CS223: Efficient Algorithms for Detecting & Classifying Credit Communities

Nao Ouyang, Juspreet Singh Sandhu, Mark York

December 2019

1 Motivation

Useful information exists in social networks, but it is difficult to access. One example is lenders in the developing world, who wish to know which borrowers are likely to repay. Currently lenders face widespread defaults and must charge exorbitant interest rates [HRH18, MMM+13]. Solving this problem could increase incomes, improve food security, and reduce civil unrest [Cab19].

Companies, including Branch and Tala, use machine learning models to determine creditworthiness of individual borrowers, but reliable data is hard to find. Fortunately, community members already know who is creditworthy. If lenders could provide performance-based incentives to community members to recommend creditworthy peers, the lenders could discover the creditworthy borrowers in the network, lend only to them, and make more stable returns while charging lower interest. Designing an efficient algorithm to gather and use these recommendations requires reasoning about the graph representing the social network.

We consider graphs where each vertex represents a community member, and each edge represents a relationship between them. There are a number of design considerations for our graph and search algorithms. We could consider some vertices to be recommenders, some to be borrowers, and some to be both, or we could restrict our consideration to only borrower nodes. Borrower nodes may be thought of as binary (creditworthy/ not creditworthy) or as existing on some normalized, continuous scale; we will restrict our consideration to the binary case. Similarly, graphs may be represented with weighted or unweighted edges to represent the level of knowledge, trust, or a combination of the two between community members. We will consider the edge weights to have one continuous weight value representing trust, with a value of zero being completely neutral. Additionally, we note that many social graphs consist of densely connected disjoint sub-communities with sparse connections between two communities. Throughout our paper, we work in this regime. It is also important to consider varying levels of recommender truthfulness and knowledge, and various baseline rates for creditworthy vs. non-creditworthy borrower vertices. Target performance metrics include false positive rates of borrower recommendations, number of iterations required to discover 50%, 90%, and 100% of creditworthy nodes, and total cost to discover all creditworthy nodes (assuming there is some cost per recommendation and a higher cost for false positives). These metrics will be formalized in the introduction below.

The paper is organized as follows: Section 2 introduces notation and basic definitions, our model and problem statements, and summarizes our contributions. Section 3 surveys related approaches that we consider for our algorithms. Section 4 covers our formulation of the problem and related theoretical results. Section 5 shows the results of our simulations, and Section 6 includes our conclusion and next steps.

2 Introduction

We are interested in using social knowledge to find creditworthy borrowers. This knowledge exists among potential borrowers' social contacts, and we want to model how to use this network to discover these contacts. We first introduce basic notations and definitions, and then rigorously introduce our model, assumptions about it, and the problem we wish to investigate.

2.1 Notation & Definitions

We represent a graph as $G = (V, E)$, where V denotes the set of vertices and $E \subseteq V \times V$. All graphs are assumed to be simple; that is, they have no self-loops. A graph can be undirected, in which case E is symmetric, or directed, in which case E is not symmetric. The set $\Gamma(v) = \{u \in V \mid (v, u) \in E\}$ denotes the vertices to which v can flow. A weight function $w : E \to \mathbb{R}$ maps edges to some real number. Often, we will normalize this weight so that $w(e) \in [-1, 1], \forall e \in E$. Unless otherwise stated, $H : \textit{string} \to \mathbb{Z}$ is a hash function that maps strings to integers. $\mathbf{C}$ will denote a trust matrix and contain *normalized trust values* for all edges in the graph.

We now introduce a few basic definitions that are consistently used in this paper:

Definition 1 (Bipartite Clique)**.** *A bipartite clique of a graph $G = (V, E)$ is a disjoint subset of vertices $A, B \subset V$, such that, $e = (a, b) \in E, \forall a \in A, b \in B$.*

Definition 2 (Dense Bipartite Subgraph)**.** *A dense bipartite subgraph of a graph $G = (V, E)$ is a disjoint subset of nodes $A, B \subset V$, such that, $|E'| = |\{e = (a, b) \mid a \in A, b \in B\}| \geq c\sqrt{|A||B|}$ for some fixed $c \in (0, 1)$.*

This definition is chosen from the one defined in [NSC+08]. In general, *k-Dense Subgraph* refers to the induced subgraph of k vertices with maximum average degree (or, equivalently, maximum number of edges). As it turns out, a *k-Dense Subgraph* can be efficiently approximated up to a factor of $\mathcal{O}(n^{\delta})$, for some $\delta < \frac{1}{3}$. Definition 2 refers specifically to dense bipartite subgraphs. However, since **MAX-CUT** is known to be NP-Hard, so is computing the *most dense* bipartite subgraph.

Definition 3 (Jaccard Similarity)**.** *The Jaccard similarity between 2 sets $A, B \subset U$ computes the normalized intersection between them and is given by $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$.*

Definition 4 (c-Shingle)**.** *A c-Shingle estimates the Jaccard similarity between 2 sets $A, B \subset U$ by computing $\frac{1}{c}\sum_{i=1}^{c} \mathbb{1}_{v_i = w_i}$, where, $\mathbf{v_c} = (\min_{a \in A} \pi_1(a), .., \min_{a \in A} \pi_c(a))$ and $\mathbf{w_c} = (\min_{b \in B} \pi_1(b), .., \min_{b \in B} \pi_c(b))$, for permutations π_j chosen i.i.d. uniformly at random from the group of permutations $S_{|U|}$.*

Notice that a *c-Shingle* simply computes the probability that the element with the minimal image under some π_j chosen uniformly at random is the same for both A and B. This is the same as $J(A, B)$. A simple Chernoff bound tells us that this is ϵ-close to the true similarity with probability $\geq 1 - exp(-\epsilon^2 c)$. For $\epsilon = 10^{-2}$ and $c = c' 10^4$, this tells us our estimate is always ϵ-close to the true similarity with probability $\geq 1 - 1/c'$. This is desirable when n is very large.
We can naturally generalize the definition of a c-Shingle, which estimates the single-element similarity of 2 sets A, B to estimate similarity for subsets of A, B of size s.

Definition 5 ((c,s)-Shingle)**.** *A (c, s)-Shingle of elements $A = \{a_1, .., a_n\}$ is constructed as a vector $v_c = (z_1, .., z_c)$, where each z_i is constructed by hashing the string formed by concatenating the s minimal elements of A under the action of c random permutations $\pi_1, .., \pi_c$. More concretely, $z_i = H('y_1 y_2 .. y_s')$, where $\{y_1, .., y_s\}$ are the s minimal elements of A under the action of π_i.*

2.2 Problem Definition & Contribution

Community members exist on an unknown social graph $G = (V, E)$. This graph will be treated as either directed or undirected, depending on the specific sub-problem we model. We further assume G is connected, and that it is a disjoint union of dense components (which can be big, but occupy no more than a constant fraction of the graph). We will assign weights to the edges of G to represent different semantic values for different sub-problems. There are 2 types of vertices in the graph G:

- Potential borrowers who are creditworthy (CW)
- Potential borrowers who are not creditworthy ($\overline{CW}$)

The general problem involves detecting all CW borrowers as quickly as possible with minimal cost. The system operator can pay graph members (vertices) for recommending neighboring graph members who are creditworthy. Specifically, under the assumptions above, we propose algorithms to answer the following questions:

1. How do we detect the set (or an approximation to) of CW borrowers using recommendations from community members ?
2. How would the distribution induced by the density of CW borrowers (and their neighborhood) affect the rate of convergence of a discovery algorithm on the graph ?
    - Due to high heterogeneity of borrower skill levels and payback likelihood [HRH18], it is reasonable to assume that the set of good potential borrowers is sparse in the set of the nodes of the graph.
3. Assuming some cost function on the vertices for every query to determine whether a node is CW, a cost for false positives, and a cost for wait time to discovery of CW borrowers, what is an efficient algorithm for discovering the set of vertices representative of CW borrowers ?

We approach each of these problems with different models, detailed in Section 4.

2.3 Design Choices

Social networks are complex, dynamic systems. Simplifying assumptions must be made to model them in a tractable way. We list here some of these design decisions we made.

1. *The creditworthiness of underlying borrowers is binary.* We denote the underlying fraction of the population which is creditworthy as p_{CW}. We allow this to vary, as it is a key characteristic of a population. We denote borrower i as b_i, and if borrower i is creditworthy we say $b_i = CW$. In future work we may consider a continuous level of creditworthiness.
2. *We assume that the distribution of creditworthy (CW) and non-creditworthy agents is random* with adjacent borrowers independent of each other. It is possible that CW and non-CW people exist in concentrated groups, affecting algorithm performance. We denote this independence as $Pr(b_i = CW | neighbor(b_i) = CW) = Pr(b_i = CW)$.
3. *Variable percent of initial vertices are selected as initial recommenders* $p_{init} = \frac{|v_{recommenders\_initial}|}{|v|}$ We vary this and assess performance. In reality, there is a cost associated with finding more initial recommenders, setting up a potential tradeoff.
4. *Recommendation language:* We ask recommenders to give a binary positive or negative recommendation (+1 or -1) for each of their neighbors for the main algorithm. In the Eigentrust case, recommenders must give a continuous score $s_{ij} \in [0, 1]$, with 0 being least creditworthy.

5. *Conditional probability* that a recommender recommends a potential borrower given that they are creditworthy - $Pr(rec_{ij} = CW | b_j = CW)$ We allow this to vary and assess performance. rec_{ij} represents the recommendation from recommender i about borrower j. In the EigenTrust case, we assume a normal distribution with $\mu = cond\_probability$ and $\sigma = 0.1$, where values $\notin [0, 1]$ are truncated to be in $[0, 1]$.

6. *Conditional probability* that a recommender recommends a potential borrower given that they are not creditworthy - $Pr(rec_{ij} = CW | b_j = \overline{CW})$

7. *How to take recommendations from recommenders:* we have recommenders give us ratings on all of their neighbors at once. However, one can imagine having recommenders make a small set of recommendations, evaluating those recommendations, then making further lending decisions with high-performing recommenders.

8. *Weight of in-community recommendations vs. out-of-community recommendations* By using dense-subgraph-identification techniques outlines below, we can assign agents to dense communities, and determine whether recommendations are coming from in-community or ex-community. Empirically, this has been shown to be important. Hussam et al. found that recommenders were 24-35% less accurate when resource allocation was on the line, and that this effect was strongest for family members and close friends [HRH18]. This is outside of scope for our current research and we do not consider it, but it would make for interesting future work.

3 Related Work

As far as we are aware, no one has applied graphical models to credit analysis before. However, the subcomponents of our problem have seen extensive study, and we cover this in the section below. First, we discuss the EigenTrust algorithm designed for peer-to-peer file sharing networks, used later in sections 4 and 5 for our simulation. Next, we consider algorithms for isolating dense, highly-connected subgraphs in the overall graph so that EigenTrust might be run on them separately to see if this provides more accurate trust scores than the global EigenTrust results. We then briefly touch on a range of graph discovery and population modeling algorithms that have been used in other contexts. Finally, we discuss an algorithm that uses Bayesian updates for probabilistic assessments of trust in web services, which could be applied in a non-binary credit scoring system.

3.1 EigenTrust

The EigenTrust framework [KSGM03] is a means for devising global trust scores for an agent in a networked system based on the local trust scores of its peers. Kamvar et al. developed EigenTrust for the context of peer-to-peer file-sharing systems where some peers are reliable sharers of quality content, while others are not. The five design considerations include:

1. The system should be de-centrally *self-policing* without a central authority (this differs from our lending scenario which has a central bank)

2. The system should maintain *anonymity* so that peers are not known by an externally-associated identifier (e.g. IP address) (we would prefer unique identifiers for members of our lending system, such as government ID numbers, which are surprisingly widespread in the developing world).

3. The system should not assign *profit to newcomers* such that it is not advantageous for peers with poor reputations to drop out of the network and re-enter with newcomer status (relevant for us)

4. The system should have *minimal overhead* in terms of computation, infrastructure, and message complexity (applies to us as well - a country may have millions of inhabitants, and the rating system cannot

be overly complex)

5. The system should be *robust to malicious collectives* of peers who know one another and attempt to collectively subvert the system

Each agent i calculates its local trust score of agents with whom it has interacted as

$$s_{ij} = sat(i,j) - unsat(i,j) \tag{1}$$

where $sat(i,j)$ and $unsat(i,j)$ represent the number of satisfactory and unsatisfactory interactions, respectively. Each agent then normalizes its trust scores to be between 0 and 1 as follows.

$$c_{ij} = \frac{max(s_{ij}, 0)}{\sum_j max(s_{ij}, 0)} \tag{2}$$

Assuming a type of transitive property of trust, an agent may assess its trust of it's peers' peers by multiplying the weighted trust scores

$$t_{ik} = \sum_j c_{ij} c_{jk} \tag{3}$$

This can be represented for all 2nd degree peers using matrix notation as follows.

$$t = C^T c_i \tag{4}$$

and by taking C^T to successively higher powers, for some k the value of t converges to a stable, globally-uniform vector. This vector has several interesting properties:

$$t = (C^T)^k c_i \tag{5}$$

- No matter which player's initial c_i is used in 5 above, the limit vector t will be the same global value. This is the left-principal eigenvector of the matrix C.
- Convergence happens rather quickly; Kamvar et al. found in their simulation with 100 graph members that convergence typically happened with values of $k = 5$ to $k = 10$.
- It is possible for the global value to be computed separately by each agent when successive reports of intermediate t values are shared by every agent's peers at each round of calculation

The pseudocode for running this base algorithm is as follows:

Algorithm 1 EigenTrust Base Algorithm

$\overrightarrow{t}^{(0)} = \sum_i \frac{\overrightarrow{c}_i}{n}$
repeat
  $\overrightarrow{t}^{(k+1)} = C^T \overrightarrow{t}^{(k)}$
  $\delta = ||t^{(k+1)} - t^k||$
until $\delta < \epsilon$

One issue faced by this algorithm is the presence of malicious peers who know each other, and give each other high ratings and their neighbors low ratings to skew the global trust value of the system. To forestall this, Kamvar et al. considered that most networks have founding members who are known to be trustworthy, a set of peers they denote as p. In our case, these could be carefully-chosen initial recommenders. The algorithm is then modified to include weighting between the original matrix multiplication and the $\overrightarrow{p}$ vector at each step. $\overrightarrow{p}$ is also used to fill in normalized trust vectors for agents who do not provide any ratings. For some constant $a \in (0, 1)$, the algorithm becomes

$$\overrightarrow{t}^{(k+1)} = (1-a)C^T \overrightarrow{t}^{(k)} + a\overrightarrow{p} \tag{6}$$

And the pseudocode becomes

Algorithm 2 EigenTrust Trusted Peer Algorithm

$\overrightarrow{t}^{(0)} = \overrightarrow{p}$
repeat
  $\overrightarrow{t}^{(k+1)} = C^T \overrightarrow{t}^{(k)}$
  $\overrightarrow{t}^{(k+1)} = (1-a)\overrightarrow{t}^{(k+1)} + a\overrightarrow{p}$
  $\delta = ||t^{(k+1)} - t^k||$
until $\delta < \epsilon$

To evaluate the runtime of the above two algorithms, note that multiplying an nxn matrix by an nx1 vector will require $\mathcal{O}(n^2)$ steps, and since this may need to be done for a maximum of n iterations before convergence, the total algorithm runs in $\mathcal{O}(n^3)$ steps, which is polynomial time in n.

EigenTrust provides a framework for calculating trust in a somewhat dense environment of ratings. For this, it would be best to have recommenders give non-binary ratings to their level of financial trust to each neighbor. It is an open question to what degree this could work in sparse ratings datasets. In addition, it is unclear how to map global trust values to lending decisions, and repayment results to recommender compensation. One idea for recommender compensation would be the Vickrey-Clarke-Groves mechanism, which would reward recommenders based on the difference their knowledge made to overall system welfare vs. the world without their recommendations [PS16]. A weakness of the EigenTrust aglorithm is that it does not differentiate between neutral situations (i.e. where there have been no interactions between nodes) and negative experiences.

3.2 k-Dense Subgraphs

The topic of community detection often involves detecting a set of densely connected networks $C = \{G_1, .., G_m\} \subset \mathcal{P}(G)$ within a large graph $G = (V, E)$. In its most natural form, the "density" of a graph can also be measured by its average degree - Intuitively, this corresponds to how many edges the graph has. Therefore, Graph Density is defined as:

Definition 6 (Graph Density)**.** *Given a directed graph $G = (V, E)$, its density is defined as $d(G) = \frac{|E|}{|V|(|V|-1)}$*

Notice that, for a directed graph $G = (V, E)$, $|E| = \sum_{v \in V} out\_deg(v)$. So, for some fixed $k > 0$, finding the densest sub-graph G' in G with fixed size k corresponds to finding G' with the largest average degree. This is known as the k-Dense Graph problem (**DkS**). Unfortunately, the **DkS** problem is known to be NP-Hard. There exists a polynomial time approximation algorithm, as discussed in Section 2.1. Formally, the following theorem holds:

Theorem 1 ($O(n^{\delta})$-approx for DkS, [FPK01]). *$\exists$ a $poly(n)$ time algorithm A that finds a subgraph $G^{'}$ of size k with average degree $\geq \frac{d^*(G,k)}{2n^{\frac{1}{3}}}$, where $d^*(G,k)$ is the true max. average degree of a k-vertex subgraph in G.*

However, its approximation ratio is $O(n^{\delta})$ for $\delta < \frac{1}{3}$ [FPK01]. It is meaningful to ask whether the approximation ratio given above can be improved upon, but [Kho06] show that **DkS** is NP-Hard to approximate up to a constant factor under a reasonable complexity theoretic assumption about the class NP. As a result, since we cannot solve the problem of detecting dense subgraphs (of fixed size) up to some constant approximation in $poly(n)$ time, we use the structure of our given social community.
We work under 2 models:

1. For Algorithm 6, we assume that our community is a sparsely connected set of disjoint, bipartite dense graphs.
2. For Algorithm 7, we do not assume our community is bipartite. Instead, we assume access to data about the community structure, such as the population (sizes) of the communities.

For Algorithm 6, we use the $DenseSubraph()$ algorithm defined in [GKT05], which enumerates as many disjoint, dense bipartite graphs as possible.
It is well known that when k is not fixed, there exist $poly(n)$ times algorithms that return the densest subgraph (which we call $ExtractComponent()$) [Gol84]. Specifically, with access to population data, we can run this algorithm, identify which community is identified, remove it from the graph, and re-run the algorithm till we get an empty set of vertices. Notice that the number of communities can be no more than $O(n)$ in our structure (since the communities must be disjoint). Therefore, this approach allows us to extract all the dense communities in $poly(n)$ time. This approach is used in Algorithm 7.
The reason dense bipartite communities are handled separately from dense communities (in general) is that the $DenseSubgraph()$ algorithm proposed by [GKT05] uses (c, s)-Shingles and can handle very large graphs (on the order of millions of nodes) tractably. In contrast, we know of no standardized practical implementation of the algorithms proposed in [Gol84].

3.3 Discovering Dense Subgraphs in Massive Graphs

To extract dense bipartite components from our graph, which we assume to be sparse overall, we use the **DenseSubgraph**($v, \Gamma(v)$) routine given in [GKT05]. We provide a brief but formal explanation of how this routine works, along with pseudocode for the its main call.

The algorithm has two main components, where the first one is a 2-step "shingling" approach. The first component of the 2-step shingling approach is to compute a (c, s)-Shingle for a given 1-neighborhood $\Gamma(v)$ of every vertex $v \in V$. Therefore, to every node $v \in V$, c shingles are attributed. This essentially computes c s-subsets of $\Gamma(v)$, each of which is represented with equal probability over the total number of s-subsets. Additionally, a list of nodes is associated with every shingle s - This marks the dense, common subsets of vertices to which other vertices flow into. The second component of the shingling is to find the set of vertices that share a sufficiently large number of shingles in common. This means another (c, s)-Shingle s' is created for every set of nodes $\{v\}$ that are common to some shingle s (which was created on the neighborhood of a vertex $u \in V$).

Having found sets of nodes that share a large number of destinations in common, the second component of the algorithm performs a "clustering". Two shingles s_1, s_2 computed by the first step of the shingling are related iff they share a shingle s' created by the second step. This leads to an equivalence class of shingles. Finding dense subgraphs is now a simple application of finding connected components in the graph representing the equivalence relation on the aforementioned shingles. This is achieved by a simple use of the classical union-find algorithm [HU73].

A sketch of this Algorithm is given by the pseudocode below:

Algorithm 3 DenseSubgraph()

$c_1, s_1, c_2, s_2 \longleftarrow$ tuned values of c_1, s_1, c_2, s_2
$i = 0$
$CC = \phi$
repeat
  $C = \text{Shingle}([v_i, \Gamma(v_i)], c_1, s_1)$
  $CC \leftarrow CC \cup C$
until $i = |V| - 1$
$\mathcal{S} = \text{Shingle}(CC[], c_2, s_2)$
$\mathcal{C} = \text{ComputeCluster}(\mathcal{S})$
$j = 0$
$\mathcal{D} \leftarrow \phi$
repeat
  $C = \mathcal{C}[j]$
  $\mathcal{D} = \mathcal{D} \bigcup_{c \in C} \Gamma(c)$
until $j = |V| - 1$
return $\mathcal{D}$

3.4 Graph Exploration Models

Some of the earliest work in this space concerned finding the shortest path which touches a desired set of terminals in a weighted directed graph when the structure and weights (costs) are known. Prim's algorithm from 1957 [Pri57] is such an example. Prim's algorithm sought to find a minimum spanning tree (MST), and indeed is guaranteed to do so in connected graphs. The algorithm proceeds by simply adding the nearest non-visited node to the current tree until all nodes are added.

Three defining factors of our model are 1) the graph structure is not known *a priori*, 2) the costs come from falsely believing a non-creditworthy node to be creditworthy, and in that sense occur with some random probability vs. the fixed costs of traversing a weighted graph, and 3) one recommender can create several new recommenders, each of which can create several of its own new recommenders, making the spreading model a bit like a particle filter.

There are numerous algorithms for exploring unknown graphs, including depth-first search (DFS) and breadth-first search (BFS) algorithms. As there is a time lag required to verify each new vertex's credibility, BFS makes more sense for our setting. Since at each step we receive the recommenders' recommendations about neighboring nodes, we have enough information to execute Prim's minimum spanning tree algorithm, which is guaranteed to find a minimum spanning tree. This approach is captured by algorithm 4 in Section 4.

Another algorithm we mention is the SIR model which assesses the spread of disease through a population [SM04]. This model uses differential equations based on the rate of change of the proportion of the population which is susceptible (S), infected (I), and recovered (R). A key parameter is b, the number of people with which each infected person comes into contact each day. Analogously for us, this could be thought of as the average degree of each vertex $|v|$. Another important parameter is k, the percentage of the infected population which recovers each day. For us, k would assume a fixed value of 1, since the each recommender gives his or her recommendations and then drops out of the system. The basic differential equations from Smith et al. include change in infected population

$$\frac{di}{dt} = b * s(t) * i(t) - k * i(t) \tag{7}$$

and change in susceptible population

$$\frac{ds}{dt} = -b * s(t) * i(t) \tag{8}$$

We could consider susceptible, infected, and recovered population to correspond to our unknown population, recommenders, and known populations. However, this assumes that people move about freely and don't exist in a static network, which does not apply to our situation (see [**?**]).

There is a considerable body of recent work on minimal-cost graph traversal, including traversing all vertices and edges, but for our purposes, we would prefer only to visit all creditworthy nodes, and visit as few edges as possible.

3.5 Bayesian Trust - Integrating Behavioral Trust in Web Service Compositions

In contrast to our assumption of binary creditworthiness, Paradesi et. al in *Integrating Behavioral Trust in Web Service Compositions* [PDS09] assume that each web service has a continuous probability $p_i \in [0, 1]$ of generating a positive experience for users who interact with it. Furthermore, an observer has imperfect knowledge of this p_i, so this imperfect knowledge is modeled as a beta density function which is updated using Bayesian updating as new experiences are observed. in this way, intermediate beliefs about the p_i and encapsulated. The Bayesian belief probability is written as:

$$B(p_i; a, b) = \frac{1}{\beta} p_i^{\alpha-1}(1 - p_i)^{b-1} \tag{9}$$

Where a and b represent one plus the number of positive and negative experiences, respectively.

4 Our Formulation

4.1 Recommender-guided graph discovery

Our first graph-discovery algorithm is similar to traditional graph exploration models. The lender lends to any borrower who is positively recommended by one or more recommenders. If even one recommendation is negative, the bank will not lend to that borrower. Upon lending, the bank determines which borrowers are CW and which are not, and the CW borrowers then give recommendations for all of their neighbors. The algorithm terminates at the round when no new lendable recommendations are given.

Algorithm 4 Recommender-Guided Graph Discovery

1: $R \leftarrow rand(V, p_{init})$ (randomly choose p_{init} fraction of graph members as recommenders)
2: $cost = 0$
3: $B_{discard} = \emptyset$ (initialize set of borrowers which the bank will not consider)
4: $B_{keep} = \emptyset$ (initialize set of good borrowers)
5: **repeat**
6: $\quad cost = cost - |B_{keep}|$ (Each good borrower reduces cost by 1 each cycle due to profitable lending)
7: $\quad B = neighbors(R) \backslash B_{discard}$
8: $\quad R = \emptyset$

Algorithm 5 Recommender-Guided Graph Discovery

for $b_i \in B$ **do**
  if $b_i = CW$ **then**
    $prob\_rec = Pr(rec_{ij} = CW | b_j = CW)$
  else
    $prob\_rec = Pr(rec_{ij} = CW | b_j = \overline{CW})$
  end if
  $rand = uniform([0, 1])$
  if $rand < prob\_rec$ **then** (Bank lends to this borrower)
    if $b_i = CW$ **then**
      $R \leftarrow b_i$
      $B_{keep} \leftarrow b_i$
      $cost- = 1$ (bank earns small profit from finding profitable borrower)
    else
      $B_{discard} \leftarrow b_i$
      $cost+ = 5$ (Significant cost to bank for making a loan which is not repaid)
    end if
  else
    $B_{discard} \leftarrow b_i$ (Bank never lends to this borrower)
  end if
end for
until $R = \emptyset$

4.2 EigenTrust

In this formulation, recommenders give weighted ratings for each of their neighbors as in section 2. The algorithm is run until convergence (as in Algorithms 1 and 2 in the EigenTrust subsection), and the bank lends to borrowers whose global trust scores are above a certain threshold. The main measure in both the algorithms here is that they compute eigentrust for each community, ignoring the connections across each community. Since the original data comes in the form of one graph, algorithms are first applied to extract communities. Following that, eigentrust is run on each community. The credit worthy borrowers from each local community are identified and then put together to form the final list. Outside of comparing the difference in the set of credit worthy borrowers that local versions of eigentrust computes (with respect to a global eigentrust run), both the following algorithms can be parallelized when computing eigentrust to increase computational efficiency.

Algorithm 6 Localized EigenTrust for Dense Bipartite Communities

$c, s \longleftarrow$ tuned values of c, s
$i = 0$
$CC = \phi$
repeat
  $C =$ DenseSubgraph$(v_i, \Gamma(v_i))$
  $CC \leftarrow CC \cup C$
until $i = |V|$
$sc[] =$ [Eigentrust_Base_Algorithm$(CC[1])$, .., Eigentrust_Base_Algorithm$(CC[n])$]
$cs = \bigcup sc[]$

Algorithm 7 Localized EigenTrust for Dense Commmunities

$CC \leftarrow \phi$
$G^{'} = G$
repeat
  $C = ExtractComponent(G^{'})$
  $G^{'} = RemoveFromGraph(G^{'}, C)$
  if $C \in$ Valid Community **then**
    $CC \leftarrow CC \cup C$
  else
    Continue
  end if
until $G^{'} = \phi$
$sc[] = [\text{Eigentrust\_Base\_Algorithm}(CC[1]), .., \text{Eigentrust\_Base\_Algorithm}(CC[n])]$
$cs = \bigcup sc[]$

4.3 Analytical / Theoretical Questions

1. *How will the local EigenTrust results differ from the global ones?*

   We discuss our empirical findings to this question in Section 5.3. However, an analytical answer to this question would be helpful. Given a network of disjoint dense graphs that are sparsely connected, the eigenvalues locally will be quite different from the global ones. This is clear since the subgraphs graphs are dense and will presumably come close to being cliques (which have rank-1 in the adjacency matrix). However, with weights, the rank of the matrix even for dense graphs may be bounded away from 1. As such, it is unclear exactly how to bound the gap between the global eigenvalue and the maximum eigenvalue of a densely connected subgraph (or community). The closest result we could find in the literature is by [PMDC14], which essentially gives a bound on the relationship between the maximal eigenvalue of the graph and the approximation factor of locating a k-dense subgraph in it.

2. *What are the expected false positive rate and cost, given an underlying rate of CW borrowers,* $Pr(rec_{ij} = CW|b_i = CW)$*, and* $Pr(rec_{ij} = CW|b_i = \overline{CW})$*?*

   By the laws of probability, the false positive rate is

   $$fp = Pr(rec_{ij} = CW|b_i = \overline{CW}) * (1 - p_{CW}) \tag{10}$$

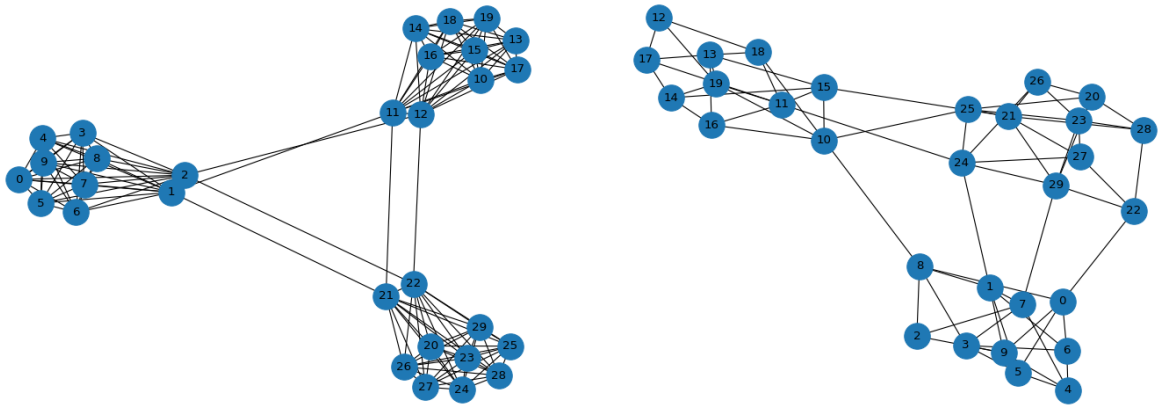
   So the false positive rate will increase as the conditional probability increases or p_{CW} decreases.

   As defined in the psuedocode for Algorithm 4, the cost for a failed loan is 5 while the cost for a successful loan is -1 (roughly corresponding to a 20% interest payment in the successful case and a 100% loss in the unsuccessful case). Thus, our expected cost becomes

   $$\frac{\mathbb{E}[cost]}{n * pct\_discovered} = 5(Pr(rec_{ij} = CW|b_i = \overline{CW})(1-p_{CW}))-Pr(rec_{ij} = CW|b_i = CW)\cdot p_{CW} \tag{11}$$

   In the case of EigenTrust, this will be determined partially be the threshold global trust score we set as the criteria to make loans. We discuss this further in Section 5.

3. *What percent of creditworthy borrowers would we expect to discover with Algorithm 4?*



(a) Complete graph

(b) Bipartite Graph

Figure 1: We used the above two graphs in our experiments. Graph (a) is comprised of three complete graphs of 10 nodes each, and six connecting edges that are manually added between the subgraphs. Graph (b) is similarly comprised of three bipartite graphs (p=0.9) of 10 nodes each, then we add three connecting edges between subgraphs 2 and 3 as well as 3 and 1, but only one connecting edge between 1 and 2 (to keep things interesting).

In the EigenTrust cases, all borrowers will have a trust score, so the question becomes how we set our threshold trust score to make loans, and what percentage of good borrowers fall below that.

In the Recommender-guided graph discovery case, we can think of the recommenders at each step as a set of particles which may produce more particles, or which may be extinguished.

4. *What is the expected time to convergence given the above parameters, plus number of recommendations k allowed to be made by each recommender, and average degree $|v|$ of each recommender?*

   We do not know an explicit expression for this, but the number of simulations needed to converge on average are discussed in Section 5.3.

5 Simulation

5.1 Setup

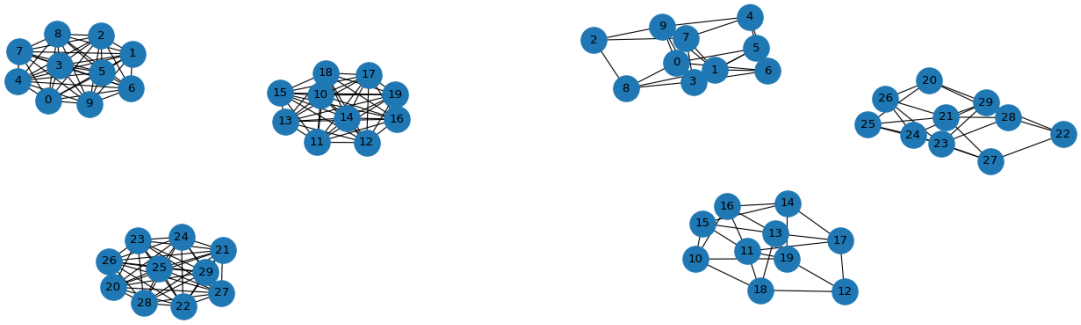
We simulated three of our algorithms from section 4 to observe behavior and motivate further theory work. We simulated 2 graphs, each with 30 nodes, including:

1. Three fully-connected subgraphs with sparse connections between them
2. Three bipartite subgraphs with sparse connections between them

Visual representations or our graphs can be seen in Fig. 1.

For each of these two graphs, we ran three algorithms to discover creditworthy borrowers:

1. Graph Discovery as in algorithm 4
2. Global EigenTrust as laid out in [KSGM03]. Note that we used a value of $\epsilon = .001$ and that we lent only to borrowers with a global trust score of $t_i > .03$.
3. EigenTrust run separately on each of the dense subgraphs, with recommended borrowers being the



(a) Complete Subgraphs

(b) Bipartite Subgraphs

Figure 2: The subgraphs used in our experiments.

union of the recommended borrowers selected from each subgraph. We assume that algorithms such as *DenseSubgraphs()* or *ExtractComponent()* could be used to identify these subgraphs before EigenTrust is run, though we did not run them explicitly. Note that as with Global EigenTrust, we used a value of $\epsilon = .001$ and that we lent only to borrowers with a global trust score of $t_i > .03$. Visual representations of these subgraphs can be found in 2.

In addition, we ran simulations with every possible combination of the following parameters:

1. Underlying rate of creditworthy borrowers, p_{CW}, with values of 0.5, 0.8, and 0.95
2. Conditional probabilities of recommenders giving the correct response. We simulated with two pairs of values, $Pr(rec_{ij} = CW|b_j = CW) = .95, Pr(rec_{ij} = CW|b_j = \overline{CW}) = .05$ and $Pr(rec_{ij} = CW|b_j = CW) = .8, Pr(rec_{ij} = CW|b_j = \overline{CW}) = .2$
3. Proportion of population selected as initial recommenders (selected randomly from CW members of population): $p_{init} \in \{.1, .2\}$

This gives 72 different simulation settings. We ran each one ten times and reported the average values of these ten repetitions.

Performance indicators we measured include:

1. Cost
2. Number of rounds to convergence
3. False-positive rate
4. Percentage of CW borrowers discovered

We also produced explanatory graphs for the discovery algorithm, see Fig. 3

Our code is available online: https://github.com/nouyang/cs223_creditgraphs.

5.2 Results

The full results are rather extensive, and we report them in the Appendix [7]. Here we report on several interesting results arising from the simulation.

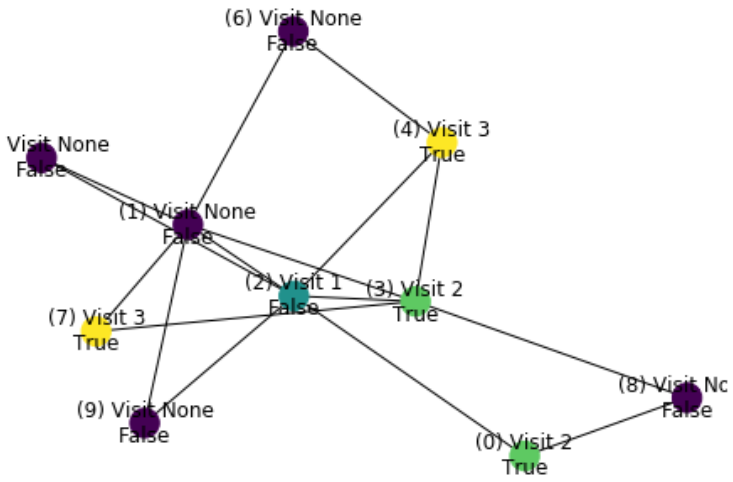


Figure 3: An explanatory example of the Discovery Algorithm. The node number is indicated in parenthesis, followed by the steps taken to reach the node, and the label below shows whether the node is creditworthy or not. The number of steps is also shown in the color of the nodes. Here, we start at node 2, which recommends nodes 0 and 3 to visit next.

5.2.1 False Positive Rate

We found that the false positive rate for the Graph Discovery algorithm closely followed the theoretical levels which ranged from .01 to .1. The false positive rate was consistently lower for Global EigenTrust, and somewhat erratic for Local EigenTrust.

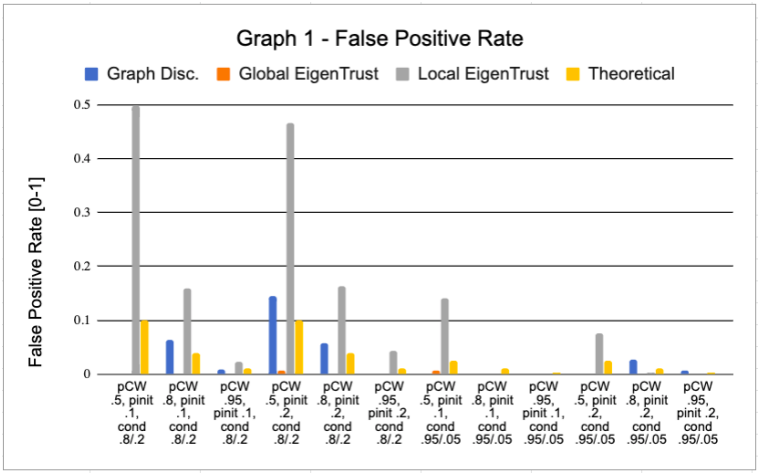


Figure 4: False positive rate for our three algorithms vs. theoretical graph discovery rate for complete graph.

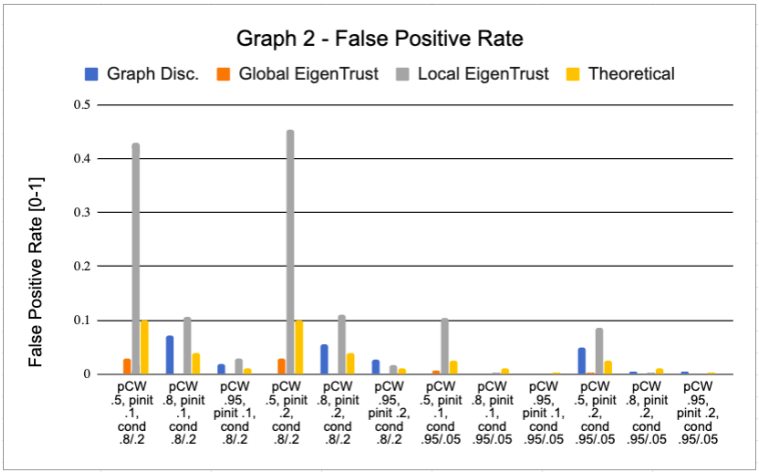


Figure 5: False positive rate for our three algorithms vs. theoretical graph discovery rate for bipartite graph.

5.2.2 Rounds to Convergence

The number of rounds to convergence represents the number of lending cycles in the Graph Discovery algorithm, and the number of times the normalized ratings matrix had to be multiplied by itself in the Global and Local EigenTrust algorithms. The values were moderate for Graph Discovery, high for Global EigenTrust, and very low for Local EigenTrust.

Algorithm	Connected Graph	Bipartite Graph
Graph Discovery	2.65	4.08
Global EigenTrust	83.48	467.61
Local EigenTrust	1.00	1.01

Table 1: table caption

5.2.3 Percent of Creditworthy Borrowers Discovered

The Graph Discovery algorithm discovered between 50% and 95% of creditworthy borrowers, with a positive correlation between the underlying percentage of creditworthy borrowers and the percent discovered. Global EigenTrust is much better, and Local EigenTrust discovers 100% of creditworthy borrowers nearly every time.

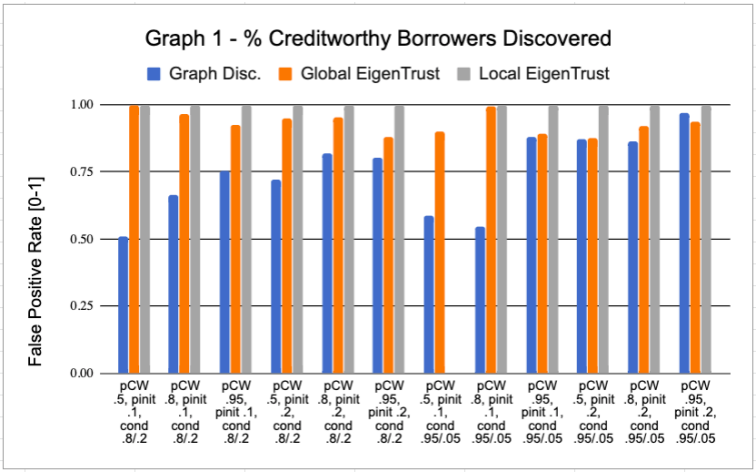


Figure 6: Percent of creditworthy borrowers discovered for our three algorithms for complete graph.

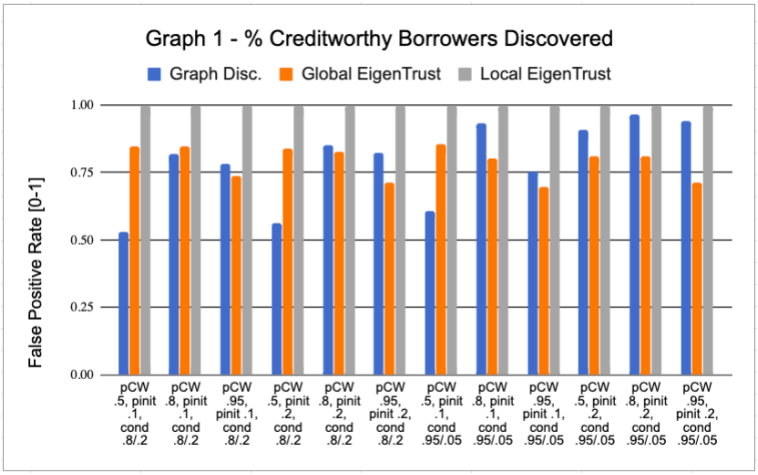


Figure 7: Percent of creditworthy borrowers discovered for our three algorithms for bipartite graph.

5.3 Discussion

Graph Discovery and Global EigenTrust were fairly robust to false positives, while Local EigenTrust had high false positives, especially in the low overall creditworthiness scenarios. For Graph Discovery, this is almost entirely dictated by recommender skill, while in the EigenTrust cases, this is a calibration of the threshold global trust score for lending. The tradeoff that the system designer can make is between lending more liberally and detecting all creditworthy borrowers versus being more conservative and lowering false positives.

The runtime is much better for Graph Discovery and Local EigenTrust than Global EigenTrust. This is due to having multiple "particles" in the case of Graph Discovery and the small subgraph size in the Local EigenTrust case. In the real world, it is not clear that this would matter, and we also need to investigate how these runtimes would scale with increasing graph size.

6 Conclusion and Next Steps

Suggested next steps would include algorithmic and/or analytical results that give precise bounds for questions 1 and 4 in Section 4.3. As mentioned in [PDS09], it would be interesting to run simulations where we consider borrowers on a non-binary (continuous, normalized) scale in the social network and see how results change for all 3 algorithms.

Further work could include tuning our algorithms, which are sensitive to the initial percentage of creditworthy borrowers and how many of those are randomly chosen as seeds by the bank, to become more robust and predictable in behavior under these parameters. In fact, a precise bound on question 2 in Section 4.3 could be obtained by answering the previous question.

References

[Cab19] Lídia Cabral. Tractors in africa: Looking behind the technical fix. 2019.

[FPK01] Uriel Feige, David Peleg, and Guy Kortsarz. The dense k-subgraph problem. *Algorithmica*, 29(3):410–421, 2001.

[GKT05] David Gibson, Ravi Kumar, and Andrew Tomkins. Discovering large dense subgraphs in massive graphs. In *Proceedings of the 31st international conference on Very large data bases*, pages 721–732. VLDB Endowment, 2005.

[Gol84] Andrew V Goldberg. *Finding a maximum density subgraph*. University of California Berkeley, CA, 1984.

[HRH18] Reshmaan Hussam HBS, Natalia Rigol, and Benjamin Roth HBS. Targeting high ability entrepreneurs using community information: Mechanism design in the field. 2018.

[HU73] John E. Hopcroft and Jeffrey D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973.

[Kho06] Subhash Khot. Ruling out ptas for graph min-bisection, dense k-subgraph, and bipartite clique. *SIAM Journal on Computing*, 36(4):1025–1071, 2006.

[KSGM03] Sepandar D Kamvar, Mario T Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *Proceedings of the 12th international conference on World Wide Web*, pages 640–651. ACM, 2003.

[MMM+13] Pushkar Maitra, Sandip Mitra, Dilip Mookherjee, Alberto Motta, Sujata Visaria, et al. *Agent Intermediated Lending: A New Approach to Microfinance*. Monash Univ., Department of Economics, 2013.

[NSC+08] Amit Anil Nanavati, Rahul Singh, Dipanjan Chakraborty, Koustuv Dasgupta, Sougata Mukherjea, Gautam Das, Siva Gurumurthy, and Anupam Joshi. Analyzing the structure and evolution of massive telecom graphs. *IEEE Transactions on Knowledge and Data Engineering*, 20(5):703–718, 2008.

[PDS09] Sharon Paradesi, Prashant Doshi, and Sonu Swaika. Integrating behavioral trust in web service compositions. In *2009 IEEE International Conference on Web Services*, pages 453–460. IEEE, 2009.

[PMDC14] Dimitris Papailiopoulos, Ioannis Mitliagkas, Alexandros Dimakis, and Constantine Caramanis. Finding dense subgraphs via low-rank bilinear optimization. In *International Conference on Machine Learning*, pages 1890–1898, 2014.

[Pri57] Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[PS16] David C Parkes and Sven Seuken. Economics and computation. *Book in preparation*, 2016.

[SM04] David Smith and Lang Moore. The sir model for spread of disease-the differential equation model. *Journal of Online Mathematics and its Applications*, 2004.

7 Appendix

Attached below are Excel screenshots that explain all parameters that were measured while running all 3 algorithms we tested. More specifically, each column corresponds to a metric we measured on each of the 2 graphs.

The following are abbreviations:

1. **pCW**: Reflects the percentage of total number of users in the network that are creditworthy.
2. **pinit**: Reflects the percentage of users that were initialized as the number of credit worthy lenders.
3. **ALG1**: Refers to Algorithm 4.
4. **ALG2**: Refers to Algorithm 6.
5. **ALG3**: Refers to Algorithm 7.

Graph 1 Results					
RESULTS TIME 20:44					
ALG1, COMPLETE	**COST**	**False positive**	**Pct Creditwort**	**Number of Rounds to Convergence**	
pCW .5, pinit .1, cond .8/.2	-12.8000	nan	0.5085	2.4	
pCW .8, pinit .1, cond .8/.2	-37.1000	0.0628	0.6648	2.9	
pCW .95, pinit .1, cond .8/.2	-56.9000	0.0091	0.7552	3.2	
pCW .5, pinit .2, cond .8/.2	-14.4000	0.1451	0.7213	2.4	
pCW .8, pinit .2, cond .8/.2	-39.6000	0.0584	0.8196	2.7	
pCW .95, pinit .2, cond .8/.2	-51.8000	0.0000	0.8021	2.3	
pCW .5, pinit .1, cond .95/.05	-25.9000	nan	0.5863	2.8	
pCW .8, pinit .1, cond .95/.05	-33.3000	nan	0.5480	2.1	
pCW .95, pinit .1, cond .95/.05	-73.2000	nan	0.8795	3.3	
pCW .5, pinit .2, cond .95/.05	-28.0000	0.0254	0.8731	2.5	
pCW .8, pinit .2, cond .95/.05	-52.1000	0.0264	0.8620	2.8	
pCW .95, pinit .2, cond .95/.05	-58.8000	0.0072	0.9683	2.4	
RESULTS TIME 21:03					
ALG2, COMPLETE, THRESH 0.03, gamma 0.001	**COST**	**False positive**	**Pct Creditwort**	**Number of Rounds to Convergence**	
pCW .5, pinit .1, cond .8/.2	-30.0000	0.0000	1.0000	70.5	
pCW .8, pinit .1, cond .8/.2	-29.1000	0.0000	0.9637	52.9	
pCW .95, pinit .1, cond .8/.2	-27.9000	0.0000	0.9259	39.9	
pCW .5, pinit .2, cond .8/.2	-29.0667	0.0067	0.9484	99.6	
pCW .8, pinit .2, cond .8/.2	-28.9000	0.0000	0.9535	59.3	
pCW .95, pinit .2, cond .8/.2	-26.6000	0.0000	0.8820	41.3	
pCW .5, pinit .1, cond .95/.05	-28.4667	0.0067	0.9009	290.1	
pCW .8, pinit .1, cond .95/.05	-29.8000	0.0000	0.9925	51.4	
pCW .95, pinit .1, cond .95/.05	-26.9000	0.0000	0.8923	42.6	
pCW .5, pinit .2, cond .95/.05	-28.1000	0.0000	0.8750	144.4	
pCW .8, pinit .2, cond .95/.05	-28.1000	0.0000	0.9210	72.2	
pCW .95, pinit .2, cond .95/.05	-28.2000	0.0000	0.9355	37.6	
TIME 21:31, EIGEN, ALG3 SUBGRAPH (10 node	**COST**	**False positive**	**Pct Creditwort**	**Number of Rounds to Convergence**	
pCW .5, pinit .1, cond .8/.2	-7.5	0.5000	1	1	
pCW .8, pinit .1, cond .8/.2	-22.8	0.1600	1	1	
pCW .95, pinit .1, cond .8/.2	-28.95	0.0233	1	1	
pCW .5, pinit .2, cond .8/.2	-9	0.4667	1	1	
pCW .8, pinit .2, cond .8/.2	-22.65	0.1633	1	1	
pCW .95, pinit .2, cond .8/.2	-28.05	0.0433	1	1	
pCW .5, pinit .1, cond .95/.05	-23.7	0.1400	nan	1	
pCW .8, pinit .1, cond .95/.05	-30	0.0000	1	1	
pCW .95, pinit .1, cond .95/.05	-30	0.0000	1	1	
pCW .5, pinit .2, cond .95/.05	-26.55	0.0767	1	1	
pCW .8, pinit .2, cond .95/.05	-29.85	0.0033	1	1	
pCW .95, pinit .2, cond .95/.05	-30	0.0000	1	1	

Figure 8: Complete data for Algorithms 1, 2 & 3 on Graph 1

Graph 2 Results					
RESULTS: TIME 20:43					
ALG1, BIPARTITE					
	COST	**False positive**	**Pct Creditwort**	**Number of Rounds to Convergence**	
pCW .5, pinit .1, cond .8/.2	-23.6000	nan	0.5296	3.8	
pCW .8, pinit .1, cond .8/.2	-73.2000	0.0712	0.8206	5.4	
pCW .95, pinit .1, cond .8/.2	-91.4000	0.0185	0.7837	4.9	
pCW .5, pinit .2, cond .8/.2	-14.3000	nan	0.5624	2.9	
pCW .8, pinit .2, cond .8/.2	-59.7000	0.0562	0.8512	3.7	
pCW .95, pinit .2, cond .8/.2	-74.1000	0.0268	0.8250	3.7	
pCW .5, pinit .1, cond .95/.05	-25.7000	nan	0.6083	3.1	
pCW .8, pinit .1, cond .95/.05	-88.1000	0.0000	0.9330	5	
pCW .95, pinit .1, cond .95/.05	-80.3000	nan	0.7547	3.8	
pCW .5, pinit .2, cond .95/.05	-57.2000	0.0485	0.9083	5.1	
pCW .8, pinit .2, cond .95/.05	-78.8000	0.0048	0.9675	3.9	
pCW .95, pinit .2, cond .95/.05	-86.6000	0.0037	0.9400	3.6	
RESULTS: TIME 21:01					
ALG2, BIPARTITE, THRESH 0.03, gamma 0.001	**COST**	**False positive**	**Pct Creditwort**	**Number of Rounds to Convergence**	
pCW .5, pinit .1, cond .8/.2	-26.9500	0.0300	0.8477	251.8	
pCW .8, pinit .1, cond .8/.2	-26.4000	0.0000	0.8467	156.8	
pCW .95, pinit .1, cond .8/.2	-22.5000	0.0000	0.7370	117.9	
pCW .5, pinit .2, cond .8/.2	-26.7500	0.0300	0.8401	311.3	
pCW .8, pinit .2, cond .8/.2	-25.7000	0.0000	0.8264	145	
pCW .95, pinit .2, cond .8/.2	-21.8000	0.0000	0.7121	113	
pCW .5, pinit .1, cond .95/.05	-27.5667	0.0067	0.8562	628.3	
pCW .8, pinit .1, cond .95/.05	-25.2000	0.0000	0.8033	228.7	
pCW .95, pinit .1, cond .95/.05	-21.2000	0.0000	0.6969	103.4	
pCW .5, pinit .2, cond .95/.05	-27.0833	0.0033	0.8110	2594.1	
pCW .8, pinit .2, cond .95/.05	-25.5000	0.0000	0.8099	838.5	
pCW .95, pinit .2, cond .95/.05	-21.8000	0.0000	0.7120	122.5	
TIME 21:32, EIGEN, ALG3 BIPARTITE	**COST**	**False positive**	**Pct Creditwort**	**Number of Rounds to Convergence**	
pCW .5, pinit .1, cond .8/.2	-10.65	0.4300	1	1	
pCW .8, pinit .1, cond .8/.2	-25.2	0.1067	1	1	
pCW .95, pinit .1, cond .8/.2	-28.65	0.0300	1	1	
pCW .5, pinit .2, cond .8/.2	-9.6	0.4533	1	1	
pCW .8, pinit .2, cond .8/.2	-25.05	0.1100	1	1	
pCW .95, pinit .2, cond .8/.2	-29.25	0.0167	1	1	
pCW .5, pinit .1, cond .95/.05	-25.35	0.1033	1	1	
pCW .8, pinit .1, cond .95/.05	-29.85	0.0033	1	1	
pCW .95, pinit .1, cond .95/.05	-30	0.0000	1	1	
pCW .5, pinit .2, cond .95/.05	-26.1	0.0867	1	1.1	
pCW .8, pinit .2, cond .95/.05	-29.85	0.0033	1	1	
pCW .95, pinit .2, cond .95/.05	-30	0.0000	1	1	

Figure 9: Complete data for Algorithms 1, 2 & 3 on Graph 2