

Allocating readers to projects

Denis Sorel

0913503

2010 Erasmus exchange short individual project

Abstract

For several years, student in the Department of Computing science are automatically assigned their projects through a web based platform where they can submit their preferences, so that nobody is left without a project or with a project that doesn't interest him. However, there is no such system when it comes to assigning readers (second markers) to these projects. This project deals with developing a software to complete this task, by reusing the same core algorithms that are used to allocate projects to students.

However, these algorithms cannot be used directly and need to be adapted to the situation, mainly because in the present case, a person is often assigned more than one project from his list of preferences. The program will permit to modify preferences lists and generate results with the aid of the matching program.

The most important feature of the developed program is interactivity, it has been conceived to be easy to use and allow the user to do some modifications over the input parameters (staff preference lists) and the results, suggesting actions by detailed displays.

Contents

1	Introduction	4
2	Background	5
2.1	Actual background	5
2.2	Matching algorithms	5
2.3	Previous work	7
3	Design	7
3.1	Functional requirements	7
3.2	Non-Functional requirements	8
3.3	Overall Design	9
3.4	GUI Design	9
4	Implementation	11
4.1	Extending preference lists	11
4.2	Displaying results and Reallocating projects	12
4.3	GUI implementation	13
5	Evaluation	14

1 Introduction

Previously, there was not really any system to allocate a reader, i.e. the second marker of the project, to each project. The project coordinator usually made the allocation with the aid of suggestions from project supervisors. So the assignment distribution was not very balanced and people were not always assigned their most preferred projects. At the same time, when students choose their project, they have access to a departmental web portal where they can browse the available projects and submit a list ordered by preference. A member of staff can then run a matching algorithm on these data to automatically assign a project to every student. A similar system could be developed using matching algorithms to make the allocation of projects to readers easier, faster, and most importantly as balanced as possible, so that everyone reads approximatively the same number of projects. The aim of the present project was to implement such a software that permits editing the preference lists, call matching algorithms to assign projects, and display the resulting allocations.

In the following parts, I will describe the different matchings that are used in the development of the program and what are their specifications. I will then explain how the software is designed, what are the requirements it should support and how its functionalities are technically implemented.

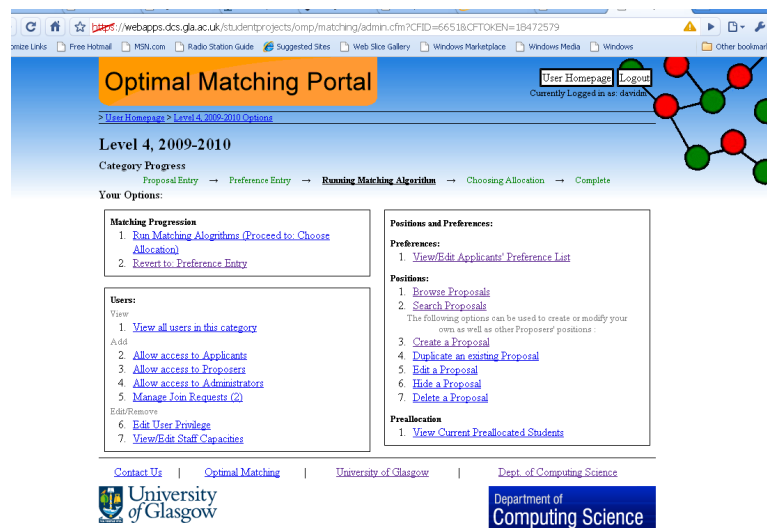


Figure 1: Matching web portal

2 Background

2.1 Actual background

This project deals with the allocation of second markers, called readers, to projects in the Department of Computing Science, University of Glasgow. There are, the current year, 114 projects of level 3, 4, 5, and MSc to be distributed between 35 members of staff. Each one is given a personal δ parameter ($0 \leq \delta \leq 1$) representing his relative marking load, which is basically 1 and is reduced under certain circumstances (sabbatical, probation). The global marking load is twice the number of project, as each project is marked by both its supervisor and its reader. We can then obtain a reference marking load by : $\frac{2 * \text{number_of_projects}}{\sum \delta}$ (in the current context 228/25.25 approximated to 9) so that every member of staff marking target, i.e. the number of project he should mark, is the product of his δ factor by this (maximum) marking reference. We can then get the number of project to be read by a person, the “reader target” by subtracting from his marking target the number of projects he is already supervising. The result of the allocation will be considered balanced if everyone marks as reader a number of projects equal to his reader target or just one less.

Another constraint that the program should ensure is that nobody is allowed to mark as reader a project he is already supervising.

Members of staff are asked to submit the longest possible preference lists (twice their reader target is the advised minimum length) so that their are assigned to mark projects related to subjects they are used to deal with. Nevertheless, as level 3 and Msc in Information Technology projects deal with more general subjects than projects from other levels, they can be allocated to anyone.

2.2 Matching algorithms

Definitions

- A matching algorithm takes as input a list of posts and applicants preference lists and assigns posts to applicants, following certain rules. The result of the algorithm, made of applicant-post couples, is called a matching. The most important criterion is that as few applicants as possible are left unmatched; this is satisfied by a *maximum matching*. In the present context of allocating readers, applicants are members of staff and posts are projects.
- the size of a matching M is its cardinality, the number of applicants that have been assigned a post.
- The degree of a matching M is the maximum i such that some applicants obtains their i^{th} -choice in M .
- The profile of M is a vector $\langle x_1, \dots, x_k \rangle$ where x_i is the number of applicants who obtain their i^{th} -choice in M ($1 \leq i \leq k$), and k is the degree of M .
- The cost of M is the sum of the ranks of the assigned posts in the applicants preference lists, i.e., it is $\sum_{i=1}^k i x_i$ where $\langle x_1, \dots, x_k \rangle$ is the profile of M .

Common algorithms

- *naive matching*, also known as Random Serial Dictatorship [1] , just consists of picking applicants at random and assigning them the first post still available in their list.

```

S:=Set of applicants
while S is not empty do
  pick an applicant a from S
  if a has an undersubscribed post on his preference list do
    match a to most preferred post
  else
    report a as unmatched
  end if
  remove a from S
end while

```

Figure 2: Pseudocode of the naive algorithm

However, the resulting matching of such algorithm has no guarantee of optimality with respect to size, cost and profile.

- A matching is a *greedy maximum matching* [2] if it has maximum cardinality and its profile is lexicographically maximal, i.e. the number of applicants getting their first choice is maximal then, subject to this, the number of second choices is maximal, ...
- A matching is a *generous matching* [2] if it has maximum cardinality and its reverse profile $(\langle x_k, \dots, x_1 \rangle)$ is lexicographically minimal, i.e. it minimizes the number of applicants getting their k^{th} choice, and then, subject to this, minimizes the number of $(k - 1)^{th}$ choices, ..., k being the maximum length of an applicant preference list.
- A matching is a *minimum cost maximum matching* [3] if it has maximum cardinality, and, subject to this, its cost is minimum.
- A matching is a *greedy generous matching* if it has maximum cardinality, its degree is equal to the degree of a generous matching, and, subject to the first two properties, its profile is lexicographically maximum.

Comparison of matchings

The s_i in figures 3 and 4 represent the applicants, and are followed by their preference lists of posts l_j . The greedy matching assigns more applicants to their first choice but has however a higher overall cost and also assign someone his last choice. On the other hand, the generous matching assigns no post ranked more than three but only one first choice. In this first example, the greedy generous matching would be the same as the generous matching.

This second simple case (figure 4) shows how not optimal a naive matching could be. It has a weight of 15 as the result of the others “more advanced” algorithms (which is similar to all in this particular case) has a size of 6.

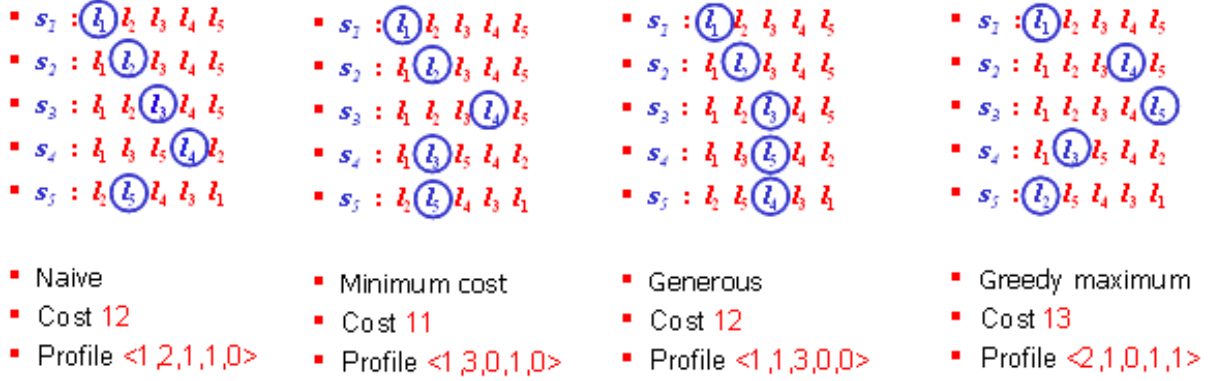


Figure 3: Matchings comparisons on a simple case

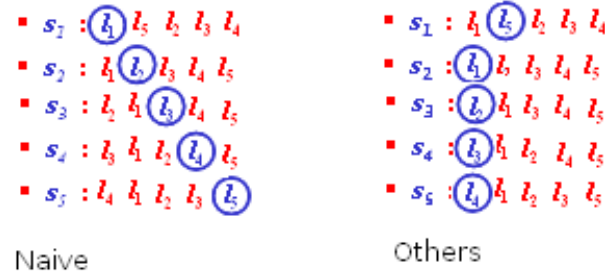


Figure 4: Naive matching vs. greedy,generous,mincost

2.3 Previous work

A software implementing algorithms to produce the different types of matching previously defined has been implemented by Dr Robert Irving. I used these algorithms for reader allocation with a modified input.

A few years ago, Alastair Scott developed as part of a MSc IT project a GUI application for reallocating posts from a matching. His program takes as input the matching software input and output files and displays the result, with coloured buttons with different meanings for different colours to highlight allocated post, applicants without post and suggest possible reallocation to the user. It also shows the cost and size of the modified result to compare with the original matching, and proposes different methods for reallocating post.

3 Design

3.1 Functional requirements

The program should support the following requirements :

3.1.1 Core requirements

Pre-matching

- The pre-matching part of the program should read in preference lists, reader targets and lists of supervised projects for each member of staff from an input file. The reader target is the number of project a person should read, as defined in part 2.1.
- The user should be able to lengthen preference lists by choosing from a number of options:
 - choose from a random selection of Level 3 and MScIT projects; these projects deal with less specific subjects than projects from other levels, everyone can be reader.
 - choose from a random selection of “not bid for” projects, that are projects that do not belong to anyone’s preference list.
 - append with an actual list of projects typed in by the user.
- It should also allow to extend all “short” lists with one of these options. Someone’s list is considered short if its length is less than twice his reader target. The number of options to extend a given short list could be increased, for instance by letting the user choose the length of the extension.
- It should finally provide functionality to run matching algorithms, and choose amongst different matchings.

Post-matching

- The post-matching part should display results of the matchings,
- Show statistics elements of the result matching such as its size, cost(weight), and profile
- Display the results either by staff member or by project.
- It should allow the user to reallocate a project from one person to another, especially in order to balance the reading load more equitably.
- The program should eventually permit export of the results to comma-separated values files in order to use them later in a spreadsheet.

3.1.2 Lower priority requirements

The program could also, in the post-matching part, allow the user to allocate to members of staff projects that have not been assigned to anyone by the matching algorithm.

The user could be allowed to undo modifications (list extensions in the pre-matching phase and project reallocations in the post-matching phase).

A web portal where preference lists are submitted by staff could be developed.

3.2 Non-Functional requirements

- The program should be platform independent, running on Windows, Mac OS, or Linux with a minimal configuration (500MHz CPU, 64MB RAM);
- It should be easy to use and include a graphical user interface.

3.3 Overall Design

The program is separated into two principal parts, namely the pre-matching and post-matching phases. The first one involves parsing the input file, managing extensions of preference lists and generating the input for the matching program. The role of the second part is to display comprehensible results from the output of the matching and give the user the possibility to reallocate a project and export results to file. The first part also passes information to the second in order to match staff members with the indices from the result in order that their original details can be recovered.

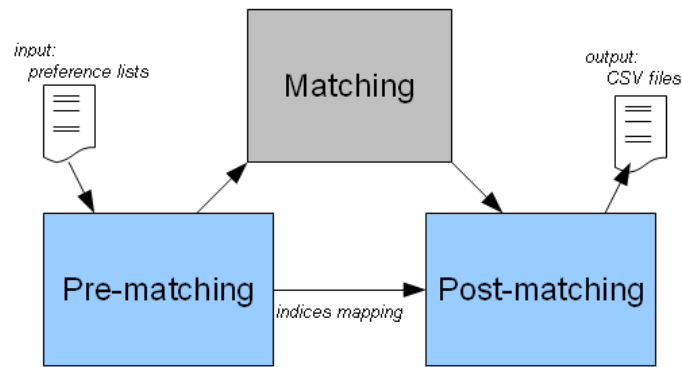


Figure 5: General overview

The input file will also need to precise the supervised projects and reader target of everyone. The implementation of both pre-matching and post-matching phases follow a Model-View-Controller pattern that separates the user interactions from the functional components.

3.4 GUI Design

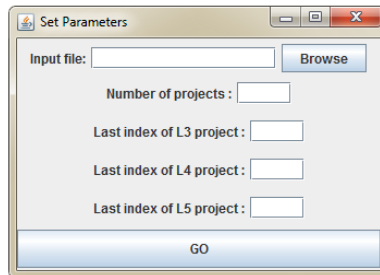


Figure 6: A window requiring details of the input preferences file and parameters relating to the projects

The user can choose the input file and define constant parameters such as the number of

projects and the index ranges for each level.

Short lists

ID	Supervised	Target	Preferences
4	20 21 22 59	5	67 68 114 34 18
9		2	
21	7 102	7	15 34 74 72 76 73 21 37 105 107 20 1
33		4	

All lists

ID	Supervised	Target	Preference
1	15	1	34 35 26 33
2	1 16 17 18 59 66 85 61	1	13 20 21 22
3	19	8	42 23 24 9 2 1 77 78 100 106 108
4	20 21 22 59	5	67 68 114 34 18
6	23 60 67 68 86	4	3 98 11 26 32 5 78 92 105 25
7	26 27	5	40 56 1 22 32 53 16 57 24 21 18
8	2 28 29 87 88 89	3	44 45 49 5 55 77 43
9		2	
10	30 62 69 90 60	4	13 42 49 83 105 2 17 68 113
11	3 31 32 91 92	4	38 76 37 93 28 5 49 94 98 29 42 76
13	33 34 35 71 72 73	3	74 15 21 53 56 66 1 20
14	74	3	15 33 34 35 71 72 73

Projects not bid for by anybody: 7 39 41 48 58 59 60 61 62 65 70 86 89 96 97 102

Extend a list

☐ With a random subset of L3 and Msc

☒ With a random subset of "not bid for"

☐ With a random subset of a set of your choice

☐ With a list of your choice

List to extend: 1

Extend Extend all 'shorts'

IDs of projects to add:

☒ Mincost ☐ Greedy ☐ Generous ☐ Greedy generous ☐ Naive greedy

Results

Figure 7: Pre-matching view with preference lists extension

“Short” lists are displayed separately and the user of the program sees which projects have not been selected, and has the possibility to choose one from different preference list extension methods and apply it by clicking the “extend” button. In the same window, radio buttons allow to choose the type of matching and open a result view (Figure 8) by clicking the “Results” button.

ID	Projects	Ranks	Target	Alloc	Cost (avg)	L3	L4	L5	MSc
1	26	3	1	1	3(3.0)	0	2	0	0
2	20	2	1	1	2(2.0)	1	4	2	2
3	2 9 23 42 78 100 106	1 2 4 5 8 9 10	8	7	39(5.5)	2	3	0	3
4	7* 60* 67 70* 102*	1 6 7 8 9	5	5	31(6.2)	1	3	2	3
6	3 11 92 98	1 2 3 8	4	4	14(3.5)	2	1	1	5
7	1 18 22 40 57	1 3 4 8 11	5	5	27(5.4)	1	5	1	0
8	44 45 49	1 2 3	3	3	6(2.0)	1	5	0	3
9	41* 86*	1 3	2	2	4(2.0)	0	1	0	1
10	13 17 83 105	1 4 5 7	4	4	17(4.2)	1	2	1	5
11	37 38 76 93	1 2 3 4	4	4	10(2.5)	1	4	0	4
13	21 53 74	1 3 4	3	3	8(2.6)	0	5	0	4
14	33 35 71	2 4 5	3	3	11(3.6)	0	2	0	2
15	30 80 82 95 99 101	1 2 4 10 12 24	6	6	53(8.8)	0	2	0	7
16	31 32 47 91	1 2 5 12	4	4	20(5.0)	0	5	0	4
17	6 29 88 90	1 3 7 8	4	4	19(4.7)	2	2	0	5
18	114	2	1	1	2(2.0)	0	0	0	2
19	51 54 103 109	6 8 9 10	4	4	33(8.2)	0	2	0	2

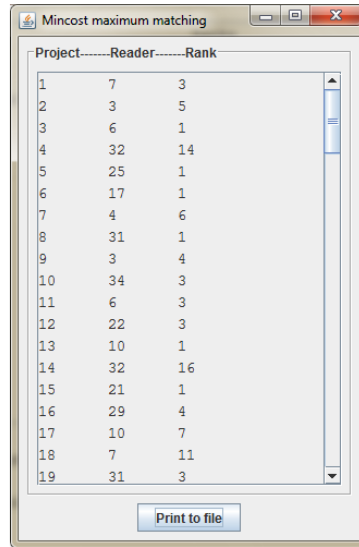
Move a project from: 1 To: 1 Project Id:

Reallocate project Project-Reader view Print to file

Size:109 Matching weight:577 Matching profile: (20,14,13,14,6,6,7,8,5,4,2,4,0,3,0,1,0,1,0,0,0,0,0,1)

Figure 8: Matching result - view by staff member and project reallocation

This window displays for each member of staff the allocated projects, ranks, cost and distribution of marking load between levels. The user can reallocate projects choosing via combo boxes and open the window shown in figure 9 to view allocations by project.



Project	Reader	Rank
1	7	3
2	3	5
3	6	1
4	32	14
5	25	1
6	17	1
7	4	6
8	31	1
9	3	4
10	34	3
11	6	3
12	22	3
13	10	1
14	32	16
15	21	1
16	29	4
17	10	7
18	7	11
19	31	3

Figure 9: Matching result - view by project

4 Implementation

4.1 Extending preference lists

The major functionality of the pre-matching phase is to allow the user to see everyone's preference list and extend one or more lists, especially those that are too 'short'. First the user is asked via a `JFrame` containing a file chooser and text fields to provide the input file and indices for the partitioning of the projects according to their levels.

Then the program parses the input file, which is a text file containing for each member of staff a line in the format:

`Id | supervised projects indices | reader target | preference list` . It creates for each line a new instance of `PreferenceList` and aggregates these objects in a `List` in the model class `ReaderAllocModel` (Figure 10). The `PreferenceList` class main attributes are : the id of the person, a list of the supervised projects, the reader target and a list of preferences. It also provides methods to extend the preference list, checking that the project id is not already supervised or part of the preferences.

While building these lists, the `ReaderAllocModel` constructor also initializes an array of the frequencies of every project in preference lists, in order to point out the projects that are

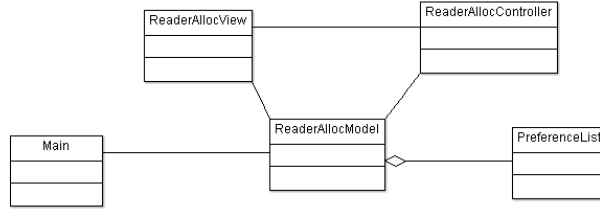


Figure 10: simplified class diagram for the pre-matching phase

not part of anybody's preferences.

Then when the user has finished extending lists, he can click on the result button that run matching algorithms and display a selected matching. The method file will so create a file formatted for the input of the matching program (previously developed by Rob Irving) by duplicating the preference lists as many times as the reader target.

4.2 Displaying results and Reallocating projects

The post matching phase is handled via **ResultModel** whose constructor first reduces the output of the matchings to only the data concerning the chosen method (using **OutputTrimmer**). All the matchings are computed the first time the user clicks on the "result" button and are not recomputed later unless new modifications occurred. Then it parses the result line by line, and puts the results in instances of **AllocationResult**, which contains a person id, his allocated projects and their ranks, reader target, and also a copy of his original preference list, so that when people have been allocated a project that did not belong to their original preferences, it will be displayed with a '*' character.

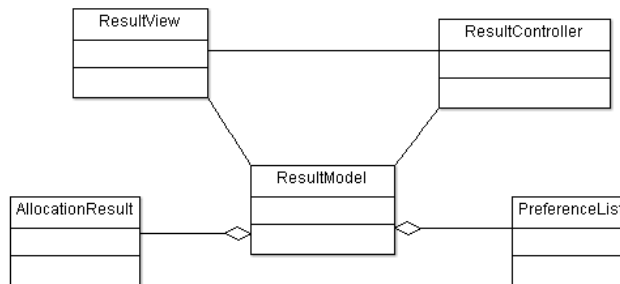


Figure 11: Post matching phase

The main issue in this part was to associate the indices from the matching result back to the indices identifying members of staff. To do that I first created from the preference lists a list of **AllocationResult** with only id and target (and supervising load for each

level) initialised and then go through this list with a **for** loop and in the same time through the result scanner with a **while** loop that allocates a project to someone (with the method `AllocationResult.addProject`) if the index of the project is less than or equal to a certain index l_i (where i represents the person's id) else we move to the next person in the list and continue the process. This "last index" l_i is calculated for everyone as the sum of the person's target and the previous person index (or equal to the target for the first person in the list): let t_i be the reader target of person with id i , so $l_i = l_{i-1} + t_i$ and $l_1 = t_1$.

In the example figure 12, person with id 1 is not allocated any projects, id 2 is allocated project 20 (that has index 2), id 3 is assigned projects with indices in range [3,10], ...

Staff members			Results	
ID	Target	last index	ID	Project Id
1	1	1	2	20
2	1	2	4	2
3	8	10	5	9
4	5	15	6	23
...			7	108
			8	78
			9	42
			10	106
			11	67
			12	68
			13	18
			14	114
			16	92
			...	

Figure 12: Mapping indices from results

For the user to reallocate projects from one reader to another, combo boxes allow the user to select which project to move. Each time the value of the source of the destination person Id is changed, the list of projects that are allowed to be reallocated is updated so that they are in the allocated projects list of the source reader but are not supervised by the destination reader. Then, the project is deleted from the source along with its associated rank and then allocated to the destination with a rank value defined by the position of the project Id in the preference list (which will have been extended with that project if not previously containing it).

A **Map** is used to associate readers to projects and allow a different view of the allocation. It is updated for every reallocation. The cardinality, the cost and the profile of the matching are also updated at every reallocation.

4.3 GUI implementation

The graphical interface is implemented following a model-view-controller pattern. I used the **Swing** toolkit of Java; the view classes `ReaderAllocView`(fig 2) and `ResultView`(fig 3.) extend the `JFrame` class. I mainly used `GridLayout` to place the different elements (`JPanel`)

and the `setPreferredSize` method to resize the components. In the pre-matching view, the preference lists are displayed in a `JTable` component as the results are in a scrollable part containing `JLabels`.

The user's actions apply via `JButtons` and `JRadioButton` when multiple options are available. Event handlers are added to these buttons, that link to the classes implementing the actions (respectively `ReaderAllocController` and `ResultController`) by determining which button was pressed and delegate the action to the model class (`ReaderAllocModel` or `Result`) before refreshing the view.

5 Evaluation

Supervised by David Manlove, I met him every week and at each meeting new parts of the program were evaluated and the requirements evolved continuously. At first the principal aim was to parse the preference lists, create a formatted input for the matching algorithms and display the results in a readable way. Then the different kinds of extensions for lists and the possibility to reallocate projects were added.

The program has been tested against input errors: if the input file does not match the required format, the program is exited; if the input parameters (figure 6) are incorrect (indices not in order, last index greater then number of projects, input file not found), the user is asked by a warning window to enter them again; in the pre-matching phase (figure 7) text field, anything that is not a valid project is ignored. I had to make everything remain meaningful not to obtain absurd results. For example, an error in the early version of the indices mapping part caused that readers were allocated projects that didn't belong to their lists. The constraint that the reader of a project should be different from the supervisor was also checked regularly. In the post-matching phase, constraints are verified first so that the user can only apply reallocations that don't create any inconsistency.

As it is designed to be used only by one person, the software has not been evaluated by a variety of different users. An unfinished version of the software was actually used to allocate readers to this year's projects in the Department of Computing Science, University of Glasgow. The result was quite satisfactory and experience of using the program led to some refinements, such as highlighting readers that have been assigned too few projects in the post-matching phase.

References

- [1] Atila Abdulkadiroglu and Tayfun Sonmez, Random Serial Dictatorship and the Core from Random Endowments in House Allocation Problems, *Econometrica*, 1998.
- [2] Robert W. Irving, Greedy and Generous matchings via a variant of the Bellman-Ford Algorithm, unpublished manuscript, 2007
- [3] Harold N. Gabow and Robert E. Tarjan, Faster scaling algorithms for network problems, *SIAM Journal on Computing*, Vol. 18, No. 5, pp. 1013-1036, 1989.