Computational Microeconomics:
Game Theory, Social Choice, and Mechanism Design
Homework 0: Linear and Integer Programming

Due: September 19 before class

Please read the rules for assignments on the course web page
(http://www.cs.duke.edu/courses/fall18/compsci590.2/).
Use Piazza (preferred) or directly contact Harsh (harsh.parikh@duke.edu), Hanrui (hrzhang@cs.duke.edu), or Vince (conitzer@cs.duke.edu) with any questions. Use Sakai to turn in the assignment. Please use clear variable names and write comments in your code where appropriate (you can put comments between `/*` and `*/`).

0. Installing the GNU linear programming kit.

You can find the GNU linear programming kit on the Web (`https://www.gnu.org/software/glpk/`). You are free to install it on your own computer. If you have trouble, below are some instructions for installing glpk on your login.oit.duke.edu user space that might be helpful (but could be a little tedious, so you are encouraged to try installing on your own machine first). If you still have trouble, please let us know. After you have successfully installed everything you are highly encouraged to check out the "examples" directory for examples of how to use the modeling language, as well as the examples from class which are on the course website.

Instructions for getting onto OIT computers

Of course, your first option is to work at a computer on campus. Otherwise, using ssh, login to your account on a Duke OIT machine and do your work there. Here is a good manual on how to do this on Duke's network.

`http://www.cs.duke.edu/~alvy/courses/Remote_Access.pdf`

You will have to work from the command line, but if you don't already know how this is a great time to learn! Here is a good tutorial for some of the basics.

`http://www.cs.duke.edu/~alvy/courses/unixtut/`

It may also be good to work from a command line text editor, like vim or emacs.

Installation Instructions for GLPK

Individual students need to install GLPK in their login.oit.duke.edu user spaces with the following commands:

```
mkdir ~/glpk
cd ~/glpk
wget ftp://ftp.gnu.org/gnu/glpk/glpk-4.64.tar.gz
tar -xzvf glpk-4.64.tar.gz
cd glpk-4.64
./configure
make
```

The program can then be run from the following directory.

`~/glpk/glpk-4.64/examples`

If you want to solve an LP/MIP expressed using the modeling language, navigate to the above directory and type

`./glpsol --math`

(Use `--cpxlp` instead of `--math` for the "plain" LP language.) You will also need to specify the file that you want to solve, e.g.

`./glpsol --math problem.mod`

and you will also need to specify a name for a file in which the output will be stored, preceded by –output. So, typing

`./glpsol --math problem.mod --output problem.out`

will instruct the solver to solve the LP/MIP problem.mod, and put the solution in a new file called problem.out.

You will need an editor to read and edit files. One such editor is emacs (but any text editor will do). For example, typing

`emacs problem.out`

will allow you to read the output file.

You will need an editor to read and edit files. One such editor is emacs. Going to the right directory and typing

`emacs problem.out`

will allow you to read the output file.

Inside emacs there are all sorts of commands. You can find emacs commands on the Web, but a few useful ones are:

- Ctrl-x Ctrl-c: exit emacs
- Ctrl-x Ctrl-s: save the file you are editing
- Ctrl-s: search the file for a string (string=sequence of characters)
- Ctrl-r: search the file backwards
- Ctrl-g: if you accidentally typed part of some emacs command and you want to get back to editing, type this
- ESC-%: allows you to replace one string with another throughout the file; for each occurrence it will check with you, press spacebar to confirm the change, n to cancel it
- Ctrl-k: delete a whole line of text
- Ctrl-Shift-_: undo

Try playing around with all of this. In particular, check that you can solve the example files, and read the solutions. Then, solve the following questions.

1. Practice with GLPK: a currency transfer problem.

In this problem, there are a set of agents and a set of currencies. Every agent may either owe, or be owed, some amount of each currency to/from the pool of agents. For example, Alice may owe 2 dollars, and be owed 3 euros. We denote this as $d_{\text{Alice,dollars}} = -2$ and $d_{\text{Alice,euros}} = 3$. You may assume that for all currencies j, $\sum_{i \in \text{agents}} d_{ij} = 0$. That is, every (say) dollar that is owed by someone is, in some sense, owed to someone else. (However, we don't say that Alice owes a dollar specifically *to* Bob.)

Every agent in fact holds plenty of each currency, and our goal is to transfer money to settle all these debts. There are no exchange rates between currencies in this problem. Money can be transferred between two agents only if they *meet*. For each pair of agents, it is known whether they meet or don't meet. Let $m_{\text{Alice,Bob}} = 1$ indicate that Alice and Bob meet and $m_{\text{Alice,Bob}} = 0$ indicate that they do not. We will denote by $x_{\text{Alice,Bob,dollars}}$ the number of dollars that Alice transfers directly to Bob in their meeting (which must be 0 if $m_{\text{Alice,Bob}} = 0$).

a Create a linear program that minimizes the *total number of times that a dollar / euro / other unit of currency moves*. Write it up in the `.mod` format and run it on **Instance 2** given below.

b Create a mixed integer linear program that minimizes *the number of meetings where nonzero amounts of money are transferred*. If multiple currencies are transferred in a single meeting, this still just contributes only 1 point to the objective; also, it does not matter whether, in a single meeting, money changes hands in both directions (for different currencies) or just in one direction. Write it up in the `.mod` format and run it on **Instance 2** given below.

Instance 1 below is an example, with the solutions to these two problems given. You are encouraged to try out your code on Instance 1 first, making sure you have the solution right, before trying it on Instance 2. But the files you turn in should be for Instance 2.

1.1 Instance 1

Table 1 shows how much everyone owes/is owed in each currency, and Table 2 shows who meets whom.

	Alice	Bob	Eva	Jack
dollar	3	-5	2	0
euro	-2	-2	4	0
yen	4	2	-2	-4

Table 1: Instance 1: Matrix d

	Alice	Bob	Eva	Jack
Alice	-	1	1	0
Bob	1	-	1	0
Eva	1	1	-	1
Jack	0	0	1	-

Table 2: Instance 1: Matrix m

Optimal solution for Problem (a): See the transfers in Table 3.

dollar	(Bob to Alice, 3), (Bob to Eva, 2)	5
euro	(Bob to Eva, 2), (Alice to Eva, 2)	4
yen	(Jack to Eva, 4), (Eva to Bob, 2), (Eva to Alice, 4)	10
total		19

Table 3: Problem (a): optimal solution for Instance 1

Problem (b): One optimal solution is as follows: there are transfers in the Alice-Bob, Bob-Eva, and Eva-Jack meetings only, for a total of 3 meetings in which currencies change hands. The transfers are then as in Table 4.

Note that the total number of times a unit of currency changes hands is now higher than before, namely 25 instead of 19. That is fine, because minimizing that is no longer our objective. (Similarly, the solution for (a) had transfers in 4 meetings, which would not be optimal for the objective here.) Also note that there are other optimal solutions as well. For example, there is an optimal solution where there are transfers only in the Alice-Eva, Bob-Eva, and Eva-Jack meetings. It is fine if your code returns such a different optimal solution instead—but it has to be one with only 3 meetings in which transfers take place, of course.

dollar	(Bob to Alice, 5), (Alice to Eva, 2)	7
euro	(Bob to Alice, 2), (Alice to Eva, 4)	6
yen	(Jack to Eva, 4), (Eva to Alice, 6), (Alice to Bob, 2)	12
total		25

Table 4: Problem (b): an optimal solution for Instance 1

1.2 Instance 2

Again, Table 5 shows how much everyone owes/is owed in each currency, and Table 6 shows who meets whom.

	Alice	Bob	Eva	Jack	Rose	Tom
dollar	3	-6	5	1	-1	-2
euro	2	-10	5	-3	3	3
yen	-15	-5	10	-2	2	10

Table 5: Instance 2: Matrix d

	Alice	Bob	Eva	Jack	Rose	Tom
Alice	-	1	0	0	0	1
Bob	1	-	1	0	0	1
Eva	0	1	-	1	1	1
Jack	0	0	1	-	1	0
Rose	0	0	1	1	-	0
Tom	1	1	1	0	0	-

Table 6: Instance 2: Matrix m