Homework: Code the Simplex Algorithm

— Deadline: 23h59 19th October 2016 —

The goal of this homework is for you to code the simplex algorithm and experiment with it. The choice of the programming language is up to you: C / C++ / Java / Python. If you want to use another language, please ask us beforehand.

1 Program Specifications

Your program should accept a linear program in canonical form: the objective is a maximization function; each constraint is in the form of $\leq$; and all variables are non-negative. In other words, your input linear program will be given as follows:

$$\begin{aligned} & & \max & \quad \sum_{i=1}^{n} c_i x_i \\ \sum_{j=1}^{n} a_{ji} x_i & \leq & b_j, & \quad \forall j \in [1, m], \\ x_i & \geq & 0, & \quad \forall i \in [1, n]. \end{aligned}$$

The input file must be in the exact format described here:

```
n
m
c1 c2 ... cn
b1 b2 ... bm
a11 a12 ... a1n
a21 a22 ... a2n
...
am1 am2 ... amn
```

This input file format must be strictly followed since this will allow us to easily test your program. Your program should be able to handle input files that contain whole numbers and fractions (represented as rational numbers: p/q). Input files may also contain negative numbers.

The requirements for the format of the output file are not as strict as in the case of the input file. However, we require that your program provide two output modes. The first is the "concise" output mode. If this option is chosen by a user/tester/grader of your program, the output file must contain the following information:

- Say that the problem is INFEASIBLE, or
- Say that the problem is FEASIBLE but the solution value is UNBOUNDED, or
- If the problem is FEASIBLE and BOUNDED:
  - An optimal solution, i.e. the values of the variables $\{x_1, \ldots, x_n\}$;
  - The objective value of the optimal solution.

The second output mode is the "verbose" output mode. If this option is chosen by the user of your program, the output file must contain the following information:

- An optimal solution, i.e. the values of the variables $\{x_1, \ldots, x_n\}$;
- The objective value of the optimal solution;
- The number of pivots taken by your program;
- Which pivot rule was used to obtain this output;

- The dictionary for each pivot;
- The entering and leaving variables at each pivot.

Even if the input program is infeasible or the solution is unbounded, the verbose mode output should give some evidence for this conclusion: i.e. show dictionaries, pivots, etc.

We do not require that you include the input linear program in the output (although you can give the user the option to do this) since the input linear program may be large. Here are some examples of what input files your program must be able to handle and what output your program could return.

1.1 Example 1

For the following linear program:

$$\begin{aligned} \max \quad & 5x_1 + 4x_2 + 3x_3 & & \\ \text{subject to:} \quad & 2x_1 + 3x_2 + x_3 & \leq & 5 \\ & 4x_1 + x_2 + 2x_3 & \leq & 11 \\ & 3x_1 + 4x_2 + 2x_3 & \leq & 8 \\ & x_1, x_2, x_3 & \geq & 0. \end{aligned}$$

The input file will be formatted as follows:

```
3
3
5 4 3
5 11 8
2 3 1
4 1 2
3 4 2
```

A concise output file may look as follows:

```
OUTPUT
The input linear program is:

Maximize  5x_1 +4x_2 +3x_3
Such that 2x_1 +3x_2 +1x_3 <= 5
          4x_1 +1x_2 +2x_3 <= 11
          3x_1 +4x_2 +2x_3 <= 8
          x_1, x_2, x_3 are non-negative
An optimal solution is:  x_1 = 2, x_2 = 0, x_3 = 1
The value of the objective for this solution is:  13
The number of pivots is:  2
The pivot rule used:  Random
```

A verbose output file may look as follows:

```
OUTPUT
The input linear program is:

Maximize  5x_1 +4x_2 +3x_3
Such that 2x_1 +3x_2 +1x_3 <= 5
          4x_1 +1x_2 +2x_3 <= 11
          3x_1 +4x_2 +2x_3 <= 8
          x_1, x_2, x_3 are non-negative

The initial dictionary is:

x_4 = 5 -2x_1 -3x_2 -1x_3
x_5 = 11 -4x_1 -1x_2 -2x_3
x_6 = 8 -3x_1 -4x_2 -2x_3
----------------------------------------
z   = 0 +5x_1 +4x_2 +3x_3

The entering variable is x_1
The leaving variable is x_4

x_1 = 5/2 -1/2x_4 -3/2x_2 -1/2x_3
x_5 = 1 +2x_4 +5x_2 +0x_3
x_6 = 1/2 +3/2x_4 +1/2x_2 -1/2x_3
----------------------------------------
z   = 25/2 -5/2x_4 -7/2x_2 +1/2x_3

The entering variable is x_3
The leaving variable is x_6

x_1 = 2 -2x_4 -2x_2 +1x_6
x_5 = 1 +2x_4 +5x_2 +0x_6
x_3 = 1 +3x_4 +1x_2 -2x_6
----------------------------------------
z   = 13 -1x_4 -3x_2 -1x_6

One optimal solution is:  x_1 = 2, x_2 = 0, x_3 = 1
The value of the objective for this solution is:  13
The number of pivots is:  2
The pivot rule used:  Random
```

1.2 Example 2

For the following linear program:

$$\begin{aligned} \max \quad & 5x_1 + \frac{4x_2}{9} - 3x_3 & & \\ \text{subject to:} \quad & 2x_1 + 3x_2 + x_3 & \leq & \ 5 \\ & 4x_1 + x_2 - \frac{2x_3}{3} & \leq & \ 11 \\ & 3x_1 + 4x_2 + 2x_3 & \leq & \ 91 \\ & x_1 + x_2 & \leq & \ 2 \\ & -x_2 - x_3 & \leq & \ -6/7 \\ & -x_1 - x_2 & \leq & \ -34 \\ & x_1, x_2, x_3 & \geq & \ 0. \end{aligned}$$

The input file will be formatted as follows:

```
3
6
5 4/9 -3
5 11 91 2 -6/7 -34
2 3 1
4 1 -2/3
3 4 2
1 1 0
0 -1 -1
-1 -1 0
```

A concise output file may look as follows:

```
OUTPUT
This linear program is INFEASIBLE.
```

2 Design of Program

Here are some helpful guidelines given for this assignment in previous years. They describe functions, types and variable names that may be useful in designing your program. Note that these specifications were written for a C program.

1. *(Fractional Numbers Data Structure)* Write a new type `frac` to handle fractional numbers and the corresponding functions.
   Here is a (possibly non-exhaustive) list of functions you will need:

`frac frac_convert(int a, int b)`	Converts a couple of integers into a `frac`
`frac frac_opposite(frac f)`	Computes the opposite of `f`
`frac frac_inverse(frac f)`	Computes the inverse of `f`
`frac frac_null(void)`	Returns the fractional null element
`bool frac_isnull(frac f)`	Tests if a fractional is null
`bool frac_isneg(frac f)`	Tests if a fractional is negative
`bool frac_ispos(frac f)`	Tests if a fractional is positive
`int frac_compare(frac f1, frac f2)`	Returns 0 if $f1 = f2$, $(f2 - f1)/\|f2 - f1\|$ otherwise
`frac frac_product(frac f1, frac f2)`	Returns the product of `f1` and `f2`
`frac frac_sum(frac f1, frac f2)`	Returns the sum of `f1` and `f2`
`void frac_simplify(frac* f)`	Simplifies the fraction `f`
`void frac_print(frac f)`	Prints on the standard output the fraction `f`

2. *(Linear Program Data Structure)*
   a) *(Data Structure)* Write a new type `lp` representing a linear program and a function to print it on the standard output (`void lp_print(const lp* prog)`).
   We propose (but you are free to choose differently) a structure of a `frac` array, for the objective function, a `frac` matrix, for the constraints, along with the size of the program, *i.e.* the number of variables and the number of constraints.
   b) *(Parsing)* Write a function parsing the file given to the program.
   For example, write the function `void lp_parse(const char* filename, lp* prog)` parsing the file `filename` and filing the variable pointed by `prog` with the linear program.

3. *(Dictionary Data Structure)* Write the data structure `dict` representing a dictionary, and the corresponding functions.
   You can choose to represent a dictionary with a matrix of `frac` along with a list of basic variables and one of non-basic variables. Here is a possible list of functions:

`dict* dict_convert(const lp* prog)`	Write the initial dictionary of `prog` in `dictionary`
`void dict_do_pivot(dict* dictionary, int entering_variable, int leaving_variable)`	Apply the pivot transformation on `dictionary`
`void dict_print(const dict* dictionary)`	Prints on the standard output `dictionary`
`int* dict_get_basic(const dict* dictionary)`	Returns the array of basic variables
`int* dict_get_nonbasic(const dict* dictionary)`	Returns the array of non-basic variables

4. *(One Phase Simplex)* Write the following functions:

```
int simplex_choose_entering(const dict* dictionary)
int simplex_choose_leaving(const dict* dictionary, int entering_var)
void simplex_one_phase(const dict* dictionary)
```

   This last function should display at each step of the simplex the current dictionary.

5. *(Two Phase Simplex)* Write the complete Simplex algorithm, performing the complete algorithm and printing at the end its result:

   ```
   void simplex_solve(const lp* program)
   ```

3 Assignment

Your assignment is to program three pivot rules:

1a. Maximum Coefficient Rule: At each step the entering variable is chosen such that its coefficient in the objective function (last line of the dictionary) is maximized.

1b. Bland's Rule.

1c. Design your own pivot rule.

After implementing these pivot rules, you should thoroughly test your program on various linear programs. For eventual submission and grading, you should run your problem on the following problems:

2a. Exercises from the second exercise sheet:

- Thief problem
- Power Plants problem,
- Fraction problem.

2b. Random linear programs containing n variables and m constraints for n and m as large as possible.

2c. Find some "interesting" linear programs from any source you like (e.g. class, the internet, your friends, yourself) and run your algorithm on these linear programs. Try to find examples that show the limitations of your implementation: i.e. an input linear program that requires many pivots; an input linear program that does very well for one pivot rule but badly for Bland's Rule; an input linear program that cycles on a simple pivot rule, etc.

4 Submission

The following files will be emailed to the TAs by the deadline. You should zip the files and email to:

- `florent.brehard@ens-lyon.fr` and `ngoc-khang.le@ens-lyon.fr`

Your submission should include a readme file (filename: `lastname-README.txt`). This file should specify how to run the program on an input file from the command line. For example,

```
holland-exec -v -r input.dat
```

would be the command to run Mr. Holland's program in verbose mode using a random pivot on the input file `input.dat`.

The readme file should also contain the known limitations of your program that you have determined by thoroughly testing it. For example,

```
holland-simplex.c runs on inputs containing at most 20 variables and 1000 constraints.
The coefficients in the input file must be whole numbers (it does not work with fractional coefficients).
```

Files to be submitted:

- Readme file: `lastname-README.txt`
- Your program: `lastname-simplex.c`
- An executable file: `lastname-exec`

- A solution for the Thief problem: `lastname-thief.out`
- A solution for the Power Plant problem: `lastname-pp.out`
- A solution for the Fraction problem: `lastname-fraction.out`
- Three examples of random linear programs. Include input files and your output files: `lastname-random1.dat`, `lastname-random1.out`
- Other files containing your "interesting" examples: `lastname-example1.dat`, `lastname-example1.out`.
- A discussion file: `lastname-discusssion.txt`. It should contain answers to the following questions:
  - Explanation of your choice/design of the third pivot rule.
  - What differing behavior (if any) did you find for the various pivot rules?
  - What were the examples of linear programs that you chose [for 2c.] and why are they interesting? i.e. What notable behaviour did they cause your program to exhibit?
  - To which parameter of the program is your implementation the most sensitive? The number of variables or the number of constraints?

5 Evaluation

Your program will be evaluated as follows. 50% of the points will come from your submitted files and the other 50% will be test instances designed by the TAs. The submitted files will be graded roughly as follows:

Solutions for the Thief, Power Plants and Fraction problems. (*20 points*)

Solutions on the three random instances. (*10 points*)

Discussion of the interesting examples that you found. (*10 points*)

Design of your pivot rule: correctness of implementation/ideas. (*10 points*)

The "surprise" tests run by the TAs will be roughly divided into the following three categories:

Easy instances: ≤ 5 variables (*20 points*)

Medium instances: ≤ 20 variables (*20 points*)

Hard instances: ≤ 100 variables (*10 points*)

The test cases will contain a variety of instances so that you can receive partial credit even if your program does not meet all of the requirements. For example, if your program only works on whole numbers (and not on fractions) or if your program only works when the b_j values are positive (i.e. you did not implement Phase I of the simplex), then you should still get points for some of the test cases.

6 Collaboration Policy

You may discuss anything related to this project (e.g. how does the simplex algorithm work, ideas for new pivot rules, how to design an instance requiring a huge number of pivots) with your classmates. However, you must write your own program; you are not allowed to copy code from your classmates or from the internet for the main body of your simplex program.

If you wish to use supplemental functions (e.g. a random number generator for generating random instances, or an algebraic subroutine for your pivot rule), you may use code that you did not write yourself, but you must cite it appropriately (i.e. where did you find it and who wrote it).