Handout 10
CSCI 136: Spring, 2004
3 May

Lab 10

Due: 9 May

Final exams

1 Introduction

You are to write a program which will help the registrar schedule final exams so that no student has two exams at the same time. You are to use a "greedy" algorithm to determine an assignment of classes to exam slots so that:

1. No student is enrolled in two courses assigned to the same exam slot.

2. Any attempt to combine two slots into one would violate rule 1.

Thus we wish to get by without gratuitously wasting exam slots (students would like to get out of here as soon as possible, after all).

2 What

Input to the program will be a file generated by my program "Register", which is in file "Register.java" (see the assignment web page). Information on each student is written on the file using the writeUTF method of DataOutputStream. For each student, 5 strings are written. The first for the name, and the next 4 for courses selected. The program insists that each student takes exactly 4 courses. A possible file would include

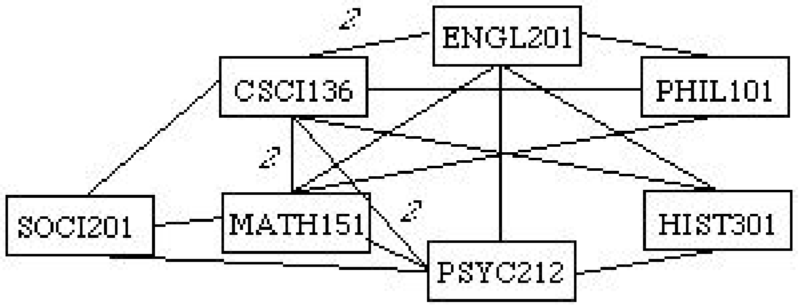
- Kim Bruce taking CSCI 136, MATH151, ENGL201, and PHIL101;
- Peter Murphy taking PSYC 212, ENGL 201, HIST 301, and CSCI 136; and
- David Edwards taking SOCI 201 CSCI 136, MATH 151, and PSYCH 212.

The output of the program should be a list of time slots with the courses whose final will be given at that slot.

3 How

The key to doing this assignment is to build a graph as you read in the file of students and their schedules. (An end-of-file condition will be signalled by a student name "$$".)

Each node of the graph will be a course taken by at least one student in the college. An edge will be drawn between two nodes if there is at least one student taking both courses. The label of an edge could be the number of students with both classes (though we don't really need the weights for this program). Thus if there are only the three students listed above, the graph would be as given below (edges without a weight label have weight 1).



A "greedy" algorithm to find an exam schedule satisfying our two constraints would work as follows. Choose a course (say, PHIL 101) and stick it in the first time slot. Search for a course to which it is *not* connected. If you find one (e.g., HIST 301), add it to the time slot. Now try to find another which is not connected to any of those already in the time slot. If you find one (e.g., SOCI 201), add it to the time slot. Continue until all nodes in the graph are connected to at least one element in the time slot. When this happens, no more courses can be added to the time slot (why?). (By the way, the final set of elements in the time slot is said to be a *maximal independent set* in the graph.)

If there are remaining nodes in the graph, pick one and enter it in a new time slot and then try adding other courses to the same slot as before. Continue adding time slots for remaining courses until all courses are taken care of. Print the exam schedule. For the graph shown, a possible exam schedule is:

```
Time 1: PHIL 101, HIST 301, SOCI 201
Time 2: MATH 151
Time 3: CSCI 136
Time 4: ENGL 201
Time 5: PSYC 212
```

Notice that no pair of time slots can be combined without creating a time conflict with a student. Unfortunately, this is not the minimal schedule as one can be formed with only 4 time slots. (See if you can find one!) Thus a greedy algorithm of this sort will give you a schedule with n slots, no two of which can be combined, but a different selection of courses in slots may result in fewer than n slots. Any schedule which satisfies are our constraints will be acceptable (though see below for extra credit).

4 Hints for building the final exam schedule:

You are to represent graphs as adjacency lists. (Why does that make the most sense for this application?) Vertex labels should be the course names.

Here is one possible way to find a collection of maximal independent sets from the graph. Represent each slot by some sort of a list (or, better yet, a binary search tree). To find a maximal independent set for a slot, pick any vertex of the graph and add it to the list. Cycle through all other vertices of the graph. If a vertex is not connected to any of the vertices already in the slot, throw it in. Continue until you've tried all vertices. Now delete all vertices in the slot from the graph. Fill successive slots in the same way until there are no vertices left in the graph.

5 Extra Credit:

A wide variety of extra credit is possible. Here are some options:

1. Always generate the best possible exam schedule (that is, the one with the fewest number of slots).
2. Allow students to take more or fewer than four courses.
3. Print out a final exam schedule for each student.
4. Print out a final exam schedule ordered by course number.
5. Arrange the time slots in an order which tries to minimize the number of students who must take exams in three consecutive time slots. *Warning: This part is hard!*

Feel free to add other useful bells and whistles. As usual, be sure sure to indicate in the heading of your program what extras you have included.

6 Implementation Hints:

I suggest that you look very carefully at my program for generating the files in order to see how I wrote the files. You will use similar file operations (except reading instead of writing). You should print the final exam schedule in a text area in a frame. An important operation for this is "`append(String newText)`" which adds `newText` to the end of whatever has been written so far in the `TextArea`. Sending the message `setEditable(false)` to the TextArea will ensure that the user cannot accidentally override the information printed there (only the program can write information there).

7 Using Dialog Boxes in Java

Java provides a built-in class, `FileDialog`, in the AWT package to allow the user to select files for saving or loading. The constructors for `FileDialog` include

```
FileDialog(Frame, String)
FileDialog(Frame, String, int)
```

The constructor creates a file dialog window associated with the specified frame and with the specified title for loading or saving a file. The optional `int` argument is `FileDialog.LOAD` or `FileDialog.SAVE`. If omitted the default is `FileDialog.LOAD`. The following sample code shows how a `FileDialog` can be used in a program to get a file name to be read from.

```
String fileName;
do {
    dialog = new FileDialog(myFrame,"Load a schedule file");
    dialog.show();
    fileName = dialog.getFile();
} while (fileName == null);
```

This code attaches a dialog box to myFrame. When executed, the code creates and shows the dialog. The `dialog.getFile()` command either returns a string representing the file name to be loaded or `null` (if the cancel button is clicked from the dialog box). Thus the loop continues creating and displaying the dialog box until a file is selected.

The programmer now must write commands to open and read from the file.

8 Writing to files

In this section, we provide the code used to write a file from the `Register` class. You will need to do something similar in your `Schedule` class except that you will be reading instead of writing, output will be changed to input, etc.

```
// Open file and go to data entry field when click on start button
public void actionPerformed(ActionEvent evt){
    String fileName;
    do {
        FileDialog dialog = new FileDialog(Register.this,"New file for students:",FileDialog.SAVE);
        dialog.show();
        fileName = dialog.getFile();
    } while (fileName == null);

    try {
        outFile = new DataOutputStream(new FileOutputStream(fileName));
        cardManager.last(contentPane);
    } catch(IOException e) {
        System.out.println("Not legal name!  Re-enter file name:");
    }
}

...

    try {
        outFile.writeUTF(name);
        for (int courseNo = 0; courseNo < 4; courseNo++) {
            outFile.writeUTF(courses[courseNo]);
        }

        ...
    } catch (IOException e) {
        message.setText("Bad data caused write failure");
    }
```

You will read the data out exactly as it was read in, using `readUTF()` on a file obtained from the `DataInputStream` and `FileInputStream` constructors.