

## An Effective Strategy for Dynamic Mapping of Peer Reviewers

Edward F. Gehringer

Yun Cui

North Carolina State University

{efg@ncsu.edu, ycui4@unity.ncsu.edu}

### Abstract

Although much work has been done on peer review in an academic setting, virtually no work has been done on strategies for mapping reviewers. In most cases, an instructor merely collects papers, shuffles them, and passes them out to students for peer review. When peer review is electronic and asynchronous, better strategies are needed. A random mapping, arranged statically in advance, is frustrated by students who drop the course during the assignment period, or who do not submit their assignments. The result is that some students get reviewed by more reviewers than other students, and some students do not get reviewed at all.

We have developed a dynamic strategy that adapts to situations like students dropping the course or not doing their assigned reviews, to ensure that all students' work is reviewed by approximately the same number of other students, and that no students are assigned to review their own work. The strategy is very similar to the Banker's Algorithm for allocating resources in a computer system. It assigns reviewers in such a way as to maximize the chance that all future mapping assignments will be able to be assigned successfully (i.e., without assigning a student to review him/herself). The strategy has been extended to second-level reviews—that is, where one student reviews another student's *review*, to give students an incentive to do a careful job of reviewing. It has also been extended to the situation where team projects are reviewed by individual students. This strategy offers great advantages for peer review of student work, and opportunity for extension to other problems like asynchronous review periods, where students themselves negotiate review deadlines.

### 1. Introduction

Peer review in the classroom is a technique that is becoming increasingly popular, with over 100 papers published on the topic in the past ten years. Much work has been performed on assessing usefulness of the technique (students generally like it, and learn well from it) and its validity (students do in general rate better work more highly, though some effort needs to be invested in the assessment procedure to assure this). However, very few published reports discuss appropriate strategies for matching reviewers with reviewees. In his 1998 survey paper [Topp 98], Topping says, "How peer assessors and assessees should best be matched ... is discussed surprisingly little in the literature." In most cases, he says, a single assessor was matched with an assessee. Most papers omit entirely any indication of how reviewers are chosen; others just say they are matched "randomly." In other cases, multiple assessors were used.

Random matching is relatively easy to do. In an offline environment where papers are collected at the start of a class period and then shuffled, it can be guaranteed that each student will get a paper to review, and each student will be reviewed by another student. When review takes place

outside the classroom, or even when papers are collected in one class period and distributed in the next [KPD 95], we have no such guarantee. Some students who are assigned to do reviews will not do them, either because they just don't do their homework, or because they drop the course. Consequently, some students will fail to receive the requisite number of reviews. Conversely, some students will not submit their homework, so the reviewers who are assigned to them will have nothing to do. In a small class, this is not a major problem, as the instructor can reassign reviews that were to be done by students who don't do them. In a large class, it is a major administrative overhead. Further, if grades are being assigned based on these reviews, the instructor or TA will need to do reviews that students don't.

These considerations led us to reject a "static" strategy of mapping reviewers to reviewees that is, a strategy that assigns all reviewers before any reviews are done. Instead, we investigated "dynamic" strategies that wait until a student asks to do a review before assigning a submission for him/her to review. Using a dynamic strategy, no student will be assigned to review work that has not been submitted. Conversely, a student who has submitted the work will be assigned a "live" reviewer.

Unfortunately, such strategies are prone to "paint themselves into a corner," where the last student to do a review must review him/herself, all other students already having been reviewed by others. The problem becomes more acute when multiple levels of review are used. For example, in our PG system [Gehr 99], the *reviews* done by students are themselves reviewed by other students, to give students an incentive to submit good reviews. In this case, no student may review his own work, nor may (s)he review her own review. A mapping that satisfies these constraints is called a "valid" mapping.

## 2. Overview of our strategy

The basic idea of this algorithm is similar to the banker's algorithm [SGG 01] used for resource allocation in operating systems.

**Definition 1.** A mapping assignment of a reviewer to a reviewee is *valid* if it does not cause a reviewer to review his/her own submission, and leaves sufficient valid mapping assignments for all future reviewers.

**Definition 2.** A mapping assignment is *invalid* either if it does not leave enough valid mapping assignments for all future reviewers, or it causes the reviewer to review his/her own submission.

Assuming that all previous mapping assignments are valid, our algorithm allows us always to find a valid mapping assignment for the next reviewer. We keep on doing this until all the mapping assignments are done.

## 3. Outline of the Basic Algorithm

First, let's consider the case of assigning reviewers for each author's submission. The goal is that none of the reviewers will review their own submission.

Let's assume that there are  $n$  total students, and every original submission needs  $r$  reviewers (this is the same as saying that every reviewer needs to review  $r$  original submissions). We also assume that  $r$  is less than  $n$ . We index the set of students from 1 to  $n$ .

An easy way to understand this algorithm is to use a matrix. Each row is for a particular reviewer, and each column is for a particular submission. Let's say  $review[1:n][1:n]$  is the matrix. Each value in the matrix shows whether the mapping assignment of a particular reviewer to reviewee would be valid or invalid. Let's use "1" to designate a valid mapping assignment that has already been assigned, "0" to mean an invalid mapping assignment and "-1" as a candidate valid mapping assignment that could be made but has not been mapped yet. For example,

- $review[i][j] = 1$  means that student  $j$ 's original submission has been assigned to student  $i$  to review;
- $review[i][j] = 0$  means that student  $j$ 's original submission cannot be assigned to student  $i$ , and
- $review[i][j] = -1$  means that student  $j$ 's original submission can be assigned to student  $i$ , but we have not assigned this mapping assignment yet.

The basic logic of this algorithm is to prevent the number of "1"s from becoming greater than  $r$  in any row or column. Otherwise it would be an invalid mapping assignment. Similarly, the number of "0"s cannot be greater than  $(n-r)$  in any row or column. If the number of "1"s has reached the limit  $r$ , no more mapping assignment are needed for this reviewer or for this submission. So, the remaining elements in the same row or column that are "-1" should now be set to "0". Similarly, if the number of "0"s has reached the limit  $(n-r)$ , the remaining elements in the same row or column that are "-1" need to be set to "1".

We should remember that changing an element in a matrix affects both a row and a column. So, after we change a value in a row, we need to check the corresponding column and vice versa. A change to one matrix element may leave only one valid choice in another row or column. So, after this algorithm sets a value in a row (for a specific reviewer), it checks the corresponding column and makes a change if necessary. After it changes a value in a column, it checks whether it is still valid for its corresponding row and makes a change if necessary. If the algorithm cannot make the required change without violating some other constraint, then it "backs out," by undoing changes already made pursuant to this mapping assignment, and choosing another of the possible mapping assignments.

Since the student cannot be mapped to his own original submission, we can set "0"s on the matrix's diagonal. The other elements in the matrix are all initialized to "-1".

**Definition 3.** A mapping is *valid* if the number of "1"s in each row and column is not greater than  $r$ , and the number of "0"s in each row and column is not greater than  $(n-r)$ , and if  $review[i][i]=0$  (for  $i = 1, 2, \dots, n$ ).

Assuming that the number of reviewers per student  $r$  is  $\leq n$ , one valid mapping is to map student  $k$  to student  $(k+1), (k+2), \dots, (k+r) \bmod n$ . We will prove that there is a legal mapping

assignment for the  $i^{\text{th}}$  student, assuming that the  $(i-1)^{\text{st}}$  student's mapping assignment has been done successfully ( $i = 2, \dots, n-1, n$ ).

When student  $i$  asks to review, we search the  $i^{\text{th}}$  row and select an element that has the value “-1”. We then check whether the submission corresponding to this element has been submitted; if it has not been submitted, we make another choice. Let's say we select  $review[i][j]$ . There are two types of check we will do after we set  $review[i][j] = 1$ .

- First, we count the number of “1”s in this row. If it has reached the limit  $r$ , we need to set all the remaining “-1” elements to “0”. For each “-1” element (in column  $j'$ ) set to 0 in this row, we check the  $j'^{\text{th}}$  column. Since the number of “1”s has not been changed, we only check the number of “0”s in the  $j'^{\text{th}}$  column. If it has reached the limit  $(n-r)$ , the remaining “-1” elements will be set to “1”.
- The second type of check is to check the  $j^{\text{th}}$  column. Since we have set  $review[i][j] = 1$ , the number of “0”s has not been changed, a check of the number of “1”s is enough. If the number of “1”s in the  $j^{\text{th}}$  column is still less than  $r$ , it is valid. If the number of “1”s in the  $j^{\text{th}}$  column has already reached the limit  $r$  (this means the original submission from student  $j$  has enough reviewers), further action is needed. First, since no more reviewers are needed for this original submission from student  $j$ , we need to change all the remaining “-1”s in the  $j^{\text{th}}$  column to “0”s. Let's say the row number of such an element is  $i'$ . Then, we check the  $i'^{\text{th}}$  row to see whether it is still a valid mapping assignment. Here only the number of “0”s is changed. So we check whether the number of “0”s has exceeded  $(n-r)$ .

During this process, we will keep track of the changed elements. If all of the above checks are valid, then the mapping assignment  $review[i][j] = 1$  is valid. If any of the above checks fails, then our current mapping assignment  $review[i][j] = 1$  is invalid. We need to set  $review[i][j] = 0$ , restore all the changed elements back to “-1” and try another element in the  $i^{\text{th}}$  row.

Example 1 shows how this process takes place. It might be helpful to peruse it now, and again when studying the algorithm presented in Section 5.

Keeping in mind that a student may drop the course during the submission and review process, it is impossible to guarantee that all submissions are assigned to “live” reviewers and that each reviewer has something to do. But we try to maximize the possibility that a submission has as many “live” reviewers as possible and a reviewer has as many “live” submissions as possible. To do this, the algorithm keeps track of how many reviewers we have assigned to each submission, that is, the number of “1”s in each column. When we want to select a submission for a reviewer, we select the submission that has had the fewest reviewers assigned up to now.

#### 4. Mapping for reviews of reviews

Now, let's consider the situation where a student is to be assigned to review another student's review. The student must not review his own review or the review of his own original submission. Because of this, the author/reviewer mapping assignments from the previous stage are needed to do this mapping assignment.

### Example 1: How the reviewer-mapping algorithm works

Suppose there are four students, and each student is assigned two reviews; that is,  $n=4$  and  $r=2$ . To make the example easy to understand, we assume that all submissions are valid and we assign one review to a student when that particular student requests to do a review.

We assume that students request to do reviews in the sequence 0, 2, 3, 1; that is, student 0 requests to do the review first, student 2 requests second, student 3 requests third and student 1 requests last.

Initially, the matrix *review* is as shown at the right.

$$\begin{array}{l}
 \begin{array}{c} \text{Student doing reviewing} \\ \text{Student being reviewed} \end{array}
 \begin{array}{c} \text{review} \\ \text{zeros} \end{array}
 \begin{array}{c} \text{row}_- \\ \text{ones} \end{array} \\
 \begin{bmatrix} 0 & \mathbf{1} & -1 & -1 \\ -1 & 0 & -1 & -1 \\ -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}
 \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}
 \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \\
 \text{col\_zeros} \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \\
 \text{col\_ones} \begin{bmatrix} 0 & \mathbf{1} & 0 & 0 \end{bmatrix}
 \end{array}$$

**Second request:** Then, let's assume that student 0 asks to do another review. We pick student 2, and set  $review[0][2] = 1$ .

$$\begin{array}{l}
 \begin{array}{c} \text{Student doing reviewing} \\ \text{Student being reviewed} \end{array}
 \begin{array}{c} \text{review} \\ \text{zeros} \end{array}
 \begin{array}{c} \text{row}_- \\ \text{ones} \end{array} \\
 \begin{bmatrix} 0 & 1 & \mathbf{1} & \mathbf{0} \\ -1 & 0 & -1 & -1 \\ -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}
 \begin{bmatrix} \mathbf{2} \\ 1 \\ 1 \\ 1 \end{bmatrix}
 \begin{bmatrix} \mathbf{2} \\ 0 \\ 0 \\ 0 \end{bmatrix} \\
 \text{col\_zeros} \begin{bmatrix} 1 & 1 & 1 & \mathbf{2} \end{bmatrix} \\
 \text{col\_ones} \begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix}
 \end{array}$$

$$\begin{array}{l}
 \begin{array}{c} \text{Student doing reviewing} \\ \text{Student being reviewed} \end{array}
 \begin{array}{c} \text{review} \\ \text{zeros} \end{array}
 \begin{array}{c} \text{row}_- \\ \text{ones} \end{array} \\
 \begin{bmatrix} 0 & -1 & -1 & -1 \\ -1 & 0 & -1 & -1 \\ -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}
 \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}
 \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \\
 \text{col\_zeros} \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \\
 \text{col\_ones} \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}
 \end{array}$$

**First request:** When student 0 requests to do a review, we randomly pick one of the  $-1$ s in row 0 of the *review* matrix. Let's say we pick  $review[0][1]$ . This assigns student 0 to review student 1. Having done this, we set  $review[0][1] = 1$ .

Values in *row\_zeros* and *col\_zeros* change appropriately.

In the diagram to the left, values that have changed are shown in **boldface**.

$$\begin{array}{l}
 \begin{array}{c} \text{Student doing reviewing} \\ \text{Student being reviewed} \end{array}
 \begin{array}{c} \text{review} \\ \text{zeros} \end{array}
 \begin{array}{c} \text{row}_- \\ \text{ones} \end{array} \\
 \begin{bmatrix} 0 & 1 & \mathbf{1} & -1 \\ -1 & 0 & -1 & -1 \\ -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}
 \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}
 \begin{bmatrix} \mathbf{2} \\ 0 \\ 0 \\ 0 \end{bmatrix} \\
 \text{col\_zeros} \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \\
 \text{col\_ones} \begin{bmatrix} 0 & 1 & \mathbf{1} & 0 \end{bmatrix}
 \end{array}$$

Now, the constraint vectors change to  $row\_ones[0] = 2$ ,  $col\_ones[2] = 1$ .

Since  $row\_ones[0] = n - r = 2$ , student 0 cannot review anyone other than his current reviewees, and thus  $review[0][3] = 0$  is set.

Example 1 (cont.): How the reviewer-mapping algorithm works

$$\begin{array}{l}
 \text{Student doing reviewing} \\
 \begin{array}{c}
 \text{review} \\
 \text{Student being reviewed}
 \end{array} \\
 \begin{bmatrix} 0 & 1 & 1 & 0 \\ -1 & 0 & -1 & \mathbf{1} \\ -1 & -1 & 0 & \mathbf{1} \\ -1 & -1 & -1 & 0 \end{bmatrix} \\
 \begin{array}{c}
 \text{row\_} \\
 \text{zeros} \\
 \text{row\_} \\
 \text{ones}
 \end{array} \\
 \begin{bmatrix} 2 \\ 1 \\ 1 \\ 1 \end{bmatrix} \\
 \begin{bmatrix} 2 \\ 1 \\ 1 \\ 0 \end{bmatrix} \\
 \text{col\_zeros} \begin{bmatrix} 1 & 1 & 1 & 2 \end{bmatrix} \\
 \text{col\_ones} \begin{bmatrix} 0 & 1 & 1 & \mathbf{2} \end{bmatrix}
 \end{array}$$

Now, another boundary condition is met, that is  $col\_zeros[3] = 2$ . Two of the four students are now ineligible to review student 3, so the other two students must review this student. We set  $review[1][3] = 1$  and  $review[2][3] = 1$ .

**Third request:** Similarly, when student 2 requests, we randomly select student 0 as the reviewee and set the matrix *review* as follows.

Now student 2 has been assigned to do two reviews, so we know that (s)he can review no other students.

$$\begin{array}{l}
 \text{Student doing reviewing} \\
 \begin{array}{c}
 \text{review} \\
 \text{Student being reviewed}
 \end{array} \\
 \begin{bmatrix} 0 & 1 & 1 & 0 \\ -1 & 0 & -1 & 1 \\ 1 & \mathbf{0} & 0 & 1 \\ -1 & -1 & -1 & 0 \end{bmatrix} \\
 \begin{array}{c}
 \text{row\_} \\
 \text{zeros} \\
 \text{row\_} \\
 \text{ones}
 \end{array} \\
 \begin{bmatrix} 2 \\ 1 \\ \mathbf{2} \\ 1 \end{bmatrix} \\
 \begin{bmatrix} 2 \\ 1 \\ 2 \\ 0 \end{bmatrix} \\
 \text{col\_zeros} \begin{bmatrix} 1 & \mathbf{2} & 1 & 2 \end{bmatrix} \\
 \text{col\_ones} \begin{bmatrix} 1 & 1 & 1 & 2 \end{bmatrix}
 \end{array}$$

$$\begin{array}{l}
 \text{Student doing reviewing} \\
 \begin{array}{c}
 \text{review} \\
 \text{Student being reviewed}
 \end{array} \\
 \begin{bmatrix} 0 & 1 & 1 & 0 \\ -1 & 0 & -1 & 1 \\ \mathbf{1} & -1 & 0 & 1 \\ -1 & -1 & -1 & 0 \end{bmatrix} \\
 \begin{array}{c}
 \text{row\_} \\
 \text{zeros} \\
 \text{row\_} \\
 \text{ones}
 \end{array} \\
 \begin{bmatrix} 2 \\ 1 \\ 1 \\ 1 \end{bmatrix} \\
 \begin{bmatrix} 2 \\ 1 \\ \mathbf{2} \\ 0 \end{bmatrix} \\
 \text{col\_zeros} \begin{bmatrix} 1 & 1 & 1 & 2 \end{bmatrix} \\
 \text{col\_ones} \begin{bmatrix} \mathbf{1} & 1 & 1 & 2 \end{bmatrix}
 \end{array}$$

Thus we set  $review[2][1]$  to 0.

But this means that student 1 cannot be reviewed by students 1 or 2, so (s)he must be reviewed by student 3. So the following change is made.

$$\begin{array}{l}
 \text{Student doing reviewing} \\
 \begin{array}{c}
 \text{review} \\
 \text{Student being reviewed}
 \end{array} \\
 \begin{bmatrix} 0 & 1 & 1 & 0 \\ -1 & 0 & -1 & 1 \\ 1 & 0 & 0 & 1 \\ -1 & \mathbf{1} & -1 & 0 \end{bmatrix} \\
 \begin{array}{c}
 \text{row\_} \\
 \text{zeros} \\
 \text{row\_} \\
 \text{ones}
 \end{array} \\
 \begin{bmatrix} 2 \\ 1 \\ 2 \\ 1 \end{bmatrix} \\
 \begin{bmatrix} 2 \\ 1 \\ 2 \\ \mathbf{1} \end{bmatrix} \\
 \text{col\_zeros} \begin{bmatrix} 1 & 2 & 1 & 2 \end{bmatrix} \\
 \text{col\_ones} \begin{bmatrix} 1 & \mathbf{2} & 1 & 2 \end{bmatrix}
 \end{array}$$

Example 1 (cont.): How the reviewer-mapping algorithm works

**Fourth request:** Student 3 now requests to do a review. The system randomly selects student 0 to be reviewed.

$$\begin{array}{l}
 \begin{array}{c} \text{Student doing reviewing} \\ \text{Student being reviewed} \end{array}
 \begin{array}{c} \text{review} \\ \text{Student being reviewed} \end{array}
 \begin{array}{c} \text{row\_} \\ \text{zeros} \end{array}
 \begin{array}{c} \text{row\_} \\ \text{ones} \end{array} \\
 \begin{bmatrix} 0 & 1 & 1 & 0 \\ -1 & 0 & -1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & -1 & 0 \end{bmatrix}
 \begin{bmatrix} 2 \\ 1 \\ 2 \\ 1 \end{bmatrix}
 \begin{bmatrix} 2 \\ 1 \\ 2 \\ 2 \end{bmatrix} \\
 \text{col\_zeros} \begin{bmatrix} 1 & 2 & 1 & 2 \end{bmatrix} \\
 \text{col\_ones} \begin{bmatrix} 2 & 2 & 1 & 2 \end{bmatrix}
 \end{array}$$

$$\begin{array}{l}
 \begin{array}{c} \text{Student doing reviewing} \\ \text{Student being reviewed} \end{array}
 \begin{array}{c} \text{review} \\ \text{Student being reviewed} \end{array}
 \begin{array}{c} \text{row\_} \\ \text{zeros} \end{array}
 \begin{array}{c} \text{row\_} \\ \text{ones} \end{array} \\
 \begin{bmatrix} 0 & 1 & 1 & 0 \\ -1 & 0 & -1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & \mathbf{0} & 0 \end{bmatrix}
 \begin{bmatrix} 2 \\ 1 \\ 2 \\ \mathbf{2} \end{bmatrix}
 \begin{bmatrix} 2 \\ 1 \\ 2 \\ 2 \end{bmatrix} \\
 \text{col\_zeros} \begin{bmatrix} 1 & 2 & \mathbf{2} & 2 \end{bmatrix} \\
 \text{col\_ones} \begin{bmatrix} 2 & 2 & 1 & 2 \end{bmatrix}
 \end{array}$$

Now student 3 has been assigned to review students 0 and 1. So (s)he cannot be assigned to review student 2. Therefore, a 0 is placed in  $review[3][2]$ .

But this means that two students (students 2 and 3) cannot review student 2, so the remaining student, that is, student 1, must review him.

$$\begin{array}{l}
 \begin{array}{c} \text{Student doing reviewing} \\ \text{Student being reviewed} \end{array}
 \begin{array}{c} \text{review} \\ \text{Student being reviewed} \end{array}
 \begin{array}{c} \text{row\_} \\ \text{zeros} \end{array}
 \begin{array}{c} \text{row\_} \\ \text{ones} \end{array} \\
 \begin{bmatrix} 0 & 1 & 1 & 0 \\ -1 & 0 & \mathbf{1} & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}
 \begin{bmatrix} 2 \\ 1 \\ 2 \\ 2 \end{bmatrix}
 \begin{bmatrix} 2 \\ 2 \\ 2 \\ 2 \end{bmatrix} \\
 \text{col\_zeros} \begin{bmatrix} 1 & 2 & 2 & 2 \end{bmatrix} \\
 \text{col\_ones} \begin{bmatrix} 2 & 2 & \mathbf{2} & 2 \end{bmatrix}
 \end{array}$$

$$\begin{array}{l}
 \begin{array}{c} \text{Student doing reviewing} \\ \text{Student being reviewed} \end{array}
 \begin{array}{c} \text{review} \\ \text{Student being reviewed} \end{array}
 \begin{array}{c} \text{row\_} \\ \text{zeros} \end{array}
 \begin{array}{c} \text{row\_} \\ \text{ones} \end{array} \\
 \begin{bmatrix} 0 & 1 & 1 & 0 \\ \mathbf{0} & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}
 \begin{bmatrix} 2 \\ \mathbf{2} \\ 2 \\ 2 \end{bmatrix}
 \begin{bmatrix} 2 \\ 2 \\ 2 \\ 2 \end{bmatrix} \\
 \text{col\_zeros} \begin{bmatrix} 2 & 2 & 2 & 2 \end{bmatrix} \\
 \text{col\_ones} \begin{bmatrix} 2 & 2 & 2 & 2 \end{bmatrix}
 \end{array}$$

Now student 1 has two review assignments, so (s)he cannot review anyone else. The algorithm therefore sets  $review[1][0]$  to 0.

Thus, even before the last student (student 1) makes a request, (s)he has already been assigned two students to review (students 2 and 3), since these are the only two valid mappings left.

The basic idea of the algorithm is the same as above. The difference is that we are using a 3-dimensional matrix (the  $x$ -axis is for reviews of reviews, the  $y$ -axis is for the reviews, and the  $z$ -axis is for the original submissions). Let's call this matrix  $ror[1:n][1:n][1:r]$ . But it will be much easier to think it as a 2-dimensional matrix with vector elements ( $x$ -axis for reviews of reviews and the  $y$ -axis is for students who has done review of original submissions). Each element in this array is an  $r$ -element vector, where  $r$  is the number of reviews each original submission has. As in the previous algorithm, we use "1" to denote a valid mapping assignment that has already been mapped, "0" as an invalid mapping assignment and "-1" to denote possible valid mapping assignments that have not been assigned. For example,

- $ror[i][j][k]=1$  means the  $k^{\text{th}}$  review that student  $j$  did has been assigned to student  $i$ ;
- $ror[i][j][k]=0$  means the  $k^{\text{th}}$  review that student  $j$  did cannot be assigned to student  $i$ , and
- $ror[i][j][k]=-1$  means the  $k^{\text{th}}$  review that student  $j$  did can be assigned to student  $i$ , but we have not assigned this mapping assignment yet.

The important part of this algorithm is the initialization. It consists of two steps. The first step is to prevent a student from reviewing his/her own review, and the second step is to prevent the student from reviewing a review of his/her own submission. The previous mapping assignment information between review and original submission is needed. Let's say we can get it from the matrix  $review[1:n][1:n]$ . Here is the way we do it.

```

for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    if (i==j) // student cannot review his own review
      for (k=0; k<r; k++)
         $ror[i][j][k] = 0;$ 
    else {
      k=0;
      for (m=0; m<n; m++) {
        if (review[j][m] == 1) {
          // student j reviews student m's original submission
          if (m==i)
             $ror[i][j][k] = 0;$ 
          else  $ror[i][j][k] = -1;$ 
          k++;
        }
      }
    }
  }

```

After this part is done, the remaining part is quite similar to the previous algorithm; we just control of number of "1"s and number of "0"s in each row and each column.

## 5. Mapping Algorithm for Reviews

Now we proceed to give the realization of the algorithm outlined in Section 3. We begin with a list of important variables.

- *review*[][]: the basic matrix used to hold the mapping;
- *row\_zeros*[]): keeps track of the number of "0"s in each row;
- *row\_ones*[]): keeps track of the number of "1"s in each row;
- *col\_zeros*[]): keeps track of the number of "0"s in each column;
- *col\_ones*[]): keeps track of the number of "1"s in each column;
- *cells\_changed*[]): keeps track of the elements which have been changed during an iteration; each one is a structure to keep track of the position of the original element. It records the *row\_id* and *col\_id* of the element;
- *num\_changed*: the total number of elements that have been changed during one iteration;

Here is the algorithm for assigning student *i* to do a review.

```

initialize the matrix review[][] with information on the mapping assignments already
  assigned;
calculate initial values for row_zeros[], row_ones[], col_zeros[] and col_ones[] based on
  review[][];
back up this row (review[i][*]);
for each j such that review[i][j] == "-1" {
  back up this column (review[*][j]);
  back up row_zeros[], row_ones[], col_zeros[], col_ones[];
  num_changed=0;
  review[i][j] = 1;
  row_ones[i]++;
  col_ones[j]++;
  if (col_ones[j] == r)
    {check_col(j)}; // Would it be impossible for i to review submission j?
  if check_col(j) returned "successful" && row_ones[i] == r {
    // Back up the cells which may need to be restored later if this choice is invalid
    for each remaining "-1" in this row (say its column id is j') {
      review[i][j'] = 0;
      row_zeros[i]++;
      col_zeros[j']++;
      // If we do this assignment, will it force some student to do too many reviews?
      if col_zeros[j'] == (n - r) {
        if check_zeros_col(j') is "unsuccessful", then break; // and try
        // another value for j
      }
    }
  }
}
if any of the above checks (check_col(j) or check_zeros_col(j')) was "unsuccessful" {
  restore the backed-up row and backed-up col
  according to the row_id and col_id in the cells_changed[], set these elements to
  "-1";
  restore row_zeros[], row_ones[], col_zeros[] and col_ones[];
}

```

```

        review[i][j] = 0;
        row_zeros[i]++;
        col_zeros[j]++;
    } else break; // We've found a reviewee j.
}

```

Here are the four subroutines used by the above procedure:

```

check_row(int i) {
// Precondition: This reviewer has enough submissions to review. So, set remaining "-1"s to 0s
    for each remaining "-1" in this row (say its column ID is j) {
        cells_changed[num_changed].row_id = i;
        cells_changed[num_changed].col_id = j;
        num_changed++;
        review[i][j] = 0;
        row_zeros[i]++;
        col_zeros[j]++;
        if (col_zeros[j] > (n - r)) return "unsuccessful";
        else if (col_zeros[j] == (n - r))
            // Will this cause a reviewer to do too many reviews?
            check_zero_col(j);
            if check_zero_col(j) returns "unsuccessful"
                return "unsuccessful";
    }
    return "successful"
}

```

```

check_zeros_row(int i) {
// Precondition: This reviewer already has (n-r) invalid mapping assignments; set remaining
// "-1"s to "1"s
    for each remaining "-1" in this row (say column ID is j) {
        cells_changed[num_changed].row_id = i;
        cells_changed[num_changed].col_id = j;
        num_changed++;
        review[i][j] = 1;
        row_ones[i]++;
        col_ones[j]++;
        if (col_ones[j] > r) return "unsuccessful";
        else if (col_ones[j] == r)
            // Will this cause a student to do too few reviews?
            check_col(j);
            if check_col(j) returns "unsuccessful"
                return "unsuccessful";
    }
    return "successful"
}

```

```

check_col(int j) {
// Precondition: All reviewers for this submission have been mapped; change remaining -1s to 0s
  for each remaining "-1" in this column (say row ID is i) {
    cells_changed[num_changed].row_id = i;
    cells_changed[num_changed].col_id = j;
    num_changed++;
    review[i][j] = 0;
    row_zeros[i]++;
    col_zeros[j]++;
    if (row_zeros[i] > (n - r)) return "unsuccessful";
    else if (row_zeros[i] == (n - r))
      // Will this cause a submission to have too many reviews?
      check_zeros_row(i);
      if check_zeros_row(i) returns "unsuccessful"
        return "unsuccessful";
  }
  return "successful"
}

```

```

check_zeros_col(int j) {
// Precondition: This reviewee already has (n-r) impossible choices, & so must be reviewed by
// all the rest
  for each remaining "-1" in this column (say row ID is i) {
    cells_changed[num_changed].row_id = i;
    cells_changed[num_changed].col_id = j;
    num_changed++;
    review[i][j] = 1;
    row_ones[i]++;
    col_ones[j]++;
    if (row_ones[i] > r) return "unsuccessful";
    else if (row_ones[i] == r)
      // Will this cause a submission to be reviewed by too few students?
      check_row(i);
      if check_row(i) returns "unsuccessful"
        return "unsuccessful";
  }
  return "successful"
}

```

## 6. Mapping Algorithm for Reviews of Reviews

Finally, we give the algorithm for mapping reviews of reviews. Here are some important variables used in the algorithm:

- *review*[][]: the mapping between the review and the original work; *review*[*i*][*j*] = 1 means that student *i* reviewed student *j*; this matrix has been created by the algorithm in the previous section.
- *ror*[][][]: each element of this matrix is an array (we have introduced this 3-dimensional matrix in section 4);
- *row\_zeros*[]): keeps track of the number of "0"s in each row;
- *row\_ones*[]): keeps track of the number of "1"s in each row;
- *col\_zeros*[]): keeps track of the number of "0"s in each column;
- *col\_ones*[]): keeps track of the number of "1"s in each column;
- *cells\_changed*[]): keeps track of the elements that have been changed during an iteration, each one is a structure to keep the position of the original element. It records the *row\_id*, *col\_id* and *vector\_id* of the element;
- *num\_changed*: the total number of elements which have been changed during an iteration;

Here is the algorithm. Let's assume that each student needs to do *rr* review of reviews.

```

When a student asks to do a review of review,
// say his/her index in the matrix is i, i.e., the ith row of the matrix contains information about
// his/her reviewer mapping assignments
count the number of reviews already mapped to this student to do review of review;
if rr reviews have already been mapped to this student, then return;
// this student's mapping is already done

initialize the matrix ror[][][], as described in section 4;
// set the elements to 0, 1 and -1
for all i
    calculate the initial values to row_zeros[i], row_ones[i] based on ror[i][*][*];
    // i.e., add up all of the "0"s or "1"s in the other two dimensions.
for all j
    calculate the initial values for col_zeros[j] and col_ones[j] based on ror[*][j][*];
back up this row (ror[i])
for each pair of (j, k) such that ror[i][j][k] == -1
// need to check whether this review has been submitted. If not, select another "-1"
    back up this column, ror[*][j][*]
    back up row_zeros[], row_ones[], col_zeros[], col_ones[];
    num_changed=0;
    ror[i][j][k] = 1;
    row_ones[i]++;
    col_ones[j]++;
    if (col_ones[j] == rr)
        check_col(j);
    if check_col(j) returns "successful" && row_ones[i] == rr {
        // this reviewer has enough reviews to review. So, set remaining "-1"s to "0"s
        // to back up the cells which may need to be restored later if this mapping is invalid

```

```

for each remaining "-1" in this row (say column id is  $j'$   $k'$ ) {
     $ror[i][j'] [k'] = 0;$ 
     $row\_zeros[i]++;$ 
     $col\_zeros[j']++;$ 
    if  $col\_zeros[j'] == (n - rr)$  {
         $check\_zeros\_col(j');$ 
        if  $check\_zeros\_col(j')$  is "unsuccessful"
            break; // and try another pair of  $(j,k)$ 
    }
}
}
}
if any of the above checks ( $check\_col(j)$  or  $check\_zeros\_col(j')$ ) was "unsuccessful" {
    restore the backed-up row and backed-up col
    according to the  $row\_id$ ,  $col\_id$  and  $vector\_id$  in the  $cells\_changed[]$ , set these
    elements to -1;
    restore  $row\_zeros[]$ ,  $row\_ones[]$ ,  $col\_zeros[]$  and  $col\_ones[]$ ;
     $ror[i][j][k] = 0;$ 
     $row\_zeros[i]++;$ 
     $col\_zeros[j]++;$ 
}
}

```

Here are the four subroutines used by the above procedure:

```

 $check\_row(int i)$  {
// Precondition: This reviewer has enough reviews to review. So, set remaining "-1"s to "0"s
    for each remaining "-1" in this row (say column ID is  $j$ , vector ID is  $k$ ) {
         $cells\_changed[num\_changed].row\_id = i;$ 
         $cells\_changed[num\_changed].col\_id = j;$ 
         $cells\_changed[num\_changed].vector\_id = k;$ 
         $num\_changed++;$ 
         $ror[i][j][k] = 0;$ 
         $row\_zeros[i]++;$ 
         $col\_zeros[j]++;$ 
        if  $(col\_zeros[j] > (n - rr))$  return "unsuccessful";
        else if  $(col\_zeros[j] == (n - rr))$  {
            // Will this cause a student to do too many reviews?
             $check\_zero\_col(j);$ 
            if  $check\_zero\_col(j)$  is "unsuccessful";
                return "unsuccessful";
        }
    }
}
return "successful"
}

```

```

check_zeros_row(int i) {
// Precondition: This reviewer has (n - rr) invalid mapping assignments. So, set remaining "-1"
// to "1"s
    for each remaining "-1" in this row (say column ID is j, vector ID is k) {
        cells_changed[num_changed].row_id = i;
        cells_changed[num_changed].col_id = j;
        cell_changed[num_changed].vector_id = k;
        num_changed++;
        ror[i][j][k] = 1;
        row_ones[i]++;
        col_ones[j]++;
        if (col_ones[j] > rr) return "unsuccessful";
        else if (col_ones[j] == rr) {
            // Will this cause a student to do too few reviews?
            check_col(j);
            if check_col(j) is "unsuccessful"
                return "unsuccessful";
        }
    }
    return "successful"
}

```

```

check_col(int j) {
// Precondition: All reviewers for this submission have been mapped. So, set the remaining "-1"s
// to "0"s
    for each remaining "-1" in this column (say row ID is i, vector ID is k) {
        cells_changed[num_changed].row_id = i;
        cells_changed[num_changed].col_id = j;
        cells_changed[num_changed].vector_id = k;
        num_changed++;
        ror[i][j][k] = 0;
        row_zeros[i]++;
        col_zeros[j]++;
        if (row_zeros[i] > (n - rr) ) return "unsuccessful";
        else if (row_zeros[i] == (n - rr) ) {
            // Will this cause a review to be reviewed by too many students?
            check_zero_row(i);
            if check_zero_row(i) is "unsuccessful"
                return "unsuccessful";
        }
    }
    return "successful"
}

```

```

check_zeros_col(int j) {
// Precondition: The review already has (n-rr) invalid mapping assignments. So, must be
//reviewed by all the rest
  for each remaining "-1" in this column (say row ID is i, array vector ID is k) {
    cells_changed[num_changed].row_id = i;
    cells_changed[num_changed].col_id = j;
    cell_changed[num_changed].vector_id = k;
    num_changed++;
    ror[i][j][k] = 1;
    row_ones[i]++;
    col_ones[j]++;
    if (row_ones[i] > rr) return "unsuccessful";
    else if (row_ones[i] == rr) {
      // Will this cause a review to be reviewed by too few students?
      check_row(i);
      if is "unsuccessful"
        return "unsuccessful";
    }
  }
}
return "successful"
}

```

## 7. Future work

The algorithms we have presented in this paper are capable of producing valid mappings for reviewers, and reviewers of reviews, dynamically. However, in special situations they fail to produce valid mappings. If a student drops a class after an assignment has started, it may be impossible to produce a valid mapping. Assume that there are four students and our algorithm would have produced the mapping (A reviews B, B reviews A, C reviews D, and D reviews C). If the first two mapping assignments ( $A \rightarrow B$  and  $B \rightarrow A$ ) have already been made, and D drops the course, we have no alternative but to assign C to review himself. Therefore, further work is necessary to see how to minimize the probability that an invalid mapping will result.

Also, sometimes we desire individuals to review project teams. Even if some project teams have more members than others, we would still desire that each team get nearly the same number of reviews. This raises the situation where the set of reviewees is different (and smaller than) the set of reviewers.

Finally, if a class is large enough, it should be possible to do some reviews asynchronously. That is, not all students should need to have the same deadline. Such a capability would be very helpful, since there is otherwise no way to give a student an extension on a peer-reviewed assignment. We will work on extending the algorithm to cover these special cases.

## Bibliography

[Gehr 99] Gehringer, Edward F., "Peer grading over the Web: enhancing education in design courses," *Proc. American Society for Engineering Education 1999 Annual Conference and Exposition*, Session 2532.

[Gehr 00] Gehringer, Edward F., "Strategies and mechanisms for electronic peer review," *Proc. Frontiers in Education 2000*, Session F1B.

[Gehr 01] Edward F. Gehringer, "Assignment and quality control of peer reviewers," 2001 ASEE Annual Conference and Exposition, American Society for Engineering Education, Albuquerque, June 26, 2001

[KPD 95] Kerr, Peter M., Park, Kang H., and Domazlicky, Bruce R., "Peer grading of essays in a principles of microeconomics course," *Journal of Education for Business* 70:6, July 1995, pp. 357 ff.

[Topp 98] Topping, Keith, "Peer assessment between students in colleges and universities," *Review of Educational Research* 68:3 (Fall 1998), pp. 249-276.

[SGG 01] Silbershatz, Abraham, Galvin, Peter, and Gagne, Greg, *Operating System Concepts*, 6<sup>th</sup> ed., John Wiley and Sons, 2001.

#### EDWARD F. GEHRINGER

Edward Gehringer is an associate professor in the Department of Electrical and Computer Engineering and the Department of Computer Science at North Carolina State University. He has been a frequent presenter at education-based workshops in the areas of computer architecture and object-oriented systems. His research interests include architectural support for persistence and large object systems, memory management and memory-management visualization, and garbage collection. He received a B.S. from the University of Detroit(-Mercy) in 1972, a B.A. from Wayne State University, also in 1972, and the Ph.D. from Purdue University in 1979.

#### YUN CUI

Yun Cui is currently an M.S. student in the Department of Computer Science at North Carolina State University. Her research interests include object-oriented systems and network security. She received a B.S. from Zhejiang University in P.R. China in 1994.