COMPSCI 323 - Computational Microeconomics

Programming Assignment 1: linear and mixed integer programming (due Jan. 31 before class)

Please read the rules for assignments on the course web page (`https://www2.cs.duke.edu/courses/spring20/compsci323d/`). Use Piazza (preferred) or directly contact Hyoung-Yoon (hk236@duke.edu), Jiali (jx76@cs.duke.edu), Caspar (cpo11@duke.edu) or Vince (conitzer@cs.duke.edu) with any questions. 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 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-4.65.tar.gz
tar -xzvf glpk-4.65.tar.gz
cd glpk-4.65
./configure
make
```

The program can then be run from the following directory.

`~/glpk/glpk-4.65/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, 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. (10 points.) Knapsack. Modify[1] the knapsack.lp file (*not* knapsack.mod) so that object B has weight 3.75kg, volume 4.25 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. (20 points.) Hot dogs. Modify the hotdogs.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. (40 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. (30 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.