An Investigation of Federal Election Donation Networks from 1980 to 2010

Candidate 130530

March 30, 2012

Contents

Abstract

The study of American politics has increasingly turned to both quantitative and qualitative analyses of political networks to understand voting patterns and behavior, policy ideas, public opinion, impacts on legislation, interest group influences and coalitions, and other political practices. The goal of this project is to understand and analyze the network structure of election donation data from the Federal Election Commission as it changes over three decades. We see differences in transitivity and reciprocity between presidential and midterm cycles, suggesting that the two have different structure. We also observe a large jump in transitivity between 1986 and 1988, reflecting the increase in the use of soft-money in the 1988 presidential campaign. We implement an algorithm published by Ball, Karrer and Newman in 2011 that detects overlapping communities, and we run this against a modified projection onto the candidates.

Acknowledgments

I would like to thank, in no particular order, Andrew Waugh for curating and supplying the data; my supervisor for his wisdom, guidance and critical eye; Lucas Jeub for visualization code and providing help and coding improvements; and my parents, for all their support, pecuniary and otherwise.
I must also thank former President Richard Milhous Nixon–without Tricky Dick's creative campaign financing, the FEC might not have existed.

Chapter 1

Introduction

One of the main issues of political science is that of campaign finance by which political candidates fund their campaigns. In the United States, the issue regularly enters the news. In the 1970s, the US congress passed the Federal Election Campaign Act to regulate campaigns(Mann, 2003). The act was later amended to establish the Federal Election Commission (FEC), a bipartisan committee that enforces campaign rules(Mann, 2003).

The disclosure of donations is important to show how politicians are influenced, especially when donations to relevant elected officials are an integral part of modern lobbying. The raw data collected by the FEC is available to the public for analysis. However, I used data that Andrew Waugh, a political scientist and collaborator from U. C. San Diego, supplied. The supplied data is a proprietary version of what is available to the public, which he has amalgamated, formatted and parsed for use in network analysis. The goal of my investigation is to see how network properties like transitivity, reciprocity and community structure have changed over the years and if they reflect changes in electoral laws.

Sections 1.1 and 1.2 focus on the mathematical background to the analysis. Chapter 2 gives an overview of political networks and more information about how I used the data in this investigation. Chapter 3 describes the algorithm I implemented, re-derived and used in my analysis. Chapters 4 and 5 provides discussion and conclusions, respectively. Python and MATLAB code I wrote is given in Appendices A through C. Codebooks describing the structure of the supplied data are in Appendices E and D.

1.1 Graphs

The usual representation of a network is a *graph* $G := (V, E)$ with a set of vertices $V = \{v_i\}_{1 \leq i \leq N}$ and a collection of edges $E \subseteq V \times V$. The connections between vertices are represented as ordered pairs of vertices $(v_i, v_j) \in E$ in these *directed* graphs, as shown in Figure 1.1a. When the direction of the connection is unimportant, the order of the pair does not matter. In such *undirected* graphs (as shown in Figure 1.1b), only the existence of a connection matters(Newman, 2010).

A particularly large network is the World Wide Web, with webpages as vertices and hyperlinks as edges. The edges of a graph representing this network must be directed because a hyperlink connects in only one direction. Because a webpage may link to itself, the graph representing it can have self-edges– edges of the form (v_i, v_i). In Figure 1.1a, vertex 1 has a self-edge.

In certain networks (including the World Wide Web) multiple edges can exist between two vertices(Newman, 2010). In such graphs, known as *multigraphs*, an edge can be labeled to distinguish it from the others. For example, the edges in a temporal network are of the form (v_i, v_j, t), where t is the time of edge's occurrence. In Figure 1.1c, the multiple edges between vertices 0 and 2 are distinguished by their color. Graphs without multiple or self edges are called *simple*.

If a graph's set of vertices can be split into two disjoint sets U and W where every edge connects a vertex in U to one in W and vice versa, then the graph is said to be *bipartite*, as depicted in Figure 1.1d. For example, consider a network of consumers and products where an edge exists between a consumer and product if the consumer purchased the product. Because we can split the set of vertices into the disjoint sets of consumers and products, this network is bipartite.

Edges in any type of graph may also be *weighted* by a real number to represent the strength of the interaction between two vertices (Newman, 2010). The graphs described previously are *unweighted*. How to weight an edge depends on the context of the analysis. For example, consider a graph where the vertices are countries and the edges represent shipping lanes. A weighting by total value of goods shipped across each lane could be one choice; another could be by total number of ships.

1.1.1 The adjacency matrix

A convenient representation of a simple, unweighted graph is by using an *adjacency matrix* A_{ij} defined as

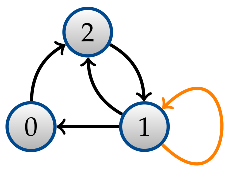
$$A_{ij} := \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}. \tag{1.1}$$

In graphs with weighted edges and without self-edges, the adjacency matrix is defined as

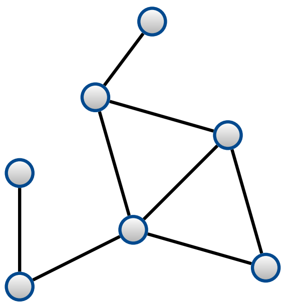
$$A_{ij} := \begin{cases} \text{Weight}((v_i, v_j)) & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}. \tag{1.2}$$

The adjacency matrix of a graph encodes the graph in a form suitable for calculations. Many network algorithms require the ability to quickly check the existence of an edge between two vertices and the weight of that edge, if the graph is weighted. Representing a graph as a full adjacency matrix allows the retrieval of this information within $O(1)$ time. However, if the adjacency matrix has many entries equal to 0, then representing it in a sparse matrix format can save memory. Unfortunately, checking for the existence of an edge in an adjacency matrix in a sparse format is slower than $O(1)$.

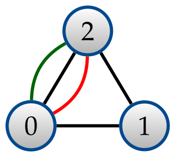
Other properties may be ascertained quickly from the adjacency matrix. For example, undirected graphs have a symmetric adjacency matrix because the existence of the edge (v_i, v_j) requires the existence of the edge (v_j, v_i), so $A_{ij} = A_{ji}$. In an undirected graph, a self-loop is encoded by setting A_{ii} equal to



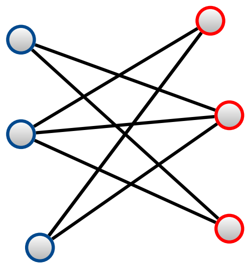
(a) A directed triangle graph with a self-edge in orange



(b) An undirected graph



(c) A multigraph with multiple edges between vertices 0 and 2



(d) A bipartite graph, with groups labeled by colors.

Figure 1.1: Various types of unweighted graphs

2. In a directed graph, a self-loop is encoded by setting A_{ii} equal to 1(Newman, 2010).

1.2 Network properties

1.2.1 Degree

In an undirected graph without self-edges, the degree of a vertex is the number of edges emanating from that vertex. The degree must be non-negative as well as less than the total number of vertices minus one(Newman, 2010). In a directed graph, we must consider the *in-degree*, which is the number of edges pointing into a vertex, as well as the *out-degree*, the number of edges emanating from that vertex.

The cumulative degree distribution of a graph G is defined as

$$C_G(d) := \frac{\text{\# of nodes with degree} \geq \text{d}}{\text{total number of vertices in } G} \tag{1.3}$$

This gives an insight into how often vertices with high or low degrees appear in the graph. Generating a graph with a fixed number of vertices, by sampling the degree of a vertex from a known distribution such as Poisson or power law, is a common way of creating null models against which to compare real-world networks.

1.2.2 Neighborhood of a vertex

In a network, the *neighborhood* of a vertex are the other vertices that connect to it. The neighborhood in an undirected graph of vertex v_i is defined $\Gamma_i = \{u \in V : (u, v_i) \in E\}$. The degree of vertex i is then $|\Gamma_i|$.

In a directed graph, there is a distinction between the in-neighborhood $\Gamma_i^{\text{in}} = \{u \in V : (u, v_i) \in E\}$ and the out-neighborhood $\Gamma_i^{\text{out}} = \{u \in V : (v_i, u) \in E\}$, which are, respectively, the vertices that have an edge pointing at v_i and vertices for which v_i is in their in-neighborhood. The in-degree of vertex i is then $|\Gamma_i^{\text{in}}|$ and the out-degree is $|\Gamma_i^{\text{out}}|$.

1.2.3 Transitivity and Reciprocity

The *transitivity* or *global clustering coefficient* is defined as three times the ratio of the number of triangles to the number of connected triples. A triangle occurs between vertices u, v, and w if the edges (u, v), (v, w), and (w, u) are present in the graph, whereas a connected triple occurs if any two of those three edges exist. The *local clustering coeffcient* is defined for each vertex as the ratio of the number of connected triangles containing that vertex to the number of triples containing that vertex.

In undirected graphs, a triangle is the smallest loop that can occur, so graphs with a high clustering coefficient represent situations where the "friends of my friends are also my friends." Because undirected graphs formally represent a relation on the set of their vertices, this quantity measures the degree to which the relation is transitive. For example, an undirected graph that represents an equivalence relation has a transitivity of 1. In contrast, the transitivity of a tree is 0.

In directed graphs, the loop between two vertices is the smallest that can occur. One can then measure how often a vertex reciprocates an edge. In other words, how often A_{ij} equals A_{ji} when either equals 1. The *reciprocity* is

$$r := (\text{\# of edges})^{-1} \sum_{i,j} A_{ij} A_{ji} \tag{1.4}$$

Graphs with weighted edges are treated as if they were unweighted. $A_{ij} \in \{0, 1\}$) is the ratio of reciprocated edges to the total number of edges present in the graph, and it lies between 0 and 1. A high reciprocity occurs when vertices are likely to link back to each other, if one links to the other. An undirected graph has a reciprocity of 1.

1.2.4 Projections

In bipartite networks, one can construct a graph onto one of the subsets of vertices, creating a unipartite graph. Let U be the set of vertices of one of the two distinct groups of vertices of the graph and let W be the other. The projected graph P has vertices U and adjacency matrix that is equivalent to computing AA^T, keeping the rows and columns corresponding to U and setting the diagonal elements to 0(Newman, 2010). It is equivalent to a projection in linear algebra, whence the name. The projected graph P is an undirected, weighted network.

However, the reduction in vertices comes at a price: the constructed graph does not encode the same information. Most of the information in the vertices that were *not* projected upon (vertices in W) is lost. Furthermore, if the graph is weighted, then it is not straightforward as to how to incorporate the weightings of the original graph into the projected one. A more complex issue is how to relate the network properties of the constructed graph to those of the original. There is no straightforward way to interpret those properties in the context of the original, because the information encoded in the projected graph differs from the information encoded in the original.

1.2.5 Communities

One of the main areas of study in network science is the algorithmic detection of a *community* of vertices, intuitively a subset of the vertices where there are more links within the subset than to vertices outside of the subset(Porter et al., 2009). Communities may be distinct, where a vertex belongs to one and only one community, or overlap, where a vertex could belong to two or more communities simultaneously. If communities overlap, then a vertex may belong more strongly to one community than another. The extent to which a vertex belongs to a particular community is derived from the placement of the network's edges. There is no consensus on the precise definition of community.

For example, in a network with vertices representing US Senators where an edge exists between two senators if they co-sponsored a bill, one might expect to see communities corresponding to party affiliation. Conversely, if a community detection algorithm shows the presence of a community for which there is no previously known identification, further investigation may be worthwhile(Porter et al., 2005).

There are various approaches to detecting communities. One approach maximizes a quantity called modularity, which assigns vertices to a community based upon how different the distribution of edges are to the expected distribution of edges of a specific random null-model of a graph(Newman, 2006; Porter et al., 2009). Other algorithms work by partitioning the graph by selectively removing edges, thereby splitting a graph into unconnected components through a function such as geodesic betweenness, which measures how frequently an edge occurs on the shortest paths between two vertices(Newman, 2010).

In Zachary's Karate Club network, which links members of a university karate club if they interacted outside of club sessions, one can find the existence of multiple communities as depicted in Figure 3.1(Zachary, 1977). These could signify the existence of separate groups within the club, and in fact the club split into two due to disputes between two factions over club administration(Zachary, 1977). However, one of the members joined the opposing faction's club after the split. This person could therefore reasonably belong to more than one community–assigning him or her membership in a particular community would ignore the ties to other communities.

In order to deal with ambiguities in assigning a vertex to a particular community, an issue that is common in real-world networks, an algorithm that allows communities to overlap is preferred. This means that a vertex can belong to multiple communities, and certain algorithms can compute the degree to which the vertex belongs to a particular community.

The runtimes of these algorithms vary, but any algorithm that runs in time greater than linear time (i.e., not in time directly proportional to the number edges or vertices) becomes impractical as the number of vertices or edges grows large. Real-world networks frequently consist of millions of vertices and edges, so an algorithm that can detect communities in a reasonable amount of time (minutes to hours, or less) is preferred to one that takes days or months(Newman, 2010).

Chapter 2

Political Networks

2.1 Context

The analysis of political networks quantifies relationships between and within political entities and seeks to reveal structure from these relationships. Researchers studying these relationships seek insights into their strength, how they affect decision-making, and how influence and information travels through them. A wide array of data has been studied, including the committee relationships in the House of Representatives, the treaty, trade and military alliances between countries, and the personal relationships between politically important people(Lazer, 2011). Studying the pattern of wars between nations has led to the observation that democracies rarely go to war with each other,(Ward et al., 2011) and the analysis of the personal communication between the 9/11 plotters showed that lead hijacker Mohammed Atta had direct ties to most of the other terrorists(Ward et al., 2011).

Finding previously unknown structure in networks has also been a focus of network research. When Porter, Mucha, Newman and Warmbrand applied community detection algorithms to co-sponsorships of bills in the House of Representatives, the community structure found therein reflected the committee assignments of members and showed strong ties between committees with disparate remits(Porter et al., 2005).

In summary, network methods allow the relationships between political entities to be quantified–for example, one can see the extent to which money flows between people. Because power and influence follow ties between people, organizations and countries, the structure of political networks can show who influences whom in national politics.

By applying network methods, political scientists seek to discover structure that could have political consequences. Knowledge of this structure could then inform political decision making.

2.2 Data

Poprietary data assembled by Andrew Waugh of U. C. San Diego from public FEC data and supplied to us, consist of the information reported by campaigns during the campaign trail of congressional elections conducted in the US from

1980 to 2010. The Federal Election Commission requires that campaigns report detailed information on donations over $200. This information includes the amount donated, when the donation took place, how the donation was reported, the registered FEC identification number (FEC ID) of both the recipient and donor, and how the money was transferred.

Waugh supplied the data as CSV files and RData images–stored runtimes of the R statistical programming environment. I then imported the RData images into the igraph(Csardi and Nepusz, 2006) format to calculate common network quantities like degree and clustering coefficient. I also constructed sqlite3 databases from the CSV files.

The resulting directed graph edges (each edge (v_i, v_j, t) corresponds to a reported donation made by v_i to v_j at time t) encodes all the donations made in that election cycle. I also constructed an undirected graph, with edges weighted by total amount donated between v_i and v_j. In the following, this will be referred to as the 'collapsed graph' for the cycle.

Metadata describing various personal and professional details about donors were also supplied by Waugh as a CSV file with structure defined in Appendix E. The most important descriptor of a vertex is the FEC ID assigned to it; this uniquely identifies each vertex and is consistent inside an election cycle. I uniquely identified a vertex in all of the graphs I constructed by its FEC ID. Furthermore, the IDs allow us to distinguish candidates from non-candidates. Donations may occur between any two FEC IDs. A candidate may give to another candidate, for example, if he or she has retired from the race and has excess funds. Political committees often donate to each other, as well.

A major goal in analyzing the data was to seek an understanding of how the community structure of the network changes within an election cycle. A challenge was the size of the data. There are sixteen cycles, each hacing up to millions of vertices and edges. Moreover, the BKN algorithm, as originally described, only works on undirected graphs.

To simplify the collapsed graph before running the BKN algorithm, I removed the candidate-candidate donations and the donations between non-candidates. Although PAC to PAC donations occur, and some of that money is forwarded to the candidate, I focused on direct donations. Candidate to candidate donations were removed because they occur only in exceptional circumstances. This created a bipartite network, so I could construct a projection onto the candidates. I then modified the projected graph by ignoring the weights, because the BKN algorithm cannot handle weights, creating an unweighted, undirected graph with edges between candidates that share a common donor. Because there are 1000 to 4000 candidates in any year, the projected graph was much smaller than the original (reducing the number of edges by a factor of 4 or more).

Chapter 3

The Ball-Karrer-Newman (BKN) algorithm

3.1 Introduction

The algorithm that I implemented to detect communities in the FEC data (code in Appendix C) was published by Ball, Karrer and Newman in 2011 and addresses many of the shortcomings, such as speed and memory usage, of previous approaches to detecting overlapping communities in undirected graphs. Specifically, it is a generalized expectation-maximization procedure that requires time and space directly proportional to the number of edges in the network. For simplicity, the derivation allows the existence of self-edges and multiple edges between vertices, though this may not be realistic(Ball et al., 2011). This derivation follows the one described in their paper(Ball et al., 2011).

3.2 Derivation

The algorithm is derived from considering the statistical likelihood of generating the adjacency matrix of a given graph from a graph with n vertices and K communities, where the propensity of vertex i to have an edge in community z is θ_{iz}. In particular, the number of edges of community z between two distinct vertices i and j is independently Poisson-distributed with mean $\theta_{iz}\theta_{jz}$, and the number of self-edges of vertex i of community z is $\frac{1}{2}\theta_{iz}^2$. Therefore, because the sum of Poisson distributions is also a Poisson distribution, the total number of edges between distinct vertices i and j is Poisson distributed with mean

$$\mu_{ij} = \sum_z \theta_{iz}\theta_{jz} \tag{3.1}$$

and the total number of self-edges of vertex i satisfies

$$\mu_{ii} = \sum_z (1/2)\theta_{iz}^2 \tag{3.2}$$

Thus the likelihood, given the nK propensities θ, of generating the adjacency matrix $A = (A_{ij})$ of the given graph is

$$L(A|\theta) = \prod_{i=1}^{n}\prod_{j=1}^{i-1} \frac{\mu_{ij}^{A_{ij}}}{(A_{ij})!} \exp(-\mu_{ij}) \times \prod_{i=1}^{n} \frac{\mu_{ii}^{A_{ii}/2}}{(A_{ii})/2!} \exp(-\mu_{ii}). \tag{3.3}$$

The procedure then attempts to find the value of the θ such that this quantity is maximized. A few simplifications are needed to sidestep having to optimize a coupled non-linear function of the θ. Maximizing the likelihood is equivalent to maximizing the logarithm of the likelihood, because log increases monotonically.

Furthermore, since the value of the (log-)likelihood is unimportant, we can shift and scale the log-likelihood by positive constants. This does not change the *position* of the maximum, hence we can eliminate additive and multiplicative constants so that maximizing the likelihood is equivalent to maximizing the simplified log-likelihood

$$\log L(A|\theta) = \sum_{ij} \left[A_{ij} \log\left(\mu_{ij}\right) - \mu_{ij} \right]. \tag{3.4}$$

Unfortunately, differentiating equation 3.4 also results in coupled non-linear equations for the θ. We can bypass this obstacle by applying the conditional form of Jensen's inequality to change the equality in (3.4) to an inequality, by exploiting the concavity of the logarithm. Specifically, by introducing arbitrary probabilities $q_{ij}(z)$ satisfying the constraint (for fixed i and j) $\sum_z q_{ij}(z) = 1$, the relation

$$\log(\mu_{ij}) \geq \sum_z q_{ij}(z) \log \frac{\theta_{iz}\theta_{jz}}{q_{ij}(z)} \tag{3.5}$$

holds, with equality holding if and only if

$$q_{ij}(z) = \frac{\theta_{iz}\theta_{jz}}{\mu_{ij}}. \tag{3.6}$$

Hence, if we choose the $q_{ij}(z)$ to force equality,

$$\log L(A|\theta) = \sum_{ij} \left(A_{ij} q_{ij}(z) \log \frac{\theta_{iz}\theta_{jz}}{q_{ij}(z)} - \mu_{ij} \right). \tag{3.7}$$

Now, differentiating (3.7) with respect to θ_{iz} (holding the $q_{ij}(z)$ fixed) we find, after rearranging, that the optimal values of θ_{iz} satisfy

$$\theta_{iz} \sum_s \theta_{sz} = \sum_j A_{ij} q_{ij}(z). \tag{3.8}$$

Summing over i then yields

$$\left(\sum_i \theta_{iz} \right)^2 = \sum_{ij} A_{ij} q_{ij}(z) \tag{3.9}$$

and upon combining 3.8 and 3.9 we get

$$\theta_{iz} = \frac{\sum_j A_{ij} q_{ij}(z)}{\sqrt{\sum_{ij} A_{ij} q_{ij}(z)}}. \tag{3.10}$$

3.2.1 Naïve BKN algorithm

Now the problem of maximizing the log-likelihood proceeds in the following manner:

1. Initialize the $q_{ij}(z)$ to random values, making sure they satisfy the constraint $\sum_z q_{ij}(z) = 1$.
2. Calculate the new values of θ using equation 3.10 and the current values of $q_{ij}(z)$.
3. Calculate the new values of $q_{ij}(z)$ using equation 3.6 using the current values of θ.
4. If the maximum difference $\epsilon = \max_{ij} \left| q_{ij}^{\text{new}} - q_{ij}^{\text{old}} \right|$ is negligible (less than, for example, 10^{-4}), then continue to step 5; otherwise discard the current values of θ and using the new values of $q_{ij}(z)$ go back to 2 . Alternatively, after a set amount of iterations, say 1000, go to step 5.
5. Calculate the log-likelihood (using 3.4) corresponding to the converged θ and store it and the θ.
6. Repeat steps 1 to 5 as many times as practical (the authors recommend between 10 and 100 times, I picked 25 because anything greater did not improve the results) and keep the θ corresponding to the greatest calculated log-likelihood. Because the initialization of the $q_{ij}(z)$ is random, the algorithm avoids getting caught at a local maximum in the iterative optimization of steps 2 and 3; this increases the possibility of finding the global maximum.

Once we have calculated the optimal values of the θ and $q_{ij}(z)$, we assign the edges between vertex i and j to community

$$\arg\max_{1 \le z \le K} q_{ij}(z).$$

The extent to which vertex i belongs to community z is then the proportion of incident edges (i.e, edges of the form (i, j), for some j) that are assigned to community z.

The algorithm described in Section 3.2.1, although linear in time and space, consumes a significant amount of memory. The dual optimization procedure requires that the mK values of the $q_{ij}(z)$ and the nK values of the θ be tracked, resulting in high memory usage because virtually all real-world graphs have more edges than vertices.

3.2.2 Expectation-Maximization

The procedure in section 3.2.1 is an example of a generalized expectation-maximization (GEM) technique. A GEM scheme attempts to find optimal parameters for a model that can be expressed using a log-likelihood function (equation 3.4 in the case of naïve BKN). Steps 2 and 3 in the naïve algorithm guarantee that the log-likelihood monotonically increases–thereby ensuring convergence(Borman, 2004). These correspond to the maximization phases of GEM; the use of a random (in my case sampled from a uniform distribution) starting point guards against the algorithm getting caught at a local maximum.

3.2.3 Practical Improvements

In order to reduce the amount of memory used by the naïve BKN algorithm, equations 3.6 and 3.10 can be replaced by new equations involving the quantities

$$k_{iz} = \sum_j A_{ij} q_{ij}(z) \tag{3.11}$$

and

$$\kappa_z = \sum_i k_{iz} \tag{3.12}$$

so that (from equation 3.10)

$$\theta_{iz} = \frac{k_{iz}}{\sqrt{\kappa_z}} \tag{3.13}$$

and (from equation 3.6)

$$q_{ij}(z) = \frac{k_{iz} k_{jz}}{\kappa_z} \left(\sum_\zeta \frac{k_{i\zeta} k_{j\zeta}}{\kappa_\zeta} \right)^{-1}. \tag{3.14}$$

Steps 2 and 3 can then be replaced by a single step wherein only the k_{iz} are stored and new values $k_{iz}^{\text{new}} = \sum_{ij} A_{ij} q_{ij}(z)$ are calculated using the values of $q_{ij}(z)$ from equation 3.14(Ball et al., 2011). This requires the storage of only nK values, a substantial savings in most cases. We still assign edges to their communities by assigning them to the community that corresponds to the highest value of $q_{ij}(z)$, and the extent to which vertex i belongs to community z is again the proportion of edges in community z (Ball et al., 2011).

Other Optimizations

Further optimizations include the δ-approximation, where values of k_{iz} that are less than a small parameter δ $(0 \leq \delta \leq 1/K)$ are set to zero. Because $k_{iz} = 0$ implies that k_{iz} will remain zero for all future iterations, we can avoid updating this value. Moreover, if a fixed vertex i has only one non-zero k_{iz}, then that vertex will definitely belong to community z–it is considered to have converged to a particular community(Ball et al., 2011). If both ends of an edge (i, j) have converged, then the maximum of the probabilities of that edge being in a particular community (i.e., the $q_{ij}(z)$) will not change. Hence, the edge's community is fixed and we ignore it in future iterations.

In my runs, ignoring an edge did not result in a speed increase until networks had more than approximately 16000 edges. The reason that graphs with fewer edges do not benefit is likely to be the increased bookkeeping required–in my implementation, I used approximately nK more quantities.

3.2.4 Drawbacks of the BKN algorithm

A significant drawback of the BKN algorithm is the requirement that the number of communities be pre-specified, even though it may not be known in advance. In addition, the algorithm cannot handle directed and weighted graphs.

3.3 Computational Constraints

My implementation of the BKN algorithm was limited by my system, a 2011 Macbook Air with an Intel Core i7 processor and 4 GB of RAM. In order to improve execution time and memory (even after implementing the optimizations in Section 3.2.3) I chose to construct graphs by modifying a projection onto the candidates of each cycle, rather than processing the full, directed graphs that arise from the FEC data. Depending on how many candidates contested elections, this yielded graphs of 1000 to 4000 vertices.

By focusing on detecting communities in the modified projection, I was able to run the BKN algorithm in a reasonable amount of time and space–each run (25 iterations of the core optimization described in Section 3.2.3) took about fifty seconds and under 25 megabytes of RAM. On small networks, like Zachary's Karate Club (shown in Figure 3.1), the BKN algorithm executes much more quickly – under 100 ms for 25 iterations of the core optimization. Appendix B lists code that I wrote for the projection process, which is a faster version of the operation described in Section 1.2.4 and the modification given in Section 2.2.

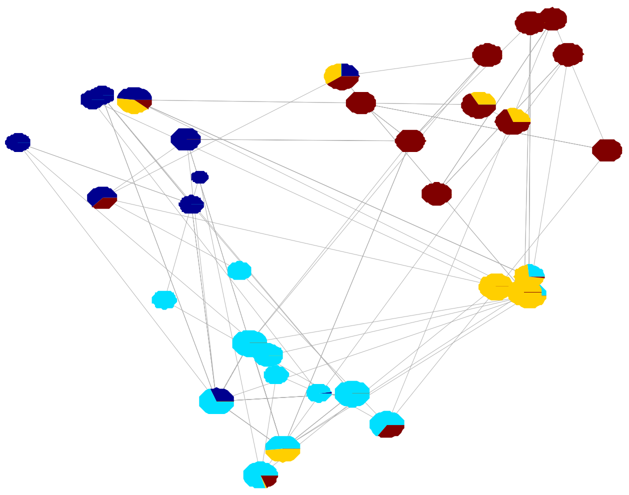


Figure 3.1: My implementation of the BKN algorithm run on Zachary's Karate Club network. Colors represent communities; vertices that belong to more than one community are represented as a pie chart with sections corresponding to extent of membership. The visualization was drawn using code from Lucas Jeub and used position data from a modifed Kamada-Kawaii spring force layout algortihm that positions communities using the Fruchterman-Reingold algorithm(Traud et al., 2009)

Chapter 4

Discussion

4.1 Changes in transitivity over time

I calculated both the transitivity and the reciprocity (Section 1.2.3) of the complete graph (including candidate-candidate and donor-donor donations as well as refunds) but treated as undirected, of all donations in a particular cycle, from 1980 to 2010 (except for 2000 and 2002 due to technical errors in igraph, which returned spurious NaNs). As can be seen in Figure 4.1, the transitivity tends to increase from 1980 until 1990 whereupon it tends to decrease. There is an oscillating pattern where presidential years have a lower transitivity than the subsequent midterm election cycle (midterms). The change from an increasing trend to a decreasing trend in 1990 suggests a change in structure.

Recalling the definition of transitivity (Section 1.2.3), where a connected triangle would correspond to a loop of donations including one candidate-candidate or donor-donor edge and an unconnected triple could correspond to a donor-candidate-donor setup, it is not surprising that midterms have a higher transitivity than presidential election cycles. Presidential years tend to have a higher number of donors who donate directly to a single candidate or PAC, rather than PACs who often donate to allied organizations. This could relate to the increase in transitivity in midterms, as PACs could donate proportionately more in midterm elections.

4.2 Changes in reciprocity over time

In the context of the FEC data, reciprocity represents the relative proportion of donations that have been returned as well as mutual donations between, for example, PACs. I calculated the reciprocity for the complete graph of each year from 1980 to 2010 (except 2000 and 2002, again due to the same technical error in igraph) as shown in Figure 4.2. In the 1980s, there is an increasing, linear trend ($R^2 \approx .95$ with $p \approx .027$ for the slope), whereas from 1988 to 1992 it is decreasing.

After 1992, an oscillatory pattern develops where presidential cycles have a low, decreasing reciprocity, while midterm years have a reciprocity that decreases until 2010, but at a lower rate than midterms. Also interesting is the jump in reciprocity from 1986 to 1988, which coincides with a change in the structure

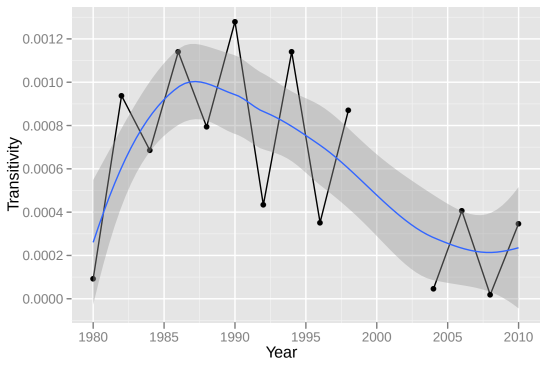


Figure 4.1: Transitivity versus year. A fitted LOESS curve is shown in blue.

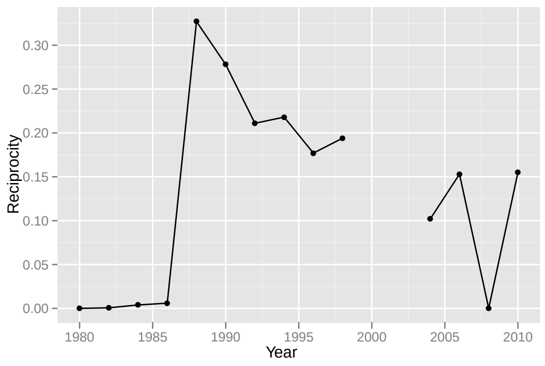


Figure 4.2: Plot of reciprocity versus year.

of donations(cfs). A historically high amount of new individual donors donated in 2008, because individuals are unlikely to have their donations refunded, which could explain why the reciprocity is 0.

4.3 Degree and Clustering Coefficient

I calculated the cumulative in-degree distribution of each election cycle. In the context of the FEC data, this represents the number of donations received by a person or organization. As typified by the 1980 cycle (shown in Figure 4.3), a negative trend appears in the frequency of vertices with a bump around 100. From degree between 1 and 100, the cumulative degree distribution decreases rapidly, even on this log scale, suggesting that the number of vertices with high degree is dramatically lower than those with low degree. This is not surprising because individual donors, who make up the vast proportion of the vertices, rarely donate to other individual donors.

At degree equal to 50, there is a noticeable inflection point and the shape of the distribution straightens while still decreasing. At the higher end of the graph (around degree > 1000), there is data sparsity. These overall features

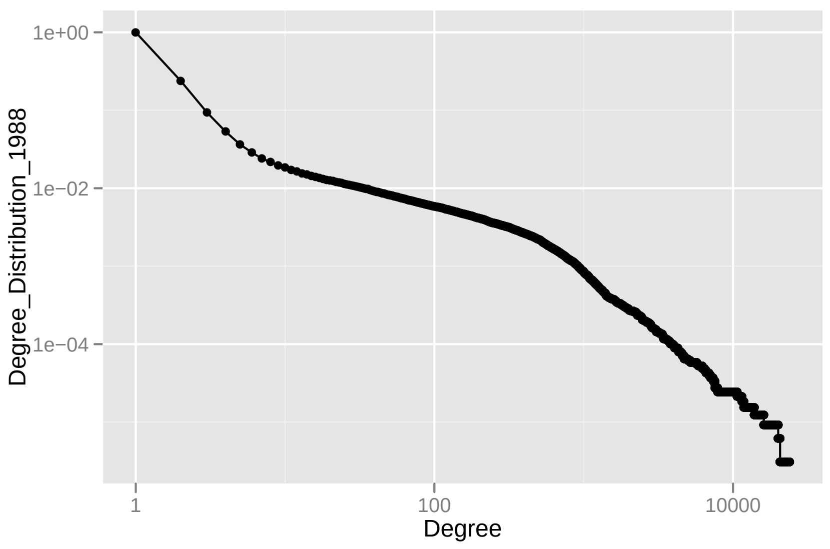


Figure 4.3: Cumulative degree distribution in 1980; note the undulations.

(rapid decrease, inflection point, straightening out and roughness) appear in the graph for cycles from 1980 to 2010.

I also calculated and plotted the local clustering coefficient (see Section 1.2.3) versus the (total) degree. Figure 4.4 is a typical example. All of the cycles had negative trend and a banana-shaped pattern in the 10 to 1000 degree range. This may show that as donations increase, the fraction of inter-organizational donations decreases.

To investigate the community structure of the networks, I ran the BKN algorithm on a graph created by ignoring candidate-candidate and donor-donor edges to create a bipartite network that I then projected onto the candidates using the process described in Section 1.2.4. This projected graph connects candidates if they shared a donor but does not take into account the strength of that connection, because the BKN algorithm cannot handle weights. Communities in this graph would group candidates who share donations and thus gain money from the same people. In order to see if they correspond to binary political attributes such as party affiliation, I enforced just two communities in the BKN algorithm to speed execution time. I found the structure depicted in Figure 4.5; the colors represent communities, however I found they do not correspond to party affiliation.

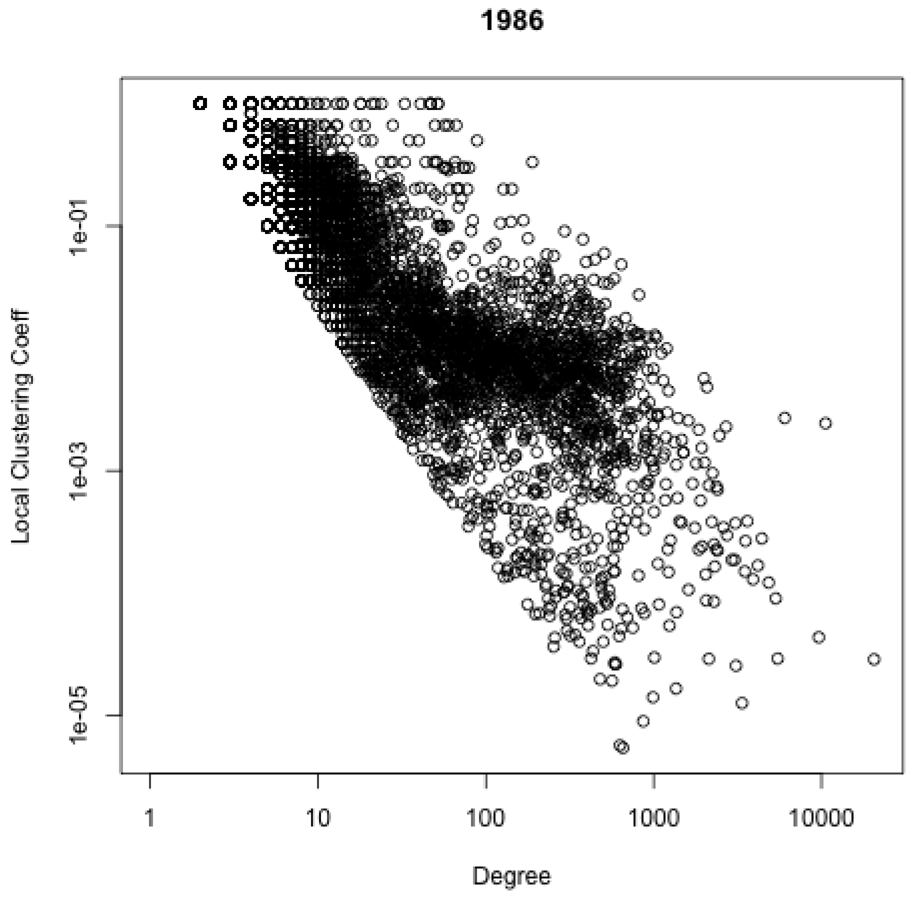


Figure 4.4: Local clustering coefficient versus degree in 1986.

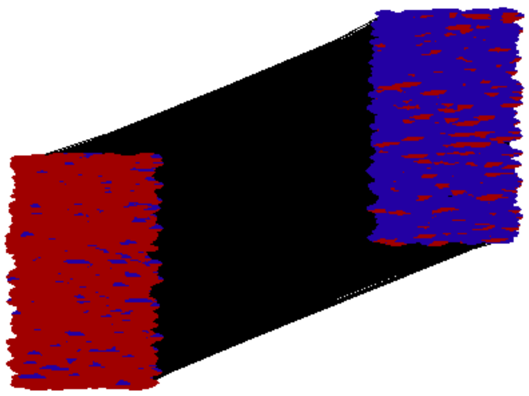


Figure 4.5: I enforced 2 communties in this run of the BKN algorithm on the modified projection of the 2002 election cycle. Red corresponds to the first community (grouped on the left) and blue to the second (on the right). The communities and colors do not correspond to party affiliation.

Chapter 5

Conclusion

In conclusion, I analyzed election donation data on US congressional elections from the 1980 presidential cycle to the 2010 midterm elections. There are three main features I found after analyzing the data:

1. The transitivity and reciprocity of the full, directed network–comprising all donations made within a cycle in a presidential cycle–is less than that of the subsequent midterm cycle. This could reflect differences in presidential and midterm cycles, particularly in the relative frequency of inter-organizational donations as compared to individual donors.
2. There is a large jump in reciprocity that occurs between 1986 and 1988. This corresponds to the discovery and exploitation of a 1979 ruling by the FEC that allowed more freedom in the use of soft-money, which is money not directly used by or for a particular candidate. This lead to an explosion in the number of PACs and non-candidate organizations which factored heavily into the Bush and Dukakis campaigns in 1988(cfs).
3. The transitivity peaks in 1990, tending to decline until 2010. This occurs after the 1988 soft-money discovery but could also be related to the greater use of PACs following 1988.

In order to do this, I wrote Python code, listed in Appendix A, to import the code into sqlite3 databases. I also wrote Python code to sanitize the data, and I created graphs from that data by writing SQL and Python code, also listed in Appendix A. In addition, I wrote MATLAB code, listed in Appendix C, that implements the Ball-Karrer-Newman (BKN) algorithm described in Section 3.1. This is an implementation of a recently published community detection algorithm for undirected, unweighted graphs that allows for overlapping communities. Because the BKN algorithm cannot handle directed or weighted graphs (detailed in Section 3.2.4), I ran it on a modified projection of the original graph onto the candidates.

I wrote Python code to develop a modified projection without weights, which is detailed in Appendix B. The number of communities in the BKN algorithm must be pre-specified, I specified 2 communities for simplicity and increased performance and then ran the BKN algorithm on the modified projections. The community structure I found does not correspond to party affiliation (see Figure 4.5).

5.1 Future Work

There are more political questions left to answer:

1. How do rule changes in election laws influence community structure in the full network?
2. Which overall network quantities of the full network correspond to rule changes, if any?
3. How do other network properties differ between mid-term and presidential election cycles? I found some differences in transitivity and reciprocity, but we would like to see how other quantities, such as modularity and betweenness, change.
4. How does the community structure of the full network change over time?

As an extension to the project, I plan to implement the BKN algorithm in a language that has lower runtime overhead (such as C++) and explore approaches to incorporating weighted and directed edges. I also plan on programmatically selecting the number of communities by first running a different community detection algorithm and using the number of communities detected by that algorithm as an input to the BKN algorithm. Other algorithms, such as the non-negative matrix factorization, do not pre-suppose the number of communities(Psorakis et al., 2011).

Appendix A

Python Code: Processing Raw Text Files

I wrote this code to process the raw text files of data supplied by Andrew Waugh. This involved importing the raw CSV formatted files into a sqlite3 database followed by significant preprocessing to remove spurious values (for example, malformed dates often occurred).

```python
import networkx as nx
import csv
import sqlite3
import os
import itertools
import matplotlib.pyplot as plt
import matplotlib
import datetime


def make_db(year):
    conn = sqlite3.connect('/Users/Nimish/projects/fec/text/graph' + year + '.db')
    c = conn.cursor()
    c.execute('''CREATE TABLE edgelist%s (sendID text not null, recID text not null,amount

    c.execute('''CREATE TABLE nodes%s (nodeid tex,nodetype text,name text,incumb text, dis

    conn.commit()
    c.close()

def grouper(n, iterable, fillvalue=None):
    "grouper(3, 'ABCDEFG', 'x') --> ABC DEF Gxx"
    args = [iter(iterable)] * n
    return itertools.izip_longest(fillvalue=fillvalue, *args)

def parse_amt(amtstr):
    factor =1
    amtstr = amtstr.lstrip('0').rstrip(']')
    if len(amtstr) <= 1:
        return 0
    lastchr = amtstr[-1]
    if lastchr in 'Kk':
        factor = 1000
    elif lastchr in 'Mm':
        factor = 1000*1000
    else:
        return int(amtstr)
    return int(amtstr) * factor


def import_year(year):
    dbname = 'graph'+year+'.db'
    ntn = 'nodes'+year
    etn = 'edgelist'+year
    nodefilename = 'fec_nodelist_txt_'+year+'.txt'
    edgefilename = 'fec_edgelist_txt_'+year+'.txt'
    print "Processing year %s\n" % year
    print "Processing nodes\n"
    nf = open(nodefilename,'rU')
    ef = open(edgefilename,'rU')
    conn = sqlite3.connect(dbname)
    nr = csv.reader(nf)
    print nr.next() # header
    for idx,chunk in enumerate(grouper(250000,nr,fillvalue=None)):
        #c.execute('BEGIN TRANSACTION')
        print 'Processing chunk %s' % (idx)
        c = conn.cursor()
        for node_line in chunk:
            if node_line is not None:
                c.execute('insert into %s values (?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)' %n
        #c.execute('END TRANSACTION')
        conn.commit()
        c.close()
    er = csv.reader(ef)
    print 'Processing edges'
    print er.next() #headers
    num_bad_edges = 0
    for idx,chunk in enumerate(grouper(250000,er,fillvalue=None)):
        #c.execute('BEGIN TRANSACTION')
        print "Processing chunk %s" % (idx)
        c = conn.cursor()
        for edge_line in chunk:
            if edge_line is not None:
                #edge_line[2:6] = map(int,edge_line[2:6])
                try:
                    edge_line[2] = int(edge_line[2]) # make amount an int
                    c.execute('insert into %s values (?,?,?,?,?,?,?,?,?,?,?)' %etn ,tuple(
                except ValueError,ve:
                  num_bad_edges += 1
                    #  print edge_line
                  #  print ve
        conn.commit()
        c.close()
    print 'End processing %s ; Had %s bad edges.' %(year,num_bad_edges)
    conn.close()
        #c.execute('END TRANSACTION')

def cleanup_year(year):
    dbname = 'graph'+year+'.db'
    ntn = 'nodes'+year
    etn = 'edgelist'+year
    conn = sqlite3.connect(dbname)
    c = conn.cursor()
    print("Begin Processing %s"%(year))
    print ("Adding edgename")
    c.execute('alter table %s add column edgename text'% etn)
    print ("Adding date")
    c.execute('alter table %s add column date' %etn)
    print("Updating edgename")
    c.execute('update %s set edgename = sendID || \"->\" || recID' % etn)
    print("Updating date")
    c.execute('update %s set date = (case when cast(year as integer) >= 60 then \"19\" els
    print("Adding is_candidate")
    c.execute('alter table %s add column is_candidate integer' %ntn)
    print("Updating is_candidate")
    c.execute('update %s set is_candidate = case when status != \"NA\" then 1 else 0 end'
    print("Creating collapsed_edges")
    c.execute('create table collapsed_edges%s as select sendID,recID,sum(amount) as total_
    print("Committing")
    conn.commit()
    c.close()
    print("End Processing %s" %(year))
    conn.close()

def fix_dates(year):
    dbname = 'graph%s.db'%year
    etn = 'edgelist' + year
    conn = sqlite3.connect(dbname)
    c = conn.cursor()
    print "Fixing dates for %s" % year
    c.execute('update %s set date = case when length(date) > 10 then substr(date,3) else d
    print "Deleting strange dates"
    c.execute('delete from edgelist%s where cast(strftime(\"%%Y\",date) as integer) > %s'
    conn.commit()
    c.close()
    conn.close()


```

```python


def make_collapsed(year):
    dbname = 'graph'+year+'.db'
    ntn = 'nodes'+year
    etn = 'edgelist'+year
    conn = sqlite3.connect(dbname)
    c = conn.cursor()






    #os.system('sqlite3 -separator , %s \".import %s %s\" ' % (dbname,nodefilename,ntn))
    #print "Processing edges\n"
    #os.system('sqlite3 -separator , %s \".import %s %s\"' %(dbname,edgefilename,etn))
    #print "End processing year %s" % year



def plot_donations(year):
    dbname = 'graph%s.db' % year
    etn = 'edgelist'+year
    conn = sqlite3.connect(dbname)
    c = conn.cursor()
    l = list(c.execute('select date, count(*) from %s group by date'%etn))
    conn.close()
    dates =[]
    nums =[]
    for pair in l:
        try:
            dates.append(datetime.datetime.strptime(pair[0],'%Y-%m-%d'))
            nums.append(pair[1])
        except ValueError,ve:
            print pair
    plt.plot(dates,nums)
    y = int(year)
    y1 = y - 1
    y2 = y+1
    x1 = datetime.datetime.strptime(str(y1)+'-11-01','%Y-%m-%d')
    x2 = datetime.datetime.strptime(str(y2)+'-01-01','%Y-%m-%d')
    print x1
    print x2
    plt.xlim(x1,x2)
    plt.savefig('dons_per_day%s.png'%year)




```

```python



if __name__ == '__main__':
    for i in xrange(1982,2010,2):
        make_db(year)
```

Appendix B

Python Code: Creating Projections

I wrote this code to process the data stored in sqlite3 databases into a graph form handled by the NetworkX(Hagberg et al., 2008) library. In addition, it contains the projection code for implementing a faster projection algorithm than matrix multiplication.

```python
import networkx as nx
import csv
#from itertools import *
import sqlite3 as sql3
import scipy.io as sio
import fish
import os

def mk_gyear(year):
    os.system('sqlite3 -separator , -header graph%s.db "select * from collapsed_edges%s" >

def process_year_simple(year,filename):
    year = str(year)
    print 'Making digraph'
    dgr,cands = mk_digraph(year)
    print 'Projecting (%d candidates)' %len(cands)
    pr = simple_pc(dgr,cands)
    print 'Outputting'
    to_matrix(pr,filename,year,cands=cands)


def mk_digraph(year):
    f = open('/Users/nimish/projects/projects_old/fec/text/g%s.csv' % str(year),'rb')
    dr = csv.DictReader(f)
    gr = nx.DiGraph()
    cands = []
    for i,de in enumerate(dr):
        sid = de['sendID']
        is_candidate = sid.startswith(('H','S','P'))
        if is_candidate:
            cands += [sid]
        gr.add_node(sid,attr_dict={'is_candidate':is_candidate})
        #TODO: make sure ints are parsed correctly in attr_dict
        gr.add_edge(de['sendID'],de['recID'],attr_dict=de)
        fish.animate()
    f.close()

    return gr,cands

def simple_pc(dgr,cands):
    g = nx.Graph()
    num_cands = len(cands)
    p = fish.ProgressFish(total=num_cands)
    for i,c in enumerate(cands):
        #if (i % 100) == 0:
        #    print 'Processed %d/%d candidates.' % (i,num_cands)
        succs = dgr.successors(c)
        preds = dgr.predecessors(c)
        for pred in preds:
            pred_succs = dgr.successors(pred)
            for ps in pred_succs:
                shared_succs = set(dgr.successors(ps)) & set(succs)
                if  len(shared_succs) > 0:
                    g.add_edge(c,ps)
        p.animate(amount=i+1)

    return g


def mp_proj_cand():
    pass
def project_cands(dgr,cands):
    g = nx.Graph()
    num_cands = len(cands)
    p = fish.ProgressFish(amount = num_cands)
    for i,c in enumerate(cands):
        #if (i % 10) == 0:
        #    print 'Processed %d/%d candidates.' % (i,num_cands)
        succs = dgr.successors(c)
        preds = dgr.predecessors(c)
        for pred in preds:
            pred_succs = dgr.successors(pred)
            for ps in pred_succs:
                shared_succs = set(dgr.successors(ps)) & set(succs)
                num_shared_nbrs = len(shared_succs)
                if num_shared_nbrs > 0:
                    tot_shared_dons = 0
                    tot_num_shared_dons = 0
                    for ss in shared_succs:
                        for ss_nbr,ss_nbr_info in dgr[ss].iteritems():
                            tot_num_shared_dons += int(ss_nbr_info['num_donations'])
                            tot_shared_dons += int(ss_nbr_info['total_sum'])
                    g.add_edge(c,ps,attr_dict=
                            {'num_shared_nbrs':num_shared_nbrs
                             ,'tot_shared_dons':tot_shared_dons
                            , 'tot_num_shared_dons':tot_num_shared_dons})
                else:
                    pass
        p.animate(amount=i+1)

    return g

def to_matrix(g,filename,year,cands = None,weight='weight'):
    mat = None
    if cands is not None:
        mat = nx.to_numpy_matrix(g,nodelist=cands.sort(),weight=weight)
    else:
        mat = nx.to_numpy_matrix(g)
    sio.savemat(filename,{'mat' + str(year):mat})
```

Appendix C

MATLAB Code: Ball-Karrer-Newman

The following is my implementation of the core optimization of the BKN algorithm, described in Section 3.1. It is run 25 times and the run with the highest log-likelihood, called fitness in the code, is chosen as the best assignment of edges to communities.

```
function [k,varargout] = BKN(A,K)
% Returns k,[colors]
%A = adjacency matrix
%K = number of communities
[r,c] = size(A);
n = r;
%initialize q's instead
k = zeros(n,K);
k2 = zeros(n,K);
iter = 0;
max_iter = 50;
tol = .0001;
delta = max(0.1,1/K * .9);
active = sparse(A);
nodes = 1:n;


%if A is symmetric (as the BKN algo requires), this is fine
[indrow,indcol]=find(A);

edges=[indrow(indcol>indrow),indcol(indcol>indrow)];
node_converged = false(n,1);
num_edges = size(edges,1);

%randomly initialize the q's (enforcing that they are probabilities)
%then calculate the k's
%for m=1:num_edges
%     i = edges(m,1);
%     j = edges(m,2);
%     qij=rand(1,K);
%     qij=qij/sum(qij);
%     k(i,:)=k(i,:)+qij;
%     k(j,:)=k(j,:)+qij;
%end
k = rand(n,K) + 1;

mmerr = max(max(abs(k-k2)));


while mmerr > tol && iter < max_iter

    kappa = sum(k,1);
    k2 = zeros(n,K);

    %find the active edges (edges for which at least one node is
    %unconverged)
    [ir,ic] = find(active);
    active_edges = [ir(ic>ir),ic(ic>ir)];
    num_active_edges = size(active_edges,1);
    fprintf('Active edges left:%d; active nodes left = %d\n',num_active_edges,sum(~node_co




    %update the k's
    for en = 1:num_active_edges

        i = active_edges(en,1);
        j = active_edges(en,2);
        if node_converged(i) && node_converged(j)
            fprintf('i = %d,j=%d\n',i,j);
            active(i,j) = 0;
        else
            qij = (k(i,:) .* k(j,:)) ./ kappa;
            D = sum(qij);
            qij = qij / D;
            %update new values of k
            k2(i,:) = k2(i,:) + qij;
            k2(j,:) = k2(j,:) + qij;
        end
    end

    %ensure that the converged value of k is preserved
    converged_nodes = nodes(node_converged);
    k2(converged_nodes) = k(converged_nodes);

    %find the converged nodes early
    unconverged_nodes = nodes(~node_converged);
    for i=unconverged_nodes
        %can't vectorize this for parsing reasons
        node_converged(i) = ((sum(k(i,:)>delta)==1));
    end

    mmerr = max(max(abs(k-k2)));

    iter = iter + 1;

    m = mod(iter,50);
    if m == 0
        fprintf('Iter #%d ; Error = %d\n',iter,mmerr);
    end




    k = k2;





end

fprintf('Error = %d\n',mmerr);
fprintf('Iteration stopped at iter = %d\n',iter);
%sum(k) = deg node; initialize q's
%for z = 1:K
%        kappa(1,z) = sum(k(:,z));
%end

kappa = sum(k,1);
colors=zeros(n,n);
theta = k;
for i = 1:n
    theta(i,:) = k(i,:) ./ sqrt(kappa);
end

fitness = 0;

for en = 1:num_edges
        i = edges(en,1);
        j = edges(en,2);
        etn = dot(theta(i,:),theta(j,:));
        % etn is the expected total # of edges between i and j

        if etn > 0
            fitness = fitness + A(i,j)*log(etn) - etn;
        end


        % calc only if we need the edge's communities

        if nargout > 1
            qij = (k(i,:) .* k(j,:)) ./ kappa;

            %following two lines unnecessary since we only care about max
            D = sum(qij);
            qij = qij / (D);

            [max_q,idx] = max(qij);
            colors(i,j) = idx;
        end

end

if nargout > 1
    varargout{1} = colors;
end
fprintf('Log-likelihood is: %d\n',fitness)
```

Appendix D

Codebook for FEC Edgelist Files

The structure of the files containing edge data is described by the following key, supplied by Andrew Waugh and adapted from the FEC website (Detailed Files About Candidates, Parties and Other Committees).

```
1. sendID - ID of the sender (identified in FEC Node List file)

2. recID - ID of the recipient (identified in FEC Node List file)

3. amount - Dollar amount of the transfer

4. month - Month of transfer

5. day - Day of transfer

6. year - Year of transfer

7. amend - Amendment Indicator
A Amendment
C Consolidated
M Multi-Candidate
N New
S Secondary
T Terminated

8. reptyp - Report Type
10D PRE-ELECTION
10G PRE-GENERAL
10P PRE-PRIMARY
10R PRE-RUN-OFF
10S PRE-SPECIAL
12C PRE-CONVENTION
```

```
12G PRE-GENERAL
12P PRE-PRIMARY
12R PRE-RUN-OFF
12S PRE-SPECIAL
30D POST-ELECTION
30G POST-GENERAL
30P POST-PRIMARY
30R POST-RUN-OFF
30S POST-SPECIAL
60D POST-ELECTION
ADJ COMP ADJUST AMEND
CA COMPREHENSIVE AMEND
M1 JANUARY MONTHLY
M10 OCTOBER MONTHLY
M11 NOVEMBER MONTHLY
M12 DECEMBER MONTHLY
M2 FEBRUARY MONTHLY
M3 MARCH MONTHLY
M4 APRIL MONTHLY
M5 MAY MONTHLY
M6 JUNE MONTHLY
M7 JULY MONTHLY
M8 AUGUST MONTHLY
M9 SEPTEMBER MONTHLY
MY MID-YEAR REPORT
Q1 APRIL QUARTERLY
Q2 JULY QUARTERLY
Q3 OCTOBER QUARTERLY
TER TERMINATION REPORT
YE YEAR-END
90S POST INAUGURAL SUPPLEMENT
90D POST INAUGURAL
48H 48 HOUR NOTIFICATION
24H 24 HOUR NOTIFICATION

9. trnstyp - Transaction Type
10 NON-FEDERAL RECEIPT FROM PERSONS LEVIN (L-1A)
11 TRIBAL CONTRIBUTION
12 NON-FEDERAL OTHER RECEIPT LEVIN  (L-2)
13 INAUGURAL DONATION ACCEPTED
15 CONTRIBUTION
15C CONTRIBUTION FROM CANDIDATE
15E EARMARKED CONTRIBUTION
15F LOANS FORGIVEN BY CANDIDATE
15I EARMARKED INTERMEDIARY IN
15J MEMO (FILER'S \% OF CONTRIBUTION GIVEN TO JOIN
15T EARMARKED INTERMEDIARY TREASURY IN
15Z IN-KIND CONTRIBUTION RECEIVED FROM REGISTERED
16C LOANS RECEIVED FROM THE CANDIDATE
16F LOANS RECEIVED FROM BANKS
```

```
16G LOAN FROM INDIVIDUAL
16H LOAN FROM CANDIDATE/COMMITTEE
16J LOAN REPAYMENTS FROM INDIVIDUAL
16K LOAN REPAYMENTS FROM CANDIDATE/COMMITTEE
16L LOAN REPAYMENTS RECEIVED FROM UNREGISTERED EN
16R LOANS RECEIVED FROM REGISTERED FILERS
16U LOAN RECEIVED FROM UNREGISTERED ENTITY
17R CONTRIBUTION REFUND RECEIVED FROM REGISTERED
17U REF/REB/RET RECEIVED FROM UNREGISTERED ENTITY
17Y REF/REB/RET FROM INDIVIDUAL/CORPORATION
17Z REF/REB/RET FROM CANDIDATE/COMMITTEE
18G TRANSFER IN AFFILIATED
18H HONORARIUM RECEIVED
18J MEMO (FILER'S \% OF CONTRIBUTION GIVEN TO JOIN
18K CONTRIBUTION RECEIVED FROM REGISTERED FILER
18S RECEIPTS FROM SECRETARY OF STATE
18U CONTRIBUTION RECEIVED FROM UNREGISTERED COMMI
19 ELECTIONEERING COMMUNICATION DONATION RECEIVE
19J MEMO (ELECTIONEERING COMMUNICATION \% OF DONAT
20 DISBURSEMENT - EXEMPT FROM LIMITS
20A NON-FEDERAL DISBURSEMENT LEVIN (L-4A) VOTER R
20B NON-FEDERAL DISBURSEMENT LEVIN (L-4B) VOTER I
20C LOAN REPAYMENTS MADE TO CANDIDATE
20D NON-FEDERAL DISBURSEMENT LEVIN (L-4D) GENERIC
20F LOAN REPAYMENTS MADE TO BANKS
20G LOAN REPAYMENTS MADE TO INDIVIDUAL
20R LOAN REPAYMENTS MADE TO REGISTERED FILER
20V NON-FEDERAL DISBURSEMENT LEVIN (L-4C) GET OUT
22G LOAN TO INDIVIDUAL
22H LOAN TO CANDIDATE/COMMITTEE
22J LOAN REPAYMENT TO INDIVIDUAL
22K LOAN REPAYMENT TO CANDIDATE/COMMITTEE
22L LOAN REPAYMENT TO BANK
22R CONTRIBUTION REFUND TO UNREGISTERED ENTITY
22U LOAN REPAID TO UNREGISTERED ENTITY
22X LOAN MADE TO UNREGISTERED ENTITY
22Y CONTRIBUTION REFUND TO INDIVIDUAL
22Z CONTRIBUTION REFUND TO CANDIDATE/COMMITTEE
23Y INAUGURAL DONATION REFUND
24A INDEPENDENT EXPENDITURE AGAINST
24C COORDINATED EXPENDITURE
24E INDEPENDENT EXPENDITURE FOR
24F COMMUNICATION COST FOR CANDIDATE (C7)
24G TRANSFER OUT AFFILIATED
24H HONORARIUM TO CANDIDATE
24I EARMARKED INTERMEDIARY OUT
24K CONTRIBUTION MADE TO NON-AFFILIATED
24N COMMUNICATION COST AGAINST CANDIDATE (C7)
24P CONTRIBUTION MADE TO POSSIBLE CANDIDATE
24R ELECTION RECOUNT DISBURSEMENT
```

```
24T EARMARKED INTERMEDIARY TREASURY OUT
24U CONTRIBUTION MADE TO UNREGISTERED
24Z IN-KIND CONTRIBUTION MADE TO REGISTERED FILER
29 ELECTIONEERING COMMUNICATION DISBURSEMENT(S)

10. primgen - Primary-General Indicator
C Convention
G General
P Primary
R Runoff
S Special

11. altrecID - Alternate Recipient ID
(Candidate ID for candidate committees, Committee ID for candidates, if applicable)
```

Appendix E

Codebook for FEC Nodelist Files

The structure of the files containing node data is described by the following key, supplied by Andrew Waugh and adapted from the FEC website(Detailed Files About Candidates, Parties and Other Committees).

```
1. nodeid - Node Identification Number

2. nodetype - Node Type
C Committee
H House Candidate
S Senate Candidate
P Presidential Candidate
I Individual
U Individual/Other Stored in the Committee-Committee FEC Files

3. name - Node Name

4. incumb - Incumbency Status (for Candidates only)
I Incumbent
C Challenger
O Open Seat

5. district - Congressional District (for House Candidates only)

6. elecyear - Election Year (for Candidates only)

7. party - Primary Political Party Affiliation

8. city - City
```

```
9. state - State

10. zip - Zip Code

11. altid - Alternate ID
(Candidate ID for candidate committees, Committee ID for candidates, if applicable)

12. occup - Occupation (for Individuals)

13. status - Candidate Status (for Candidates only)
C Statutory Candidate
(declared candidate who has raised or spent $5000)
F Statutory Candidate for a future election
N Not yet a Statutory Candidate
(candidate has declared but not raised or spent $5000)
P Statutory candidate for a prior electoral cycle

14. desig - Committee Designation (for Committees)
A Authorized by Candidate
J Joint Fund Raiser
P Principal Campaign Committee of a Candidate
U Unauthorized
B Lobbyist/Registrant PAC
D Leadership PAC

15. ctype - Committee Type (for Committees)
C Communication Cost
D Delegate
H House
I Independent Expenditure
N Non-party Non-qualified
P Presidential
Q Qualified Non-Party
S Senate
X Non-qualified Party
Y Qualified Party
Z National Party Organization, Non Federal Account.
E Electioneering Communications

16. filefreq - Filing frequency with the FEC (for Committees)
A Administratively Terminated
D Debt
M Monthly Filer
Q Quarterly Filer
T Terminated
W Waived

17. intcat - Interest Group Category
(for Committee types N and Q only)
C Corporation
```

```
L Labor Organization
M Membership Organization
T Trade Association
V Cooperative
W Corporation without Capital Stock

18. sponsor - The Reported Name of a Committee's Sponsor (for Committees)
```

Bibliography

Brian Ball, Brian Karrer, and M. E. J. Newman. Efficient and principled method for detecting communities in networks. *Phys. Rev. E*, 84, 2011.

Sean Borman. The expectation maximization algorithm – a short tutorial. July 2004. URL `http://www.seanborman.com/publications/EM_algorithm.pdf`.

Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006. URL `http://igraph.sf.net`.

Detailed Files About Candidates, Parties and Other Committees. Detailed files about candidates, parties and other committees. November 2011. URL `http://www.fec.gov/finance/disclosure/ftpdet.shtml`.

Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, 2008.

D. Lazer. Networks in political science: Back to the future. *PS: Political Science & Politics*, 44(01):61–68, 2011.

Thomas E. Mann. Linking knowledge and action: Political science and campaign finance reform. *Perspectives on Politics*, 1(1):pp. 69–83, 2003. ISSN 15375927. URL `http://www.jstor.org/stable/3687813`.

Mark E. J. Newman. *Networks: An Introduction*. Oxford University Press, Inc., New York, NY, USA, 2010.

M.E.J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006.

M.A. Porter, P.J. Mucha, M.E.J. Newman, and C.M. Warmbrand. A network analysis of committees in the us house of representatives. *Proceedings of the National Academy of Sciences of the United States of America*, 102(20):7057, 2005.

M.A. Porter, J.-P. Onnela, and P.J. Mucha. Communities in networks. *Notices of the AMS*, 56(9):1082–1097, 2009.

Ioannis Psorakis, Stephen Roberts, Mark Ebden, and Ben Sheldon. Overlapping community detection using bayesian non-negative matrix factorization. *Phys. Rev. E*, 83, 2011.

A.L. Traud, C. Frost, P.J. Mucha, and M.A. Porter. Visualization of communities in networks. *Chaos*, 19(4), 2009.

M. D. Ward, K. Stovel, and A. Sacks. Network analysis and political science. *Annual Review of Political Science*, 14:245–264, 2011.

Wayne W. Zachary. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research*, 33(4):pp. 452–473, 1977.