Graduate Admissions at MIT & Comparison-based Rank Aggregation: A Case Study

by

Katherine Carbognin Szeto

B.Sc. EECS, Massachusetts Institute of Technology, 2012

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2013

[June 2013]

Author . . . .

Department of Electrical Engineering and Computer Science

May 24, 2013

Certified by . . . [illegible] . . . .

Dr. Devavrat Shah

Jamieson Associate Professor of Electrical Engineering, Laboratory for Information and Decision Systems

Thesis Supervisor

Certified by . . [illegible] . . . .

Dr. Gregory Wornell

Sumitomo Professor of Electrical Engineering, Research Laboratory of Electronics

Thesis Supervisor

Certified by . [illegible] . . . .

Dr. Frans Kaashoek

Charles Piper Professor of Computer Science, Computer Science and Artificial Intelligence Laboratory

Thesis Supervisor

Accepted by . . . [illegible] . . . .

Prof. Dennis M. Freeman

Chairman, Masters of Engineering Thesis Committee

Admission to MIT & Comparison-based Rank Aggregation: A Case Study

by
Katherine C. Szeto

Submitted to the Department of Electrical Engineering and Computer Science

May 24, 2013

In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Admission to the Graduate program in EECS at MIT is done via an all-electronic system. Applicants submit materials through a web interface and faculty "reviewers" read the applications and make notes via a different interface. Among other comments, reviewers provide a numerical score between 1 and 4 to each application. Admissions decisions are reached via a lengthy process involving discussion among faculty, which is often guided by the numerical scores of the applications. Past data show the scores of the applicants still under consideration during the final stages of the process are almost all 4's. Because of this uniformity, the scores do not provide much resolution into the differences in quality of the applications, and are therefore not as helpful as they could be in guiding admissions decisions.

In this work, we present the use of an additional scoring system that is based on pairwise comparisons between applicants made by faculty reviewers. We present the design we created for this scheme and the code written to implement it, and analyze the data we obtained when the code went live during the 2012-2013 admissions cycle.

Contents

1 Introduction

In many scenarios (like graduate school admissions), it is necessary to form a rank, or ordering, of a collection of items (in our case, applicants). However, it is necessary to do so based on the opinions of multiple people (i.e., the admissions committee). Say each member has formed a rank of the applicants according to his expertise. How, then, do you combine these ranks into a single rank that represents the group's collective opinion so that the committee can decide whom to admit? This illustration of collective decision-making is a specific application of *rank aggregation*, the problem of taking many individual ranks and using them to produce a single, global rank.

Rank aggregation is an old problem - indeed, we will be citing the work of Condorcet from 1785. Nevertheless, despite its age, it remains a relevant and challenging problem today. It is a relevant problem because of its wide range of modern applications. These applications include, but are not restricted to: election systems, recommendation systems such as Netflix, admissions systems (which, of course, is the central focus of this work), paper rating tools for conference management software like HotCRP [Kohler: 2007], and even gaming systems like Microsoft's TrueSkill©. It is a challenging problem because there are both fundamental philosophical difficulties (arising from social choice theory results) and computational difficulties (arising from the fact that most modern applications require working with large datasets).

For this thesis project, we took a novel rank aggregation algorithm and applied it in the context of graduate admissions at MIT in the Electrical Engineering and Computer Science (EECS) department. In particular, we augmented the preexisting online system so that it takes individual preferences about applicants from professors and aggregates these preferences into a global rank by using comparative, or ordinal, information. Our vision in doing so was threefold. First, we wanted to see how such a new algorithm (as yet unimplemented in a real system) would fair in practice. Second, we wanted to provide another example of a successful ordinal-information based ranking system. As will be described in the following sections, most ranking systems in use today use cardinal ranking - although there are some exceptions like HotCRP [Kohler: 2007]. Nevertheless, many (including myself and my research group) believe that ordinal information leads to better ranks. Lastly, and most importantly, we wanted to improve the graduate admissions system by providing additional functionality that hopefully streamlines the decision-making process for admissions committee members at MIT.

We begin by providing the reader with the necessary background regarding admissions at MIT, as well as the background for understanding the philosophical motivations for the project described above. Next, we present the design for the modules we added to the system, followed by a description of relevant implementation detail. We conclude by evaluating the results we obtained when our additions to the system went live during the January 2013 admissions decision process and by discussing directions for future work.

2 Background

Each year, MIT's Graduate Program in Electrical Engineering and Computer Science receives upwards of 2,000 applications. The admissions process is managed through an all-electronic system called *gradapply* that was developed by professors Frans Kaashoek and Robert Morris of MIT, starting in the summer of 2009. In this section, we discuss how this electronic system works and what issues therein motivated my thesis project.

EECS applicants must specify a particular subarea to which they are applying, such as Artificial Intelligence, Theoretical Computer Science, or Circuits, to name a few. Once the submission deadline passes, the chairs of said areas perform a preliminary scan of the applications, or *folders*, in their area. They then assign each folder to a handful of reviewers whom they believe are particularly qualified to read that application. This assignment is done electronically through an interface provided by the website.

On average, each folder is assigned somewhere between two and five reviewers. Reviewers are typically faculty on the admissions committee, but can also include other specialists that were given accounts on the website by an area chair. It is the reviewer's job to carefully read each folder to which he is assigned in order to assess the applicant's potential fit at MIT. The reviewers keep notes about each application like:

- Which classes could the applicant TA?

- Who would be potential advisors for the applicant?
- Does the applicant have work experience?
- Does the applicant have teaching experience?
- Does the applicant have research experience?
- Please write a short review of the application.

Importantly, for the purposes of this work, reviewers also provide each application with a numerical score of 1, 2, 3, or 4. Later, we will introduce these scores as having to do with "implicit pairwise information," but for now, we will refer to them as 1,2,3,4-scores. Higher numbers are better than lower numbers. In particular, in EECS, these numbers are interpreted as:

- 1: Reject (bottom%80)
- 2: Possible accept (top %20)
- 3: Likely accept (top %10)
- 4: Must accept (top %5)

An issue we will bring up again later is that these scores are quite coarse-grained. Toward the end of the review period, the admissions faculty gather together in person to jointly decide whom to admit. Typically, the CS and EE faculty have separate meetings. Given the limited positions available and the large number of highly qualified applicants, this becomes the most difficult stage of the admissions process.

Through a UI provided by the website, it is possible to display a table of all the students still under consideration at this final stage in the admissions process. Each row contains information like the student's name and school, but also shows the 1,2,3,4-score he received by each reviewer. If you click on the student's name, which is a hyperlink, you can read the reviews each reviewer gave the student. This table is implemented in such a way that you can sort the students in it according to some function of the 1,2,3,4-scores they received, say the average 1,2,3,4-score.

After lengthy debate, which is usually guided in part by the information in the table described above, the faculty collectively decide whom to admit.

2.1 Application-specific Project Motivation

Past data reveals that students who score 1's or 2's are almost certainly rejected. Usually, only students with 4's and occasionally 3's are considered for admission. This breakdown reveals that the 1,2,3,4-scores are useful for dividing students into coarse "buckets" at the beginning of the process, but that they do no provide much information during the final stages of the admissions process - applicants left in the running at that point almost all have 4's. In other words, these scores provide no resolution into the differences in the quality of the applications during the final stages of the admissions process.

A central goal of this project, therefore, is to provide a way for faculty members to enter in more fine-grained pereferences for students "at the top", and then aggregate these preferences into a single, more informative ranking of students. It is worth underscoring that the aim of generating such a rank is *not* to help automate admissions decisions. Rather, it is our hope that this more informative scoring system will serve as a springboard for discussion for the faculty members during their final-stage admissions meetings, and help guide conversation in a productive way.

2.2 Philosophical Project Motivation and Related Work

In this section, we discuss very briefly why rank aggregation is an inherently difficult (and interesting) problem. We also mention related work, and point the curious reader towards additional sources.

Recall that the goal of rank aggregation is to take the preferences of individuals within a community and somehow merge them into a single rank that reflects what the group as a whole believes. To make this less abstract imagine a group of people in which each member has a preference over ice cream flavors (i.e. person A believes chocolate is better than vanilla is better than strawberry etc.) and we are concerned with combining these individual beliefs into a single rank representative of the group's belief. One natural approach is to look at a majority vote. You could ask each person whether he likes chocolate better or vanilla. If more people think chocolate is better, chocolate should appear above vanilla in the global rank. Repeat this procedure for all pairs of flavors. However, no such solution is guaranteed to exist as there can be cycles. The reader can verify this himself by thinking about a group of three people in which one individual likes chocolate more than vanilla more than strawberry, one likes vanilla more than strawberry more than chocolate, and the third likes strawberry more than chocolate more than vanilla. This phenomenon was first articulated by Condorcet in 1785 and is often referred to as Condorcet's paradox [Condorcet: 1785].

It turns out that problems with rank aggregation go far beyond Condorcet's paradox. In the late 20th century, economist Kenneth Arrow proved that there is no way to aggregate individual ranks such that the resulting global rank simulateously satisfies a set of properties, which are desireable of global ranks [Arrow: 1963]. Such properties include "non-dictatorship" and "Pareto efficiency". His findings led to the birth of social choice theory.

As the above discussion illustrates, rank aggregation is an inherently hard problem. It is not obvious if such a thing as a "correct" aggregate rank exists in the first place, but even if we define a correct aggregate rank as having a certain set of "nice" properties, Arrow proved that this is impossible.

In spite of these difficulties, we have witnessed an outpouring of new approaches to this problem in recent years (many of them statistical in nature), probably on account of the immense commercial value such algorithms have (as mentioned earlier, they are the heart and soul of most recommendation systems). Many of these algorithms, however, do not interest us since they rely on cardinal information. As alluded to briefly in the introduction, many ranking algorithms ask users to give each item a numerical score, then sort the items according to some function of the score (e.g., the five stars scale for restaurants). For a myriad of reasons, we believe that it is far better to present two options to a user and simply ask which one he likes better (this is ordinal, or comparative information) as opposed to asking him to give all the items a numerical score (this is cardinal information). The idea is that a user's ordinal beliefs are much more stable than numerical beliefs and that in many contexts, getting numerical information is not even possible. Imagine a consumer facing a wall of products and picking shampoo X over shampoo Y; we can derive no cardinal information from this, but we can infer ordinal information in the sense that the user prefers X to Y. Thus, ordinal information is much more ubiquitous. For a more in-depth discussion on why we have restricted our attention to rank aggregation algorithms that rely on comparative information, see the introduction of [Ammar, Shah: 2012].

One of the more recent rank aggregation algorithms that satisfies our constraint of working with ordinal information is Negahban's [Negahban, Oh, Shah: 2012]. It is this algorithm that we used in *gradapply* because it is simple and outperforms other contenders like Braverman and Mossel's [Braverman, Mossel: 2008] and Ammar's [Ammar, Shah: 2012], which also rely on user-supplied comparisons.

Armed with this abridged overview, readers can hopefully better understand our vision. By using Negahban's algorithm, we are providing the community with one more successful example of a ranking system that relies on comparative information. By implementing his algorithm in a real system, we are getting a sense of how it performs in practice. By applying it in the context of graduate admissions at MIT, we are providing an automated way to aggregate the individual preferences of professors over students (which we know is fundamentally challenging), and therefore are adding value to the admissions process.

2.3 Relevant Background on User Experience of Pre-Existing System

We conclude this introductory section by providing some additional background information on the *gradapply* system, which will help the reader bettter understand the changes we made to the system (to know what has changed, one must have a reference point).

As was briefly alluded to before, the user experience for the typical reviewer (i.e., not an area chair, who is also responsible for administrative duties like assigning folders within his area to suitable faculty reviewers) is centered around the folder-table, which, as the name suggests, is just an html table that contains folders. When the reviewer first logs in, he is told that if he wants to review the applications to which he has been assigned, he should go to the "Your Folders" page. The "Your Folders" page contains a table where each row corresponds to a folder and is populated by information about that folder like name, gender, college etc. To make a review (i.e., give the folder a 1,2,3,4-score and enter in other comments), users must click on the name of the candidate in the folder table, which doubles as a hyperlink to the review form. There is a search bar above the folder-table that allows the user to submit queries on folders, the results of which will be displayed in the table. For instance, if you would like to see a list of all the students who were admitted in the current application cycle, you could enter in the query: 'tag:mit_admit & year:2010'. In fact, the "Your Folders" page is simply this same table with the search bar pre-populated with the query: "reader: your_username_here".

The folder-table has useful features such as the fact that it is sortable. By sortable, we mean that if you click on any of the header names, it will sort the rows in the table according to the element in that header-column. For instance, if you click on "Name: Last, First", the rows in the table will get sorted by last name from A-Z. If you click on that header again, the rows will be sorted by last name, but this time in the opposite order, from Z-A.

The folder-table is also customizeable by area chairs. An area chair can control which columns (from a fixed set of hard-coded choices) appear or do not appear in the table. One of the options, for instance, is to add a column that displays the average of a candidate's 1,2,3,4-scores. So, for instance, if candidate X was assigned readers A and B who gave him a 3 and 4 respectively, then 3.5 would show up in the new column in the row corresponding to candidate X. Then, if any given reviewer would like to get a sense of how a set of students compare to one another, he can submit a query for the set of students, then sort the students in the table by clicking on the column header corresponding to the average score. We took advantage of this fact in the design of our system, as will be explained later.

3 Design

In this section, we present the design for our additions to *gradapply*. We break the discussion into two sections, one centered around the input and the other centered around the output. We interleave discussion of the user interface throughout. However, for clarity, we begin by defining several terms, which will be used frequently.

3.1 Definitions and Overview

A crucial point about Negahban's algorithm is that it relies on "pairwise comparisons". A pairwise comparison is a comparison involving two applicants, say A

and B, where the outcome of the comparison is of the type

1. A is preferred to B (in which case we will say "A beat B" or $A > B$)
2. B is preferred to A (in which case we will say "B beat A" or $B > A$)
3. A and B are preferred equally (in which case we will say "A and B are tied" or $A = B$).

The algorithm will be explained in greater detail in Section 3.3.1.

As mentioned previously, faculty submit 1,2,3,4-scores for each student to whom they are assigned. If a certain professor gives student X a 4 and student Y a 3, this can be interpreted as the professor *implicitly* making the pairwise comparison, $X > Y$. We will therefore refer to this type of pairwise comparison information as "implicit pairwise comparison information," or "implicit PWC's", for short. Since this data comes for free (in the sense that the existing system manages input and storage of the data already, and that faculty are already accustomed to providing it), it made sense to use this type of pairwise comparison information as fodder for my ranking module. That said, the premise of this work is that the 1,2,3,4-scores do not provide sufficient resolution into the varying quality of applications (since usually all the people of interest get 4's, so the most you can say is $A = B$), so using only this type of PWC information is not enough.

To augment this information, we decided to allow faculty to *explicitly* give pairwise comparisons by submitting partial orderings of the students to whom they were assigned. A "partial ordering" is merely an extension of a pairwise comparison. It allows more than two people to be compared at a time, but does not allow for ties. For instance, using the notation introduced above, a partial ordering of students A, B, C, and D is : $D > B > A > C$ and the pairwise comparisons encoded by this partial ordering are: $D > B$, $D > A$, $D > C$, $B > A$, $B > C$, and $A > C$. We will refer to the pairwise comparisons encoded by partial orderings as "explicit pairwise comparisons" or "explicit PWC's". Note that introducing partial orderings clearly addresses the resolution problem; imagine A, B, C, and D all received scores of 4 - in this case, reviewers can create partial orderings like $D > B > A > C$ to differentiate among the previously equal candidates.

A reviewer "expresses an opinion" on a particular student if he makes a pairwise comparison (explicit or implicit) involving that student. A "personal rank" corresponding to reviewer X is an ordering (from best to worst) of all students about whom reviewer X has expressed an opinion. A "global rank" for the EECS department is an ordering (from best to worst) of all students about whom *some* reviewer in EECS has expressed an opinion.

Now that we have developed such vocabulary, we can talk in greater depth about the design of the pieces we added to *gradapply*.

To accomplish the goals outlined in the previous section, we added the following functionality to *gradapply*:

1. ability for reviewers to submit partial orderings (and thereby explicit pairwise comparison information) involving applications from the current admissions cycle

2. computation and display of a "personal rank" for each reviewer based on both implicit and explicit pairwise comparison information

3. computation and display of a" global rank" based on both implicit and explicit pairwise comparison information

Figure 1 helps explain at a high level what the main components of our system are and how information flows through them. The "Pairwise Comparison Information: Collection & Processing" Module takes in explicit PWC information in the form of partial orderings, and implicit PWC information in the form of 1,2,3,4 scores and converts them into actual pairwise comparisons (it also handles persistence of some of this information, which will be explained in more detail in Section 3.2.2). The "Ranking Module" takes in pairwise comparisons and outputs both personal and global ranks.

We will discuss both modules in greater detail in the ensuing sections, explaining design choices made along the way.

3.2 Pairwise Comparison Information: Collection & Processing (PCICP) Module

Figure 2 allows us to see inside the PCICP Module. Explicit PWC information in the form of partial orderings must be collected from reviewers through some interface. Furthermore, processing on the partial orderings is needed to extract the

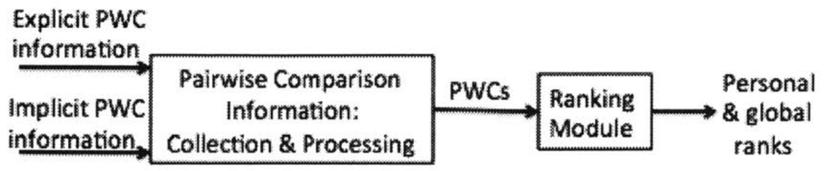


Figure 1: Schematic of the abstract components added to *gradapply* and how information flows through them.

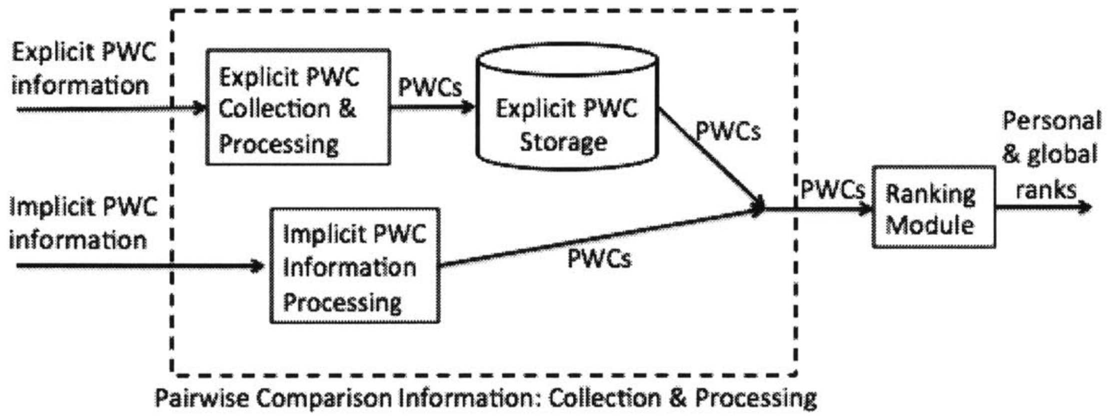
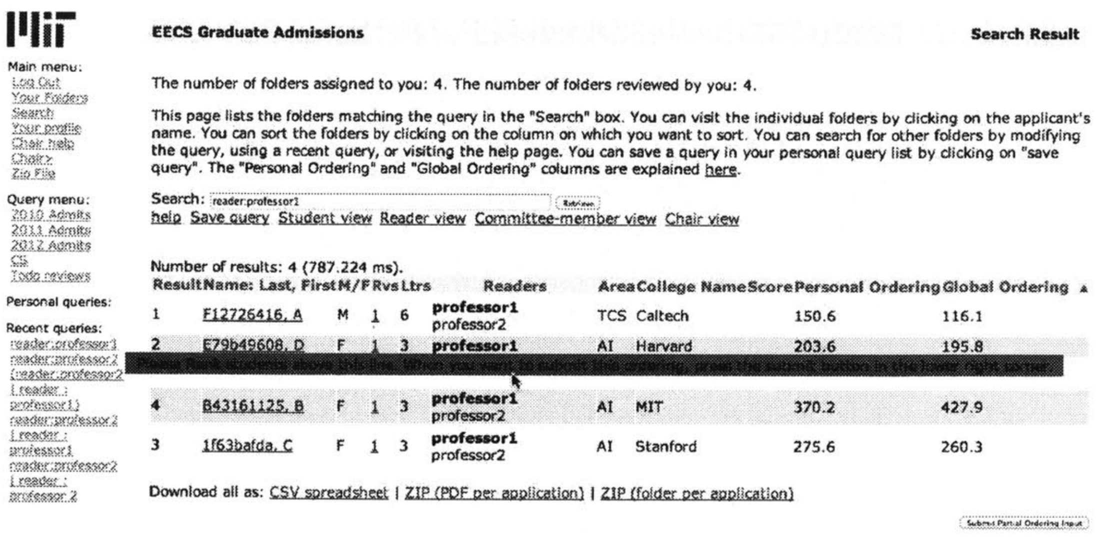

Explicit PWC information
Explicit PWC Collection & Processing
PWCs
Explicit PWC Storage
PWCs
Implicit PWC information
Implicit PWC Information Processing
PWCs
PWCs
Ranking Module
Personal & global ranks
Pairwise Comparison Information: Collection & Processing

Figure 2: Schematic revealing how the PCICP Module works.

PWCs from them. These two actions are the responsibility of the "Explicit PWC Collection & Processing" sub-module, shown in Figure 2. Next, these explicit PWCs must be persisted somehow. If the explicit PWCs were passed directly into the ranking module without being persisted first, this information would be lost forever; even if the resulting rank was saved, there would be no way to infer from that rank what PWCs gave rise to it. The "Explicit PWC Storage" sub-module in Figure 2 handles this.

The lower branch of the PCICP module handles implicit pairwise comparison information. Recall that the existing system already handles 1,2,3,4 scores - that is, it already provides a way for reviewers to input these 1,2,3,4-scores and persists them. Therefore, to obtain 1,2,3,4-scores from reviewers, we need only query the database (these scores are stored in a table called *Reviewv1*). The 1,2,3,4-scores, like the partial orderings, must be processed in order to obtain the underlying pairwise comparisons. This is the responsibility of the "Implicit PWC Information Processing" sub-module in Figure 2. Once the implicit PWCs have been derived from the 1,2,3,4-scores, an important question arises: should these PWCs be persisted, like the explicit PWCs are persisted?

We decided they should not be persisted. The pros of persisting the implicit PWCs is that it saves us from doing repetitive work. If a personal rank for reviewer X is asked for, we get process reviewer X's 1,2,3,4-scores *once* and then save the resulting PWCs in the database. The next time, say a global rank is requested (which of course will include reviewer X's opinions), we do not need to re-compute reviewer X's implicit PWCs from his 1,2,3,4-scores. Instead, we can simply query a database for them. However, there is a rather large downside of persisting the PWCs, namely: data consistency. The information stored in the 1,2,3,4-score table and the information that would be stored in the implicit PWC table are linked. Every time reviewer X adds a 1,2,3,4-score for a student, removes a 1,2,3,4-score for a student, or changes a 1,2,3,4-score, the PWCs in the implicit PWC table would need to be updated accordingly. While it is possible to keep these two tables roughly in sync, it is difficult to do it well, and leaves the system prone to the normal problems of synchronicity when data replication is involved. Given that the computation required to process the 1,2,3,4-scores and produce implicit PWCs is not very heavy, we decided that the cost of computing these "live" each time a rank is computed was well worth avoiding the headaches and dangers of



EECS Graduate Admissions **Search Result**

The number of folders assigned to you: 4. The number of folders reviewed by you: 4.

This page lists the folders matching the query in the "Search" box. You can visit the individual folders by clicking on the applicant's name. You can sort the folders by clicking on the column on which you want to sort. You can search for other folders by modifying the query, using a recent query, or visiting the help page. You can save a query in your personal query list by clicking on "save query". The "Personal Ordering" and "Global Ordering" columns are explained here.

Search: reader:professor1

Number of results: 4 (787.224 ms).

Result	Name: Last, First	M/F	Rvs	Ltrs	Readers	Area	College Name	Score	Personal Ordering	Global Ordering ▲
1	F12726416, A	M	1	6	**professor1** professor2	TCS	Caltech		150.6	116.1
2	E79b49608, D	F	1	3	**professor1**	AI	Harvard		203.6	195.8
Please Rank students above this line. When you want to submit this ordering, press the submit button in the lower right corner.										
4	B43161125, B	F	1	3	**professor1** professor2	AI	MIT		370.2	427.9
3	1f63bafda, C	F	1	3	**professor1** professor2	AI	Stanford		275.6	260.3

Download all as: CSV spreadsheet | ZIP (PDF per application) | ZIP (folder per application)

Figure 3: Screenshot showing a user (whose cursor is represented by the arrow head) dragging the separator bar upwards in the folder-table.

data inconsistency. This explains the asymmetry between the upper and lower branches of the PCICP module.

The next three sections discuss each of the three PCICP sub-modules in greater detail.

3.2.1 Explicit PWC Collection & Processing Sub-Module

As described above, this sub-module has two responsibilities: collection of explicit PWCs via partial orderings and processing of explicit PWCs.

To collect partial orderings, we decided to use the folder table. In particular, we made this table "draggable" - that is to say, users can reorder the rows of the table by clicking on and dragging various rows to new locations in the table. We also added a submit button and a "separator bar", which itself is a draggable row that starts off as the first row in the table. Figure 3 shows the separator being dragged upwards in the folder table.

To submit partial orderings, users populate the folder-table with applicants via some query (typically "reader: your_username_here", which is the query used when users navigate to the "Your Folders" page). They then drag the subset of students (rows) whom they want to rank *above* the separator bar, then order those students as they see fit (again, by dragging the rows around). When they are satisfied with the ordering, they press the submit button. To be clear, only the ordering of rows above the separator bar gets looked at by the system. There are a few natural questions to ask here:

(a) Why use the folder-table as the main interface? This severely limits the possible input-gathering mechanisms.

(b) Why use a separator bar?

(c) Why dragging?

(d) Since what you really want are PWCs and not partial orderings, why don't you just directly collect pairwise comparisons?

The answer to (a) is two-fold. The first reason is that, as explained in Section 2.3, the folder table is the center of activity for most reviewers. Reviewers are used to interacting with the folder table, and would be more likely to adopt a new tool if it involved something they already know (as opposed to having to learn a new structure). The second is that the folder-table is implemented in a very secure and robust way. It also has nice features like being easily extensible and sorttable. It made sense to leverage this powerful and safe structure instead of re-inventing the proverbial wheel.

Our reasoning for (b) was as follows. Without a separator bar, reviewers would have to order every folder (row) in the table. The average use case corresponds to the "Your Folders" page, where the table is populated with the folders to whom the reviewer has been assigned. In many cases, this number is greater than 100, and we felt that having to rank all 100 people at once would be an eye-crossing experience. The separator bar allows reviewers to rank any subset of students at a time (of size greater than or equal to two and higher). It should be noted here, however, that we plan to omit the separator bar in future versions of gradapply since its behavior was confusing to some users.

The answer to (c) involves aesthetic choice. We found that dragging seemed to be the most natural way of ordering a collection of items (and was far superior to other schemes we tried out). This choice was inspired in large part by the UI design of HotCRP [Kohler: 2007], mentioned in the introduction.

As for (d), we did contemplate an alternative design that did *not* involve partial orderings, and only pairwise comparisons. It involved selecting two folders from the table, and then selecting the winner. The problem with this design (and others similar to it) is that to make a single comparison, you need to perform at least 3 actions (selecting the two folders and designating the outcome). A single partial ordering involving n students, automatically encodes $\binom{n}{2}$ PWCs. To give the system the same number of PWCs, the alternative designs would require reviewers to perform $3*\binom{n}{2}$ actions. Therefore, the partial ordering scheme seemed like a much more efficient way of having faculty submit PWCs.

Now that we have discussed partial ordering collection, we can discuss the partial ordering processing. The goal of the processing is to take the partial orderings submitted by users, derive the explicit PWCs encoded therein, and send those to the Explicit PWC Storage Sub-Module (described in next section) for persistence. We have already seen that $A > B > C > D$ encode $A > B$, $A > C$, $A > D$, $B > C$, $B > D$, $C > D$. As this example demonstrates, the processing is simple, but there is one catch.

Users can submit as many partial orderings as they want. In most cases, reviewers use this to submit PWCs involving a new subset of applicants, however in some cases, they might submit partial orderings involving a subset of students whom they have already ordered. My system handles this by only storing the most recent PWC for each distinct pair of applicants for each reviewer (i.e. there is at most one explicit PWC per reviewer per folder-pair). For example, take the scenario in Table 1:

First, reviewer X submits a partial ordering involving A, B, C and D. This is his first partial ordering submitted, so all derived PWCs are saved. Next, reviewer X submits an ordering involving D and E. This is a PWC involving a new pair of students, so it just gets added to all the other PWCs. The third partial ordering is the interesting one because it involves C, B, and E. We see that C compared to B is an old comparison, but that C vs E and B vs E are new comparisons. This means that $C > B$ overrides the old PWC involving B and C, namely $B > C$, and that $C > E$ and $B > E$ also get saved.

Lastly, it has been implicit in this discussion that we store PWCs, as opposed

Table 1: An example of sequential partial ordering submissions

Partial Ordering Submitted	Derived PWCs	Total Saved PWCs
$A > B > C$	$A > B$, $A > C$, $A > D$, $B > C$, $B > D$, $C > D$	$A > B$, $A > C$, $A > D$, $B > C$, $B > D$, $C > D$
$D > E$	$D > E$	$A > B$, $A > C$, $A > D$, $B > C$, $B > D$, $C > D$, $D > E$
$C > B > E$	$C > B$, $C > E$, $B > E$	$A > B$, $A > C$, $A > D$, $C > B$, $B > D$, $C > D$, $D > E$, $C > E$, $B > E$

This table shows how the system behaves when multiple partial orderings are submitted sequentially. The first row corresponds to the first submission, the second row to the second submission, and the third row to the third submission. The i'th row of the "Total Saved PWC's" column shows the total pairwise comparisons that the system remembers after the ith submission.

to the partial orderings from which they were derived. This was a logical choice because PWCs are the natural building blocks or atomic units for our system. Partial orderings are just a "vehicle for submission," but at the end of the day, what the system cares about are the pairwise comparisons. Said differently, with the PWCs, we know everything we need - it is knowledge at the most granular level. Furthermore, there are more concrete reasons like space and computation savings: instead of storing multiple orderings (corresponding to multiple submissions) per reviewer, you save only the most recent PWC per pair for each reviewer and you only do the computation once (before saving) rather than every time you need to generate a ranking. In fact, in an earlier prototype, we implemented a schema that stored partial orderings. When we changed this to store PWCs as opposed to partial orderings, the code simplified by a factor of at least two.

3.2.2 Explicit PWC Storage Sub-Module

The obvious choice regarding persistence of explicit PWCs is to use a database. This begets the question of which schema to use. We developed a schema that matches one's intuitive idea of a pairwise comparison. A pairwise comparison is comprised of the two individuals being compared, the reviewer who is comparing them, and the judgement made by the reviewer (i.e. who won?). Because of the way the UI works, the system only receives strict orderings of students, so no ties are possible in explicit pairwise comparisons. As a result, we decided to have *folderid_lower* and *folderid_upper* be two columns that together store the identities of the two applicants being compared. An additional column, *username*, stores the identity of the reviewer who made the comparison. Lastly, the "lower" and "upper" tags implicitly store the result of the outcome. That is, the applicant who won is stored in the *folderid_upper* column and the applicant who lost is stored in the *folderid_lower* column.

Let N be the number of students who submitted applications in the current admissions cycle. Since there are $\binom{N}{2}$ distinct pairs of students, there are $O(N^2)$ pairs. Let K be the number of reviewers involved in the admissions process this year. The system is designed so that any reviewer can give explicit pairwise comparison information about any subset of students (including those to whom they were not assigned), so in principle, each of the K reviewers can make a judgment on each of the $O(N^2)$ pairs. As described in Section 3.2.1, the system only remembers the most recent judgment made by a reviewer concerning a particular pair. This means that for each set consisting of one reviewer and two students, there can be at most one row in the table. Combining these observations together gives that the number of rows in the table is $O(KN^2)$. This number can be high considering N and K were around 3000 and 300 respectively, this year. However, this bound is extremely conservative since in reality, faculty generally only enter in pairwise comparisons for the students to whom they have been assigned. A better (though still high) estimate would be given by:

1. Approximate number of reviewers: ≈ 300
2. Average number of students per reviewer: ≈ 100
3. Approximate number of pairwise comparisons per reviewer: $\binom{100}{2} = 4950$
4. Approximate total number of pairwise comparisons: 1.5×10^6.

This number is still an overestimate because, based on the data collected this year, some faculty only entered in scores for a small subset of their assigned students (usually the top portion that are still under consideration). Regardless,

1.5×10^6 is a reasonable number for PostgreSQL (the database used by *gradapply*) on an average server.

As this is the first time we use numbers to justify our design, we make an important clarification here. The statistics mentioned in this section and in following sections assume that both the EE and CS deparments are using the system. However, for reasons discussed later, the implementation was used by the CS deparment only. This means that we cannot completely validate our design decisions since we do not know if our additions would be fully functional when used across the entire department (we can only make this claim within the CS department).

3.2.3 Implicit PWC Information Processing Sub-Module

This module is called whenever a personal or global rank needs to be (re)computed. If it is a personal rank for reviewer X, the module gets all 1,2,3,4-scores that reviewer X made, then infers all possible implicit PWCs from this, and outputs a list of tuples that contain these PWCs. If it is a global rank, the same procedure is followed except *all* 1,2,3,4-scores made by reviewers this year are used. The tuples are of the form (folderid_lower, folderid_upper, istie, username), where folderid_lower, folderid_upper and username have the same meaning they did as in Section 3.2.2 for the explicit pairwise comparisons. The main difference here is that in implicit pairwise comparisons, ties are allowed (since two students can both have the same 1,2,3,4-score). The istie boolean takes care of this - it takes on a value of true if folderid_lower and folderid_upper had the same 1,2,3,4-score (thereby making the _lower and _upper tags meaningless). For instance, if we query the database and find that reviewer X gave A a 4, B a 4, and C a 2 (where A, B, and C are folder ids), the module would output the following list of tuples:

[(A, B, True, X), (C, A, False, X), (C, B, False, X)].

We have avoided code and pseudocode here so that readers can focus on the high level functionality. The implementation section contains further detail.

3.3 Ranking Module

The ranking module takes in pairwise comparisons and uses Negahban's algorithm to output a ranking of the applicants based on the (incomplete and perhaps contradictory) information in those comparisons. The same algorithm is used to generate both personal rankings and the global ranking. What differentiates these rankings therefore is just what you give the algorithm as input.

The personal rank is generated by running Negahban's algorithm on all explicit and implicit pairwise comparisons made by that reviewer during the current admissions cycle. This means a personal rank can be interpreted as an ordering of all the applicants on which the reviewer has expressed an opinion this year, and which was generated using only opinions from that reviewer. The output is a real number associated with each applicant involved, which is interpreted as the score of that applicant. The final ordering is determined by sorting the applicants according to their scores, from highest to lowest (i.e., higher scores are better). There is at most one "personal rank" associated with each reviewer.

A global rank is an ordering of all the applicants who applied this year which were judged by at least one reviewer (i.e. they were involved in at least one pairwise comparison by at least one reviewer this year). There is only one "global rank" for all of EECS. The global rank is generated in the same way as the personal rank. All explicit and implicit pairwise comparisons made by all reviewers this year are fed into Negahban's algorithm, which produces a score for each student. The final ordering is again determined by sorting the applicants according to their score from highest to lowest.

For both the personal rank and global rank, once the final scores of the applicants have been computed, we simply display that number in new columns entitled Personal Ordering and Global Ordering. Then, because the folder-table is sortable, users can get the actual rank by clicking on the Personal/Global ordering header, which will then sort the rows according to the personal/global score.

In this section, we focus first on the algorithm, then on storage of the ranks, and finally on the user experience.

3.3.1 Algorithm

The algorithm uses a random walk approach to ranking. The input is a directed graph $G = (V, E)$. Nodes represent applicants and there is an edge $(i, j) \in E$ if and only if applicant i has been compared to applicant j. The algorithm can be viewed as consisting of two parts: 1) constructing a probability transition matrix P over the graph, and 2) computing the corresponding stationary distribution

π. The stationary distribution will associate a real number with each node i (representing the steady-state probability of being in state i). It is this number that is interpreted as the score of applicant i and the number according to which we sort the applicants to produce the final ordering.

We now give the algorithm in full. Let $N_{i>j}$ denote the number of times that applicant i defeated applicant j. In the case of a tie, let $N_{i>j} = N_{j>i} = \frac{1}{2}$. As stated before, an edge $(i, j) \in E$ if and only if $N_{i>j} + N_{j>i} > 0$. Define:

$$Q_{ij} = \begin{cases} \frac{N_{j>i}+1}{N_{i>j}+N_{j>i}+2} & \text{if } (i,j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

Since $N_{i>j} + N_{j>i}$ is just the total number of times i and j were compared, Q_{ij} can be interpreted loosely as the fraction of times that j won when compared to i.

Let $C = max_i \sum_{j \neq i} Q_{ij}$. Viewing Q_{ij} as a matrix, C is the maximum of the sums of each row (with the diagonal terms excluded from the sums). We are now ready to generate the transition matrix over our graph:

$$P_{ij} = \begin{cases} \frac{Q_{ij}}{2 \cdot C} & \text{if } i \neq j \\ 1 - \sum_{j \neq i} P_{ij} & \text{if } i = j. \end{cases}$$

We see that P_{ij} is essentially Q_{ij} scaled to form a valid probability matrix. Dividing each row by $2 \cdot C$ ensures that no row has a sum greater than one and setting the diagonal elements to $1 - \sum_{j \neq i} P_{ij}$ ensures each row sums to exactly one.

Now that we have computed a transition matrix for our graph, we need only compute its stationary distribution. By construction, our Markov chain is aperiodic. Unfortunately, when applied to *gradapply*, it is quite possible that the graph will be composed of several disconnected (but irreducible) components. This means that the stationary distribution might depend on the starting state (how we handle this is described in Section 4.3.2). Let $p_t(i) = P(X_t = i)$ be the distribution of the random walk at time t, with $\boldsymbol{p_0} = [p_0(i)] \in \mathbb{R}^n_{>0}$ denoting an arbitrary starting distribution on over the n states. We have that, for all $t \geq 0$,

$$\boldsymbol{p_{t+1}^T} = \boldsymbol{p_t^T} P \tag{1}$$

so we can approximate the steady state distribution of our Markov Chain by performing sufficiently many iterations of equation 1. Intuitively, the edge from i

to j is roughly proportional to the fraction of times j beat i. Using π_j, the longterm frequency of being in state j, is therefore a natural measure of j's "goodness" since this number will be high if j beat other applicants with high longterm frequencies (i.e. other good candidates) or if it won against many applicants.

We conclude this section by mentioning that, when run on data generated by the Bradley-Terry-Luce model, Negahban's algorithm performs as well as the Maximum Likelihood estimator for that model. We direct the curious reader to Negahban's paper [Negahban, Oh, Shah: 2012] for further information.

3.3.2 Storage of Ranks

We decided to store the personal and global ranks computed by Negahban's algorithm in the database. While this was by no means mandatory, since the server can always pass back the results to the browser when computation is finished, it seemed logical to make the system more "static". Since the personal ranks and global rank do not change frequently after their initial computation, it made sense to load potentially stale data from a database rather than perform the computation every time reviewers navigated to their "folders" page (this would pose a heavy load on the server).

Given the decision to persist the output of various calls to the algorithm, we now discuss the schema. Recall that Negahban's algorithm produces a score for each applicant and that there are two types of orderings: personal and global. Therefore, an ordering is fully specified by identifying which applicants are in the rank, associating a score with each applicant, and indicating whether it is a personal or global rank (of course, if it is a personal rank, then we must also specify whose ordering it is). To achieve this, we have a *folderid* column to denote the applicant, a *finalscore* column to denote the score that our rank module output for said person, and a *username* column to denote whether this applicant/score datum is part of a global or personal rank. If it is a personal ordering, the *username* fields gets assigned the username of the reviewer to whom the ordering belongs. Otherwise, it is a global ordering, so the *username* field is assigned the string, "global".

Every time the ranking algorithm is run, it produces a new rank, which is stored in the table described above. However, the table only keeps the most recent rank, so there is at most one rank per reviewer. As an aside, it makes sense to keep only the most recent rank; the rank algorithm is recomputed whenever there is new information to be had, so a user should only be interested in the most recent rank, which reflects the most up-to-date information. To analyze the schema, we again let K and N be the number of reviewers and number of people who submitted applications in the current admissions cycle, respectively. Since there is at most one rank per reviewer, there are $O(K+1) = O(K)$ ranks per admissions cycle (the plus one is for the separate global rank). Each rank can involve at most N students, thereby contributing $O(N)$ rows per rank to the table. Thus, during any given admissions cycle, there can be $O(KN)$ rows in the table. Using $N \approx 3000$ and $K \approx 300$, which is representative of this year's data, we get approximately $900,000$ rows, which is more than manageable for PostgreSQL on an average server.

3.3.3 Overall Flow

In this section we clarify when ranks get recomputed. As alluded to briefly in Section 3.3.2, the personal and global scores are loaded with information from the database whenever one navigates to the folder page. The personal rank gets recomputed for a given professor whenever that professor submits new partial orderings. It does *not* get recomputed when the professor changes, or adds new 1,2,3,4-scores. This entails that users may find themselves looking at slightly stale personal orderings. To rectify this situation, we decided to have the button for partial ordering submissions double as a "refresh button". If a user does not drag the separator bar down or move any rows, clicking this button will cause the personal rank to be recomputed and re-displayed. This gives concerned users a way of ensuring they are looking at the most up-to-date version of their personal ordering.

The global rank is a different entity. Given that its computation is heavy, and that its purpose only comes into play once or twice at the end of the admissions process (when final admissions decisions must be made), it made sense for the computation to be triggered by a human only when needed. As such, we made the global rank computable only by area chairs (letting any reviewer trigger it at any time would put the server at risk). The expected use case is that an area chair would run the computation the night before he convenes a meeting with other professors to finalize admissions decisions.

The design choices made above were governed by the goal of making the "system" as static as possible subject to the constraint that users still receive acceptably accurate data. Keeping in line with this goal, future editions of *gradapply* will not compute personal ordering scores as described. Rather, the personal ordering will simply reflect the order in which the user has dragged his students (this information will still get written to the database in the form of explicit PWCs, but the algorithm will no longer have to run on said information to give the UI an ordering).

4 Implementation

The goal of this section is not to go over every detail of the code we wrote. Rather, it is intended to provide the reader with certain facts which may interest him and provide greater detail when such detail is necessary or elucidating.

4.1 Technologies Used

The *gradapply* system was developed primarily by Professors Frans Kaashoek and Robert Morris of MIT. It uses Apache as the webserver, PostgresSQL for the database, and Django 1.3 and Python 2.7 as the web framework. Other technologies like jQuery UI were also used to make the folder table draggable ("//code.jquery.com/ui/1.10.0/jquery-ui.js").

4.2 Models

We added two models to the system: *PairwiseComparison* and *AggregateOrderv1*, which correspond to the two tables discussed in Sections 3.2.2 and 3.3.2. The former is the one that stores explicit pairwise information from all reviewers and the latter stores the personal ranks of each reviewer, as well as the global rank.

PairwiseComparison is a simple Django model that contains three fields, which map to the columns introduced earlier. The code below shows that *folderid_lower* and *folderid_upper* are both foreign keys to the Folder table, which was a pre-existing table used to store folders. This was done to help maintain data consistency and gain efficiency. The *related_name* argument is necessary so that Django can resolve the backward relation since there are two foreign keys within the same table.

```
class PairwiseComparison(models.Model):
    folderid_lower = models.ForeignKey(Folder, db_column='folderid_lower',
        related_name = 'folder_lower')
    folderid_upper = models.ForeignKey(Folder, db_column='folderid_upper',
        related_name = 'folder_upper')
    username = tf()
```

Similarly, *AggregateOrderv1* is a simple Django model whose three fields map to the columns introduced earlier.

```
class AggregateOrderv1(models.Model):
    folderid = models.ForeignKey(Folder, db_column='folderid')
    finalscore = models.FloatField()
    username = tf()
```

We see from the code that the *folderid* field is a foreign key to the Folder table and *finalscore* is a float representing the score that our rank module output for said person. If it is a personal ordering, the *username* fields gets assigned the username of the reviewer to whom the ordering belongs. Otherwise, it is a global ordering, so the *username* field is assigned the string, "global".

4.3 Algorithm

Recall from Section 3.3.1 that the algorithm is simply a random walk over a graph. To implement this, it is necessary to represent the graph in some way, compute the appropriate weightings, then compute the corresponding stationary distribution.

4.3.1 Data Structures

To represent the graph, we used a matrix since it is an easy data structure to work with and a fairly efficient one. In particular, we defined a two dimensional $n \times n$ matrix, $N_{i>j}$, such that each index $i = 1, ..., n$ represents an applicant, and where the i,jth entry represents the number of times that applicant i won when it was compared to applicant j. We maintained a mapping from applicants to their column numbers by using a dictionary whose keys were the folder ids of the students involved in the rank and whose values were integers representing the column index of the given applicant. Note that if $[N]_{ij}$ and $[N]_{ji}$ are both 0, this implies that applicants i and j have never been compared.

To populate the matrix for professor X's personal rank, we followed the procedure below:

- determine which applicants received a 1,2,3,4 score or were involved in a PWC (or both) by professor X. Assume there are n such students.
- create an $n \times n$ matrix of zeros and create a dictionary that maps folder ids to column indices of the matrix.
- iterate over all pairwise comparisons made by professor X. For each one, let k be the index of the winner and m be the index of the loser (ties cannot happen). Set $[N]_{k,m} = [N]_{k,m} + 1$.
- get all 1,2,3,4 scores made by professor X and infer the PWCs implied by these scores. Iterate through these implicit PWCs. If k is the index of the winner and m is the index of the loser, increment the $[N]_{k,m}$ by 1. If they are tied increment $[N]_{k,m}$ and $[N]_{m,k}$ by 0.5 each.

If we are performing a global rank as opposed to a personal rank, you repeat the above procedure for each active reviewer this year, but instead of starting with a matrix of zeros each time, you pass on the matrix from the previous iteration so the same matrix gets updated once per reviewer.

Note, that once populated, the $N_{i>j}$ matrix contains all the information we need in order to find the probability transition matrix. We can use the matrix $N_{i>j}$ to get a new matrix, Q_{ij}, whose ijth entry represents the *fraction* of times i won when compared to j, and then derive the transition matrix by following the procedure from Section 3.3.1.

4.3.2 Stationary Distribution Computation

There are several ways to compute a stationary distribution given a transition matrix P_{ij}. We opted for starting with a uniform stationary distribution, then computing $\boldsymbol{p}_{t+1}^T = \boldsymbol{p}_t^T P$ for a "sufficiently large" number of iterations. What we did in detail is the following:

- start with a uniform distribution over all applicants involved ($\boldsymbol{p}_0$ = uniform)
- $t = 1$
- while True:

```
    $\boldsymbol{p}_{t+1}^T = \boldsymbol{p}_t^T P$
    compute the $\ell^1$ norm of the difference between $\boldsymbol{p}_{t+1}$ and $\boldsymbol{p}_t$
    if the difference is less than $\frac{1}{n^2 \log^2 n}$ or $t$ exceeds some threshold:
        return $\boldsymbol{p}_{t+1}$
    $\boldsymbol{p}_t$=$\boldsymbol{p}_{t+1}$
    $t = t + 1$
```

Essentially, when the estimates of the stationary distribution from successive iterations start differing by a sufficiently small amount, we take that as a good approximation to the true limit and return that vector. We used $10n$ as a threshold because $n \log_e n$ is less than $10n$ when $n = 3000$ (again, the exact number of applicants this year was 2970). We needed to make sure that this "timeout" variable was high enough such that even the worst graphs would have time to converge. An example of an ill-behaved graph is a ring graph, which is known to have a mixing time of $O(n)$. Furthermore, the $\log n$ factor is needed to get a small error in chi-square distance [Shah: 2009]. In practice, we found this heuristic worked well; the loop was always exited due to the termination condition being met and not because it timed out (i.e. that the stationary distribution"converged" within our worst-case expectations).

The reader may wonder why we chose to compute the stationary distribution by starting with a uniform distribution over the states, then repeatedly performing $\boldsymbol{p}_{t+1}^T = \boldsymbol{p}_t^T P$. The reason is because in this application, it is quite possible that our comparison graph is comprised of several disconnected components. Recall that an edge exists between nodes i and j if and only if i and j have been compared by at least one reviewer. Now imagine we pick a candidate i from the Circuits subarea and a candidate j from Theoretical Computer Science. It is quite likely that no reviewer would have submitted a partial ordering involving these two candidates, or given them both 1,2,3,4-scores. Extrapolating from this example, it is easy to imagine the comparison graph consisting of several disconnected components where each component roughly corresponds to a research area. In this case, there does not exist a unique stationary distribution because the stationary distribution will very much depend on what state the chain started in - every node not in the component containing the start state will have a steady state probability of 0, and will also therefore have a global score of 0. Starting with a stationary distribution skirts this issue by assigning probability mass to each component of the graph according to the component's size and, subsequently, no students will end up with a global score of 0.

5 Evaluation

In this section, we argue that the features we implemented *worked* and *added value.* Unfortunately, this is not a straightforward task; as our discussion in Section 2.2 revealed, it is not clear that there is any correct aggregate rank against which we can check our output. Another way of saying this is that there is no "ground truth" (except, perhaps, the list of students who were admitted, which is data I lack), so we cannot measure how far we are from that. To get around this, we take two approaches. First, we try to show, using a synthesized example, that our ranking module produces results consistent with our expectations. Second, we turn to real data we collected and show that the ranks generated from such data pass a series of "sanity checks". Third, we look more closely at the ranks generated from the real data and try to argue that the results make sense and are helpful (this is harder than with the synthesized example because the sample space is much larger).

5.1 Synthesized Example

Imagine there are four Students, A, B, C, and D, and two professors, *professor1* and *professor2*. For simplicity, assume that all four applicants are assigned to each professor for review. Furthermore, imagine we know *professor1* submitted the partial ordering $A > C > B > D$ and *professor2* submitted the partial

Table 2: 1,2,3,4-scores given to Students A, B, C, and D by each Reviewer

Reviewer	Score for A	Score for B	Score for C	Score for D
professor1	4	3	4	1
professor2	4	4	4	2

This table shows the 1,2,3,4-scores that *professor1* and *professor2* gave their students in our "Synthesized Example".

Table 3: N matrix for *professor1*

	A	B	C	D
A	-	2	1.5	2
B	0	-	0	2
C	.5	2	-	2
D	0	0	0	-

This table shows the N matrix for *professor1* for our synthesized example. Each row and column is labelled with a student name. The cell whose row is A and whose column is B, for example, gives the number of times *professor1* thought A won against B.

ordering $C > A > B > D$ in conjunction with the 1,2,3,4-scores shown in Table 2:

Based on this information, we see that *professor1* thinks A and C are the best, followed by B, then D. Further more, *professor1*'s partial ordering reveals that he thinks A is better than C. As for *professor2*, she also thinks A and C are the best, but thinks C is better than A. We therefore expect the aggregate ordering to rank A and C as the top two, followed by B and D.

The N matrix (recall from Section 4.3.1 that the i, jth entry of this matrix represents the number of times i won when it was compared to j) for *professor1* is as shown in Table 3. The N matrix for *professor2* is as shown in Table 4.

The resulting stationary distribution for the global rank (determined by summing the two N matrices, normalizing it appropriately, and computing the stationary distribution from a uniform starting distribution) is given by $[\pi_A, \pi_B, \pi_C, \pi_D] =$

Table 4: N matrix for *professor2*

	A	B	C	D
A	-	1.5	.5	2
B	.5	-	.5	2
C	1.5	1.5	-	2
D	0	0	0	-

This table shows the N matrix for *professor2* for our synthesized example. Each row and column is labelled with a student name. The cell whose row is A and whose column is B, for example, gives the number of times *professor2* thought A won against B.

Table 5: 1,2,3,4-scores given to Students A, B, C, and D by *professor3*

Reviewer	Score for A	Score for B	Score for C	Score for D
professor3	4	2	4	3

This table shows the 1,2,3,4-scores that *professor3* gave his students in our "Synthesized Example".

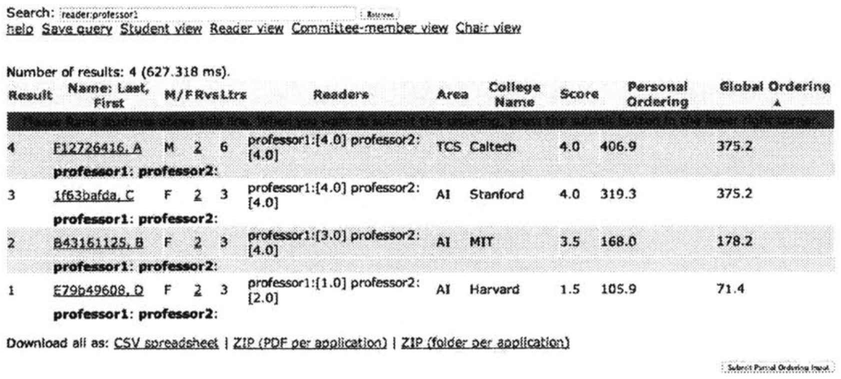
$[.3752, .1782, .3752, .071]$. In other words, A and C are tied with a score of 375.2, B follows with a score of 178.2; and D comes in last place with 71.4 (where we scaled the probabilities by a factor of $1,000$ to get the score).

Note that A and C receive the same score, which makes sense because they were both given 4's, and because *professor1* liked A more than C but *professor2* liked C more than A. Because of the symmetry, there is no reason why one should be ranked above the other. The rest of the scores are consistent with what we expect: B is third to last and D is last (since he lost to everyone else).

Now, imagine we add an additional reviewer, *professor3*, who submitted the partial ordering, $A > C > D > B$ and the 1,2,3,4-scores in Table 5.

As a result of *professor3*'s submissions, the N matrix changes to 6 and the resulting steady state distribution is now given by $[\pi_A, \pi_B, \pi_C, \pi_D] = [.4261, .1211, .3779, .075]$.

A gets a higher score than C, which is good because now two professors rank $A > C$ and only one ranks $C > A$. Furthermore, D is still the worst candidate,


Search: reader:professor1

help Save query Student view Reader view Committee-member view Chair view

Number of results: 4 (627.318 ms).

Result	Name: Last, First	M/F	Rvs	Ltrs	Readers	Area	College Name	Score	Personal Ordering	Global Ordering ▲
Please Rank students above this line. When you want to submit this ordering, press the submit button in the lower right corner.										
4	F12726416, A	M	2	6	professor1:[4.0] professor2: [4.0]	TCS	Caltech	4.0	406.9	375.2
	professor1: professor2:									
3	1f63bafda, C	F	2	3	professor1:[4.0] professor2: [4.0]	AI	Stanford	4.0	319.3	375.2
	professor1: professor2:									
2	B43161125, B	F	2	3	professor1:[3.0] professor2: [4.0]	AI	MIT	3.5	168.0	178.2
	professor1: professor2:									
1	E79b49608, D	F	2	3	professor1:[1.0] professor2: [2.0]	AI	Harvard	1.5	105.9	71.4
	professor1: professor2:									

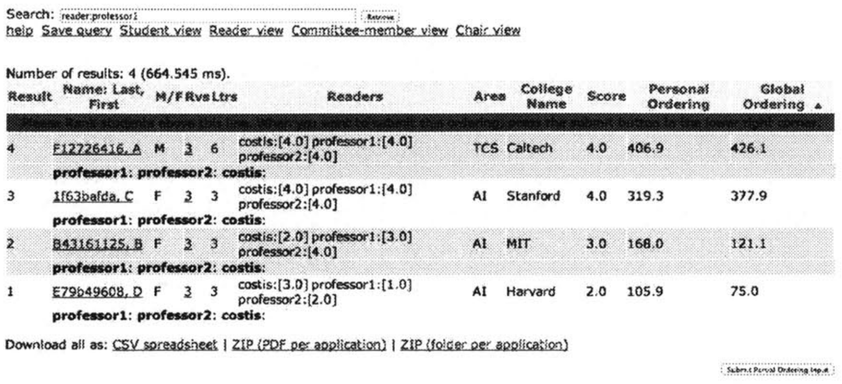
Download all as: CSV spreadsheet | ZIP (PDF per application) | ZIP (folder per application)

Figure 4: This figure shows the folder-table seen by *professor1* after both *professor1* and *professor2* have submitted their opinions of their students and after the global rank has been computed.

Table 6: Global N matrix after *professor3*'s submissions

	A	B	C	D
A	-	5.5	3.5	6
B	.5	-	.5	4
C	2.5	5.5	-	6
D	0	2	0	-

This table shows the global N matrix after all three professors have submitted their opinions. The columns and rows are again labelled by student names. The cell indexed by row *A* and column *B* denotes the number of times *A* won when it was compared to *B* (acounting for all the professors' opinions).



Search: reader:professor1 Retrieve
help Save query Student view Reader view Committee-member view Chair view

Number of results: 4 (664.545 ms).

Result	Name: Last, First	M/F	Rvs	Ltrs	Readers	Area	College Name	Score	Personal Ordering	Global Ordering ▲
Please Rank students above this line. When you want to submit this ordering, press the submit button in the lower right corner.										
4	F12726416, A	M	3	6	costis:[4.0] professor1:[4.0] professor2:[4.0]	TCS	Caltech	4.0	406.9	426.1
	professor1: professor2: costis:									
3	1f63bafda, C	F	3	3	costis:[4.0] professor1:[4.0] professor2:[4.0]	AI	Stanford	4.0	319.3	377.9
	professor1: professor2: costis:									
2	B43161125, B	F	3	3	costis:[2.0] professor1:[3.0] professor2:[4.0]	AI	MIT	3.0	168.0	121.1
	professor1: professor2: costis:									
1	E79b49608, D	F	3	3	costis:[3.0] professor1:[1.0] professor2:[2.0]	AI	Harvard	2.0	105.9	75.0
	professor1: professor2: costis:									

Download all as: CSV spreadsheet | ZIP (PDF per application) | ZIP (folder per application)

Submit Partial Ordering Input

Figure 5: This figure shows the folder-table *professor1* would see after *professor3* submitted his opinions and the global rank was re-computed.

but we see his score moved up from 71.4 to 75.0 (which is good because *professor3* ranked $D > B$ - i.e. before, B was consistently ranked over D, but now there is at least one dissenting vote). Figure 5 summarizes this information.

This example illustrates that our ranking module performs rank aggregation reasonably. While there is no "ground truth" per say, the algorithm combined *professor1* and *professor2*'s personal ranks in an intuitively pleasing way. Furthermore, after adding *professor3*'s preferences, the rank changed in all the ways in which we expected it to. There are numerous other small examples that we coded as part of a suite of unit tests, and the results thus obtained, adhered similarly to common-sense expectations.

5.2 Real Data

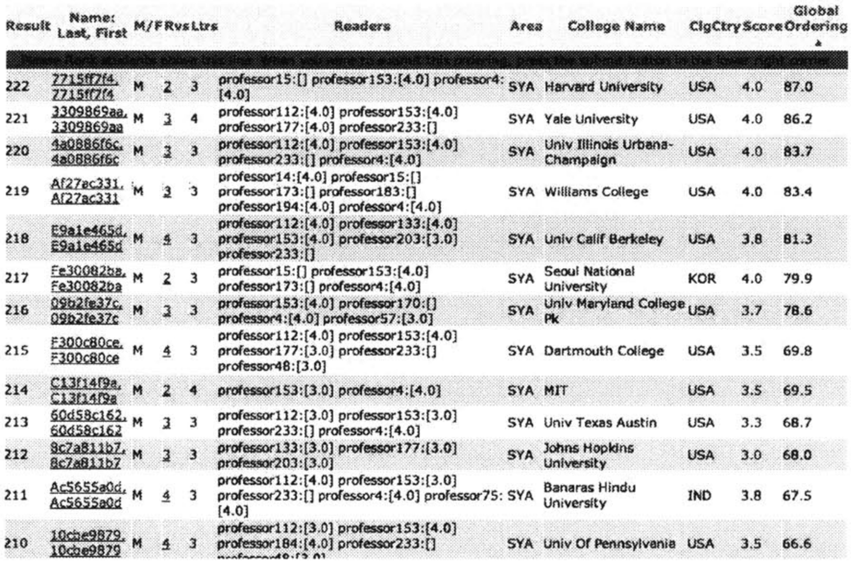
Now we transition to looking at the actual data collected from the January 2013 trial. As mentioned before, since we cannot argue directly that the global rank is "correct", we first perform several sanity checks, then argue why this feature was a useful addition to the system.

On January 24, the night before several of the CS faculty were to gather and finalize admissions decisions, the new scoring system we implemented went live. CS professors attending the meeting were asked to submit partial orderings by late that evening. Seventeen reviewers fulfilled this request and submitted orderings. Due to a bug that has since been fixed, the new scoring system was taken down soon after (before the EE faculty could enter their partial orderings), and a global rank was never displayed to the faculty. Because our code was live for a very short time period and only seventeen users got to interact with it, we received little to no user feedback, and will therefore not address feedback in this section.

We did, however, receive an anonymized copy of the database (taken after the CS faculty submitted their partial orderings). Because we had access to their partial orderings, we were able to compute personal ranks and the global rank, locally on my computer. It is these results, which we analyze in this section. Note that all figures show the folder table populated by the query, (reader:professor112 | reader:professor153 | reader:professor177 | reader:professor4). We chose these four professors because they were among the seventeen professors who submitted partial orderings and they all happened to have a lot of assigned applicants in common, due to their shared research interests of operating systems, storage systems, and security analysis. We also removed the Personal Ordering column from many of the figures for ease of readability since it is primarily the global rank which concerns us.

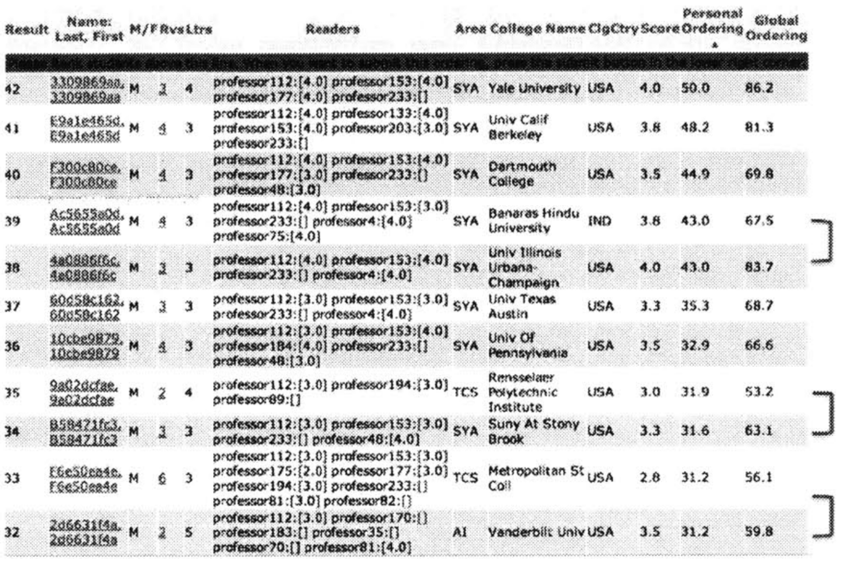
The first sanity check we perform is to see that the scores our ranking module produces are roughly in line with the 1,2,3,4-scores submitted by faculty. In other words, we expect our algorithm to rank people who got a lot of 4's highly, people who got a lot of 3s less highly, and so on. Figure 6 shows a segment of the folder table, where the rows in the table have been sorted according to the global score. We see that, scanning from top to bottom, the 4s are located mainly at the top, followed by the 3s. The picture was truncated for brevity, but the full rank continues this pattern.

The second sanity check we perform is to take a few select professors and check that their personal ranks never vary too much from the global scores. Of course, they cannot match up perfectly, but if any professor's preferences differed drastically from the global scores, that would indicate that the aggregation is not working well since it did not seem to take into account that professor's views. Figures 7, 8, and 9 juxtapose three professors' personal ranks with the global rank. The personal ranks are all sorted according to personal score. You can see the differences because the global numbers will sometimes be out of order. In other



Result	Name: Last, First	M/F	Rvs	Ltrs	Readers	Area	College Name	ClgCtry	Score	Global Ordering
[illegible]										
222	7715ff7f4, 7715ff7f4	M	2	3	professor15:[] professor153:[4.0] professor4:[4.0]	SYA	Harvard University	USA	4.0	87.0
221	3309869aa, 3309869aa	M	3	4	professor112:[4.0] professor153:[4.0] professor177:[4.0] professor233:[]	SYA	Yale University	USA	4.0	86.2
220	4a0886f6c, 4a0886f6c	M	3	3	professor112:[4.0] professor153:[4.0] professor233:[] professor4:[4.0]	SYA	Univ Illinois Urbana-Champaign	USA	4.0	83.7
219	Af27ac331, Af27ac331	M	3	3	professor14:[4.0] professor15:[] professor173:[] professor183:[] professor194:[4.0] professor4:[4.0]	SYA	Williams College	USA	4.0	83.4
218	E9a1e465d, E9a1e465d	M	4	3	professor112:[4.0] professor133:[4.0] professor153:[4.0] professor203:[3.0] professor233:[]	SYA	Univ Calif Berkeley	USA	3.8	81.3
217	Fe30082ba, Fe30082ba	M	2	3	professor15:[] professor153:[4.0] professor173:[] professor4:[4.0]	SYA	Seoul National University	KOR	4.0	79.9
216	09b2fe37c, 09b2fe37c	M	3	3	professor153:[4.0] professor170:[] professor4:[4.0] professor57:[3.0]	SYA	Univ Maryland College Pk	USA	3.7	78.6
215	F300c80ce, F300c80ce	M	4	3	professor112:[4.0] professor153:[4.0] professor177:[3.0] professor233:[] professor48:[3.0]	SYA	Dartmouth College	USA	3.5	69.8
214	C13f14f9a, C13f14f9a	M	2	4	professor153:[3.0] professor4:[4.0]	SYA	MIT	USA	3.5	69.5
213	60d58c162, 60d58c162	M	3	3	professor112:[3.0] professor153:[3.0] professor233:[] professor4:[4.0]	SYA	Univ Texas Austin	USA	3.3	68.7
212	8c7a811b7, 8c7a811b7	M	3	3	professor133:[3.0] professor177:[3.0] professor203:[3.0]	SYA	Johns Hopkins University	USA	3.0	68.0
211	Ac5655a0d, Ac5655a0d	M	4	3	professor112:[4.0] professor153:[3.0] professor233:[] professor4:[4.0] professor75:[4.0]	SYA	Banaras Hindu University	IND	3.8	67.5
210	10cbe9879, 10cbe9879	M	4	3	professor112:[3.0] professor153:[4.0] professor184:[4.0] professor233:[]	SYA	Univ Of Pennsylvania	USA	3.5	66.6

Figure 6: This figure shows the top portion of the folder table when populated by the query, (reader:professor112 | reader:professor153 | reader:professor177 | reader:professor4).

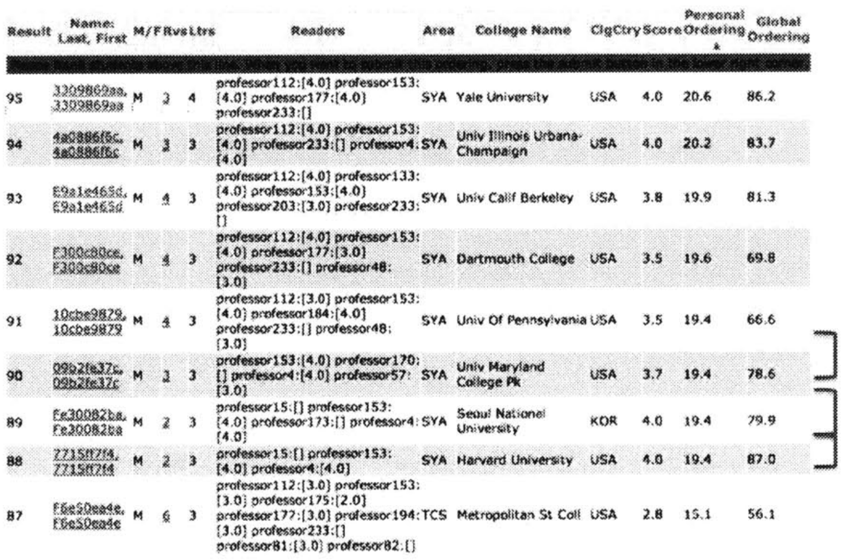


Result	Name: Last, First	M/F	Rvs	Ltrs	Readers	Area	College Name	ClgCtry	Score	Personal Ordering ▲	Global Ordering
[illegible]											
42	3309869aa, 3309869aa	M	3	4	professor112:[4.0] professor153:[4.0] professor177:[4.0] professor233:[]	SYA	Yale University	USA	4.0	50.0	86.2
41	E9a1e465d, E9a1e465d	M	4	3	professor112:[4.0] professor133:[4.0] professor153:[4.0] professor203:[3.0] professor233:[]	SYA	Univ Calif Berkeley	USA	3.8	48.2	81.3
40	F300c80ce, F300c80ce	M	4	3	professor112:[4.0] professor153:[4.0] professor177:[3.0] professor233:[] professor48:[3.0]	SYA	Dartmouth College	USA	3.5	44.9	69.8
39	Ac5655a0d, Ac5655a0d	M	4	3	professor112:[4.0] professor153:[3.0] professor233:[] professor4:[4.0] professor75:[4.0]	SYA	Banaras Hindu University	IND	3.8	43.0	67.5
38	4a0886f6c, 4a0886f6c	M	3	3	professor112:[4.0] professor153:[4.0] professor233:[] professor4:[4.0]	SYA	Univ Illinois Urbana-Champaign	USA	4.0	43.0	83.7
37	60d58c162, 60d58c162	M	3	3	professor112:[3.0] professor153:[3.0] professor233:[] professor4:[4.0]	SYA	Univ Texas Austin	USA	3.3	35.3	68.7
36	10cbe9879, 10cbe9879	M	4	3	professor112:[3.0] professor153:[4.0] professor184:[4.0] professor233:[] professor48:[3.0]	SYA	Univ Of Pennsylvania	USA	3.5	32.9	66.6
35	9a02dcfae, 9a02dcfae	M	2	4	professor112:[3.0] professor194:[3.0] professor89:[]	TCS	Rensselaer Polytechnic Institute	USA	3.0	31.9	53.2
34	B58471fc3, B58471fc3	M	3	3	professor112:[3.0] professor153:[3.0] professor233:[] professor48:[4.0]	SYA	Suny At Stony Brook	USA	3.3	31.6	63.1
33	F6e50ea4e, F6e50ea4e	M	6	3	professor112:[3.0] professor153:[3.0] professor175:[2.0] professor177:[3.0] professor194:[3.0] professor233:[] professor81:[3.0] professor82:[]	TCS	Metropolitan St Coll	USA	2.8	31.2	56.1
32	2d6631f4a, 2d6631f4a	M	2	5	professor112:[3.0] professor170:[] professor183:[] professor35:[] professor70:[] professor81:[4.0]	AI	Vanderbilt Univ	USA	3.5	31.2	59.8

Figure 7: This figure shows the folder-table as viewed by *professor112* when sorted by *professor112*'s personal ordering scores. The red brackets show places in which the global scores increase when scanned from top to bottom (denoting places where *professor112*'s opinion differs from the global one).

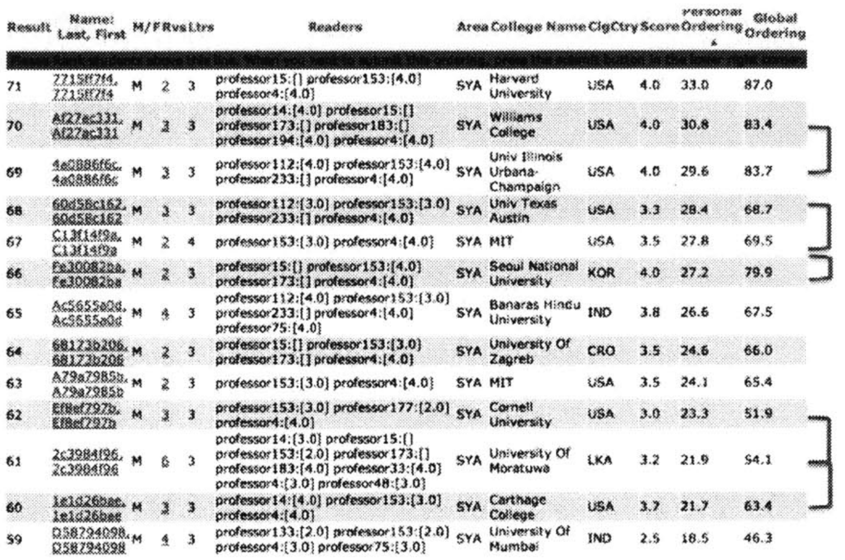
words, if you start reading at the top and move down, whenever the scores in the global ordering column jump up (instead of going down monotinically), there is a difference. We have highlighted those cases with red brackets for ease of viewing. We see that there are (necessarily) jumps in these samples, but that on average it is not excessive, which is consistent with our intuitive notion of aggregating diverse preferences.

Hopefully, the synthesized example from Section 5.1 and the above sanity checks on the real data convincingly demonstrate that the algorithm is working. We now transition to focusing on how these ranks improve upon the earlier system. Looking back to Figure 6, we see that our ranking module has provided new information. The first four candidates in the global rank - students '7715ff74f', '3309869aa', '4a0886f6c' and 'Af27ac331' - all have an average 1,2,3,4-score of 4.0, which is shown in the column entitled "Score". Prior to my additions, *gradapply* could not have ordered these students in any meaningful way, except for perhaps



Result	Name: Last, First	M/F	Rvs	Ltrs	Readers	Area	College Name	ClgCtry	Score	Personal Ordering	Global Ordering
[illegible]											
95	3309869aa, 3309869aa	M	3	4	professor112:[4.0] professor153:[4.0] professor177:[4.0] professor233:[]	SYA	Yale University	USA	4.0	20.6	86.2
94	4a0886f6c, 4a0886f6c	M	3	3	professor112:[4.0] professor153:[4.0] professor233:[] professor4:[4.0]	SYA	Univ Illinois Urbana-Champaign	USA	4.0	20.2	83.7
93	E9a1e465d, E9a1e465d	M	4	3	professor112:[4.0] professor133:[4.0] professor153:[4.0] professor203:[3.0] professor233:[]	SYA	Univ Calif Berkeley	USA	3.8	19.9	81.3
92	F300c80ce, F300c80ce	M	4	3	professor112:[4.0] professor153:[4.0] professor177:[3.0] professor233:[] professor48:[3.0]	SYA	Dartmouth College	USA	3.5	19.6	69.8
91	10cbe9879, 10cbe9879	M	4	3	professor112:[3.0] professor153:[4.0] professor184:[4.0] professor233:[] professor48:[3.0]	SYA	Univ Of Pennsylvania	USA	3.5	19.4	66.6
90	09b2fe37c, 09b2fe37c	M	3	3	professor153:[4.0] professor170:[] professor4:[4.0] professor57:[3.0]	SYA	Univ Maryland College Pk	USA	3.7	19.4	78.6
89	Fe30082ba, Fe30082ba	M	2	3	professor15:[] professor153:[4.0] professor173:[] professor4:[4.0]	SYA	Seoul National University	KOR	4.0	19.4	79.9
88	7715ff7f4, 7715ff7f4	M	2	3	professor15:[] professor153:[4.0] professor4:[4.0]	SYA	Harvard University	USA	4.0	19.4	87.0
87	F6e50ea4e, F6e50ea4e	M	6	3	professor112:[3.0] professor153:[3.0] professor175:[2.0] professor177:[3.0] professor194:[3.0] professor233:[] professor81:[3.0] professor82:[]	TCS	Metropolitan St Coll	USA	2.8	15.1	56.1

Figure 8: This figure shows the folder-table as viewed by *professor153* when sorted by *professor153*'s personal ordering scores. The red brackets show places in which the global scores increase when scanned from top to bottom (denoting places where *professor153*'s opinion differs from the global one).



Result	Name: Last, First	M/F	Rvs	Ltrs	Readers	Area	College Name	ClgCtry	Score	Personal Ordering	Global Ordering
[illegible]											
71	7715ff7f4, 7715ff7f4	M	2	3	professor15:[] professor153:[4.0] professor4:[4.0]	SYA	Harvard University	USA	4.0	33.0	87.0
70	Af27ac331, Af27ac331	M	3	3	professor14:[4.0] professor15:[] professor173:[] professor183:[] professor194:[4.0] professor4:[4.0]	SYA	Williams College	USA	4.0	30.8	83.4
69	4a0886f6c, 4a0886f6c	M	3	3	professor112:[4.0] professor153:[4.0] professor233:[] professor4:[4.0]	SYA	Univ Illinois Urbana-Champaign	USA	4.0	29.6	83.7
68	60d58c162, 60d58c162	M	3	3	professor112:[3.0] professor153:[3.0] professor233:[] professor4:[4.0]	SYA	Univ Texas Austin	USA	3.3	28.4	68.7
67	C13f14f9a, C13f14f9a	M	2	4	professor153:[3.0] professor4:[4.0]	SYA	MIT	USA	3.5	27.8	69.5
66	Fe30082ba, Fe30082ba	M	2	3	professor15:[] professor153:[4.0] professor173:[] professor4:[4.0]	SYA	Seoul National University	KOR	4.0	27.2	79.9
65	Ac5655a0d, Ac5655a0d	M	4	3	professor112:[4.0] professor153:[3.0] professor233:[] professor4:[4.0] professor75:[4.0]	SYA	Banaras Hindu University	IND	3.8	26.6	67.5
64	68173b206, 68173b206	M	2	3	professor15:[] professor153:[3.0] professor173:[] professor4:[4.0]	SYA	University Of Zagreb	CRO	3.5	24.6	66.0
63	A79a7985b, A79a7985b	M	2	3	professor153:[3.0] professor4:[4.0]	SYA	MIT	USA	3.5	24.1	65.4
62	Ef8ef797b, Ef8ef797b	M	3	3	professor153:[3.0] professor177:[2.0] professor4:[4.0]	SYA	Cornell University	USA	3.0	23.3	51.9
61	2c3984f96, 2c3984f96	M	6	3	professor14:[3.0] professor15:[] professor153:[2.0] professor173:[] professor183:[4.0] professor33:[4.0] professor4:[3.0] professor48:[3.0]	SYA	University Of Moratuwa	LKA	3.2	21.9	54.1
60	1e1d26bae, 1e1d26bae	M	3	3	professor14:[4.0] professor153:[3.0] professor4:[4.0]	SYA	Carthage College	USA	3.7	21.7	63.4
59	058794098, 058794098	M	4	3	professor133:[2.0] professor153:[2.0] professor4:[3.0] professor75:[3.0]	SYA	University Of Mumbai	IND	2.5	18.5	46.3

Figure 9: This figure shows the folder-table as viewed by *professor4* when sorted by *professor4*'s personal ordering scores. The red brackets show places in which the global scores increase when scanned from top to bottom (denoting places where *professor4*'s opinion differs from the global one).

putting '7715ff74f' below the other three, since that applicant received only two 4's as opposed to three. After my additions, the system now has a way of imposing ordering among sets of candidates with the same average 1,2,3,4-scores - namely, the global score. Does the extra layer of ordering make sense in this case? Figure 8 shows *professor153*'s students ordered according to his preferences (his personal scores). We see that he ranked '3309869aa' above '4a0886f6c' (20.6 > 20.2). Similarly, Figure 7, which shows *professor112*'s students sorted by personal score, shows that '3309869aa3' was again ranked above '4a0886f6c' (50.0 > 43.0). *Professor112* and *professor153* are the only two reviewers who reviewed both '3309869aa3' and '4a0886f6c', so we suspend further observations and conclude that it makes sense that '3309869aa' is above '4a0886f6c' in the global rank, with scores of 86.2 and 83.7, respectively. A more thorough examination of the full version of the table from Figure 6 shows that the extra ordering imposed among students with equivalent average 1,2,3,4-score by and large makes sense.

We conclude this section by noting one more thing: the global scores not only help us see that '3309869aa' is "better" than '4a0886f6c' (even though they both have an average 1,2,3,4-score of 4), but they also show us *how large* this difference is. Looking back again to Figure 6, we see that '7715ff74f' and '3309869aa' are quite close (with scores of 87.0 and 86.2, resepctively), while '09b2fe37c' and 'f300c80ce' were judged to be quite far apart (with scores of 78.6 and 69.8, respectively).

The above discussion reveals that ordinal ranking can be helpful in differentiating highly-ranked students. We have yet to collect data on whether ordinal ranking helps EECS faculty make bestter decisions faster, but are hopeful that it can and look forward to collecting more data in future admissions cycles.

6 Future Work

There are two main areas which we believe deserve special attention as this project moves forward: user experience and performance optimization. Regarding the former, we have already begun discussion of improvements and plan to have them implemented before the current academic term ends. These changes include getting rid of the "separator" bar, which some people found confusing and, potentially, changing the semantics of the Personal Ordering.

Regarding performance optimization, we believe the implementation of the ranking module merits investigation. On average, when run locally on my laptop, computation of the global rank takes between ten and twenty minutes. While this is definitely within the acceptable performance range given our expected user scenario (of the global rank being computed by a chair the night before an admissions meeting), we anticipate there are ways to speed it up. For example, using a different data structure in place of the N-matrix or implementing the matrix using a library different from *numpy* might help improve efficiency. However, while these "micro-optimizations" are beneficial in the immediate context of *gradapply*, they will be insufficient if we broaden our amibitions. By this cryptic statement, we mean that such optimizations would be useless when the data starts to become really large. The *gradapply* system has only about 6GB of information total, but in many systems where rank aggregation is used, the amount of data is so big that it cannot fit in memory all at once. Changing our ranking module to handle "big data" probably necessitates more fundamental changes like changing it so that it can be easily parallelized or changing it to work incrementally as data is added. Christina Lee (PhD candidate in EE) is already making headway in this direction under the supervision of Professors Shah and Ozdaglar. She is investigating an alternative method for computing the stationary distribution of a Markov chain by running multiple random walks and getting an empirical estimate of the expected return time, which of course is the inverse of the steady-state probability. This approach is easily parallelizable because the random walks can be run simultaneously on different computers. To read more about her work, see her Master's thesis, *Local Computation of Network Centrality* [Lee, Shah, Ozdaglar: 2013]. I am excited to see her work evolve and hope that it is eventually applied in systems like *gradapply* (or even to *gradapply* itself).

An application of this research which is of particular interest to me is how grading is performed in online education systems. Imagine a scenario where there are millions of students enrolled in a class with only ten teaching assistants. Imagine further, that the assignment is an essay, which of course is difficult to grade in an automated way. One approach to make grading this enormous amount of essays possible is to "crowdsource" the grading. Each student could be tasked with reading a few pairs of essays and picking which one they thought was better for each pair. These act as our PWCs, from which a global rank could be inferred. The teaching assistants could then randomly sample a few students assignments, grade them carefully, then use these grades to infer the grades of other students nearby in the rank. While this example is overly simplified and error prone (in present form), I think the general idea is promising, and leads to even more interesting questions (e.g. can we reconstruct the global ranking reliably even when the grades reported by students are very noisy? etc.).

7 Conclusion

To summarize, our contributions are the following:

1. enhanced *gradapply* by giving reviewers the ability to input partial orderings via a drag-and-drop user interface;
2. enhanced *gradapply* by implementing a ranking module that produces a personal ordering for each reviewer and a department-wide global rank;
3. provided the ranking community with another successful example of a ranking system, which relies on *ordinal* information as opposed to *cardinal* information;
4. provided some insight into the usability and scalability of Negahban's algorithm by implementing it in a real-world system;
5. proposed potential synergies between this application and Lee's alternative algorithm for computing stationary distributions;
6. proposed future work in the area of online education.

Our code will be fully integrated into *gradapply* by Professors Kaashoek and Shah over the summer of 2013, and will hopefully help streamline admissions decisions starting in the 2013-2014 application cycle and onwards.

8 Acknowledgements

I would first and foremost like to thank my thesis advisors, Professors Kaashoek, Shah, and Wornell, for coming up the idea for this project, finding the funding, and trusting me to take the lead on the project. Thank you, Devavrat, for taking me under your wing during 6.UAP my senior year and letting me stay on with you for my MEng. It has been a wonderful, eye-opening experience to be a part of your group, and to participate in the exciting area of rank aggregation. Thank you for explaining Sahand's algorithm to me with patience, no matter how many times I asked for clarification, for all the Skype and telephone calls, and for agreeing to take over leadership of the project when I leave. Thank you, Greg, for your role in the project formulation, for helping me understand how graduate admissions in EECS works, for lending me an ear whenever I wanted to discuss my "life-plans" with you and subsequently offering sage advice, and for inspiring me in 6.437! And thank you, Frans, for adopting me (without any external prompting) as an additional advisee, when you already had so much on your plate. Getting to work with you in a one-on-one setting was a once in a lifetime experience for me - I got to witness firsthand how you think about large scale system design, got detailed feedback from you on my code, and learned to hold myself to higher standards. You encouraged me to think for myself and I will always be grateful for it.

Many other professors deserve my thanks, including my academic advisor, Professor Tsitsiklis, for constant guidance and support, and to Professors Adalsteinsson and Winston for lending me their time and advice as I began the search for a research topic.

I would be remiss if I didn't thank my officemates/LIDS cohorts: Ammar Ammar, Kuang Xu, Christina Lee, Luis Voloch, and Sahand Negahban. Ammar, thank you for getting me started in a good direction, Sahand, for letting me use your algorithm, Christina, for the thought-provoking discussion (I wish we had started earlier!), Luis for the company on numerous "bridge loops", and Kuang for the friendship, the idea of applying this approach to ranking in online education, and for encouraging me after I decided to apply to the PhD program at MIT.

Lastly, my eternal gratitude goes to Albert Wang (S.B. '12, Computer Science), without whom this project never would have been completed. Albert, I don't know what I did to deserve a friend like you. Thank you.

References

[Kohler: 2007] Available: http://www.read.seas.harvard.edu/ kohler/hotcrp/.

[Condorcet: 1785] Condorcet, M. Essai sur l'application de l'analyse a la probabilite des decisions rendues a la pluralite des voix. l'Imprimerie Royale, 1785.

[Arrow: 1963] Arrow, K. *Social Choice and Individual Values.* Yale University Press, 1963.

[Ammar, Shah: 2012] Ammar, A. and Shah, D. (2012). Efficient Rank Aggregation Using Partial Data. Submitted to the 2012 Sigmetrics Conference.

[Braverman, Mossel: 2008] Braverman, M. and Mossel, E. Noisy sorting without resampling. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '08, pages 268-276. Society for Industrial and Applied Mathematics, 2008.

[Negahban, Oh, Shah: 2012] Negahban, S., Oh, S., Shah, D. Iterative Ranking from Pair-wise Comparisons http://arxiv.org/pdf/1209.1688v1.pdf.

[Shah: 2009] Shah, D. Gossip Algorithms. Foundations and Trends in Networking, Vol. 3, No. 1 (2008) 1-125.

[Lee, Shah, Ozdaglar: 2013] Lee, C., Shah, D., and Ozdaglar, A. Local Computation of Network Centrality. To be submitted to the MIT Department of EECS in partial fulfillment of the requirements for the degree of Master of Electrical Engineering, June 2013.