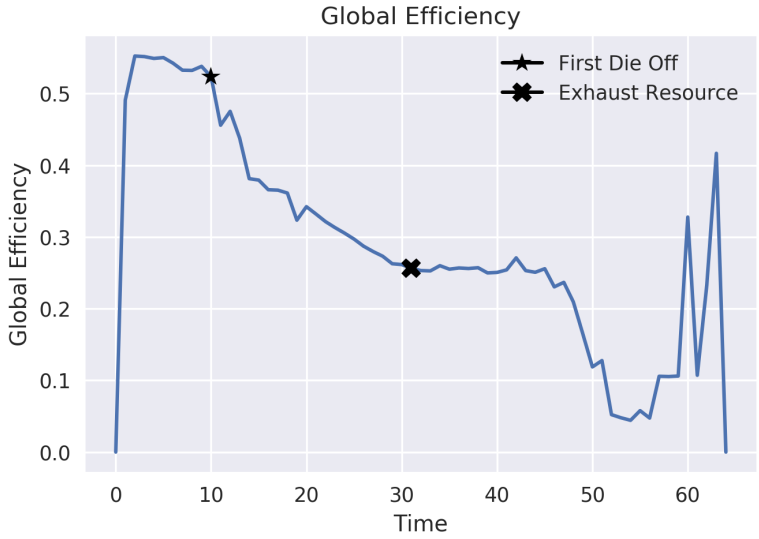

Average Harmonic Centrality
First Die Off
Exhaust Resource
Average Harmonic Centrality
Time
0
1
2
3
4
5
0
10
20
30
40
50
60

Fig. 7: text





Not sure what to say about this one

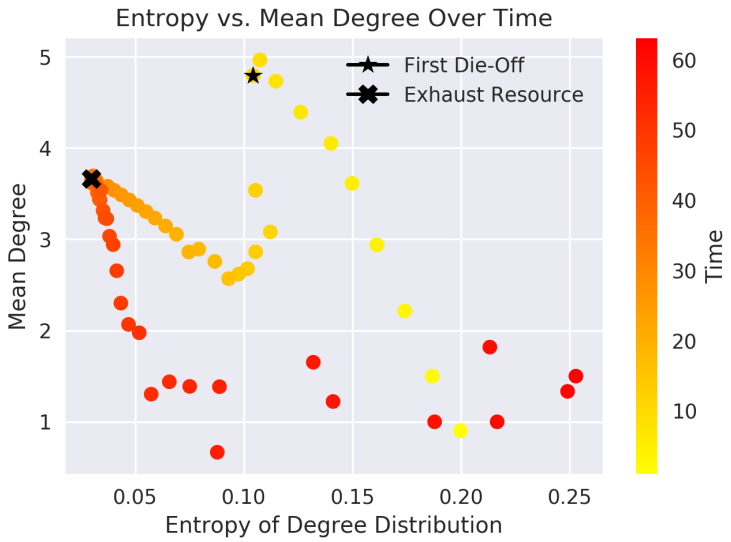


Fig. 8: At every moment in time, the mean degree of the network, plotted against the normalized entropy of the degree distribution. The pattern begins with the most saturated yellow points, and transitions through orange to red. Note the two spikes, or sharp turns, at each of the key events, the die-off and the exhaustion of the resource nodes.

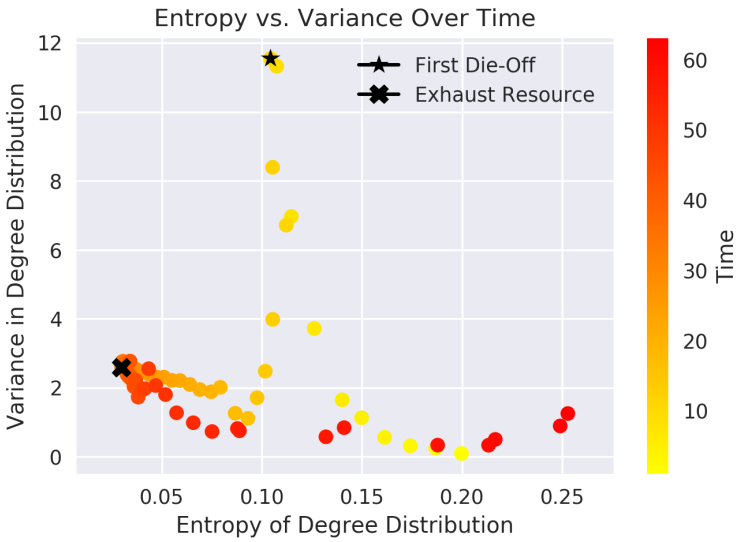


Fig. 9: The variance of the degree distribution at each moment in time plotted as a function of the entropy of the degree distribution. Here, two competing dynamics are visible: variance spikes dramatically near the initial die-off, while entropy is less affected and continues to drift downwards, until the resource node is exhausted, at which point, entropy begins to climb again, while variance falls.

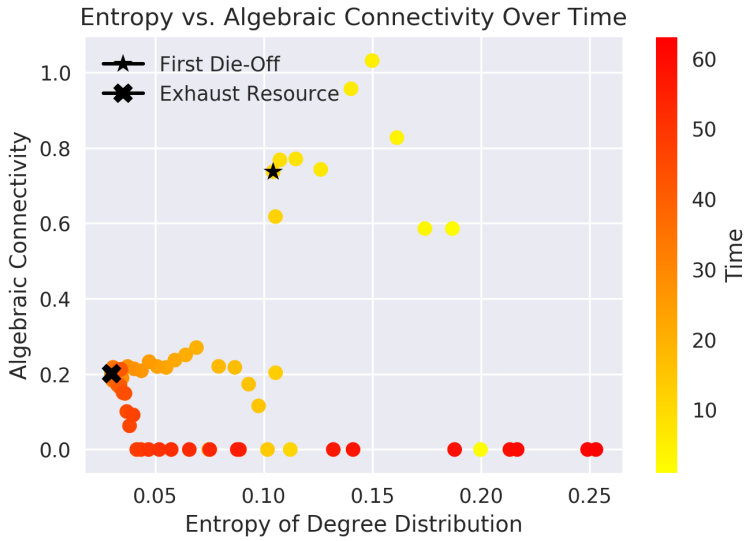


Fig. 10: The algebraic connectivity of the network at each moment in time as a function of the entropy of the degree distribution. While the trajectory is not as clean as what is visible in Figures 8 or 9, a clear pattern is visible. The algebraic connectivity over time is a noisier signal.

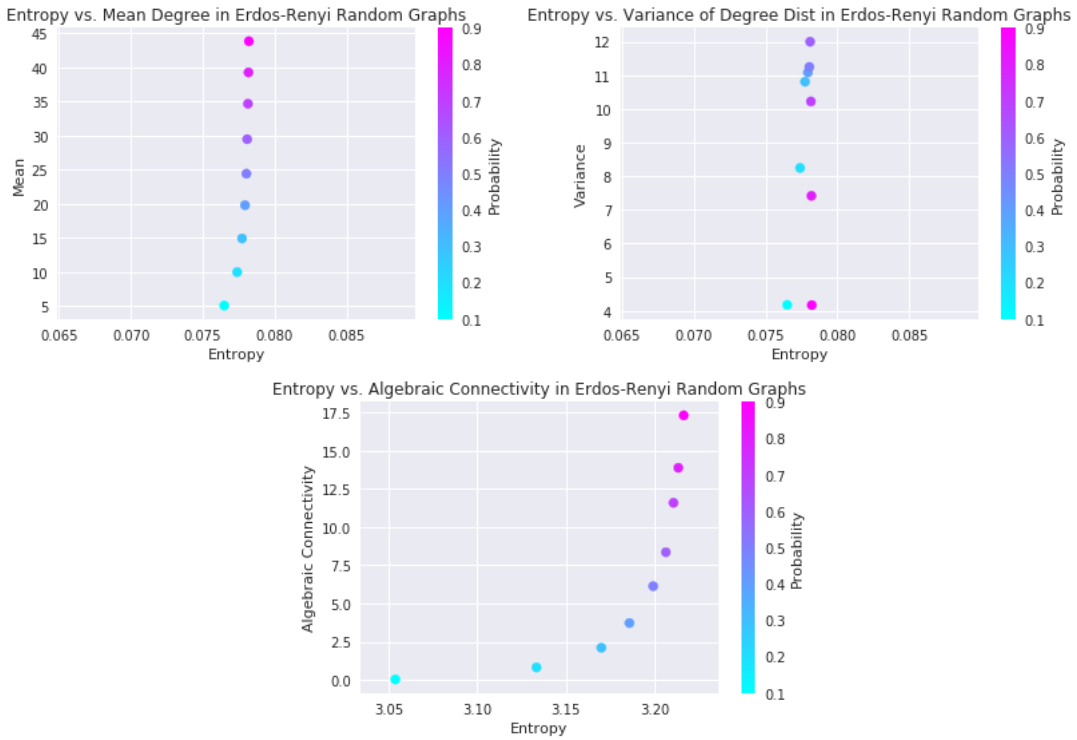


Fig. 11: Simple null models plotting the relationship between entropy, mean degree, variance in degree, and algebraic connectivity of an ensemble of 25 Erdos-Renyi graphs with 50 degrees each, created with probabilities between 0.1 and 0.9

.

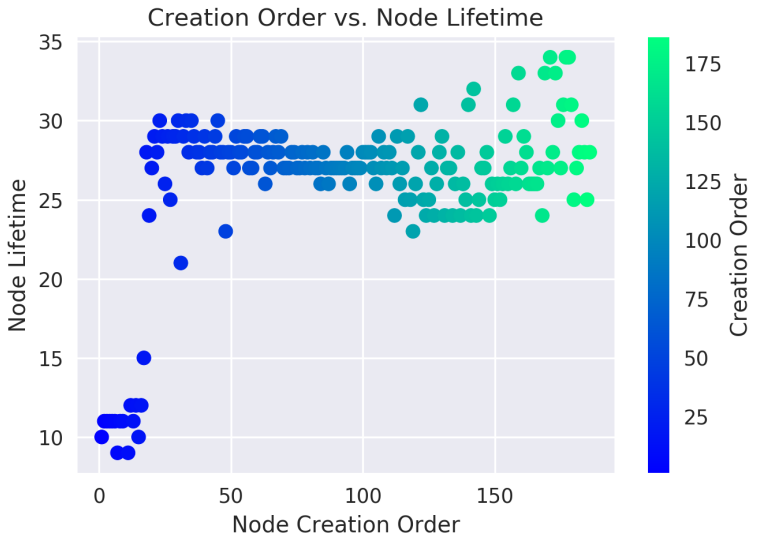


Fig. 12: The lifetime of a node (how many timesteps it persisted for), plotted as a function of the order of it's creation. Node the small cluster of very early nodes with short lifetimes that are distinct from the larger set.

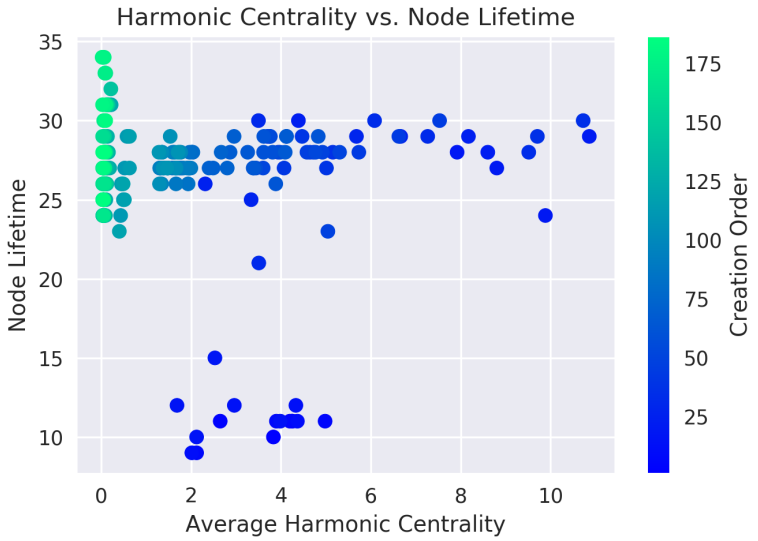


Fig. 13: The average harmonic centrality of every node, plotted against the lifetime of that node, and colored based on the node creation order. As with other plots of this type, notice a small group of early nodes that don't live very long and have low harmonic centrality. There are the nodes that come into being before the initial die-off. After the die-off, an trajectory emerges, where all subsequently created nodes have roughly similar lifetimes, however, nodes created later have almost uniformly decreases average harmonic centralities. This may be reflective of the entropy described by Figure 3, where as time goes on, the network becomes more 'lattice-like', which less obviously central nodes.

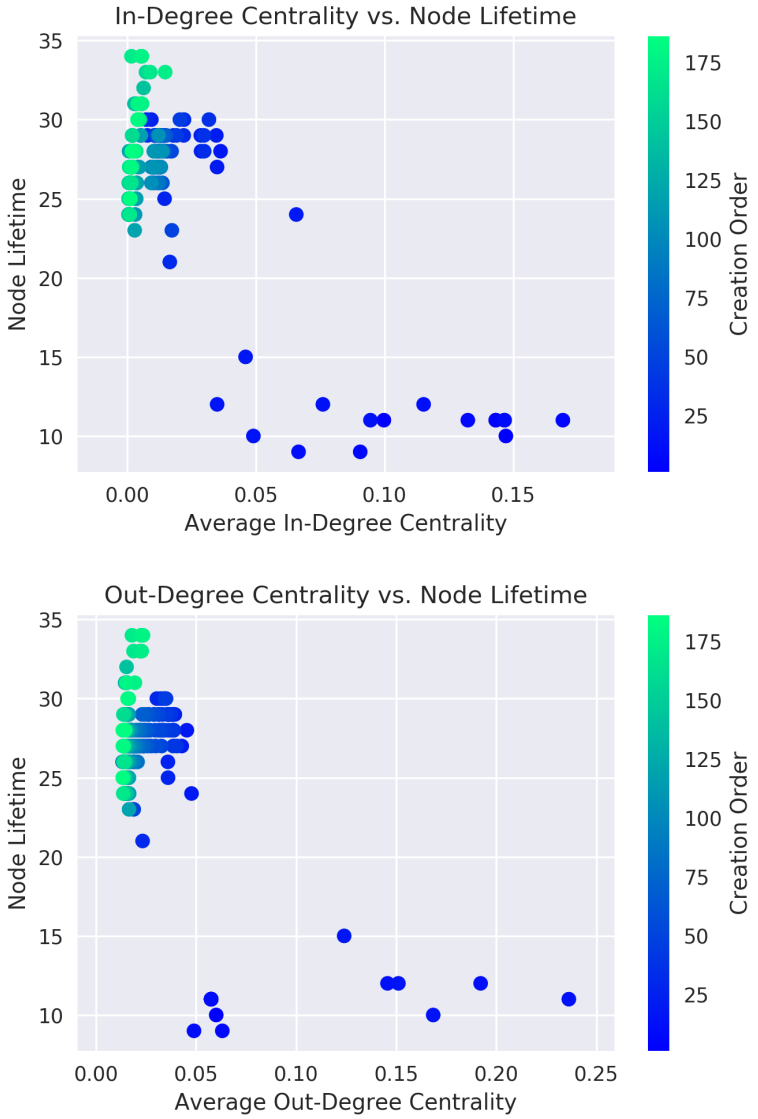


Fig. 14: Here the average in- and out-degrees are plotted as a function of the node lifetime, and colored by creation order. As with other plots of this type, there is a subset of nodes created early in the history of the system that die quickly, but have high degree centralities. After the initial die-off, subsequently created nodes don't show a strong relationship between degree centrality and lifetime, although there seems to be a strong trend that nodes created later have lower in- and out-degree centralities. This is consistent with he overall homogenization described in Figures 3 and 13.

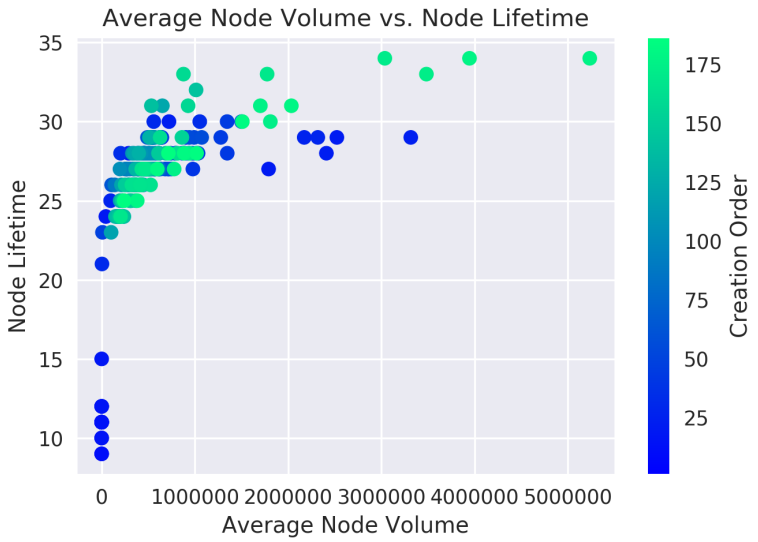


Fig. 15: Here, the average volume of a node over the course of it's life is plotted against it's lifetime. In contrast to the various centrality plots, where after the die-off there was not a consistent relationship between metric and lifetime, here there seems to be a global, logarithmic relationship between node volume and lifetime (although the break between pre-die-off nodes and post-die-off nodes is still evident), although creation order is less tightly related to either lifetime or volume here.