InvestorRank and an inverse problem for PageRank

Permalink
https://escholarship.org/uc/item/2qw4b90x

Author
Sims, Bryan

Publication Date
2012-05-10

Peer reviewed|Thesis/dissertation
InvestorRank and an Inverse Problem for PageRank

Bryan Sims

A technical report submitted in partial fulfillment of the requirements for the degree of Master of Science in Applied Mathematics

May, 2012
This is to certify that I have examined a copy of a technical report by

Bryan Sims

and found it satisfactory in all respects, and that any and all revisions
required by the examining committee have been made.

Research Advisor: ________________________________
Harish Bhat

Reading Committee: ______________________________
Roummel Marcia

Reading Committee: ______________________________
Avi Shapiro

Applied Mathematics Graduate Studies Chair: ________________
Boaz Ilan

________________________________
Date
InvestorRank and an Inverse Problem for PageRank

by

Bryan Sims

May, 2012

University of California, Merced

Abstract

We explore methods that allow us to change an existing network’s structure to achieve a desired PageRank. Being able to change the ranking of the nodes is desirable for any network for which the PageRank determines the importance of the nodes. We search for a perturbation to the original network that when added will give the desired ranking. We derive multiple algorithms to solving this Inverse PageRank problem using both constrained optimization techniques and root finding approaches. In these approaches, we ensure that the answer we find satisfies the conditions for an adjacency matrix of an undirected graph, and we explore techniques that promote sparsity of the final solution. We want sparse solutions so that we only have to change a fraction of the existing edges. We develop a general approach that can be applied to any network. We describe C++ implementations for a set of algorithms, the best of which is scalable to thousands of vertices and much faster than Matlab-based approaches we have tried. This same set of algorithms is subjected to a number of tests designed to study the relationship between the initial guess and the final solution. Our ultimate goal is to solve the inverse PageRank problem on a social network of over 6000 venture capitalists to try and reproduce the top 10 list from a previously published set of rankings. Using a root-finding algorithm where nonnegativity constraints are handled implicitly, we are able to find a highly sparse solution to this inverse problem. This thesis consists of a draft of a research paper, co-authored by my advisor and me, that we plan to submit to a peer-reviewed journal of applied/computational mathematics.
1 Introduction

PageRank was introduced by the founders of Google as a technique for ranking web pages [3]. To apply PageRank, we represent the web as a directed graph in which vertices correspond to web pages and edges correspond to hyperlinks from one page to another. The PageRank $P(A)$ is then purely a function of the adjacency matrix $A$ of the graph. We refer to the mapping $A \mapsto P(A)$ as the forward PageRank problem. In this thesis, we design and test algorithms for the solution of the inverse PageRank problem: given an adjacency matrix $A$ and a desired PageRank vector $P_0$, the inverse PageRank problem considered in this work is to find a perturbation $\varepsilon$ that minimizes the mismatch $\|P(A + \varepsilon) - P_0\|_2$. Solving this minimization problem for a particular choice of metric yields the solution of the inverse problem, which we can view as a mapping $(A, P_0) \mapsto \varepsilon$.

The inverse PageRank problem is of broad interest to several different fields. While the original PageRank algorithm has been updated and modified in several ways, it is still an ingredient in the search rankings used by Google [13]. The PageRank algorithm has been applied to rank the importance of proteins in protein interaction networks [22, 23, 16], to rank genes using data from microarray experiments [18], and to rank venture capitalists in an abstract social network of coinvestment [21]. In these applications, the graphs are typically undirected and weighted, which leads to an inverse problem that can be solved using continuous optimization and/or root-finding. The problem where we must find a perturbation $\varepsilon$ such that $A + \varepsilon$ remains unweighted—with entries either equal to 0 or 1—leads to a much more difficult discrete optimization problem that we do not discuss here.

In this thesis, we target the application of our inverse PageRank algorithms to the InvestorRank problem [21]. The venture capitalist Chris Farmer developed InvestorRank primarily using CrunchBase (http://www.crunchbase.com/), a public domain database of venture capitalists and startup companies that includes information on funding deals [24]. Farmer built a coinvestment graph in which each venture capitalist (VC) corresponds to a vertex and an edge between two VC’s means they have co-invested in the same startup company, possibly at different times. Applying PageRank to this coinvestment social network, Farmer obtained a ranking of VC’s. Note that only the top 10 VC’s were reported. In our initial work, we mined the same CrunchBase data set, produced an adjacency matrix for the coinvestment graph, and applied PageRank to obtain our own top 10 ranking. However, our top 10 did not agree with Farmer’s top 10, leading to a natural question: how can we perturb our adjacency matrix in order to obtain Farmer’s rankings? Using our algorithm, we provide an answer to this question in Section 6.2.

The applications cited above also indicate that the inverse PageRank problem may be viewed as a statistical or data mining problem. Suppose one has information about the ranking of vertices but lacks complete information about the edge weights of the graph. Then a method for solving the inverse problem is a method for inferring the missing edge weights from the available data, an example of a network inference problem [12]. Since many graph data sets are massive in size, issues of scalability often play a role in the solution of such problems. The InvestorRank problem with its approximately $6 \times 10^3$ vertices is far smaller than the PageRank problem for the web graph, as there are over $3 \times 10^{10}$ web pages indexed by Google [3]. However, a key aspect of our solution of the inverse InvestorRank problem is the ability to select a number $\tilde{n}$ of vertices whose edge weights can be modified that is much smaller than $n$, the total number of vertices in the graph. This enables us to solve the inverse PageRank problem for the InvestorRank network on a standard, multicore desktop computer.

Methods and Results. We approach the problem in two different ways. In the first approach, we formulate the inverse problem as a constrained optimization problem. The idea is to find
the smallest perturbation $\varepsilon$ that we need to add to the existing adjacency matrix $A^0$ where the PageRank constraint is satisfied for a desired PageRank $P^0$. We solve this optimization problem using Lagrange multipliers coupled with Newton’s method, using analytical gradients and Hessians of the objective function and constraints. With this approach, we are able to find solutions $\varepsilon$ for the case where the objective function is the squared Frobenius norm of $\varepsilon$. However, our results show that these solutions tend to be dense. Moreover, because any method involving either an exact or approximate Hessian will involve a matrix that grows like $n^2 \times n^2$, where $n$ is the number of vertices in the graph, we find that this approach does not scale well.

This motivates the second approach, in which the inverse problem is formulated as an underconstrained root-finding problem, where we solve the nonlinear PageRank constraint equations directly. We define a function $f : \mathbb{R}^{n^2} \to \mathbb{R}^{n+m}$ and apply Newton’s Method to find a solution to $f(\varepsilon) = 0$. Here $m$ is the number of equations required to enforce the graph-based constraints that $\varepsilon_{ii} = 0$ (no self-loops) and that $\varepsilon_{ij} \geq 0$ when $i \neq j$ (no deletion of existing edges of $A^0$). Note also that we constrain $\varepsilon$ to be symmetric by setting $\varepsilon = \frac{1}{2}(\kappa + \kappa^T)$ and solving for $\kappa$; this ensures that $A + \varepsilon$ is a valid adjacency matrix for an undirected graph.

Using Newton’s method with the pseudoinverse of the Jacobian of $f$, we first solved this problem for a full-sized $n \times n$ matrix $\varepsilon$. To enforce the graph-based constraints, we use an active-set approach that leads to a worst-case value of $m = n^2$. While this approach is only marginally more scalable than the optimization approach described above, it does yield sparse solutions, a desirable feature.

Next, we reimplement the pseudoinverse Newton’s method, this time enforcing the nonnegativity of off-diagonal elements of $\varepsilon$ by setting $\varepsilon_{ij} = \frac{1}{2} \left( \kappa_{ij}^2 + k_{ji}^2 \right)$. Now we obtain a value of $m = n$ since the only graph-based constraints that must be enforced are the constraints on the diagonal of $\varepsilon$, or equivalently, $\kappa$.

This yields a function $f : \mathbb{R}^{n^2} \to \mathbb{R}^{2n}$, leading to a much more scalable linear problem that must be solved at each iteration. We further improve the scalability of this approach by constraining $\kappa$ to be a matrix that is zero outside of an $\tilde{n} \times \tilde{n}$ upper-left block, with $\tilde{n} < n$. This approach proves to be the most successful in terms of both scalability and sparsity, and is the approach used to solve the inverse problem for the InvestorRank network.

Other Approaches and Related Work. To our knowledge, this thesis represents the first attempt at solving the inverse PageRank problem using continuous optimization and/or root-finding methods. We now review work that is related to our work and explain other approaches that have been taken.

The most directly related work is that pertaining to PageRank optimization. Here the goal is to maximize the PageRank of a particular vertex by adjusting the weights of edges that belong to a subset of all edges in the graph. Proposed solutions to this problem involve Markov decision theory \cite{12, 13}. Continuous relaxations of the problem, similar to the continuous problem studied here, have been solved using constrained nonlinear programming \cite{14, 15}. These problems are different from ours chiefly in the absence of a desired PageRank vector $P^0$. The proposed solutions do allow for coupling constraints, i.e., constraining one vertex’s PageRank to be larger than another’s, but it is unclear how these algorithms would perform if $n$ such constraints were included.

Other optimization problems such as maximizing the sum of PageRank scores for a set of web pages \cite{16} or maximizing the PageRank of a tree \cite{17} have also been considered.

A related stream of work studies the sensitivity of PageRank to perturbations in the adjacency matrix. In one of the first works on this topic \cite{18}, we can find a bound in the $L^1$ norm of the change in the PageRank vector. This work shows that perturbing the edge weights of low-
ranked vertices yields only a small change in the PageRank vector, but leaves open the question of whether the small numerical change might involve large shifts in the ranking of the vertices. More recent work \cite{7} explicitly addresses the sensitivity of the ranking provided by the PageRank vector and shows that large changes in PageRank rankings are possible with only small perturbations to the adjacency matrix. This work generalizes earlier work \cite{2} that analyzed the effect of rank-one updates to the PageRank transition probability matrix on the PageRank of one vertex. There has also been work on determining lower- and upper-bounds on the error of the PageRank vector when it is known that a fraction of the edges in the graph might be (temporarily) unobservable \cite{15}.

Issues not addressed in this thesis include proofs of convergence for the numerical methods that are proposed. Our chief aim is the construction and experimental evaluation of approaches to the inverse problem, with an eye towards developing a body of numerical evidence that explains which implementation is the best in practice and for the targeted application.

2 Forward Problem for PageRank

PageRank was introduced by Brin and Page \cite{5} as a technique for ranking web pages that match a given search query. Since its introduction, PageRank has been modified in several ways, but in this thesis, we focus our attention on the original method.

Let $A$ be the adjacency matrix for a graph $G$, and let $D$ be the diagonal degree matrix: $D_{ii} = \sum_k A_{ik}$ and $D_{ij} = 0$ for $i \neq j$. Then the PageRank vector $P$ is defined by

$$P(A) = \frac{1}{N} \left[ I - \gamma AD^{-1} \right]^{-1} 1 \tag{1}$$

where $0 < \gamma < 1$ is a fixed constant and $1$ is the $N$-dimensional vector consisting of ones. In the numerical experiments described below, we use the common choice of $\gamma = 0.85$. We write $P(A)$ to emphasize that $P$ depends on $A$. This dependence is nonlinear, first because $AD^{-1}$ is a nonlinear function of $A$, and second because of the linear solve implied by the inverse of the square bracketed matrix multiplied by $1$.

Multiplying (1) through by $[I - \gamma AD^{-1}]$ and assuming that $1^T P = 1$, we see that $P$ also satisfies

$$MP = P, \tag{2a}$$

$$M = \gamma AD^{-1} + \frac{1 - \gamma}{N} 11^T. \tag{2b}$$

The matrix $M$ is the probability transition matrix for a Markov chain on the vertices of the graph. One can check that $1^T M = 1^T$, so that $M$ is a column stochastic matrix.

When $\gamma = 1$, the Markov chain represented by $M = AD^{-1}$ is called the random walk on the graph. In this case, the probability of transitioning from vertex $i$ to vertex $j$, assuming that $(i, j)$ is an edge in the graph, is equal to the reciprocal of the degree $D_{ii}$ of vertex $i$. In other words, the random walker starting at vertex $i$ chooses uniformly at random from the vertices adjacent to vertex $i$.

For $0 < \gamma < 1$, the matrix $M$ consists of the weighted combination of the random walk transition matrix $AD^{-1}$ with the matrix $(1/N) 11^T$. This latter matrix represents the probability transition matrix for a random walk on the complete graph on $N$ vertices—here, a random walker at any given vertex has an equally likely chance of transitioning to any of the $N$ vertices, including the one at which it is currently located.

From a modeling perspective, the idea is that a web surfer will have probability $\gamma$ of following...
any of the links on a given page, and have probability $1 - \gamma$ of “teleporting” to any page on the Internet.

In the original formulation of PageRank, the web graph was taken to be a directed graph with edge weights consisting only of 0’s and 1’s. The entry $A_{ij}$ equals one if and only if there is a link from web page $i$ to web page $j$.

We seek to apply PageRank to undirected graphs where the edge weights are non-negative real numbers. The entry $A_{ij}$ equals zero if and only if there is no edge between vertices $i$ and $j$. If $A_{ij} > 0$, then $A_{ij}$—the weight of the edge $(i, j)$—indicates the strength of the connection between vertices $i$ and $j$.

3 InvestorRank

3.1 InvestorRank Top 10 List

Let us now describe in more detail the InvestorRank problem and algorithm. InvestorRank was developed by the venture capitalist Chris Farmer using CrunchBase augmented with data from individual venture capitalist websites [24]. As Farmer has not published the algorithm used to produce the coinvestment social network or the InvestorRank rankings, we will first describe what can be determined from published discussions of InvestorRank [21, 24]. First, we know that in Farmer’s coinvestment social network, if two investors invested in the same company at the same time during a round of funding, the investors are joined by an edge. Second, we infer that there are at least two mechanisms that are used to boost an investor’s rank: (i) bringing a new investor to a company’s rounds of funding boosts the prior investors’ ranks, and (ii) if an investor continues to invest in a startup company across more than one round of funding, then that investor’s rank is boosted. With these criteria, Farmer produced the following top 10 ranking of venture capitalists:

1. Andreessen Horowitz
2. Sequoia Capital
3. Accel
4. Benchmark Capital
5. Union Square Ventures
6. General Catalyst Partners
7. NEA
8. Kleiner Perkins
9. Khosla Ventures
10. Greylock

3.2 Mining CrunchBase

In an attempt to reproduce these results, we wrote Python code that downloaded data from CrunchBase in early June, 2011. In this way, we obtained data on 14304 startup companies for which we have information about at least one round of funding. Our next steps were to process this data in the same way as Farmer. First, we changed all currencies to US dollars. Second, we included only those investment rounds with total funding of at least 100 million US dollars. Finally, we included only those investments that occurred from 2006 to 2011. We now let

$$\Omega = \{I_0, I_1, ..., I_j, ..., I_n\}$$
Figure 1: Using data that we mined from CrunchBase, we plot histograms of the distribution of funding amounts (left) and $\log_{10}$ funding amounts (right) across all rounds and all companies. The means and standard deviations for the two distributions are given above the corresponding plot.

represent the set of all the investors where $I_j$ is investor $j$. We looped through our database and went through each round of funding for each individual company. If two investors invested in the same company at the same time, we set $A_{ij} = A_{ji} = 1$. In this way, we produced a $6049 \times 6049$ adjacency matrix $A$ for the CrunchBase coinvestment social network.

After computing $A$, we realized that the Top 10 rankings computed via PageRank $P(A)$ do not agree with the Top 10 rankings given in Section 3.1. A close reading of Farmer’s comments on InvestorRank [21, 24] led us to hypothesize that three additional ideas may have played a role in Farmer’s rankings.

First, we examine the idea of an existing investor bringing a new investor into the next funding round, for a given company. Let $\tilde{B} \in \mathbb{R}^{n}$ be a vector that is initially zero. For each company, let $R_i \subseteq \Omega$ be the set of investors that invest in round $i$, so that $R_{i+1} \subseteq \Omega$ is the set of investors for the following round. Looping across all companies and all rounds of funding, we increment by one the entries of $\tilde{B}$ corresponding to the set $T_i = R_i \setminus R_{i+1}$ so long as $R_{i} \cap R_{i+1} \neq \emptyset$. In words, as long as round $i + 1$ has a new investor who was not present in round $i$, all the investors in round $i$ receive an increment of +1. After doing this across the whole database, the vector $\tilde{B}$ records the number of new investors each investor has attracted.

The second idea is to keep track of investors that continue to invest in the same company. We do this in a similar way as before. Let $\tilde{C} \in \mathbb{R}^{n}$ be a vector that is initially zero, and let $R_i$, $R_{i+1}$, and $T_i$ have the same meanings as before. Looping across all companies and all rounds of funding, we increment by one the entries of $\tilde{C}$ corresponding to the set $T_i$. After the loops are finished, the vector $\tilde{C}$ will record the number of repeated investments in the same company each investor has made.

The third idea is to see how much money a group of investors invested in a round of funding, and rank the investors based on how much was invested. In Figure 2, we present a histogram that gives the distribution of funding amounts across all rounds and all companies in our database. The horizontal axis is divided into bins of funding ranges, and the vertical axis records the number of investors who have participated in a funding round whose total amount falls into the given bin.

We can see that the amount invested within a round of funding has a steep decay and a long tail. This is intuitive—there will be relatively few massive rounds of funding for certain startup
companies, and there will be relatively frequent rounds where a very small amount of funding is provided. The quartiles of the empirical distribution corresponding to this histogram are:

\[
Q_{0.25} = 2888000 \\
Q_{0.50} = 6000000 \\
Q_{0.75} = 13600000
\]

Let \( \tilde{M} \in \mathbb{R}^n \) be a vector that is initially zero. For each company, let \( X_i \) be the amount of money invested in round \( i \). Looping through all rounds and all companies in our database, for each \( j \) in \( R_i \), we increment the entry \( M_j \) by \( \xi(X_i) \) where

\[
\xi(X) = \begin{cases} 
1 & : X \leq Q_{0.25} \\
5 & : Q_{0.25} < X \leq Q_{0.5} \\
10 & : Q_{0.5} < X \leq Q_{0.75} \\
15 & : X > Q_{0.75}
\end{cases}
\]

In the end, \( \tilde{M} \) will contain scores that relay the total size of all investments made by each investor.

3.3 Naïve Attempt to Reproduce InvestorRank Results

Let \( \tilde{P} = P(A) \). We normalize \( \tilde{P}, \tilde{B}, \tilde{C} \), and \( \tilde{M} \) so that each vector sums to one. Each such vector can be viewed as offering a different ranking of the vertices. In what follows, we describe a preliminary method we employed to try to use the information in these four vectors to obtain Farmer’s rankings. The method we are about to describe was ultimately unsuccessful, motivating the continuous optimization and root-finding approaches described in subsequent sections of the paper.

For \( \tilde{z} = [z_1, z_2, z_3, z_4] \), let

\[
\tilde{V}(\tilde{z}) = z_1 \tilde{P} + z_2 \tilde{B} + z_3 \tilde{C} + z_4 \tilde{M}.
\]

The rankings according to \( \tilde{V} \) are given by

\[
\tilde{R}(\tilde{z}) = G(\tilde{V}(\tilde{z})),
\]

where \( G \) denotes the index sort of the vector \( \tilde{V} \). Let \( S_n \) denote the space of permutations of the integers \( \{1, 2, \ldots, n\} \), and let the \( j \)-th component of \( G \) be denoted \( G_j \). Then the index sort is the function \( G : \mathbb{R}^n \to S_n \) such that \( [V_{G_1}, V_{G_2}, \ldots, V_{G_n}] \) is the vector \( \tilde{V} \) sorted in descending order from largest to smallest entry. Hence \( G_j \) is the index of the \( j \)-th largest element of \( \tilde{V} \), and thus \( G \) is precisely a list of the ID numbers of the venture capitalists sorted from highest to lowest ranking.

Let \( \tilde{F} \) denote the indices of Farmer’s top 10 venture capitalists. We now set up an optimization problem to see which mixture of coefficients \( \tilde{z} \) will give us the most matches to Farmer’s top 10. Let

\[
Q(\tilde{z}) = 10 - |\tilde{R}_{1:10}(\tilde{z}) \cap \tilde{F}|.
\]

Here \( \tilde{R}_{1:10}(\tilde{z}) \) denotes the first 10 entries of the vector \( \tilde{R}(\tilde{z}) \) and \( |\cdot| \) denotes cardinality. We treat both the \( \tilde{R}_{1:10} \) and \( \tilde{F} \) vectors as sets of integers, so that \( \tilde{R}_{1:10} \cap \tilde{F} \) is the set of integers common to both sets. The cardinality of this intersection measures the number of venture capitalists in Farmer’s top 10 list that are present, in any order, in our top 10 list. The maximum cardinality is
10, and so $Q$ measures how far away we are from having a perfect match in terms of the identities—but not necessarily the ordering—of the venture capitalists across both our and Farmer’s top 10 lists.

The optimization problem is then to find

$$\tilde{z}_* = \arg\min_{\tilde{z}} Q(\tilde{z}).$$

(5)

As $Q$ is not a differentiable function of $\tilde{z}$, we use Nelder-Mead minimization in Matlab to solve this problem. Since Nelder-Mead only finds a local minimum, we run this optimization a number of times, each time with different initial guesses, to try to have a better chance of hitting the global minimum. Constraining the entries of $\tilde{z}$ to be nonnegative, we obtain the following top 10:

1. intel-capital
2. new-enterprise-associates
3. accel-partners
4. draper-fisher-jurvetson
5. kleiner-perkins-caufield-byers
6. sequoia-capital
7. first-round-capital
8. benchmark-capital
9. bessemer-venture-partners
10. venrock

Note that when we run the Nelder-Mead procedure in Matlab, the code always terminates saying that a local minimum has been reached, and no matter which minimum is reported, the top 10 list remains the same. After numerous runs, we are confident there is no choice of positive constants that will yield better results. For this top 10, we could let $\tilde{z}=[1,1,1,1]$ and obtain the same result reported above. Finally, comparing with Farmer’s top 10 list (see Section 3.1), we see that only five of the desired top 10 are present.

### 3.4 Eigenvector Centrality

Since we were unable to obtain Farmer’s top 10 using the PageRank vector $\tilde{P} = \mathbf{P}(A)$, we try again using eigenvector centrality, a different method to rank the importance of vertices in a graph [3]. In this approach, we let $\tilde{B}$, $\tilde{C}$, and $\tilde{M}$ denote diagonal matrices with the respective vectors $\tilde{B}$, $\tilde{C}$, and $\tilde{M}$ along the diagonal. We then define

$$K(\tilde{z}) = z_1 A + z_2 B + z_3 C + z_4 M,$$

where $\tilde{z} = [z_1, z_2, z_3, z_4]$. Essentially, $B$, $C$, and $M$ add diagonal perturbations to the adjacency matrix. Increasing the value on the diagonal of the adjacency matrix adds more importance to each individual node by introducing a self-loop whose weight depends on the combination of the three rankings. To calculate the eigenvector centrality, we seek

$$\tilde{V}(\tilde{z}) = \arg\max_{\tilde{v}} \tilde{v}^T K(\tilde{z}) \tilde{v},$$

the eigenvector corresponding to the maximum eigenvalue of $K(\tilde{z})$. We use this $\tilde{V}(\tilde{z})$ in exactly the same manner as given in (3), forming the objective function as described by (3). Once again,
we apply Nelder-Mead optimization in Matlab to minimize \( \tilde{Q}(\tilde{z}) \). Constraining the entries of \( \tilde{z} \) to be positive, we obtain the following top 10 venture capitalists:

1. intel-capital
2. first-round-capital
3. new-enterprise-associates
4. accel-partners
5. kleiner-perkins-caufield-byers
6. draper-fisher-jurvetson
7. sequoia-capital
8. charles-river-ventures
9. bessemer-venture-partners
10. redpoint-ventures

This time, we obtain only four of the venture capitalists in Farmer’s top 10 list.

3.5 Motivation

The main result of this section is that we are unable to reproduce Farmer’s top 10 list using the adjacency matrix \( A \) and the extra ranking vectors \( \hat{B}, \hat{C}, \) and \( \tilde{M} \). This remains true even if we ignore the ordering of the entries in the top 10 list. We conclude that the reasons for this discrepancy are that: (a) Farmer’s inclusion of investment data not present in CrunchBase leads to a fundamental change in the adjacency matrix and associated investor rankings, and/or (b) the way we are computing and/or applying the extra ranking vectors does not match Farmer’s methodology. In what follows, we assume that the main reason for the discrepancy is (a), and we use this as motivation for pursuing the inverse problem. We seek a modified adjacency matrix \( \hat{A} = A + \varepsilon \) such that \( P(\hat{A}) \) yields a top 10 ranking with the same venture capitalists that are present in Farmer’s top 10.

4 An Inverse Problem for PageRank

We now return to the inverse problem introduced in Section 1. The idea is that we have a graph with \( n \) vertices whose adjacency matrix \( A^0 \) has PageRank \( P^0 = P(A^0) \) that is undesirable. Let \( P' \) denote the desired PageRank and let \( A(\varepsilon) \) denote \( A(\varepsilon) = A^0 + \varepsilon \), so that \( \varepsilon \) is a perturbation to \( A^0 \). Then the central problem is to find

\[
\varepsilon_* = \arg\min_{\varepsilon \in \mathcal{S}} \| P(A(\varepsilon)) - P' \|_2,
\]

where \( \mathcal{S} \) is a space of admissible perturbation matrices and \( \| \cdot \|_2 \) is the vector 2-norm. In what follows, we take \( \mathcal{S} \) to be the space of symmetric matrices with zero diagonal and nonnegative off-diagonal entries. Choosing \( \varepsilon \in \mathcal{S} \) ensures that \( A(\varepsilon) \) is an adjacency matrix for a weighted graph with no self-loops.

The nonnegativity of off-diagonal entries ensures that edges that exist in \( A^0 \) cannot be deleted by \( \varepsilon \). If \( P' \) is a rearrangement of the entries of \( P^0 \), then we know there exists a solution of (6) given by (i) relabeling the vertices of the graph according to the rearrangement \( P^0 \mapsto P' \), (ii) setting \( A^0 + \varepsilon \) equal to the adjacency matrix for the relabeled graph, and then (iii) solving for \( \varepsilon \). Such an \( \varepsilon \) will always involve deleting at least one edge of \( A^0 \). This represents an undesirable, trivial solution that is eliminated by the nonnegativity constraint.
4.1 Optimization Approach

The first approach we take is to formulate (6) as the constrained optimization problem

\[
\min \frac{1}{2} \|\epsilon\|_F^2 \quad \text{subject to}
\]

\[
P(A(\epsilon)) = P' \tag{7a}
\]

\[
\epsilon = \epsilon^T \tag{7b}
\]

\[
\epsilon_{ii} = 0 \tag{7c}
\]

\[
\epsilon_{ij} \geq 0 \tag{7d}
\]

for \(1 \leq i, j \leq n\). We immediately make two modifications to this approach. First, in order to satisfy (7c), we set

\[
\epsilon(\kappa) = \frac{1}{2} (\kappa + \kappa^T) \tag{8}
\]

and rewrite the problem in terms of \(\kappa\). Second, we manipulate the PageRank constraint to make it easier to differentiate. Using (1), we see that

\[
P(A(\epsilon)) = P' \quad \text{if and only if} \quad A(\epsilon) D(\epsilon) \frac{1}{2} P_0 = Q \tag{9}
\]

where \(D(\epsilon)\) is the diagonal matrix whose \(k\)-th entry on the diagonal is the sum of the \(k\)-th row/column of \(A(\epsilon)\), and

\[
Q = \frac{1}{\gamma} \left[ P' - \left( \frac{1 - \gamma}{N} \right) 1 \right]. \tag{10}
\]

Now we define the PageRank equality constraint function

\[
c(\epsilon) = A(\epsilon) D(\epsilon)^{-1} P' - Q \tag{10}
\]

so that \(c(\epsilon) = 0\) corresponds to finding an \(\epsilon\) such that (7) is satisfied. With the above modifications, we rewrite (7) as

\[
\min_{\kappa} J(\epsilon(\kappa)) \quad \text{subject to}
\]

\[
c(\epsilon(\kappa)) = 0 \tag{11a}
\]

\[
\kappa_{ii} = 0 \tag{11b}
\]

\[
\kappa_{ij} \geq 0 \tag{11c}
\]

where

\[
J(\epsilon) = \frac{1}{2} \|\epsilon\|_F^2. \tag{12}
\]

We proceed to solve (11) using Lagrange multipliers. We write the constraints (11a-11d) using a function \(g(\kappa)\). To encode the inequality constraints, we define the \(C^1\) scalar function \(p(x) = \frac{1}{2} I_{x < 0} x^2\), where \(I_A\) stands for the indicator function on the set \(A\). Note that \(p(x) = 0\) when \(x \geq 0\). We define

\[
g_{ij}(\kappa) = \begin{cases} 
\frac{1}{2} \kappa_{ii}^2 & i = j \\
\frac{p(\kappa_{ij})}{\gamma} & i \neq j.
\end{cases} \tag{13}
\]

As written, \(g\) is a matrix-valued function of \(\kappa\). We identify \(g\) with a vector in \(\mathbb{R}^{n^2}\) by stacking the columns of the matrix representation of \(g\). In general, column stacking/unstacking is how we will
identify elements of $\mathbb{R}^{n \times n}$ with elements of $\mathbb{R}^{n^2}$.

Next, we form the Lagrangian $L : \mathbb{R}^{n^2} \times \mathbb{R}^{n+n^2} \to \mathbb{R}$ in the usual manner:

$$L(\kappa, \lambda) = \frac{1}{2} \|\varepsilon(\kappa)\|_F^2 - [\lambda_1^T \lambda_2^T] \begin{bmatrix} c(\varepsilon(\kappa)) \\ g(\kappa) \end{bmatrix}.$$  \hfill (14)

For $\lambda \in \mathbb{R}^{n+n^2}$, we have set $\lambda_1$ to be the first $n$ and $\lambda_2$ to be the final $n^2$ components of $\lambda$.

By the KKT conditions [20, Theorem 12.1], we know that a necessary condition for an optimal solution is that $\nabla L = 0$, so we proceed to use Newton’s method to find a point $(\kappa, \lambda_1, \lambda_2)$ such that $\nabla L = 0$. This will involve constructing the gradient and Hessian of $L$.

**Frobenius Norm.** By our earlier definition (12),

$$J(\varepsilon) = \frac{1}{2} \text{trace} \varepsilon^T \varepsilon = \frac{1}{2} \sum_{m=1}^{N} \sum_{k=1}^{N} \varepsilon_{mk} \varepsilon_{km} = \frac{1}{2} \sum_{m,k} \varepsilon_{km}^2,$$

so that

$$\frac{\partial}{\partial \varepsilon_{ij}} J = \varepsilon_{ij}.$$

Assembling all the element-wise derivatives of $J$ into a matrix yields the matrix $\varepsilon$. An easy consequence is the second derivative or Hessian formula

$$\frac{\partial}{\partial \varepsilon_{kl}} \frac{\partial}{\partial \varepsilon_{ij}} J = \delta_{ik} \delta_{jl},$$

where $\delta_{ik}$ is the Kronecker delta function. Thus the matrix representation of the Hessian of $J$ is the $n^2 \times n^2$ identity matrix.

**Matrix 2-norm: Gradient of the Largest Modulus Eigenvalue.** In Section 5, we describe motivating results obtained by solving (11) using the matrix 2-norm of $\varepsilon$ instead of the squared Frobenius norm. Let

$$J_2(\varepsilon) = \|\varepsilon\|_2.$$  \hfill (15)

Assume that $\varepsilon$ is a symmetric positive-semidefinite real $N \times N$ matrix. Then

$$J_2(\varepsilon) = \max_{1 \leq i \leq N} |\lambda_i|$$

where $\{\lambda_i\}_{i=1}^{N}$ is the spectrum of $\varepsilon$. Put another way, there exists $i \in \{1, 2, \ldots, N\}$ such that we have

$$J_2(\varepsilon) = \begin{cases} \lambda_i & \lambda_i \geq 0 \\ -\lambda_i & \lambda_i < 0. \end{cases}$$

Therefore, we focus on calculating

$$\frac{\partial}{\partial \varepsilon_{ij}} \lambda$$

where $\lambda$ is an eigenvalue of $\varepsilon$. Let

$$\tau = \frac{\partial}{\partial \varepsilon_{ij}}.$$  \hfill (16)
For any eigenvalue/eigenvector pair such that
\[ \varepsilon v = \lambda v, \]
both \( v \) and \( \lambda \) depend on the entries of \( \varepsilon \). Differentiating the above equation, we obtain
\[ \varepsilon' v + \varepsilon v' = \lambda' v + \lambda v'. \]
(17)

We multiply the previous equation on the left by \( v^T \) and obtain
\[ v^T \varepsilon' v + v^T \varepsilon v' = \lambda' v^T v + \lambda v^T v'. \]
Since \( \varepsilon v = \lambda v \), symmetry of \( \varepsilon \) guarantees that \( v^T \varepsilon = \lambda v^T \). Using this fact, the second term on the left-hand side cancels the second term on the right-hand side. We obtain
\[ \lambda' = \frac{v^T \varepsilon' v}{v^T v}. \]

Let us assume the eigenvector \( v \) is normalized so that it has unit norm—in fact, we are guaranteed by the spectral theorem that the real symmetric matrix \( \varepsilon \) has an orthonormal basis of eigenvectors, so this is a perfectly valid assumption. Then
\[ \lambda' = v^T \varepsilon' v. \]
(18)

Using the definition of \( \varepsilon' \) and the matrix multiplication formula, we have
\[ \frac{\partial}{\partial \varepsilon_{ij}} \lambda = \lambda' = \sum_{k,m} v_k \varepsilon'_{km} v_m = \sum_{k,m} v_k \delta_{ik} \delta_{jm} v_m = v_i v_j. \]
Therefore,
\[ \frac{\partial}{\partial \varepsilon_{ij}} J_2 = \begin{cases} v_i v_j & J_2 = \lambda \\ -v_i v_j & J_2 = -\lambda \end{cases} \]
where \( \lambda \) is the element of the spectrum of \( \varepsilon \) with largest absolute value. Note that if we put the values \( v_i v_j \) together in a matrix, we obtain \( vv^T \).

**PageRank Constraint.** We compute
\[ \frac{\partial}{\partial \varepsilon_{ij}} c(\varepsilon) = \frac{\partial A}{\partial \varepsilon_{ij}} D(\varepsilon)^{-1} P' - A(\varepsilon) D(\varepsilon)^{-2} \frac{\partial D}{\partial \varepsilon_{ij}} P'. \]
(19)

By definition, the matrix \( D(\varepsilon) \) is a diagonal matrix where the \( k \)-th entry on the diagonal is
\[ D(\varepsilon)_{kk} = \sum_m A_{mk}^0 + \varepsilon_{mk}. \]
The inverse of a diagonal matrix is easy to compute:
\[ D(\varepsilon)^{-1}_{kk} = \left( \sum_m A_{mk}^0 + \varepsilon_{mk} \right)^{-1}. \]
Also,
\[
\frac{\partial}{\partial \varepsilon_{ij}} D(\varepsilon)_{kk} = \sum_m \frac{\partial}{\partial \varepsilon_{ij}} \varepsilon_{mk} = \delta_{jk},
\]
where
\[
\delta_{jk} = \begin{cases} 
1 & j = k \\
0 & j \neq k.
\end{cases}
\]
The only other ingredient is
\[
\frac{\partial}{\partial \varepsilon_{ij}} A(\varepsilon)_{kl} = \delta_{ik}\delta_{jl}.
\]
Putting all the information together, we obtain
\[
\frac{\partial}{\partial \varepsilon_{ij}} c(\varepsilon) = \sum_{k,l,g} \delta_{ik}\delta_{jl} D(\varepsilon)_{lg}^{-1} p'_{g} - A(\varepsilon)_{kl} D(\varepsilon)_{lg}^{-2} \delta_{jg} p'_{g},
\]
which reduces to
\[
\left[ \frac{\partial}{\partial \varepsilon_{ij}} c(\varepsilon) \right]_k = \delta_{ik} D(\varepsilon)_{jg}^{-1} p'_{g} - A(\varepsilon)_{kj} D(\varepsilon)_{jj}^{-2} p'_{j}.
\]
To compute the Hessian of \( c \), we first note that
\[
\frac{\partial}{\partial \varepsilon_{ab}} \frac{\partial}{\partial \varepsilon_{ij}} A(\varepsilon) = \frac{\partial}{\partial \varepsilon_{ab}} \frac{\partial}{\partial \varepsilon_{ij}} D(\varepsilon) = 0,
\]
because both \( A(\varepsilon) \) and \( D(\varepsilon) \) depend on \( \varepsilon \) linearly. In what follows, we sometimes omit the functional dependence of \( A \) and \( D \) on \( \varepsilon \). We proceed with
\[
\left[ \frac{\partial}{\partial \varepsilon_{ab}} \frac{\partial}{\partial \varepsilon_{ij}} c(\varepsilon) \right]_k = - \left[ \frac{\partial A}{\partial \varepsilon_{ij}} D(\varepsilon)^{-2} \frac{\partial D}{\partial \varepsilon_{ab}} p' - \frac{\partial A}{\partial \varepsilon_{ab}} D(\varepsilon)^{-2} \frac{\partial D}{\partial \varepsilon_{ij}} p' + 2A(\varepsilon)D(\varepsilon)^{-2} \frac{\partial D}{\partial \varepsilon_{ab}} \frac{\partial D}{\partial \varepsilon_{ij}} p' \right].
\]
Let us simplify this expression by multiplying terms. We get
\[
[T_1]_\alpha = \sum_{f,g,h} \frac{\partial A_{af}}{\varepsilon_{ij}} D(\varepsilon)^{-2} \frac{\partial D_{gh}}{\varepsilon_{ab}} p'_{h} \\
= \sum_{f,g,h} \delta_{ia}\delta_{jg}\delta_{f} \left( \sum_m A_{mf}^0 + \varepsilon_{mf} \right)^{-2} \delta_{gh}\delta_{bg} p'_{h} \\
= \delta_{ia}\delta_{jb} \left( \sum_m A_{mj}^0 + \varepsilon_{mj} \right)^{-2} p'_{j}.
\]
By symmetry of indices (swap \( i \) and \( a \), and also swap \( j \) and \( b \)), we also know
\[
[T_2]_\alpha = \delta_{aa}\delta_{jb} \left( \sum_m A_{mb}^0 + \varepsilon_{mb} \right)^{-2} p'_{b}.
\]
Lastly, we get

\[
[T_3]_\alpha = 2 \sum_{f,g,h,\ell} A(\varepsilon)_{\alpha f} D(\varepsilon)^{-3} g\frac{\partial D_{gh}}{\partial \varepsilon_{ab}} \frac{\partial D_{h\ell}}{\partial \varepsilon_{ij}} P'_\ell
\]

\[
= 2 \sum_{f,g,h,\ell} A(\varepsilon)_{\alpha f} \delta_{fg} \left( \sum_m A_{mf}^0 + \varepsilon_{mf} \right)^{-3} \delta_{gh} \delta_{bg} \delta_{hl} P'_\ell
\]

\[
= 2 A(\varepsilon)_{\alpha f} \delta_{fb} \left( \sum_m A_{mj}^0 + \varepsilon_{mj} \right)^{-3} P'_j.
\]

**Graph-Based Constraints.** The graph-based constraints are

\[
g_{ij}(\kappa) = \begin{cases} \frac{1}{2} \kappa_{ii}^2 & i = j \\ p(\kappa_{ij}) & i \neq j. \end{cases}
\]

With this definition, we compute

\[
\frac{\partial}{\partial \varepsilon_{ij}} g_{ij}(\kappa) = \begin{cases} \kappa_{ii} & i = j \\ \kappa_{ij} I_{\kappa_{ij} < 0} & i \neq j. \end{cases}
\]

and

\[
\frac{\partial}{\partial \varepsilon_{kk}} \frac{\partial}{\partial \varepsilon_{ij}} g_{ij}(\kappa) = \begin{cases} \delta_{hi} \delta_{kj} & i = j \\ \delta_{hi} \delta_{kj} I_{\kappa_{ij} < 0} & i \neq j. \end{cases}
\]

\((23)\)

**\(\kappa\) Gradients and Hessians.** We have differentiated \(J(\varepsilon)\) and \(c(\varepsilon)\) with respect to \(\varepsilon\). Using \((8)\), it is easy to convert these expressions into derivatives with respect to \(\kappa\), which is what we desire to compute \(\nabla L\). Suppose that we have a function \(f : \mathbb{R}^{n \times n} \rightarrow \mathbb{R}\), i.e., a function that takes a matrix as its input and returns a scalar as its output. We can write

\[
y = F(\varepsilon).
\]

Let us now use \((8)\) to express \(\varepsilon\) as a function of \(\kappa\):

\[
y = F(\varepsilon(\kappa)) = f \left( \frac{1}{2} [\kappa + \kappa^T] \right).
\]

Then

\[
\frac{\partial F}{\partial \kappa_{ij}} = \sum_{a,b} \frac{\partial F}{\partial \varepsilon_{ab}} \frac{\partial \varepsilon_{ab}}{\partial \kappa_{ij}}
\]

\[
= \frac{1}{2} \sum_{a,b} \frac{\partial F}{\partial \varepsilon_{ab}} \left( \delta_{ia} \delta_{jb} + \delta_{ib} \delta_{ja} \right)
\]

\[
= \frac{1}{2} \left( \frac{\partial F}{\partial \varepsilon_{ij}} + \frac{\partial F}{\partial \varepsilon_{ji}} \right).
\]

Continuing the above derivation, we see that the two expressions on the right-hand side, \(\partial f/\partial \varepsilon_{ij}\) and \(\partial f/\partial \varepsilon_{ji}\), are themselves functions that accept a matrix as input and return a scalar as output.
Therefore, applying the same logic as above, we can derive
\[
\frac{\partial}{\partial \kappa_{hl}} \frac{\partial f}{\partial \kappa_{ij}} = \frac{1}{2} \left( \frac{\partial}{\partial \varepsilon_{hl}} \frac{\partial f}{\partial \varepsilon_{ij}} + \frac{\partial}{\partial \varepsilon_{il}} \frac{\partial f}{\partial \varepsilon_{ji}} \right) + \frac{1}{2} \left( \frac{\partial}{\partial \varepsilon_{hl}} \frac{\partial f}{\partial \varepsilon_{ji}} + \frac{\partial}{\partial \varepsilon_{il}} \frac{\partial f}{\partial \varepsilon_{ij}} \right)
\]
\[
= \frac{1}{4} \left( \frac{\partial^2 f}{\partial \varepsilon_{hl} \varepsilon_{ij}} + \frac{\partial^2 f}{\partial \varepsilon_{il} \varepsilon_{ji}} + \frac{\partial^2 f}{\partial \varepsilon_{hl} \varepsilon_{ji}} + \frac{\partial^2 f}{\partial \varepsilon_{il} \varepsilon_{ij}} \right). \tag{25}
\]

**Putting It All Together.** Since we have all the first and second derivatives we need, let us assemble the gradient \( \nabla L \) and the Hessian \( \nabla \nabla^T L \). We have from (23) that
\[
L(\kappa, \lambda) = \frac{1}{2} \|\varepsilon(\kappa)\|_F^2 - \lambda_1^T c(\varepsilon(\kappa)) - \lambda_2^T g(\kappa).
\]
We view the gradient \( \nabla L \) as a vector of size \( 1 \times (n^2 + n + n^2) \), where the first \( n^2 \) slots consist of \( \nabla_\kappa L \) and the final \( n + n^2 \) slots consist of \( \nabla_\lambda L \). Then, using (24), we have
\[
\frac{\partial L}{\partial \kappa_{ij}} = \frac{1}{2} (\varepsilon_{ij} + \varepsilon_{ji}) - \frac{1}{2} \lambda_1^T \left( \frac{\partial c}{\partial \varepsilon_{ij}} + \frac{\partial c}{\partial \varepsilon_{ji}} \right) - \lambda_2^T \left( \frac{\partial g}{\partial \kappa_{ij}} \right).
\]
Since \( \varepsilon \) is symmetric by construction, the first term on the right-hand side reduces to \( \varepsilon_{ij} \). We can write the whole equation compactly as
\[
\nabla_\kappa L = \varepsilon - \frac{1}{2} \lambda_1^T (\nabla_\varepsilon c + \nabla_\varepsilon^T c) - \lambda_2^T \nabla_\kappa g,
\]
where the first term \( \varepsilon \) on the right-hand side is obtained by stacking the columns of the matrix \( \varepsilon \) to produce a vector of size \( n^2 \times 1 \) and then taking the transpose to produce a vector of size \( 1 \times n^2 \). Taking the transpose of the whole equation, we have the column vector
\[
(\nabla_\kappa L)^T = \varepsilon - \frac{1}{2} (\nabla_\varepsilon c + \nabla_\varepsilon^T c)^T \lambda_1 - (\nabla_\kappa g)^T \lambda_2. \tag{26}
\]
where the first term \( \varepsilon \) is just the column-stacked version of the \( n \times n \) matrix \( \varepsilon \). Hence
\[
(\nabla L)^T = \begin{bmatrix} (\nabla_\kappa L)^T \\ -c(\varepsilon(\kappa)) \\ -g(\kappa) \end{bmatrix}. \tag{27}
\]
To construct the Hessian, we think of \( \nabla_\kappa \) as a \( 1 \times n^2 \) vector of first-order partial differential operators:
\[
\nabla_\kappa = \begin{bmatrix} \frac{\partial}{\partial \kappa_{11}} & \cdots & \frac{\partial}{\partial \kappa_{nm}} \end{bmatrix},
\]
so that \( \nabla_\kappa \nabla_\kappa^T \) is an \( n^2 \times n^2 \) matrix of second-order partial differential operators. Then let \( \nabla = [\nabla_\kappa, \nabla_\lambda_1, \nabla_\lambda_2] \). In this way, we can write
\[
\nabla \nabla^T L = \begin{bmatrix} \nabla_\kappa \nabla_\kappa^T L & \nabla_\lambda_1 \nabla_\kappa^T L & \nabla_\lambda_2 \nabla_\kappa^T L \\ \nabla_\kappa \nabla_\lambda_1^T L & \nabla_\lambda_1 \nabla_\lambda_1^T L & \nabla_\lambda_2 \nabla_\lambda_1^T L \\ \nabla_\kappa \nabla_\lambda_2^T L & \nabla_\lambda_1 \nabla_\lambda_2^T L & \nabla_\lambda_2 \nabla_\lambda_2^T L \end{bmatrix}. \tag{28}
\]
Since (23) is linear in the Lagrange multipliers, all second derivatives with respect to the Lagrange multipliers necessarily vanish. By symmetry, the only entries we must compute are those in the top...
row, all of which may be computed by taking \( \nabla \) of both sides of (26). For \( \nabla_\kappa \nabla^T_\kappa L \), we use (26) in conjunction with our previously computed expressions (22), (23), and (25). The mixed derivatives \( \nabla_\lambda_1 \nabla^T_\kappa L \) and \( \nabla_\lambda_2 \nabla^T_\kappa L \) are trivial to compute from (26). We therefore obtain

\[
\nabla \nabla^T L = \begin{bmatrix}
\frac{1}{2} (\nabla_\kappa \nabla^T_\kappa L) & -\frac{1}{2} (\nabla_\varepsilon c + \nabla_\varepsilon \tau c)^T & -\nabla_\kappa g)^T \\
-\frac{1}{2} (\nabla_\varepsilon c + \nabla_\varepsilon \tau c) & 0 & 0 \\
-(\nabla_\kappa g) & 0 & 0
\end{bmatrix},
\]

(29)

where \( \mathbf{0} \) stands for a zero block of the appropriate size. This Hessian has size \((2n^2 + n) \times (2n^2 + n)\), and the gradient \( \nabla^T L \) has size \((2n^2 + n)\).

Armed with the gradient and Hessian, we use Newton’s method to search for a zero of the nonlinear system \( \nabla^T L = \mathbf{0} \). Let \( \mathbf{x}_0 \) be the initial guess. Then the basic Newton step is

\[
\mathbf{x}_{i+1} = \mathbf{x}_i - \mathbf{s},
\]

(30)

where the step \( \mathbf{s} \) satisfies

\[
[\nabla \nabla^T L(\mathbf{x}_i)] \mathbf{s} = \nabla^T L(\mathbf{x}_i).
\]

(31)

Note that the \( i \)-th iterate \( \mathbf{x}_i^T = [\kappa_i, \lambda_i] \) includes both the object of interest—the matrix \( \kappa \)—together with the Lagrange multipliers.

**Pseudocode.** Suppose that \( \mathbf{A}^0 \) is given. Then the optimization-based algorithm follows the basic skeleton:

1. Set \( \kappa \) equal to an initial guess and \( \lambda \) equal to a constant vector.
2. While the stopping criteria have not been satisfied:
   (a) Compute \( \varepsilon(\kappa) \) using (8).
   (b) Compute the PageRank constraint \( \mathbf{c}(\varepsilon(\kappa)) \) using (10).
   (c) Compute the graph-based constraint \( \mathbf{g}(\kappa) \) using (13).
   (d) Compute the gradient \( \nabla L \) using (24).
   (e) Compute the Hessian \( [\nabla \nabla^T L(\mathbf{x}_i)] \) using (29).
   (f) Solve the linear problem (31).
   (g) Take a Newton step (30), updating both the matrix \( \kappa \) and the Lagrange multipliers \( \lambda \).
3. Save final matrix \( \mathbf{A}^0 + \varepsilon \) to disk.

### 4.2 First Root-Finding Approach

We now describe a different approach, where we try to solve directly the nonlinear constraint equation (11b) together with the constraints (11c-11d). We have laid most of the groundwork for this problem above, and will be reusing many of the calculations carried out in Section 4.1.

We define the function \( \mathbf{f} : \mathbb{R}^{n^2} \rightarrow \mathbb{R}^{n + n^2} \). In this function, the \( n \times n \) matrices \( \kappa \) and \( \varepsilon \) are identified with their flattened vector representations in \( \mathbb{R}^{n^2} \).

\[
\mathbf{f}(\kappa) = \begin{bmatrix}
\mathbf{c}(\varepsilon(\kappa)) \\
\mathbf{g}(\kappa)
\end{bmatrix}.
\]

(32)

Note that up to a sign change, this function consists of the final \( n + n^2 \) components of \( \nabla^T L \) given by (27). We seek \( \kappa \) such that \( \mathbf{f}(\kappa) = \mathbf{0} \). Newton’s method gives us a natural line of attack. The
only catch is that since the Jacobian matrix \( J \) is of size \( \mathbb{R}^{n+n^2} \times \mathbb{R}^{n^2} \), we use the Moore-Penrose pseudoinverse, denoted by \( \dagger \), to write an algebraic expression for the Newton step:

\[
\kappa_{i+1} = \kappa_i - [J_f(\kappa_i)]^\dagger f(\kappa_i).
\]

(33)

The Jacobian matrix can be assembled using previously computed pieces:

\[
J_f = \begin{bmatrix}
\frac{1}{2} (\nabla_\kappa c + \nabla_T \kappa c) \\
\nabla_\kappa g
\end{bmatrix}
\]

(34)

Though we have used the pseudoinverse to write (33), it is not necessary to compute the full singular value decomposition of \( J_f \) at each iteration. We describe in Section 6 the precise method we use to calculate the Newton step \([J_f]^\dagger f\).

**Pseudocode.** Suppose that \( A^0 \) is given. The first root-finding algorithm follows the basic skeleton:

1. Set \( \kappa \) equal to an initial guess.
2. While the stopping criteria have not been satisfied:
   a. Compute \( \varepsilon(\kappa) \) using (8).
   b. Compute the PageRank constraint \( c(\varepsilon(\kappa)) \) using (11).
   c. Compute the graph-based constraint \( g(\kappa) \) using (12).
   d. Form the vector \( f \) as in (32).
   e. Compute the Jacobian (34).
   f. Take a Newton step (33), updating the matrix \( \kappa \).
3. Save final matrix \( A^0 + \varepsilon \) to disk.

### 4.3 Active Set Method for Graph-Based Constraints

In the pseudocode for both the optimization and first root-finding algorithms, a key step is the computation of both the graph-based constraint \( g(\kappa) \) and the gradient of the graph-based constraint \( \nabla_\kappa g \). This gradient enters into \( (\nabla L)^T \) and \( \nabla \nabla^T L \) for the optimization method, and it also enters into \( J_f \) for the root-finding method.

We say that the constraint (13) is active if \( \kappa_{ij} < 0 \); we say the same constraint is inactive if \( \kappa_{ij} \geq 0 \). Note that for an inactive inequality constraint, both \( g_{ij}(\kappa) \) and \( \nabla_\kappa g_{ij}(\kappa) \) vanish identically. This means that for each row corresponding to an inactive \( \kappa_{ij} \) constraint, the entire row of the overall gradient matrix \( \nabla_\kappa \) will vanish, reducing the rank of either the Hessian matrix \( \nabla \nabla^T L \) or the rectangular Jacobian matrix \( J_f \).

Therefore, when we implement either the optimization or first root-finding algorithms, we use an active set method to enforce the graph-based constraints corresponding to \( g \). On each iteration, we determine which elements of \( g \) are active. We restrict both \( g \) and \( \nabla_\kappa g \) to include only those rows corresponding to active elements. Let us say that \( g \) is then a vector of size \((n+m) \times 1\), where \( m \) is the number of active inequality constraints—the \((n+m)\) arises because we do not use the active set approach for the \( n \) “zero-diagonal” equality constraints contained in \( g \), i.e., the entries of (13) corresponding to \( i = j \). Then \( \nabla_\kappa g \) will be of size \((n+m) \times n^2\).

For the first root-finding method, the active set procedure results in a Jacobian that is of size \((n+n+m) \times n^2\), while for the optimization method, the active set procedure results in a Hessian that is of size \((n^2+n+n+m) \times (n^2+n+n+m)\). Note that the active set method affects the upper-
left block through the $\nabla_\kappa \nabla_\kappa^T$ operator applied to the $-\lambda_\kappa^T g(\kappa)$ term in the Lagrangian. Essentially, we only include those terms from (23) that correspond to equality constraints and active inequality constraints.

**Justification.** In order to justify the active set procedure, we show that if

$$M' = \begin{bmatrix} M \\ 0^T \end{bmatrix},$$

then the Moore-Penrose pseudoinverse of $M'$ is

$$M'^\dagger = \begin{bmatrix} M \\ 0 \end{bmatrix}.$$

Let

$$N = \begin{bmatrix} M \\ 0 \end{bmatrix}$$

Then note that both

$$M'N = \begin{bmatrix} M \\ 0^T \end{bmatrix} \begin{bmatrix} M^\dagger \\ 0 \end{bmatrix} = \begin{bmatrix} MM^\dagger \\ 0 \\ 0 \end{bmatrix}$$

$$NM' = \begin{bmatrix} M^\dagger \\ 0 \end{bmatrix} \begin{bmatrix} M \\ 0^T \end{bmatrix} = \begin{bmatrix} M^\dagger M \end{bmatrix},$$

so both $M'N$ and $NM'$ are symmetric. Next note that

$$M'NM' = \begin{bmatrix} MM^\dagger \\ 0^T \\ 0 \end{bmatrix} \begin{bmatrix} M^\dagger \\ 0 \end{bmatrix} = \begin{bmatrix} MM^\dagger M^\dagger \\ 0 \\ 0 \end{bmatrix} = M'$$

$$NM'N = \begin{bmatrix} M^\dagger M \end{bmatrix} \begin{bmatrix} M^\dagger \\ 0 \end{bmatrix} = \begin{bmatrix} M^\dagger MM^\dagger \\ 0 \end{bmatrix} = N$$

We have shown that $N$ satisfies the four properties of $M'^\dagger$, so by uniqueness of the Moore-Penrose pseudoinverse, we must have $M'^\dagger = N$. Now suppose we want to find the solution $x$ of the underdetermined linear system

$$M'x = b$$

that minimizes $||x||_2$. This solution is

$$x = M'^\dagger b = \begin{bmatrix} M^\dagger \\ 0 \end{bmatrix} b,$$

which is equivalent to

$$x = M'^\dagger b',$$

where $b'$ is what we obtain by deleting the last element of the vector $b$. This shows that if we delete the zero row of the left-hand side matrix and the corresponding zero row of the right-hand side vector in the linear problem (35), we obtain the same solution we would have obtained for the original system. As deleting corresponding rows is precisely what we do in the active set method, we have justified the procedure for the root-finding method.

The justification of the active set procedure for the optimization method follows analogous arguments; we omit the details.
4.4 Second Root-Finding Approach

Is this approach, we reduce the size of the problem by building one of our constraints into the declaration of \( \varepsilon \). Let \( \varepsilon \) be defined element-wise by

\[
\varepsilon_{ij} = \frac{1}{2} (\kappa_{ij}^2 + \kappa_{ji}^2).
\]

This ensures that \( \varepsilon \) is both symmetric and that each element of \( \varepsilon \) is nonnegative, allowing us to eliminate the \( n^2 - n \) off-diagonal elements in the graph-based constraint function \( g \). We redefine \( g \) as

\[
g(\kappa) = \left[ \frac{1}{2} \kappa_{11}^2, \frac{1}{2} \kappa_{22}^2, \ldots, \frac{1}{2} \kappa_{nn}^2 \right]^T,
\]

so that

\[
\frac{\partial}{\partial \kappa_{ij}} g_l(\kappa) = \kappa_{il} \delta_{ij} \delta_{jl}.
\]

The definition (36) changes the derivative calculation for the Pagerank constraint function \( c(\varepsilon(\kappa)) \). The \( \nabla_\varepsilon c \) part stays the same; all that changes is \( \nabla_\kappa \varepsilon(\kappa) \). To determine this more precisely, we write

\[
y = F(\varepsilon(\kappa)) = F\left( \frac{1}{2} \left[ \kappa^2 + (\kappa^T)^2 \right] \right),
\]

where \( M^2 \) stands for the element-wise square of the matrix \( M \). Then

\[
\frac{\partial F}{\partial \kappa_{ij}} = \sum_{a,b} \frac{\partial F}{\partial \varepsilon_{ab}} \frac{\partial \varepsilon_{ab}}{\partial \kappa_{ij}}
\]

\[
= \sum_{a,b} \frac{\partial F}{\partial \varepsilon_{ab}} \left( \delta_{ia} \delta_{jb} \kappa_{ab} + \delta_{ib} \delta_{ja} \kappa_{ab} \right)
\]

\[
= \left( \frac{\partial F}{\partial \varepsilon_{ij}} \kappa_{ij} + \frac{\partial F}{\partial \varepsilon_{ji}} \kappa_{ji} \right).
\]

Referring to (21) and (24), we obtain

\[
\begin{bmatrix}
\frac{\partial}{\partial \kappa_{ij}} c(\kappa)
\end{bmatrix}_k = \delta_{ik} D(\varepsilon)^{-1} P_j^P \kappa_{ij} - A(\varepsilon)_{kj} D(\varepsilon)^{-2} P_j^P \kappa_{ij} + \delta_{jk} D(\varepsilon)^{-1} P_i^P \kappa_{ji} - A(\varepsilon)_{ki} D(\varepsilon)^{-2} P_i^P \kappa_{ji}.
\]

We now proceed with the root-finding procedure defined in Section 4.2 with \( f \) using the \( c(\varepsilon(\kappa)) \) and \( g(\kappa) \) functions defined here. Since \( f(\kappa) \in \mathbb{R}^{2n} \), the Jacobian \( Jf \) will be a \( 2n \times n^2 \) sized matrix, a significant reduction as compared with the original \( (n^2 + n) \times n^2 \) sized \( Jf \).

**Pseudocode.** The pseudocode for this method is identical to the pseudocode presented at the end of Section 4.2, except that one must use (36) in step 2(a). More obviously, one must use (37) and the Jacobian of the redefined \( g \) in steps 2(c) and 2(e), but this amounts to a simple deletion of rows from the old definition of \( g \) and its Jacobian.

5 Preliminary Results

As a purely preliminary step, we coded the solution of the optimization problem (11) in Matlab using the \texttt{fmincon} solver in the Optimization Toolbox. We supplied the solver with the analytical
gradients of the objective and constraint functions, and we used built-in parameters to implement the graph-based constraints \((11c-11d)\) on \(\kappa\). Analytical Hessians were not supplied because the construction of these Hessians involves “for” loops that are very slow in Matlab. Therefore, the Matlab solvers we use are quasi-Newton methods that use current and past gradients to approximate the Hessian at each step.

We now describe the experimental procedure used to test the Matlab solvers. We start with an initial adjacency matrix \(A^0\) generated from an Erdős-Rényi random graph on \(n = 40\) vertices, where the probability of generating an edge between two vertices is \(p = 0.5\). We then swap the maximum and minimum entries of the PageRank vector \(P(A)\) to produce \(P^0\). For each run, the initial guess for \(\kappa\) is the zero matrix. Since this matrix does not satisfy the constraint \(c(\kappa) = 0\), we found that \texttt{fmincon}’s AS (active set) and TRR (trust region reflective) methods failed to produce any results. Therefore, we include in Table 1 the average results across 100 runs for the IP (interior point) and SQP (sequential quadratic programming) algorithms.

At a minimum, the results in Table 1 show that the problem is solvable; it is possible to determine how to update edge weights in a network to change the PageRank of vertices. However, the solutions obtained are not sparse: in each case, the solution matrix \(\kappa\) returned by the solver contains non-zero entries in all off-diagonal elements. For many of the applications we have outlined in Section 1, it will simply not be possible to change edge weights for the complete graph on all vertices in order to obtain the desired PageRank.

**Sparse Solutions.** For our next numerical experiment, we replace the Frobenius norm \((12)\) by the matrix 2-norm \((15)\). We run the same set of numerical experiments as described above, except for graphs on \(n = 10\) vertices. We do not use \(n = 40\) vertices purely because the run times are prohibitively large. We use the analytical gradient of the matrix 2-norm calculated in Section 4 in conjunction with \texttt{fmincon} and both the IP algorithms. The results, averaged across 100 runs, are given in Table 2.

The most interesting observation from Table 2 is that the solutions found by IP are only 46% non-zero, while those found by SQP are only 21% non-zero. This is in contrast with the solutions found in Table 1 using the Frobenius norm objective function \((12)\), which were 100% non-zero in off-diagonal entries. We conjecture that for the optimization problem \((11)\), the matrix 2-norm objective function \((15)\) promotes sparsity of the solution.

If we stack the columns of the \(n \times n\) matrix \(\varepsilon\) and produce a vector \(\varepsilon_{\text{vec}}\) of size \(n^2 \times 1\), then
\[
\|\varepsilon\|_F^2 = \|\varepsilon_{\text{vec}}\|_2^2,
\]
the usual vector 2-norm of \(v\). Suppose we view the optimization problem \((11)\) in penalty form as a sequence of problems where we minimize
\[
\mathcal{J}_{\ell} = \frac{1}{2} \|\varepsilon\|_F^2 + \frac{\gamma_{\ell}}{2} \|c(\varepsilon)\|_2^2
\]
\[
= \frac{1}{2} \|\varepsilon_{\text{vec}}\|_2^2 + \frac{\gamma_{\ell}}{2} \|c(\varepsilon)\|_2^2
\]
for a sequence of values \(\gamma_{\ell}\) such that \(\lim_{\ell \to \infty} \gamma_{\ell} = +\infty\). The \(\|\varepsilon_{\text{vec}}\|_2^2\) term, which is equivalent to the squared Frobenius norm of \(\varepsilon\), can now be viewed as the Tikhonov regularization for the nonlinear least squares problem corresponding to the second term. Viewed in this light, the non-sparsity is not a surprise—for the underconstrained linear least squares problem \(\|Ax - b\|_2\), it is well-known that the Tikhonov regularization of \(x\) yields dense solutions.
On the other hand, the matrix 2-norm can be viewed as
\[ \| \epsilon \|_2 = \| \sigma \|_\infty, \]
where \( \sigma = \{ \sigma_1, \sigma_2, \ldots, \sigma_n \} \) is a vector containing the \( n \) singular values of \( \epsilon \). A related norm is the nuclear norm
\[ \| \epsilon \|_* = \| \sigma \|_1, \]
the sum of the singular values of \( \epsilon \). In recent years, it has emerged that for certain linear inverse problems, minimizing the nuclear norm of the matrix leads to low-rank, sparse solutions \([\text{III}, \text{II}]\). This has to do with the fact that the nuclear norm is a proxy for the rank of the matrix. Since
\[ \| \sigma \|_\infty \leq \| \sigma \|_1 \leq n \| \sigma \|_\infty, \]
we conjecture that minimizing the matrix 2-norm may be an excellent way to also push down the nuclear norm of the matrix, leading to a sparse solution.

**Root-Finding.** With the above results in mind, we implement and test the first root-finding method described in Section 4.2. In this method, we do not try to minimize any norm of \( \epsilon \)—instead, we directly solve the constraint equation \( c(\epsilon(\kappa)) = 0 \) for \( \kappa \). We use the same testing methodology described above, using random graphs with \( n = 40 \) vertices. The results averaged over 100 runs are given in Table 3. Comparing with the results from Tables 1-2, we see that (i) the root-finding method is doing a poor job of minimizing the Frobenius norm of \( \epsilon \), but that (ii) it finds solutions with sparsity that is comparable to that found by the matrix 2-norm code.

If we compare the matrix 2-norm optimization problem to the root-finding problem, there is one mathematical reason to prefer the root-finding problem: so long as all the degrees of the graph represented by the adjacency matrix \( A^0 + \epsilon \) have positive degree, the function \( c(\epsilon(\kappa)) \) is infinitely differentiable. On the other hand, the matrix 2-norm has a first derivative that is not itself differentiable. While convex optimization techniques are a possibility, the PageRank constraint is not convex, implying that the optimization problem (11) with (15) is itself non-convex. We leave the issue of studying convex relaxations of the optimization problem (11) with matrix 2-norm objective function (15) as an area for future work, and focus on testing the root-finding method against the Frobenius norm optimization problem (11).

### 6 Implementation and Final Results

We implemented the methods described in Section 4 in C++ using the Eigen template library for linear algebra (http://eigen.tuxfamily.org). Eigen supports sparse matrices, a key feature since the adjacency matrix \( A^0 \) for the InvestorRank problem is large and sparse.

The main linear algebra function required to implement either the optimization approach from Section 4.1 or either of the root-finding approaches from Sections 4.2-4.4 is the solution of either a low-rank or underdetermined system. Let us examine each in turn.

The optimization approach requires the solution of (11) at each step. This is a problem of the form \( Mw = z \) where \( M \) is square, symmetric, and positive semi-definite. In practice, however, we find that as the iteration count increases, the Hessian matrix \( M = \nabla \nabla^T L(x_i) \) develops zero singular values, i.e., the matrix loses full rank. This implies that there are many solutions of the \( Mw = z \) problem—we choose the solution \( w \) that minimizes \( \|w\|_2 \). To compute this solution, we use the Lapack routine *dgelsd*, which uses the SVD of \( M \) to compute \( w \).
Algorithm       Final $J$   Final $\|P(A^0 + \varepsilon) - P'\|_\infty$ # iterations   time (sec.)
IP              287.55   1.8420 $\times 10^{-14}$ 13       30
SQP             277.55   7.7759 $\times 10^{-17}$ 6.7       15

Table 1: Preliminary results from Matlab’s \texttt{fmincon} applied to the problem (11) for graphs with $n = 40$ vertices. All results are average results across 100 runs. The results here show that there do exist numerical solutions to the problem, and that it is possible to change the weights in a graph to create a desired PageRank. However, the solution matrices are not sparse, nor are these algorithms scalable: the run times for $n = 100$ are prohibitively large.

Algorithm       Final $J_2$   Final $\|P(A^0 + \varepsilon) - P'\|_\infty$ # iter   time   % nnz
IP              11.2889   7.7605 $\times 10^{-11}$ 209     4.4   46
SQP             11.1973   1.2763 $\times 10^{-8}$ 164     1.9   21

Table 2: Preliminary results from Matlab’s \texttt{fmincon} applied to the problem (11) with matrix 2-norm objective function (15) for graphs with $n = 10$ vertices. All results are average results across 100 runs. The results here show that the matrix 2-norm objective function (15) yields solutions of the optimization problem (11) that are much more sparse than the Frobenius norm objective function (12). However, neither algorithm scales well: the run times for $n = 40$ are prohibitively large.

Final $J$       Final $J_2$   Final $\|P(A^0 + \varepsilon) - P'\|_\infty$ # iter   time   % nnz
830.60         229.62      1.8353 $\times 10^{-17}$ 28        41.58 33

Table 3: Preliminary results from Newton’s method with pseudoinverse (33) applied to the solution of the inverse PageRank problem on graphs with $n = 40$ vertices. All results are averaged across 100 runs. The matrices found in this way are much more sparse than the ones found in Table 1.
The root-finding approach requires the computation of \( \text{det}(\mathbf{J}(\kappa)) \) at each step. The pseudoinverse part of the computation is equivalent to finding the solution \( \mathbf{s} \) of the underconstrained linear system \( [\mathbf{J}(\kappa_i)]\mathbf{s} = \mathbf{f}(\kappa_i) \) that minimizes \( \|\mathbf{s}\|_2 \). Once again we employ the Lapack routine \text{dgelSD} \( \), which accepts rectangular matrices such as \( [\mathbf{J}(\kappa_i)] \).

The routine \text{dgelSD} is implemented in both the Intel Math Kernel (MKL) and the AMD Core Math Library (ACML), both of which automatically make use of multi-core, shared memory architectures using OpenMP and Pthreads. We have developed versions of our code that work with both the MKL and ACML libraries.

We now describe a series of results obtained using our C++ codes.

6.1 Random Graph Test Results

The first set of results are for inverse problems on randomly generated graphs. These results are designed to test the algorithms and shed light on properties of the inverse problem.

6.1.1 Experimental Procedure

For a given test, we generate an Erdős-Rényi random graph on \( n \) vertices where the probability of an edge is 0.5. Let the adjacency matrix of this graph be \( \mathbf{A}^0 \). We compute the PageRank \( \mathbf{P}(\mathbf{A}^0) \), and then relabel the vertices of the graph so that they are in descending PageRank order, i.e., vertex 1 has the highest PageRank, and vertex \( n \) has the lowest PageRank. Let \( \mathbf{P}^0 \) be the PageRank vector for this relabeled graph. We set \( \mathbf{P}' = \mathbf{P}^0 \) except in the first two positions, which we swap, i.e., \( P'_1 = P^0_2 \) and \( P'_2 = P^0_1 \). We then run the inverse problem solver and obtain the solution of the inverse problem described in detail at the beginning of Section 4.

In other words, the test answers the question, “How does the network have to change so that the second-ranked and top-ranked vertices swap positions in the PageRank ranking?”

Typically, in continuous nonlinear programming or root-finding problems, the stopping criteria involves checking the change in the objective function and/or the norm of the gradient at the current iteration. However, our primary concern in solving the inverse problem is to obtain the desired ranking of vertices, not necessarily the desired values of the PageRank vector. Therefore, for the stopping criterion in our C++ codes, we use the rankings.

Recall the sort index function \( \mathcal{G} \) defined in (3). Then \( \mathbf{r} = \mathcal{G}(\mathbf{P}(\mathbf{A}^0 + \varepsilon(\kappa))) \) is the ranking corresponding to the current value of \( \kappa \) in the inverse problem solver, while \( \mathbf{r}' = \mathcal{G}(\mathbf{P}') \) is the desired ranking. Both \( \mathbf{r} \) and \( \mathbf{r}' \) are vectors of integers. Let \( \mathbf{v}_{1:10} \) denote the first 10 entries of the vector \( \mathbf{v} \). Then the ranking match stopping criterion used for the tests in this section is \( \mathbf{r}_{1:10} - \mathbf{r}'_{1:10} = 0 \). The motivation for using the top 10 entries comes from Section 3.

The constraint \( \mathbf{c}(\varepsilon(\kappa)) = 0 \) is used to guide the optimizer towards a solution that satisfies the ranking match stopping criterion. At the point where the algorithm stops, the constraint may not be satisfied. We note that there is nothing preventing us from using our codes to try to solve for \( \varepsilon \) such that the constraint is satisfied, i.e., the actual values of the PageRank vector are achieved. It is intuitive to expect that this will require many more iterations than the ranking match just described, and limited testing bears this out.

Let us extend the statements from the previous paragraph to the constraint \( \mathbf{g}(\kappa) = 0 \). In this case as well, when the algorithm stops, the constraints may not be satisfied. Let \( Z(\kappa) \) denote a version of \( \kappa \) in which all diagonal entries and all negative off-diagonal entries have been replaced by zero. In other words, \( Z(\kappa) \) is a version of \( \kappa \) such that \( \mathbf{g}(Z(\kappa)) = 0 \). When we check whether the ranking match stopping criterion is satisfied, we always use \( Z(\kappa) \) instead of \( \kappa \) itself. Stated another way, in the ranking match stopping criterion, we use the ranking vector \( \mathbf{r} = \mathcal{G}(\mathbf{P}(\mathbf{A}^0 + \varepsilon(Z(\kappa)))) \).
We also note that, in practice, the difference between $\kappa$ and $Z(\kappa)$ is negligible.

Being able to use special stopping criteria is a feature of our C++ codes. To the best of our knowledge, it is not possible to implement the ranking match stopping criterion using any of the solvers in Matlab’s optimization toolbox.

### 6.1.2 Guide to Plots

In what follows, we will include several plots of graphs. These plots will always show the graph whose adjacency matrix is $A^0 + \varepsilon$, where $\varepsilon$ is the perturbation that we solved for using one of our inverse problem solvers. In any such plot, black (solid) edges indicate edges that were present in the original adjacency matrix $A^0$ whose edge weight was *not changed* by the perturbation $\varepsilon$. The blue (dotted) edges indicate *new* or *altered* edges in $\varepsilon$.

The vertices will be labeled by descending PageRank order. Vertex 1 is the highest ranked vertex, while vertex $n$ is the lowest ranked vertex. The PageRank of a vertex is also indicated by the brightness of the shaded circle associated with the vertex: the brighter the shade, the higher the rank of the vertex.

The open-source NetworkX package ([http://networkx.lanl.gov/](http://networkx.lanl.gov/)) in Python was used to generate all graph diagrams in this thesis.

### 6.1.3 Optimization Approach

We first test the optimization approach described in Section 4.1. We run tests only for $n = 20$, because the Hessian is of size $N \times N$ where $N = n^2 + 2n + m$, and therefore the amount of memory required for this approach is $O(n^4)$.

The tests we perform are designed to study the effect of different initial conditions on the quality of the solution. All results are averaged over 500 runs. The metrics that we use to measure the quality of the solution are the percentage of runs where the desired ranking was achieved (% success), the number of seconds required to complete a run (wall clock time), the percentage of non-zero entries in the final $\kappa$ matrix, and the number of iterations required to produce the solution. Table 4 shows these results for the optimization-based approach.

We vary the initial guess in the following way. First, we test the strategy of setting the initial $\kappa$ equal to the $A^0$ matrix itself. In Table 4, this is displayed in the first line of the table under the heading *itself*. Second, we test the performance of setting the initial $\kappa$ equal to an Erdős-Rényi random graph on $n = 20$ vertices with edge probability equal to either $p = 0.0, 0.3, 0.5, 0.7, \text{ or } 1.0$. Clearly, setting the edge probability equal to 0.0 produces a fully zero matrix for the initial guess $\kappa$, while setting the edge probability equal to 1.0 produces an initial $\kappa$ that equals one at all non-diagonal entries. The results for the five choices of $p$ are displayed in the final five lines of Table 4.

We draw the following conclusions from Table 4:

- Success rate may not be a very simple function of the edge probability in the random graph initial guess. This leads us to believe that it may be very hard to use a random guess initial condition with the optimization-based method unless we know something in advance about the structure of $A^0$. In the absence of such knowledge, we should use $A^0$ itself as the initial guess.

- Our own optimization algorithm, though it uses a very simple Newton step, already produces results that are more sparse than the fully dense results obtained using Matlab’s *fmincon*—see Section 5. In other words, we see again that the choice of algorithm plays a large role in the sparsity of the final solution.
Next, we plot four representative individual results from the same optimization method. In Figure 2, we display four graphs corresponding to final adjacency matrices \( A^0 + \varepsilon \) obtained by the optimization-based inverse problem solver. From left to right, top to bottom, the edge probability for the initial random graph \( \kappa \) was 0.0, 0.25, 0.5, and 1.0. Edges that existed in \( A^0 \) and remain unchanged by \( \varepsilon \) are plotted in solid black, while new edges are plotted in dashed blue—see Section 6.1.2 for more details. Overall, we notice two trends. First, an initial guess that is less dense (more sparse) leads to a solution \( \varepsilon \) that is more dense (less sparse). Second, most of the edges incident on vertex 0 (the vertex with the highest PageRank in the original graph) are old edges. In fact, for the two plots on the left, there are no new edges at all incident on vertex 0, while there are numerous new edges incident on vertex 1. In short, to move up in the rankings, the second-ranked vertex must make many new connections while the top-ranked vertex makes none or few.

Overall, we see three potential areas of improvement and future study for this algorithm:

- Study a quasi-Newton version of the method that uses current and past gradients to approximate the Hessian at each step, and store this Hessian as a sparse matrix. This will require developing a sparse linear solver for underconstrained / low-rank systems, i.e., a sparse version of \texttt{dgelsd}.
- Combine the optimization method with idea (36) and thereby eliminate many of the graph-based constraints (13) from the problem. The idea (36) occurred to us after we had already decided to use the root-finding approach on the full InvestorRank problem. A mitigating factor is that eliminating the graph-based constraints still leaves a Hessian matrix of size \((n^2 + 2n) \times (n^2 + 2n)\); in other words, the method may not be scalable for large \( n \), even after we implement (36).
- Use a matrix-free approach that does not require storing the entire Hessian, but rather uses a function that computes the action of the Hessian on a given vector.

### 6.1.4 First Root-Finding Approach

Using the same metrics as in (6.1.3), we study the performance of the first root-finding approach described in Section 4.2. Again, we study the effect of different initial conditions on the quality of the solution, and we present in Table 5 results averaged over 500 runs. The first row of Table 5 corresponds to setting the initial \( \kappa \) equal to \( A^0 \) itself, while the remaining five rows correspond to setting the initial \( \kappa \) equal to an Erdős-Rényi random graph on \( n = 40 \) vertices with edge probability equal to either \( p = 0.0, 0.3, 0.5, 0.7, \) or 1.0.

The results indicate that the sparsity of the solution is positively correlated with the sparsity of the initial guess: as we increase the initial edge probability along the left column of Table 5, the percentage of non-zero entries in the final solution \( \varepsilon \) also increases. Indeed, when \( p = 1.0 \), the final solution is close to a complete graph on \( n = 40 \) vertices.

Interestingly, setting the initial \( \kappa \) equal to \( A^0 \), which is itself an Erdős-Rényi random graph on \( n = 40 \) vertices with edge probability \( p = 0.5 \), yields a significantly higher success rate than setting the initial \( \kappa \) to some other random graph with edge probability \( p = 0.5 \). We interpret this to mean that the structure of the original graph plays a role in the final solution, which suggests that there may be a way to exploit the structure of \( A^0 \) to obtain an even better initial guess for \( \kappa \).

Next, we plot four representative individual results from the same root-finding method. In Figure 3, we display four graphs corresponding to final adjacency matrices \( A^0 + \varepsilon \) obtained by the first root-finding inverse problem solver. From left to right, top to bottom, the edge probability for the initial random graph \( \kappa \) was 0.0, 0.3, 0.7, and 1.0. Edges that existed in \( A^0 \) and remain
Figure 2: We display representative individual results from tests of the optimization-based method described in Section 4.1. From left to right, top to bottom, we use as our initial $\kappa$ a random adjacency matrix on $n = 20$ vertices with edge probability given by $p = 0.0$, $0.25$, $0.5$, and $1.0$. In the two plots on the left, for vertex 1 to change its rank from second to first, none of the edges incident on vertex 0 are altered. Note also that an initial guess that is less dense leads to more new/altered edges. For further details, see Section 6.1.3.

unchanged by $\varepsilon$ are plotted in solid black, while new/altered edges are plotted in dashed blue—see Section 6.1.2 for more details. Note that as the initial guess for $\kappa$ becomes more dense, so does the final solution, agreeing perfectly with the findings in Table 4.

6.1.5 Second Root-Finding Approach

Using the same metrics as in (6.1.3), we study the performance of the first root-finding approach described in Section 4.4. Again, we study the effect of different initial conditions on the quality of the solution, and we present in Table 6 results averaged over 500 runs. The first row of Table 6 corresponds to setting the initial $\kappa$ equal to $A^0$ itself, while the remaining five rows correspond to setting the initial $\kappa$ equal to an Erdős-Rényi random graph on $n = 40$ vertices with edge probability equal to $p = 0.3$, $0.5$, $0.7$, and $1.0$.

In Table 7, we show corresponding results for graphs on $n = 100$ vertices with initial $\kappa$’s generated using random graphs with edge probability equal to $p = 0.0$, $0.5$, and $1.0$.

There are three main conclusions to draw from the results in Tables 6-7:

- This is by far the fastest algorithm we have for solving the inverse problem. The average
### Table 4

Here we test the performance of the optimization-based method described in Section 4.1. All results are averaged over 500 runs. In the first row, we set the initial $\kappa$ equal to the $A^0$ matrix itself. Next, in the final five rows, we test the performance of setting the initial $\kappa$ equal to an Erdős-Rényi random graph on $n = 20$ vertices with edge probability equal to either $p = 0.0$, $0.3$, $0.5$, $0.7$, or $1.0$. The table shows that the success rate is not a simple function of the initial edge probability $p$, and also that our optimization method produces results with more sparsity than those produced by Matlab’s `fmincon` in Section 5. For further details, see Section 6.1.3.

<table>
<thead>
<tr>
<th>init. prob.</th>
<th>% success</th>
<th>wall clock time</th>
<th>% non-zero</th>
<th># iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>itself</td>
<td>0.9780</td>
<td>0.9159</td>
<td>74.63</td>
<td>2.2147</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.9821</td>
<td>3.1622</td>
<td>55.90</td>
<td>2.6288</td>
</tr>
<tr>
<td>0.3000</td>
<td>0.7120</td>
<td>9.3341</td>
<td>64.85</td>
<td>12.4494</td>
</tr>
<tr>
<td>0.5000</td>
<td>0.7700</td>
<td>9.2755</td>
<td>74.40</td>
<td>18.1169</td>
</tr>
<tr>
<td>0.7000</td>
<td>0.6588</td>
<td>10.5642</td>
<td>83.64</td>
<td>34.0144</td>
</tr>
<tr>
<td>1.0000</td>
<td>0.8740</td>
<td>1.0135</td>
<td>97.41</td>
<td>11.2952</td>
</tr>
</tbody>
</table>

### Table 5

We study the performance of the first root-finding approach from Section 4.2. All results are averaged over 500 runs. The first row corresponds to setting the initial $\kappa$ equal to $A^0$ itself, while the remaining five rows correspond to setting the initial $\kappa$ equal to an Erdős-Rényi random graph on $n = 40$ vertices with edge probability equal to the indicated value. The results indicate that the sparsity of the solution is positively correlated with the sparsity of the initial guess. Interestingly, setting the initial $\kappa$ equal to $A^0$, which is itself an Erdős-Rényi random graph on $n = 40$ vertices with edge probability $p = 0.5$, yields a significantly higher success rate than setting the initial $\kappa$ to some other random graph with edge probability $p = 0.5$. For further details, see Section 6.1.4.

<table>
<thead>
<tr>
<th>init. prob.</th>
<th>% success</th>
<th>wall clock time</th>
<th>% non-zero</th>
<th># iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>itself</td>
<td>0.9640</td>
<td>0.5257</td>
<td>72.14</td>
<td>7.9606</td>
</tr>
<tr>
<td>0.0000</td>
<td>0.9880</td>
<td>0.4280</td>
<td>69.30</td>
<td>6.4393</td>
</tr>
<tr>
<td>0.3000</td>
<td>0.6800</td>
<td>0.8295</td>
<td>39.63</td>
<td>9.2500</td>
</tr>
<tr>
<td>0.5000</td>
<td>0.9860</td>
<td>0.4927</td>
<td>67.91</td>
<td>7.2252</td>
</tr>
<tr>
<td>0.7000</td>
<td>0.4486</td>
<td>0.9195</td>
<td>33.22</td>
<td>9.5922</td>
</tr>
<tr>
<td>1.0000</td>
<td>0.3320</td>
<td>0.7438</td>
<td>34.08</td>
<td>7.8313</td>
</tr>
</tbody>
</table>
Figure 3: We display representative individual results from tests of the first root-finding method described in Section 4.2. From left to right, top to bottom, the initial $\kappa$ is the adjacency matrix of a random graph on $n = 40$ vertices with edge probability given by $p = 0.0, 0.3, 0.7,$ and $1.0$—note that as this probability increases, the initial guess becomes more dense, and so does the final solution, as indicated by the much greater fraction of blue dashed edges. This is in excellent agreement with the results of Table 5. For further details, see Section 6.1.4.

- Again, the sparsity of the initial guess is directly related to the sparsity of the final solution.
- Especially for larger values of the initial edge probability, the algorithm converges in far fewer iterations than the first root-finding algorithm, as recorded in Table 5. This will become important for large problems, where the computational cost of even one iteration is very high.

Next, we plot four representative individual results from the same root-finding method. In Figure 4, we display four graphs corresponding to final adjacency matrices $A^0 + \varepsilon$ obtained by the second root-finding inverse problem solver. From left to right, top to bottom, the edge probability for the initial $\kappa$ was $0.1, 0.3,$ or $0.7$; in the lower-right plot, the initial $\kappa$ was set to $A^0$ itself. Edges that existed in $A^0$ and remain unchanged by $\varepsilon$ are plotted in solid black, while new/altered edges are plotted in dashed blue—see Section 5.1.4 for more details.
Figure 4: We display representative individual results from tests of the second root-finding method described in Section 4.4. From left to right, top to bottom, the initial $\kappa$ with $n = 40$ vertices has edge probability $p = 0.1, 0.3$, or $0.7$; in the lower-right plot, the initial $\kappa$ was set to $A^0$ itself. Note once again that as we increase the probability for the initial guess to have an edge, the less sparse of a final solution we find. The graph where the initial guess for $\kappa$ is $A^0$ is not necessarily dense, but we are making the guess that we want to change all existing edges, yielding the fully dense solution in the lower-right plot. For further details, see Section 6.1.5.

**Constraining the Sparsity.** Let $\tilde{\kappa}$ be a matrix of size $\tilde{n} \times \tilde{n}$. Let us constrain $\kappa$ to be

$$\kappa = \begin{bmatrix} \tilde{\kappa} & 0 \\ 0 & 0 \end{bmatrix}.$$ 

Here we are essentially asking if we can change only a small subgraph of the original graph and still achieve the desired ranking. With this setup, both the PageRank and graph-based constraints operate only on the small subgraph. Computing the constraint function $f$ and the Jacobian $Jf$ under this condition is easy. The only part worth mentioning is the new Jacobian calculation—here all we do is grab the columns of the original Jacobian corresponding to the flattened vector mapping of the elements in $\tilde{\kappa}$ with respect to $\kappa$.

We only have to enforce $\tilde{n}$ graph based constraints because that is the effective number of diagonal elements that must be constrained. We also introduce the parameter $n_{pr}$, the number of PageRank constraints we choose to enforce. This is implemented very simply, by changing both the PageRank constraint function $c$ and its Jacobian, so that only the first $n_{pr}$ rows of each are
computed and stored. With the above constraint on $\kappa$, it does not make sense to constrain more than $\tilde{n}$ of the PageRank entries, so we set $n_{pr} = \tilde{n}$.

Therefore, our Jacobian will be of size $2\tilde{n} \times \tilde{n}^2$, allowing us to set $\tilde{n} \ll n$ and then solve problems for graphs with very large values of $n$. Additionally, this procedure promotes sparsity since blocks of the $\kappa$ matrix are already zero.

For $n = 500$ and $\tilde{n} = 50$, we run the same style of tests as above, varying the probability of an edge in the random graph used to generate the initial $\tilde{\kappa}$ matrix. The results, summarized in Table 8, show incredible success at improving the sparsity of the solution. In the case where the initial guess for $\tilde{\kappa}$ is the upper $\tilde{n} \times \tilde{n}$ submatrix of $A^0$ itself, we also found a very high 99% success rate and a small running time of less than 2 seconds on average.

In Figure 5, we plot four representative individual results using this same method, but with $n = 40$ and $\tilde{n} = 5$. From left to right, top to bottom, the edge probability for the initial $\tilde{\kappa}$ was 0.4, 0.7, or 1.0; in the lower-right plot, the initial $\tilde{\kappa}$ was set to the upper $\tilde{n} \times \tilde{n}$ submatrix of $A^0$ itself. Edges that existed in $A^0$ and remain unchanged by $\varepsilon$ are plotted in solid black, while new/ altered edges are plotted in dashed blue—see Section 6.1.2 for more details. Since the fraction of the graph that is allowed to change is so small, one must zoom in on the top-ranked vertices in the diagrams to actually see the new/altered edges. Once again, we observe that the sparser we make the initial guess, the sparser a final solution we obtain.

Overall, the excellent results provided by the $\tilde{\kappa}$ method motivate us to use this method to solve the InvestorRank inverse problem.

6.2 Application to InvestorRank

Based on what we have learned through the algorithm tests above, we are in an excellent position to solve the inverse problem for the InvestorRank network where $n = 6049$, a much larger graph than any of the graphs we have considered in our tests.

To solve this problem, several modifications to the standard procedure were required. First, after calculating the PageRank $P^0$ of the original matrix $A^0$ (whose construction was described in detail in Section 3), we relabeled the vertices of the graph in order of descending PageRank. We then computed that the desired PageRank, i.e., the ranking necessary to match Farmer’s ranking, is as follows:

| new vertex # | 1 2 3 4 5 6 7 8 9 10 |
| old vertex # | 22 7 6 17 89 19 3 5 33 18 |

We therefore generated an artificial desired PageRank vector $P'$ using the corresponding entries of $P^0$ given in the second row, i.e., $P'_1 = P^0_{22}, P'_2 = P^0_7$, etc. We then set entries 11 through $n$ of $P'$ equal to $P^0_{\tilde{n}}$, and then normalized $P'$ so that it sums to one. Recall that the PageRank vector sums to one because it is a stationary probability distribution for a particular Markov chain on the graph.

Armed with this $P'$, we run the inverse problem solver using the second root-finding approach from Section 4.4 together with the $\tilde{\kappa}$ approach from Section 6.1.5. For the solution we present, we set $\tilde{n} = 500$ and $n_{pr} = \tilde{n} = 500$. Smaller choices lead to solutions with undesirably large edge weights. Larger choices quickly exhaust the 12 GB of available RAM on the computer used for testing—one should keep in mind that the Jacobians in the problem are of size $1000 \times 500^2$. We note for reference that the computer used for testing has two Intel i7 hexacore processors, and is capable of running 12 simultaneous threads.

Since each iteration is so expensive, and since our earlier attempts at producing the same top 10 rankings in any order failed (see Section 3.3), we modified the stopping criterion for the solver.
<table>
<thead>
<tr>
<th>init. prob.</th>
<th>% success</th>
<th>wall clock time</th>
<th>% non-zero</th>
<th># iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>itself</td>
<td>0.9860</td>
<td>0.0635</td>
<td>49.04</td>
<td>1.3631</td>
</tr>
<tr>
<td>0.3000</td>
<td>0.9160</td>
<td>0.1273</td>
<td>30.24</td>
<td>3.0175</td>
</tr>
<tr>
<td>0.5000</td>
<td>0.9560</td>
<td>0.2847</td>
<td>49.65</td>
<td>3.0586</td>
</tr>
<tr>
<td>0.7000</td>
<td>0.7940</td>
<td>0.1806</td>
<td>68.72</td>
<td>3.8665</td>
</tr>
<tr>
<td>1.0000</td>
<td>0.9400</td>
<td>0.1140</td>
<td>97.47</td>
<td>2.5149</td>
</tr>
</tbody>
</table>

Table 6: We study the performance of the second root-finding approach from Section 4.4 for graphs with \( n = 40 \) vertices. All results are averaged over 500 runs. The first row corresponds to setting the initial \( \kappa \) equal to \( A^0 \) itself, while the remaining four rows correspond to setting the initial \( \kappa \) equal to a random graph with edge probability equal to the indicated value. For further details, see Section 6.1.5. We see that the sparsity of the initial \( \kappa \) controls the sparsity of the final solution, agreeing with our results on the first root-finding method in Table 5; however, the clock times are at least a factor of 10 smaller than the times for that method.

<table>
<thead>
<tr>
<th>init. prob.</th>
<th>% success</th>
<th>wall clock time</th>
<th>% non-zero</th>
<th># iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>itself</td>
<td>0.7850</td>
<td>1.7710</td>
<td>49.66</td>
<td>1.6746</td>
</tr>
<tr>
<td>0.3000</td>
<td>0.4480</td>
<td>5.0410</td>
<td>0.23</td>
<td>9.7239</td>
</tr>
<tr>
<td>0.5000</td>
<td>0.7660</td>
<td>5.0776</td>
<td>0.41</td>
<td>9.6710</td>
</tr>
<tr>
<td>0.7000</td>
<td>0.7980</td>
<td>4.3570</td>
<td>0.59</td>
<td>7.9975</td>
</tr>
</tbody>
</table>

Table 7: We study the performance of the second root-finding approach from Section 4.4 for graphs with \( n = 100 \) vertices. The interpretation of results and other details are as in Table 6 and Section 6.1.5.

<table>
<thead>
<tr>
<th>init. prob.</th>
<th>% success</th>
<th>wall clock time</th>
<th>% non-zero</th>
<th># iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>itself</td>
<td>0.9902</td>
<td>1.7198</td>
<td>0.97</td>
<td>1.9527</td>
</tr>
<tr>
<td>0.3000</td>
<td>0.7460</td>
<td>5.0410</td>
<td>0.23</td>
<td>9.7239</td>
</tr>
<tr>
<td>0.5000</td>
<td>0.7660</td>
<td>5.0776</td>
<td>0.41</td>
<td>9.6710</td>
</tr>
<tr>
<td>0.7000</td>
<td>0.7980</td>
<td>4.3570</td>
<td>0.59</td>
<td>7.9975</td>
</tr>
</tbody>
</table>

Table 8: We study the performance of the second root-finding approach from the end of Section 4.4 where \( n = 500 \) and \( \tilde{n} = 50 \). The itself guess is the upper left block in \( A \) of size \( \tilde{n} \times \tilde{n} \). All other guesses follow the same format as in Table 6. Notice that the same trends exist here involving the second root-finding method as explained in Table 6 and at the end of Section 4.4.
Figure 5: These plots show test results for the case where $n = 40$ and $\tilde{n} = 5$, using the sparsity constraining scheme where only the upper $\tilde{n} \times \tilde{n}$ part of $\kappa$ is allowed to vary in the inverse problem solver. From left to right, top to bottom, the edge probability for the initial $\tilde{\kappa}$ was 0.4, 0.7, or 1.0; in the lower-right plot, the initial $\tilde{\kappa}$ was set to the upper $\tilde{n} \times \tilde{n}$ submatrix of $A^0$ itself. Zooming in on the top five nodes reveals the new/altered edges. Again, the sparser the guess, the sparser the final solution.

Using the same notation used to describe the ranking match criterion in Section 6.1.1, we can state the new stopping criterion as

$$\text{sort}(r_{1:10}) - \text{sort}(r'_{1:10}) = 0.$$ 

In other words, the solver stops if the same investors appear in both our top 10 ranking and Farmer’s top 10 ranking, even if the ordering of our top 10 ranking differs from Farmer’s.

With this stopping criterion, and with an initial $\tilde{\kappa}$ consisting of the upper-left $500 \times 500$ block of the original $A^0$ matrix, the solver converged in two iterations to a solution. The desired and obtained rankings are:

| desired # | 22 7 6 17 89 19 3 5 33 18 |
| obtained # | 22 7 6 5 3 89 19 18 33 17 |

The number of nonzero entries in the original $A^0$ is 79532, and the number of nonzero entries
in the solution $A^0 + \varepsilon$ is 79842. This implies that our solution requires 155 new edges to be formed.

We plot the solution in Figure 6 and Figure 7. The solution in Figure 6 shows the entire network, which contains one giant component together with many small components that are disconnected from one another. In Figure 7, we plot only the giant component. Zooming in on the central black tangle allows one to see, plotted in red, the new/altered edges in the solution of the inverse problem.

6.3 Future Directions / Conclusion

The root-finding approach where we built the inequality constraint into the definition of $\varepsilon$ performed the best, and we will continue to make improvements to this method. We would also like to demonstrate mathematical properties for this root-finding approach, such as when the method converges, how one can control the sparsity of the solution, and whether there exist adaptive stepping schemes for the Newton iteration.

In our test results, the desired ranking vector $r'$ consisted of only swapping the number one and number two ranked nodes. Other problems can be solved, where different nodes are swapped, and the solution properties for these problems can now be studied in detail.
Figure 7: Solution of the InvestorRank inverse problem with $n = 6049$ and $\tilde{n} = 500$. Here we show only the giant component of the network. The inverse problem solution requires only 155 new edges, plotted in red. Existing edges of $A^0$ are plotted in black. No vertex discs/labels are plotted. See Section 6.2 for further details.

We obtained a solution to the inverse InvestorRank problem, but we have not examined its details in the context of financial applications. For example, an interesting direction is to examine the new edges that must be formed in order for Farmer’s ranking to be obtained—do these new edges follow a pattern based on shared investment interests? Would it be feasible for an investor to use these new edges to direct her future co-investment strategy? Also, our snapshot of the CrunchBase network was taken in 2011; since then, the amount of information in this public domain database has increased considerably. In the near future, we hope to test our methods on the most recent data we can obtain.

We were successful in finding solutions where we wanted to match the PageRank for a given adjacency matrix. We could apply this same method to solve inverse problems for other network measures. For example, we could try the same approach as above, but instead try to match rankings of the eigenvector centrality. Another avenue is to use a more recent version of the PageRank measure, one that incorporates personalized and/or topic-specific rankings.
References


