15326 - Computational Microeconomics

Programming Assignment 1: linear and mixed integer programming (due Sep. 20 before 5pm)

Please read the rules for assignments on the course web page (`http://www.cs.cmu.edu/~15326-f24/`). Use Piazza for questions and Gradescope to turn this in.

Please use clear variable names and write comments in your code where appropriate (you can put comments between `/*` and `*/`, or start a line with `#`).

0. Installing the GNU linear programming kit.
You are free to install the GNU linear programming kit however you like, but there are some specific instructions below. You can also find the GNU linear programming kit on the Web (`https://www.gnu.org/software/glpk/`). If you have trouble, please let us know.

Installation Instructions for GLPK

Navigate to the folder where you wish to install it. It will work for any folder. Then execute the following commands

```
mkdir ./glpk
cd ./glpk
wget ftp://ftp.gnu.org/gnu/glpk/glpk-5.0.tar.gz
tar -xzvf glpk-5.0.tar.gz
cd glpk-5.0
./configure
make
```

The program is called glpsol and is located in the following directory: `YOURFOLDER/glpk/glpk-5.0/examples/glpsol`

To be able to use the program from any location, you may add the following to your .bashrc file in your home folder (Linux)
`alias glpsol='YOURFOLDER/glpk/glpk-5.0/examples/glpsol'`

Usage instructions for the GLPK solver
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. It may be good to work from a command line text editor, like vim or emacs, but you can open them with any text editor.

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

`./glpsol --math problem.mod`

where `problem.mod` is your problem. You can omit `./` if you have added the alias in `.bashrc`, and do it from any folder.

(Use `--cpxlp` instead of `--math` for the "plain" LP language. Note on a Mac you would use `\glpsol --math`.)

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.

Editing files

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

1. (10 points.) Knapsack. Modify[1] the knapsack.lp file from the course website (*not* knapsack.mod) so that object B has weight 3.7kg, volume 4.2 liters, sells for $4.25, and has 3.5 units available. (Do not add integrality constraints, that is, allow the solution to be fractional.) Solve it using `glpsol --cpxlp` and put the output in knapsack.out. Check that the solution makes sense.

2. (15 points.) Hot dogs. Modify the hotdog.mod file to solve the following instance of the hot-dog problem: you now have 3 hot-dog stands (call the third one s3). The customers are now as follows:

- location: 3, #customers: 5, willing to walk: 2
- location: 5, #customers: 2, willing to walk: 3
- location: 6, #customers: 4, willing to walk: 2
- location: 8, #customers: 5, willing to walk: 1
- location: 10, #customers: 2, willing to walk: 2
- location: 11, #customers: 4, willing to walk: 6
- location: 12, #customers: 3, willing to walk: 1
- location: 15, #customers: 5, willing to walk: 7

Solve using `glpsol --math` and put the output in hotdog.out. Check that the solution makes sense.

3. (20 points.) Choosing courses.

Bob is a student at the University of Interdisciplinarity. At this university, students must obtain 10 points in the natural sciences, 10 points in the social sciences, and 10 points in the humanities, to satisfy their general education requirements. Rather cynically, Bob is only interested in minimizing the amount of effort that he has to put in to satisfy these requirements. (Let's at least say it's because Bob is passionate about a particular specialization and really wants to focus his efforts there instead...)

There are four courses that can give Bob these points.

1. "Introduction to neuroscience and its implications for social behavior" gives 8 natural science points, 6 social science points, and 4 humanities points. It requires 5 units of effort.

[1] If you want to keep the original knapsack.lp file, you can do `cp knapsack.lp knapsack.original.lp` first to create a backup.

2. "The history of the popular perception of statistical facts" gives 3 natural science points, 6 social science points, and 8 humanities points. It requires 5 units of effort.

3. "The use of biophysics in sports" gives 5 natural science points, 3 social science points, and 1 humanities points. It requires 2 units of effort.

4. "A brief introduction to global warming" gives 4 natural science points, 2 social science points, and 2 humanities point. It requires 2 units of effort.

Which courses should Bob take? What if courses can be taken partially, meaning that you take, say, half of the course, spend half of the effort, and get half of the points in each category? (Of courses, fractions other than 1/2 are also allowed, but the fraction has to be between 0 and 1.)

Write an entirely new file courses.mod (just type `emacs courses.mod`), in which you use the modeling language to solve Bob's problem instances (both without and with the option of taking courses partially). As always with the modeling language, you should write the file so that it is possible to modify only the `data` part of the file to solve similar problem instances. Your file should start with the lines:

```
set REQUIREMENTS;
set COURSES;

param points_required{i in REQUIREMENTS};
param points_contributed{i in REQUIREMENTS, j in COURSES};
param effort{j in COURSES};
```

The rest is up to you to fill in. Solve using `glpsol --math` and put the output in `courses_integral.out` and `courses_fractional.out`. Check that your solutions make sense.

4. (25 points.) Scheduling tasks.

We have a set of tasks that need to be scheduled, i.e., for each task, we need to choose a (nonnegative) time at which to do that task. For any (ordered pair of) two tasks i and j, there is a minimum wait time `time_between[i,j]` between i and j, *unless* j is scheduled (strictly) before i . There is also a minimum wait time between j and i, which may be different and applies unless i is scheduled before j. You may assume all the minimum wait times (between different tasks) are positive.

You must complete the below mixed integer linear program:

```
set TASKS;

param max_time;
param time_between{i in TASKS, j in TASKS};
var scheduled_time{i in TASKS}, >=0;
```

```
var last_time;
var earlier_than{i in TASKS, j in TASKS}, binary;

minimize total_time: last_time;
s.t. one_earlier{i in TASKS, j in TASKS}:
earlier_than[i,j]+earlier_than[j,i] <= 1;
s.t. last_one{i in TASKS}: last_time >= scheduled_time[i];
s.t. time_difference_constraint{i in TASKS, j in TASKS}: # YOUR TASK IS TO COMPLETE THIS

data;
set TASKS := tA tB tC;

param max_time := 100;

param time_between :
tA tB tC :=
  tA    0  10 10
  tB    8  0  4
  tC    5  10 0;
end;
```

Here, `scheduled_time[i]` is the time at which i is scheduled; `earlier_than[i,j]` is an auxiliary binary variable that indicates whether i is scheduled before j; and the constraint you need to complete should ensure that there is enough time between i and j *unless* j is scheduled before i. `max_time` is a number that you may assume is much greater than any of the times involved in the instance. For example, one feasible solution for this instance schedules tA at time 0, tB at time 10, and tC at time 14, for an objective value of 14 (which is not optimal).

Create a file `task_scheduling.mod` for this, solve using `glpsol --math` and put the output in `task_scheduling.out`. Check that your solution makes sense.

5. (30 points.) 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