Department of Mathematics and Computer Science
University of Southern Denmark, Odense

March 28, 2019
Marco Chiarandini

DM545/DM871 – Linear and integer programming

Computer Lab, Spring 2019 [pdf format]

Introduction

The assignment aims at introducing the students to mathematical modeling languages. They allow to write in an easy and compact way problems in Linear Programming (LP), Integer Programming (IP) and Mixed Integer Linear Programming (MILP) and to feed them in opportune forms to MILP solvers.
Well known mathematical programming languages are: AMPL, GAMS, ZIMPL and GNU MathProg. The last two are open source.
These languages are declarative type of languages, as opposed to procedural type. That is, we define the problem without saying how it must be solved. Moreover, they allow to separate the model from the data. In fact that is essentially all what they do, they *instantiate* the model on the given data. The output that is used by the solver can also be used for debugging purposes, that is, to check that the explicit model is as expected. There are two formats in which an instantiated MILP can be exported to a file: `.mps` format, which is an almost universal format among solver systems, but that is not easy to read, and `.lp` format, which is more readable (introduced by CPLEX).
The primary solvers for MILP are:

- IBM CPLEX (version 12.7),
- FICO XPRESS (Version 8)
- Gurobi (Version 7.5)

These are commericial solvers. Commercial solvers remain maybe 6-7 time faster than the main free/open-source solvers:

- SCIP
- MIP-CL
- Coin-CBC, part of the open-source initiative Coin-OR (`coin-or.org`)
- GLPK

The NEOS server (`https://neos-server.org/neos/`) provides an infrastructure to upload an instantiated model and solve it remotedly.

In this course, we will use Python with the module PySCIPOpt, a Python interface to the SCIP optimization software. The module PySCIPOpt is a library that tries to bring us very close to a mathematical programming language while not loosing all the nice facilities of Python.
You can work on your own laptop, in which case it is enough to install the software there. If you work on the machines of the terminal room then install the software in your IMADA home directory.
This is a list of things to do before the class:

1. install the latest version of Python (eg, 3.7).
2. choose and prepare your favourite Python Integrated Development Environment (IDE): for example, Jupyter, JupyterLab, Spyder3, Atom (with the package hydrogen), Emacs, Eclipse, Visual Code, etc.
3. install the SCIP optimization suite (you must be within SDU Net). In the computer lab, you have to install locally in your home directory, for example, under `~/opt/`.
4. install the PySCIPOpt module. In the computer lab, to install locally in your home directory you have first to make available the libraries from the SCIP software by typing in the `build` directory:

```
> DESTDIR=../install make install
```

and then installing PySCIPOpt:

```
> export SCIPOPTDIR=/home/marco/opt/scipoptsuite/scipoptsuite-6.0.1/install/usr/local
> pip3 install --user pyscipopt
```

5. take a tour through the documentation for PySCIPOpt with reference manual and examples available at: https://imada.sdu.dk/~marco/DM871/PySCIPOpt/index.html

In the session you will work in pairs. Although the exercise is about implementing the models at the computer, remember that the best practice is to first write the mathematical models on paper! Do that for the subtasks that asks you to derive a mathematical model.

Exercise 1 The Basics: Production Allocation

The following model is for a specific instance of the production allocation problem seen in the first lectures. We give here the primal and its dual model with the instantiated numerical parameters.

$$
\begin{array}{lrcrcrl}
\max & 5x_1 & + & 6x_2 & + & 8x_3 & = z \\
 & 6x_1 & + & 5x_2 & + & 10x_3 & \leq 60 \\
 & 8x_1 & + & 4x_2 & + & 4x_3 & \leq 40 \\
 & 4x_1 & + & 5x_2 & + & 6x_3 & \leq 50 \\
 & & & & & x_1, x_2, x_3 & \geq 0
\end{array}
\qquad
\begin{array}{lrcrcrl}
\min & 60y_1 & + & 40y_2 & + & 50y_3 & = u \\
 & 6y_1 & + & 8y_2 & + & 4y_3 & \leq 5 \\
 & 5y_1 & + & 4y_2 & + & 5y_3 & \leq 6 \\
 & 10y_1 & + & 4y_2 & + & 6y_3 & \leq 8 \\
 & & & & & y_1, y_2, y_3 & \geq 0
\end{array}
$$

Analysis of the final tableau

Solving one of the two problems provides the solution also to the other problem. The final tableau of the primal problem looks like this:

```
|------+----+----+------+----+----+----+-----|
|   x1 | x2 | x3 |   s1 | s2 | s3 | -z |   b |
|------+----+----+------+----+----+----+-----|
|   ?  |  1 |  0 |  ?   |  0 |  ? |  0 |   7 |
|   ?  |  0 |  1 |  ?   |  0 |  ? |  0 | 5/2 |
|   ?  |  0 |  0 |  ?   |  1 |  ? |  0 |   2 |
|------+----+----+------+----+----+----+-----|
| -0.2 |  0 |  0 | -0.2 |  0 | -1 |  1 | -62 |
|------+----+----+------+----+----+----+-----|
```

The question marks are for the values that are not relevant for the goals of this exercise.
We deduce that the primal solution is $x_1^* = 0, x_2^* = 7, x_3^* = 2.5$ and the dual solution is $y_1^* = 0.2, y_2^* = 0, y_3^* = 1$. The objective value is $z^* = u^* = 62$.

The three numbers in the last row for the columns of the non basic variables are called *reduced costs*. They indicate how much we should make each product more expensive in order to be worth manufacturing it. The next three values are known as *shadow prices*. After a change of sign they give us the values of the dual variables, which are interpreted as the *marginal value* of increasing (or decreasing) the capacities of the resources (that is, the value by which the objective function would improve if the constraint were relaxed by one unit, which corresponds to buying one unit more of resource). In our example, which seeks maximization, the marginal value 1 for the third slack variable corresponding to the third resource means that the objective function would increase by 1 if we could have one more unit of that resource.
It can be verified that in the primal problem at the optimum the first and third resources are fully exhausted, that is, their constraint is satisfied at the equality, while there is *slack* for the second resource, that is, the constraint holds with strict inequality. Looking at the marginal values, we see that the second resource has been given a zero valuation. This seems plausible, since we are not using all

the capacity that we have, we are not willing to place much value on it (buying one more unit of that resource would not translate in an improvement of the objective function).
These results are captured by the Complementary Slackness theorem of linear programming. If a constraint is not *"binding"* in the optimal primal solution, the corresponding dual variable is zero in the optimal solution to the dual model. Similarly, if a constraint in the dual model is not *"binding"* in the optimal solution to the dual model, then the corresponding variable is zero in the optimal solution to the primal model.

Solving the model with PySCIPOpt

Let's write the primal model in Python and solve it with SCIP. Here is the script:

```
#!/usr/bin/python3

import pyscipopt as pso
from pyscipopt import Model

# Model
model = Model("prod")

# Create decision variables
x1 = model.addVar(name="x1", vtype="C", lb=0.0, ub=None, obj=5.0, pricedVar = False)
x2 = model.addVar("x1", "C", 0, None, 6) # arguments by position
x3 = model.addVar(name="x3") # arguments by deafult: lb=0.0, ub=None, obj=0.

# The objective is to maximize (the default)
# Unecessary if we had written the obj coefficient for all vars above
model.setObjective(5.0*x1 + 6.0*x2 + 8.0*x3, "maximize")

# Add constraints to the model
model.addCons(6.0*x1 + 5.0*x2 + 10.0*x3 <= 60.0, "c1")
model.addCons(8.0*x1 + 4.0*x2 + 4.0*x3 <= 40.0, "c2")
model.addCons(4.0*x1 + 5.0*x2 + 6.0*x3 <= 50.0, "c3")

# Solve
model.optimize()

# Let's print the solution
if model.getStatus() == "optimal":
    print("Optimal value:", model.getObjVal())
    for v in model.getVars():
        print(v.name, " = ", model.getVal(v))
else:
    print("Problem could not be solved to optimality")
```

The documentation for the functions `Model.addVar()`, `Model.setObjective()` `Model.addCons()`, as well as for all other functions in PySCIPOpt is available from the Reference Manual and more specifically from the Model API page.
For the variable x_3 the lower bound, upper bound, objective coefficient and type are set to their default values that are, respectively: `vtype='C', lb=0.0, ub=None, obj=0.0` Once the model has been built, the typical next step is to optimize it (using `model.optimize()`). You can then query each variable value in the optimal solution via `model.getValue()` and the variable name with the attribute `name`.
If the code is in a file called `prod1_scip.py` then we can solve the model by calling:

```
> python3 prod1_scip.py
```

Parameter setting It is possible to specify the parameters of the solver (for example to set a time limit) by specifying them before solving the model. For example, we can avoid preprocessing to occur and set the primal simplex as the solution method:

```

# Let's deactivate presolving and heuristic solutions
```

```
model.setPresolve(pso.SCIP_PARAMSETTING.OFF)
model.setHeuristics(pso.SCIP_PARAMSETTING.OFF)
model.disablePropagation()
# let's use the primal simplex
model.setCharParam("lp/initalgorithm","p")
```

The list of parameters can be found at: https://scip.zib.de/doc/html/PARAMETERS.php.

Exporting Model Data to a File It is a good practice to check that everything is as we want it to be. In order to assess this, we can let PySCIPOpt write for us the set of parameters and the model instantiated with our data. This check can be very useful above all with implicit models as we will see later. Inspect the files created by these lines:

```
# Write the set of SCIP parameters and their settings.
model.writeParams("param.set")
# Write the instantiated model to a file
model.writeProblem("prod1_scip.lp") # lp format
model.writeProblem("prod1_scip.cip") # cip format
```

```
wrote parameter settings to file param.set
wrote problem to file b'prod1_scip.lp'
wrote problem to file b'prod1_scip.cip'
```

The file `param.set` contains a list of SCIP parameters that can be set as we did with `lp/initalgorithm` above. Inspecting this file can provide an insight on the parameters available as an alterntive to consult the web page: https://scip.zib.de/doc/html/PARAMETERS.php.
The files `prod1_scip.lp` and `prod1_scip.cip` contain the problem in two different formats. Valid suffixes for writing the model itself are `.mps, .rew, .lp,` or `.rlp`.

Your Task Try all these formats on the production allocation example above and check their contents. The MPS file is not very user friendly. This is because the format is an old format when computer technology had much more limitations than nowadays. The CPLEX-LP format is a more explicit version of the problem that may be useful to check whether the model we implemented in Python is actually the one we intended.
If everything runs fine you should get the following output:

```
presolving:
presolving (0 rounds: 0 fast, 0 medium, 0 exhaustive):
 0 deleted vars, 0 deleted constraints, 0 added constraints, 0 tightened bounds, 0 added holes, 0 changed sides, 0 changed coefficients
 0 implications, 0 cliques
presolved problem has 3 variables (0 bin, 0 int, 0 impl, 3 cont) and 3 constraints
      3 constraints of type <linear>
Presolving Time: 0.00

 time | node  | left  |LP iter|LP it/n| mem |mdpt |frac |vars |cons |cols |rows |cuts |confs|strbr|  dualbound   | primalbound  |  gap
* 0.0s|     1 |     0 |     2 |     - | 549k|   0 |   - |   3 |   3 |   3 |   3 |   0 |   0 |   0 | 6.200000e+01 | 6.200000e+01 |   0.00%
  0.0s|     1 |     0 |     2 |     - | 549k|   0 |   - |   3 |   3 |   3 |   3 |   0 |   0 |   0 | 6.200000e+01 | 6.200000e+01 |   0.00%

SCIP Status        : problem is solved [optimal solution found]
Solving Time (sec) : 0.01
Solving Nodes      : 1
Primal Bound       : +6.20000000000000e+01 (1 solutions)
Dual Bound         : +6.20000000000000e+01
Gap                : 0.00 %
Optimal value: 62.0
x1  =  0.0
x1  =  7.0
x3  =  2.5
```

The screen output of SCIP is described in Figure 1. If a letter appears in front of a display row, it indicates, which heuristic found the new primal bound, a star representing an integral LP-relaxation. In addition, the output indicates the amount of presolving that is conducted. Finally, the simplex method is applied and after 2 iterations of the primal simplex method (we set to use this method via `model.setCharParam("lp/initalgorithm","p")`) the optimal solution is found with value 62.

Most of the information associated with a PySCIPOpt model is stored in a set of attributes. Some attributes are associated with the variables of the model, some with the constraints of the model, and

```
SCIP> display display

 display column       header          position width priority status  description
 --------------       ------          -------- ----- -------- ------  -----------
 solfound                                    0     1    80000   auto  letter that indicates the heuristic which found the solution
 concsolfound                                0     1    80000   auto  indicator that a new solution was found in concurrent solve
 time                 time                  50     5     4000   auto  total solution time
 nnodes               node                 100     7   100000   auto  number of processed nodes
 nodesleft            left                 200     7    90000   auto  number of unprocessed nodes
 lpiterations         LP iter             1000     7    30000   auto  number of simplex iterations
 lpavgiterations      LP it/n             1400     7    25000   auto  average number of LP iterations since the last output line
 concmemused          mem                 1500     5    20000   auto  total number of bytes used in block memory
 maxdepth             mdpt                2100     5     5000   auto  maximal depth of all processed nodes
 nfrac                frac                2500     5      700   auto  number of fractional variables in the current solution
 vars                 vars                3000     5     3000   auto  number of variables in the problem
 conss                cons                3100     5     3100   auto  number of globally valid constraints in the problem
 curconss             ccons               3200     5      600   auto  number of enabled constraints in current node
 curcols              cols                3300     5      800   auto  number of LP columns in current node
 currows              rows                3400     5      900   auto  number of LP rows in current node
 cuts                 cuts                3500     5     2100   auto  total number of cuts applied to the LPs
 conflicts            confs               4000     5     2000   auto  total number of conflicts found in conflict analysis
 strongbranchs        strbr               5000     5     1000   auto  total number of strong branching calls
 dualbound            dualbound           9000    14    70000   auto  current global dual bound
 primalbound          primalbound        10000    14    80000   auto  current primal bound
 concprimalbound      primalbound        10000    14    80000   auto  current primal bound in concurrent solve
 gap                  gap                20000     8    60000   auto  current (relative) gap using |primal-dual|/MIN(|dual|,|primal|)
```

Figure 1: The columns in the output display

some with the model itself. To access these attributes you have to use the methods available under Model.

The Value of the Dual and Slack variables The value of the dual and slack variables can be accessed by the methods `model.getDualsolLinear()` and `model.getSlack()` on the constraints. In Python:

```
# Let's print the dual variables
for c in model.getConss():
    print(c.name, model.getSlack(c), model.getDualsolLinear(c))
```

to obtain

```
c1 0.0 0.19999999999999973
c2 2.0 0.0
c3 0.0 1.0000000000000004
```

These are the marginal values or shadow prices (here 0.2, 0.0 and 1) which correspond to the marginal value of the resources. The c1 and c3 constraints' value is different from zero. This indicates that there's a variable on the upper bound for those constraints, or in other terms that these constraints are *"binding"*. The second constraint is not *"binding"*. Indeed its slack is 2.

Your Task Try relaxing the value of each binding constraint one at a time, solve the modified problem, and see what happens to the optimal value of the objective function. Also check that, as expected, changing the value of non-binding constraints won't make any difference to the solution.

Your Task We can also access several quantities associated with the variables. A particularly relevant one is the *reduced cost*. Print the reduced costs of the variables for our example and make sure that they correspond to the expected values from the tableau above. [Hint: look for the method `model.getVarRedcost`.] What can we say about the solution found on the basis of the reduced costs?

Let's now focus on the values output during the execution of the simplex. Let's first solve another small numerical example:

```
#!/usr/bin/python
from pyscipopt import *

m = Model("infeas")

x = m.addVar(name="x") # ie, >= 0
y = m.addVar(name="y") # ie, >= 0

m.setObjective(x - y, "maximize")
```

```
m.addCons(x + y <= 2, "c1")
m.addCons(2*x + 2*y >= 5, "c2")

m.optimize()


if m.getStatus() in ["infeasible", "unbounded"]:
    print(m.getStatus())
elif m.getStatus() == "optimal":
    print('Optimal objective: %g' % m.getObjVal())
    print( m.getVal(x) )
    print( m.getVal(y) )
    exit(0)
elif m.getStatus() != "infeasible":
    print('Optimization was stopped with status %d' % m.getStatus())
    exit(0)
```

Solving it we obtain:

```
presolving:
presolving (1 rounds: 1 fast, 0 medium, 0 exhaustive):
 1 deleted vars, 1 deleted constraints, 0 added constraints, 2 tightened bounds, 0 added holes, 0 ch
 0 implications, 0 cliques
presolving detected infeasibility
Presolving Time: 0.00

SCIP Status        : problem is solved [infeasible]
Solving Time (sec) : 0.00
Solving Nodes      : 0
Primal Bound       : +1.00000000000000e+20 (0 solutions)
Dual Bound         : +1.00000000000000e+20
Gap                : 0.00 %
infeasible
```

This means that the presolve process has removed one column and identified the model as infeasible
Let's remove presolving and enforce to use the primal method:

```
presolving:
presolving (0 rounds: 0 fast, 0 medium, 0 exhaustive):
 0 deleted vars, 0 deleted constraints, 0 added constraints, 0 tightened bounds, 0 added holes, 0 ch
 0 implications, 0 cliques
presolved problem has 2 variables (0 bin, 0 int, 0 impl, 2 cont) and 2 constraints
      2 constraints of type <linear>
Presolving Time: 0.00

 time | node  | left  |LP iter|LP it/n| mem |mdpt |frac |vars |cons |cols |rows |cuts |confs|strbr|
  0.0s|     1 |     0 |     1 |     - | 545k|   0 |   - |   2 |   2 |   2 |   2 |   0 |   0 |   0 |
  0.0s|     1 |     0 |     1 |     - | 545k|   0 |   - |   2 |   2 |   2 |   2 |   0 |   0 |   0 |

SCIP Status        : problem is solved [infeasible]
Solving Time (sec) : 0.00
Solving Nodes      : 1
Primal Bound       : +1.00000000000000e+20 (0 solutions)
Dual Bound         : +1.00000000000000e+20
Gap                : 0.00 %
```

A feasible solution cannot be found and the problem is therefore infeasible. Try to change 5 to 2 in the right-hand-side of the second constraint of the model above. What happens? Explain the behaviour.

Your task If you have any of them installed, try solving the problem with other solvers, eg, `cplex`, `glpk` and gurobi. For this you have to use the MPS (Mathematical Programming System) format and run the following: [1]

```
cplex -c read prod.mps optimize
glpsol --mps prod.mps
scip -f prod.mps
```

Gurobi has also a command-line tool to solve model files:

```
gurobi_cl model.mps
```

You may also use the online solver at NEOS, the Network Enabled Optimization Server supported by the US federal government and located at Argonne National Lab. To submit an MPS model to NEOS visit `http://www.neos-server.org/neos/`, click on the icon "NEOS Solvers", scroll down to the Linear Programming or Mixed Integer Linear Programming list, click on one of those, scroll down to "Model File", click on "Choose File", select a file from your computer that contains an MPS model, scroll down to "e-mail address:", type in your email address, and click Submit to NEOS.

Exercise 2 The Diet Example

So far we have written models with embedded data. However, when building an optimization model, it is typical to separate the optimization model itself from the data used to create an instance of the model. These two model ingredients are often stored in completely different files.
There are alternate approaches to providing data to the optimization model: they can be embedded in the source file, read from an SQL database (using the Python sqlite3 package), or read them from an Excel spreadsheet (using the Python xlrd package) and more.

Diet Problem Bob wants to plan a nutritious diet, but he is on a limited budget, so he wants to spend as little money as possible. His nutritional requirements are as follows:
2000 Kcal
55 g protein
800 mg calcium
Bob is considering the following foods with corresponding nutritional values

	Serving Size	Price per serving	Energy (Kcal)	Protein (g)	Calcium (mg)
Oatmeal	28 g	0.3	110	4	2
Chicken	100 g	2.4	205	32	12
Eggs	2 large	1.3	160	13	54
Milk	237 cc	0.9	160	8	285
Apple Pie	170 g	2	420	4	22
Pork	260 g	1.9	260	14	80

With the help of Python/SCIP, find the amount of servings of each type of food in the diet.

In a file `dietmodel.py` we specify the model independently from the specific data. The file is reported in Listing 1. The `quicksum` function on the second line is an enhanced version of the sum function available in Python, used in Python/SCIP to do the computation of linear expressions more efficiently.
We set the data in another file, for example, `diet1.py`, reported in Listing 2. The dictionaries and lists are created at once by using the `multidict` function available in Python/SCIP. Here is an example of what it does:

```
keys, dict1, dict2 = multidict( {
    'key1': [1, 2],
    'key2': [1, 3],
    'key3': [1, 4] } )
print keys, dict1, dict2
```

[1]Standard MPS files do not indicate whether to minimize or maximize the objective. Thus your MPS files will come out the same whether the objective is minimize or maximize.
As you are seeing, most solvers minimize the objective by default. A solver may have a switch to tell it to maximize instead, but that is different for each solver.
If you change the signs of all the objective coefficients, while leaving the constraints unchanged, then minimizing the resulting MPS file will be equivalent to maximizing the original problem. This can be done easily by putting the entire objective expression in parentheses and placing a minus sign in front of it.

Listing 1: dietmodel.py

```
#!/usr/bin/python
from pyscipopt import *

def solve(categories, minNutrition, maxNutrition, foods, cost, nutritionValues):
    # Model
    m = Model("diet")

    # Create decision variables for the nutrition information,
    # which we limit via bounds
    nutrition = {}
    for c in categories:
        nutrition[c] = m.addVar(lb=minNutrition[c], ub=maxNutrition[c], name=c)

    # Create decision variables for the foods to buy
    buy = {}
    for f in foods:
        buy[f] = m.addVar(obj=cost[f], name=f)

    # The objective is to minimize the costs
    m.setMinimize()

    # Nutrition constraints
    for c in categories:
        m.addCons(
            quicksum(nutritionValues[f,c] * buy[f] for f in foods) == nutrition[c],
            c )

    def printSolution():
        if m.getStatus() == "optimal":
            print('\nCost: %g' % m.getObjVal())
            print('\nBuy:')
            for f in foods:
                if m.getVal(buy[f]) > 0.0001:
                    print('%s %g' % (f, m.getVal(buy[f])))
            print('\nNutrition:')
            for c in categories:
                print('%s %g' % (c, m.getVal(nutrition[c])))
        else:
            print('No solution')

    # Solve
    m.optimize()
    printSolution()
```

Listing 2: `diet1.py`

```
#!/usr/bin/python

from pyscipopt import *

categories, minNutrition, maxNutrition = multidict({
  'Calories': [1800, 2200],
  'Protein': [91, None],
  'Calcium': [0, 1779] })

foods, cost = multidict({
  'Oatmeal':  0.30,
  'Chicken':  2.40,
  'Eggs':  1.30,
  'Milk':  0.90,
  'Apple Pie':  2.00,
  'Pork':  1.90});

# Nutrition values for the foods
nutritionValues = {
('Oatmeal', 'Calories' ): 110,
('Oatmeal', 'Protein' ): 4,
('Oatmeal', 'Calcium' ): 2,
('Chicken', 'Calories' ): 205,
('Chicken', 'Protein' ): 32,
('Chicken', 'Calcium' ): 12,
('Eggs', 'Calories' ): 160,
('Eggs', 'Protein' ): 13,
('Eggs', 'Calcium' ): 54,
('Milk', 'Calories' ): 160,
('Milk', 'Protein' ): 8,
('Milk', 'Calcium' ): 285,
('Apple Pie', 'Calories' ): 420,
('Apple Pie', 'Protein' ): 4,
('Apple Pie', 'Calcium' ): 22,
('Pork', 'Calories' ): 260,
('Pork', 'Protein' ): 14,
('Pork', 'Calcium' ): 80 };
```

```
['key3', 'key2', 'key1']
{'key3': 1, 'key2': 1, 'key1': 1}
{'key3': 4, 'key2': 3, 'key1': 2}
```

The key construct that enables the separation of the model from the data is the Python module. A module is simply a set of functions and variables, stored in a file. One imports a module into a program using the import statement. One then executes the solve function of the dietmodel module by adding the following pair of statements at the end of the file `diet1.py`:

```
import dietmodel
dietmodel.solve(categories,minNutrition,maxNutrition,foods,cost,nutritionValues)
```

To solve the model we execute `diet1.py`.

Your Task A pill salesman offers Bob Calories, Protein, and Calcium pills to fulfill his nutritional needs. He needs to estimate the prices of units of serving, that is, the cost of 1 kcal, the cost of 1 g of protein, the cost of 1 mg of calcium. He wants to make as much money as possible, given Bob's constraints. He knows that Bob wants 2200 kcal, 55 g protein, and 1779 mg calcium. How can we help him in guaranteeing that he does not make a bad deal?

Solution:

The dual seeks to maximize the profit of the salesman. Let $y_i \geq 0$, $i \in N$ be the prices of the pills.

$$\begin{aligned} \min \quad & \sum_{j\in F} c_j x_j \\ \sum_{j\in F} a_{ij} x_j &\geq N_{min,i}, \qquad \forall i \in N \\ x_j &\geq 0, \qquad \forall j \in F \end{aligned} \qquad\qquad \begin{aligned} \max \quad & \sum_{i\in F} N_{min,i} y_i \\ \sum_{i\in N} a_{ji} y_i &\leq c_j, \qquad \forall j \in F \\ y_i &\geq 0, \qquad \forall i \in N \end{aligned}$$

However the values of the dual variables can be determined by the last tableau of the solution to the primal problem by printing the `Pi` attribute of the constraints.

Exercise 3 Particular Cases

The two following LP problems lead to two particular cases when solved by the simplex algorithm. Identify these cases and characterize them, that is, give indication of which conditions generate them in general. Then, implement the models in Python/SCIP and observe the behaviour.

$$\begin{array}{rrcrcl} \text{maximize} & 2x_1 & + & x_2 & & \\ \text{subject to} & & & x_2 & \leq & 5 \\ & -x_1 & + & x_2 & \leq & 1 \\ & & & x_1, x_2 & \geq & 0 \end{array}$$

$$\begin{array}{rrcrcl} \text{maximize} & x_1 & + & x_2 & & \\ \text{subject to} & 5x_1 & + & 10x_2 & \leq & 60 \\ & 4x_1 & + & 4x_2 & \leq & 40 \\ & & & x_1, x_2 & \geq & 0 \end{array}$$

Exercise 4 Pathological Cases

This exercise asks you to check the behavior of the solvers on the two pathological cases:

$$\begin{array}{rrcrcl} \text{maximize} & & & 4x_2 & & \\ \text{subject to} & & & 2x_2 & \geq & 0 \\ & -3x_1 & + & 4x_2 & \geq & 1 \\ & & & x_1, x_2 & \geq & 0 \end{array}$$

$$\begin{array}{rrl} \text{maximize} & 10x_1 - 57x_2 - 9x_3 - 24x_4 & \\ \text{subject to} & -0.5x_1 + 5.5x_2 + 2.5x_3 - 9x_4 & \leq 0 \\ & -0.5x_1 + 1.5x_2 + 0.5x_3 - x_4 & \leq 0 \\ & x_1 & \leq 1 \\ & x_1, x_2, x_3, x_4 & \geq 0 \end{array}$$

What happens with the solver? Can you detect which pathological cases are from the output of the solver? How?

Exercise 5 Shortest Path

Model the shortest path problem as an LP problem. Write the model in Python using the skeleton sp.py available from the directory `http://www.imada.sdu.dk/~marco/DM559/Files/SP/20points.txt` that reads data from `20points.tex`. In these data the source is node 1 and the target is node 20.
Model the problem in LP and solve it with Python/SCIP. Check the correctness of your solution with the help of the visualization in the template `sp.py`.
It may be worth looking at the example on the transportation problem.