Gatekeeping in Campaign Contribution Networks

June 2016 *James David Moffet III and Lihan Yao*

During this year's, for lack of a better word, unique cycle of US presidential campaigns, we have a ballot of extremely diverse candidates, whose campaign funding strategies are even more diverse than the candidates themselves. We hope that all of the spectacle can be put to some use as a frame for investigating the influence of individuals in campaign contribution networks, specifically the exercise of veto power over campaign viability, or what we term "gatekeeping" behavior.

Our research topic originated from a self-evident observation: a ballot of candidates chosen by a small percentage of voters is less democratic than a ballot of candidates chosen by a large percentage of voters. Following from this observation, we decided to investigate contribution networks as a determining factor in the selection of viable candidates in US-style elections.

How do we identify the gatekeepers of campaign viability, and what properties do they have? As a computational problem, we define a set of gatekeepers as the minimal number of nodes whose removal from the network produces components which no longer contain enough resources to surpass some threshold of viability. We hypothesized, and can now observe, that both high "contribution" value (vertex weight) and crucial topological position determine the gatekeeper nodes' collective ability to destroy the viability of every "campaign" (subgraph) in the network.

We hope to develop a computationally-efficient way to describe gatekeeping behavior in real world networks. We believe that modeling this phenomenon may yield insights for domains in which a critical mass of resources is required to change a given state.

Zachary's Karate Club Example

In our preliminary investigation, we observe relationships between three parameters and the relative size/composition of the gatekeeper set: distribution of resources 3 (as vertex weights), threshold of viability 2 (as sum of weights in a component) and network density 1. We can computationally discover gatekeeper sets and have done so for a series of graphs with a variety of values for these parameters.

The main takeaway is that distribution swamps all other parameters in terms of influencing GP set size. However, benefits of mitigating wealth inequality begins to diminish well before the distribution resembles egalitarian society. The difference between our medium and flat distributions is almost negligible when compared to the punishing version. For reference, the distribution curve for a punishing can be best fit by $y = x^3 00$ and medium nearly being $y = x^7 0$. The medium distribution still depicts a harsh distribution of wealth even though the marginal return on decreasing inequality is basically gone. From the authors' experiences working with electoral campaigns, a best fit of $x^3 00$ is our estimate for the actual distribution.

Higher threshold, unequal distribution and decreasing density implies a more autocratic gatekeeping behavior, quantitatively we mean that smaller GP set size are exhibited.

This observation holds regardless of how the other parameters are manipulated.

Computing Gatekeepers

As a computational problem, let $G = (V, E)$ be a simple, vertex weighted graph. Call the weight function $w : V \to \mathbb{R}$. Lastly, attribute to G some $t \in \mathbb{Z}$ called the threshold.

Definition 1 *If for some $X \subseteq V$ we have that $G[X]$ is connected and*

$$\sum_{v \in X} w(v) \geq t$$

then X is a **viable set**.

Definition 2 *If after removing $K \subseteq V$ from G, $G[V - K]$ contains no viable sets, and K is size minimal with respect to this property, we call K a* **gatekeeper set**

The union of gatekeeper sets are called gatekeepers. Below are two techniques for the computation of such GP sets, one by sophisticated sorting of vertex combinations of certain size, and another by measure and conquer, where we explore the tree of possible cut unions and stop whenever a terminal condition, called filters, are met.

Algorithm 1 Brute Force

function CompScore(graph)
  for $x \in V$ **do**
    $E_c \leftarrow$ compute and normalize eigenvector centrality
    $w_n \leftarrow$ normalize vertex weight
    composite score of x, $C_s \leftarrow E_c + w_n$
function GetGPSet(graph, threshold)
  for $i \leq |V|$ **do**
    Arrange vertices of V according to C_s
    Create generator g_i which yields all ($|V|$ choose i vertex combinations) according to arrangement
    Counter $\leftarrow 0$
    Terminal-Value $\leftarrow \binom{|V|}{i} 0.1$
    $C_i \leftarrow g_i$
    if $G[V - C_i]$ has no viable components **then**
      record C_i in array
      terminate after computing this i value
    if Counter = Terminal-Value **then**
      break
  return GP sets

Algorithm 2 Measure and Conquer

```
function DEL-TIL-UNVI(graph, threshold)
    D = {}                                              ▷ Stores vertices to be deleted
    while ∑_{v∈G} w(v) ≥ threshold do
        D ← highest weight vertex
        Delete highest weight vertex
    return D
function GEN-CUT(graph, size)
    yield  cut union of graph whose cardinality is smaller than size
Component-Hash = {}
function FILTER(graph, threshold)
    if ∑_{v∈G} w(v) < threshold then
        Raise exception
    D ← DEL-TIL-UNVI(graph)          ▷ updated whenever we find a smaller candidate for GP set
    Connectivity ← graph connectivity
    if Connectivity> D then                  ▷ Filter: no cut union smaller than DEL-TIL-UNVI
        Update Component-Hash with D
        return str(*+D)
     return str(D)
S ← FILTER(graph, threshold)
function MEA-CON(graph, threshold)
    if first digit of FILTER(graph, threshold) is not * then
        for C ∈ GEN-CUT(graph, S−cardinality of previous cut unions) do
            Sum |C| with previous cut unions used to reach this graph component, call it ΣC
            break if ΣC > than some D of previous components
                ▷ We do not consider this C since a smaller GP set would have been possible by taking D
some time earlier
            G_C ← G[V − C]
            for component of G_C do
                if component ∈ Component-Hash then
                    update Component-Hash
                    continue
                MEA-CON(component)
    function MAIN(graph, threshold)
        MEA-CON(graph, threshold)
     return GP-sets
```

Algorithm 3 Percolation

for $i \in |V|$ **do**

function GetV(table)

function BuildDigraph(graph, threshold) **return** GP sets

function FindNbr(table)

function UpdateTable(table)

function UpdateV

Site Percolation

A modification of the tree algorithm seen in: **A fast Monte Carlo algorithm for site or bond percolation**. The site percolation algorithm yields a collection of approximate solutions.

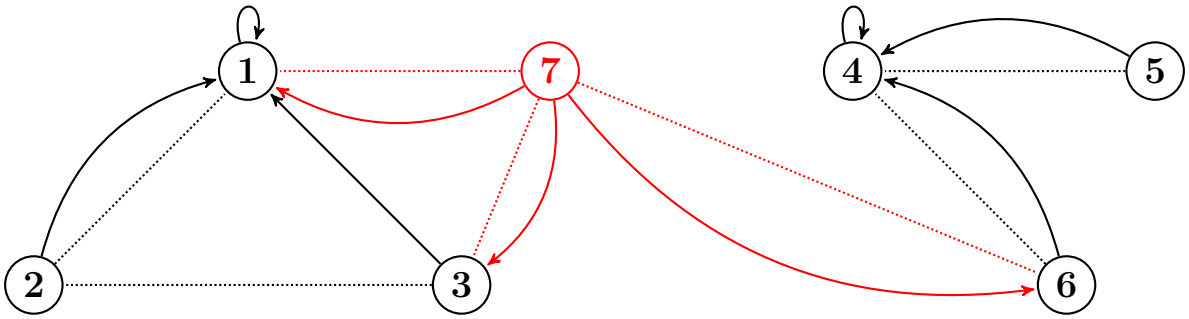
Start with a new directed graph D_0 that is initially empty. For a digraph D_i at the ith step, vertex random variable $X : V(G) \backslash (V(D_{i-1}) \cup GP) \to Z^+$ yields a vertex v_i according to a density value function f_X, where GP is the candidate gatekeeper set of this trial (see next section for specifications on X and f_X).

Upon adding v_i, if $N_G(v) \cap V(D_{i-1}) = \emptyset$, that is, the neighbors of v_i in G are not in D_{i-1}, add (v, v) to D_i. Otherwise, $\forall w \in N_G(v) \cap V(D_i)$, add (v, w) arcs to D_i.

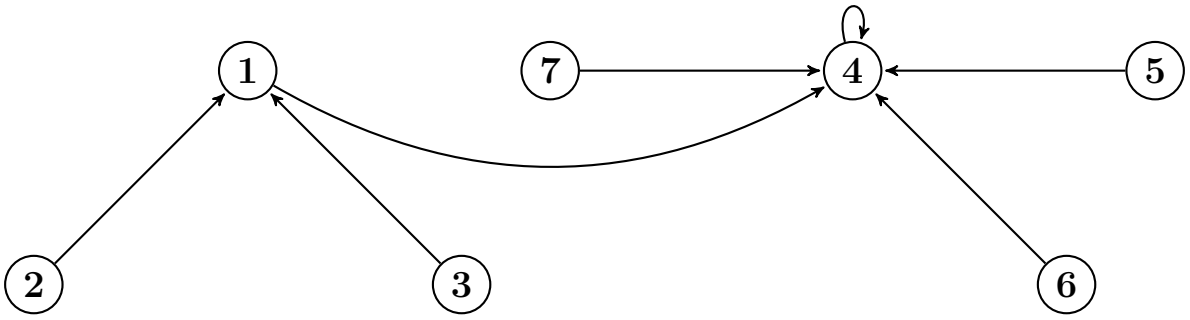
If v sends arcs to one component C in D_i, there is a unique vertex r in C with a loop. r is retrieved by simply traversing to the root of the directed tree within C. $w(r) \leftarrow w(r) + w(v)$ and $w(v) \leftarrow 0$. The node whose weight represent the total value of the component, r, is updated after the addition of v. A v-to-r arc is added to take advantage of "path compression." Path compression makes path traversal faster the next time we have to retrieve the aggregate component value $w(r)$. If $w(r) > T$ in D_{i+1}, v is noted as a member of the candidate gatekeeper set GP and p is recalculated for D_i. Discard D_{i+1}.

If v has arcs toward multiple components $C_1, C_2, \ldots, C_k$, for the j th component there is an unique vertex r_j with a loop. Pick one such vertex at random, denote it r^*. This means in G, v is connected to multiple components that will now pool together their values. Add (v, r^*) and move the value within v to r^*. Then we amalgamate the numerous component values of $C_1, \ldots C_k$ to designated node r^*. First add arcs to r^* from nodes previously representing their own components, i.e. $\{r_1, \ldots, r_k\} - r^*$, then remove all loops in D_{i+1} except (r^*, r^*).

For the j-th component C_j, whose value is represented by $w(r_j)$, we have that: $w(r^*) \leftarrow w(r^*)+w(r_j), w(r_j) \leftarrow 0, D_{i+1} - (r_j, r_j), D_{i+1} + (r_j, r^*)$. Likewise, if $w(r^*) > T$ in D_{i+1}, v is noted as belonging to GP and p is recalculated for D_i. Discard D_{i+1}.



The nodes are labeled according to the order in which they are added. Dashed lines denote edges in original graph G. 7 has connected two components, whose aggregate values are recorded by $w(v_1)$ and $w(v_4)$. In this example,$r^* = v_4$, and (v_1, v_1) will be removed from D_i. The value $w(v_1)$, in addition to $w(v_7)$, will be moved to v_4. Notice that even though there are only two components, three path traversals must be conducted by v_7 in the event that it is connected to three components in G.



D_7 after v_7 is added, with all nodes and arcs. Amongst remaining vertices of $V(G)\backslash V(D_7)$, p is recalculated for adding v_8.

Vertex Random Variable X

At the i-th step of the algorithm, possible outcomes for the vertex random variable X (it's sample space Ω) are vertices in $V(G)$ that are not yet present in D_{i-1}. Since we identify vertices with positive integers, $X : V(G)\backslash(V(D_{i-1}) \cup GP) \to Z^+$.

How is the density function f_X for this discrete random variable computed (for d.r.v the correct terminology is probability mass function)?

Suppose v_i is added (not in the sense of constructing D_i), denote the components in D_{i-1} which receive an arc from v_i in D_i, $C_1, C_2, \ldots, C_k$. Then v_i receives a score S_i reflecting it's likelihood to be a gatekeeper:

$$S_i = \alpha \prod_{i \geq 0}^{k} |C_i| + \beta \sum_{i \geq 0}^{k} w(C_i)$$

where $\alpha + \beta = 1$ and $\beta \propto \text{Var}(Y)$, Y being a random variable to the distribution given by $w(\cdot) : V(G) \to R$.

A thought experiment: if all nodes have the same weight, we do not even have to care about the second term. When we add the new v_i, it is enough to add it in a way that minimizes the size of resulting C. But what if the distribution is punishing? Then who cares about how many 0 dollar nodes you keep on adding to C, as long as the networth of C (the captured networth) is high?

Since the algorithm would like to be fed those vertices which are least likely to be GP, the probability of v_i being picked at the i-th step is then:

$$p(v_i) = 1 - \frac{S_i}{S_{max}}$$

Q1 to self: is there a better probabilistic way to pick vertices for the algorithm? I mean specifically for the expression which calculates S_i

Q2 to self: currently the first term is a large product because we know from Explosive Percolation, that minimizing product sizes delays the growth of giant components, but is slapping on α and β accurate enough to capture the value dimension of this problem? Does delaying the growth of giant components lead to topologically important nodes added last? (so approximate GPs are accurate)

v	State	Comp. Members	Comp. Value	Score	Probability	A	Pts
v_1	gatekeeper	4	4.3	10	0	5	12
v_2	undecided	3	2.2	3	0.1	9	9
v_3	digraph	2	4	5.6	0	8	7
$\vdots$							
v_k	undecided	2	1	10	0.03	8	7

[soltys]

References

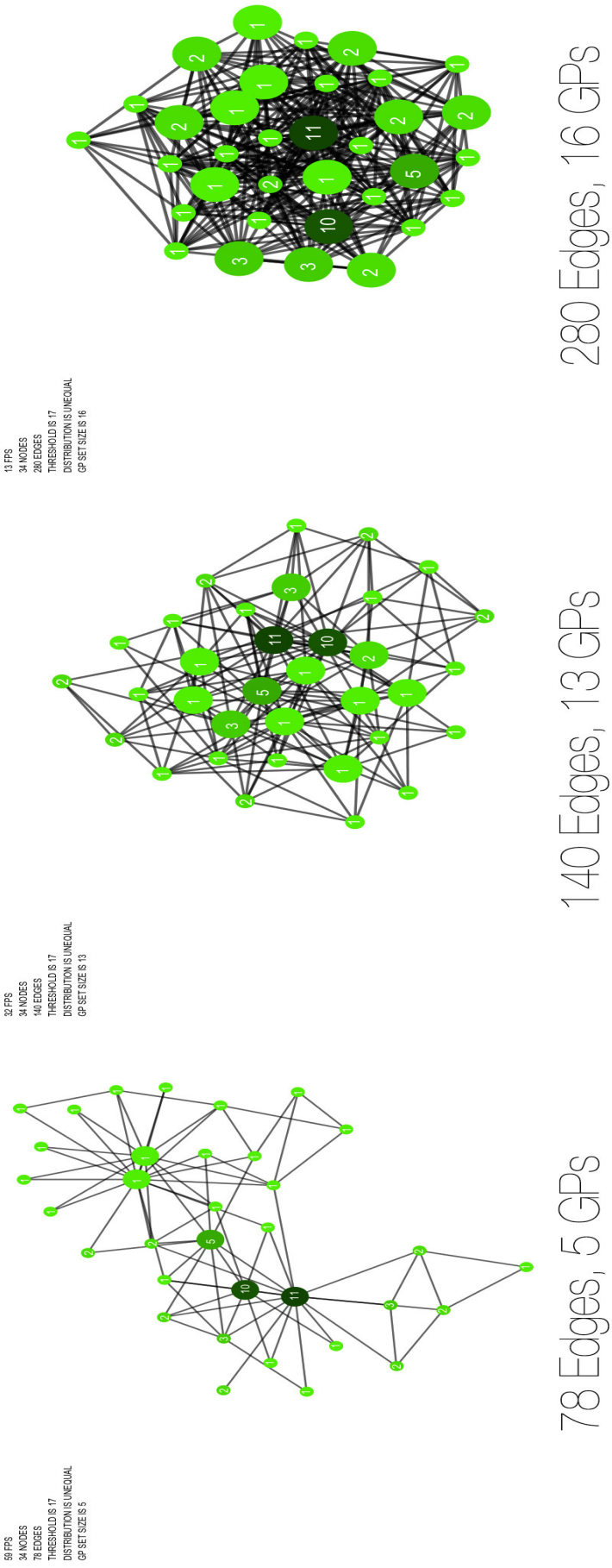


Figure 1: Varying Edge Density

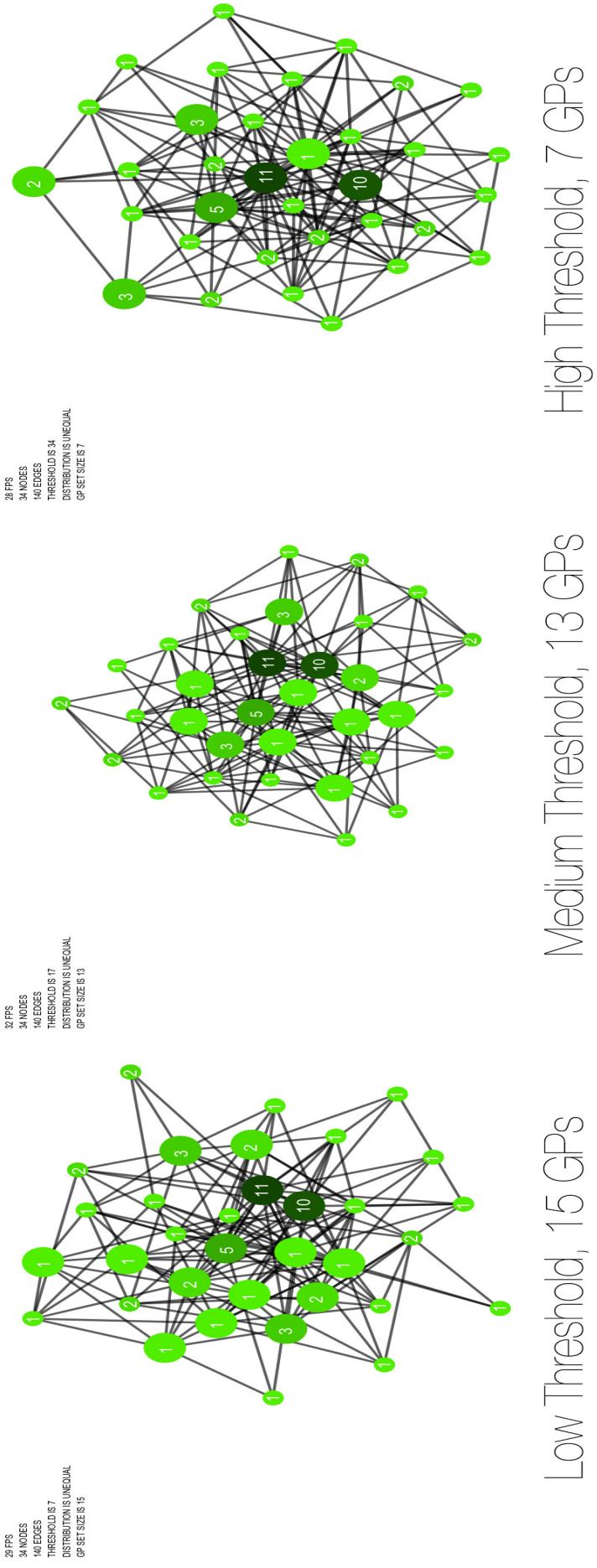


Figure 2: Varying Threshold

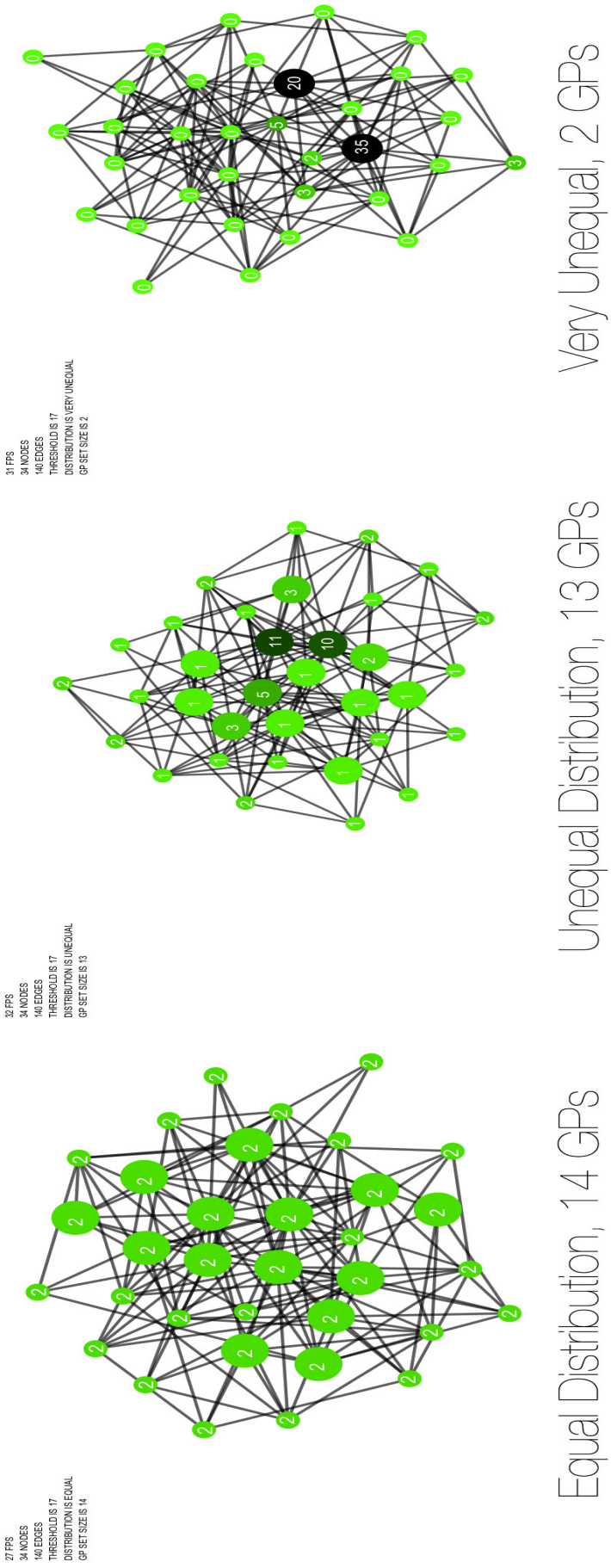


Figure 3: Varying Distribution