ILP REFORMULATION

LEKSHMI, MIKE

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Table 1: Table of Given Objects

Objects	Dimensions	Range
M	nsp X nexp	0,1,-1
INPUTS	nsp X nexp	-1,1,NaN
σ	nr X 1	-1,1

Table 2: Table of Indices and Symbols

Symbol/Index	Objects
i	index of species
j	index of experiment
k	index of reaction
nsp	number of species
nexp	number of experiments
nr	number of reactions

1 INTRODUCTION

This document contains information about the implementation of Ioannis algorithm using matrices. The aim is to provide a simpler interface as a black box to use CPLEX using scipy/numpy arrays and sympy arrays. We first explain how Ioannis code is reformulated.

2 OBJECTS

This section explains the matrix objects that are being used. This is taken from the ILP formulation implemented by Ioannis.

Table 3: Table of Objects in Optimization

Symbol	Dimension	Upper bound	Lower Bound	type	Objectwt
B	nsp X nexp	1	-1	I	0
X	nsp X nexp	1	-1	I	0
X+	nsp X nexp	1	0	B	20
X−	nsp X nexp	1	0	B	20
U+	nr X nexp	1	0	B	0
U−	nr X nexp	1	0	B	0
ABS	nsp X nexp	2	0	I	100
D	nsp X nexp	100	0	C	0

Table 4: Table of Helper Matrices in Optimization

Symbol	Dimension	Condition
R	nr X nexp	$R_{k,i} = 1$ if i is a reactant in k , else $R_{k,i} = 0$
P	nr X nexp	$P_{k,i} = 1$ if i is a product in k , else $P_{k,i} = 0$
I	nsp X nexp	$I_{i,j} = 1$ if i is an input in j , else $I_{i,j} = 0$
$\mathbf{I_{nan}}$	nsp X nexp	$I_{nan_{i,j}} = 1$ if i is an input whose value is not 'NaN' in j, else $I_{nan_{i,j}} = 0$
$\mathbf{P_{nI}}$	nr X nr	Diagonal matrix where an entry on the diagonal is 1 if the product of k is not an input $I_{i,j} = 0$
$\mathbf{R_d}$	nr X nsp	$R_{d_{k,i}} = 1$ is i is a reactant in k whose product is not an input $P_{nI_{k,k}} = 0$,else $R_{d_{k,i}} = 0$
$\mathbf{P_d}$	nr X nsp	$P_{d_{k,i}} = 1$ is i is a product in k whose product is not an input $P_{nI_{k,k}} = 0$, else $P_{d_{k,i}} = 0$
$\mathbf{M_{ind}}$	nsp X nexp	$M_{ind_{i,j}} = 1$ if i is measured in j , else $M_{ind_{i,j}} = 0$
USR	nsp X nexp	$USR_{i,j} = 0$ if species i in experiment j has a reaction upstream of it, else $USR_{i,j} = 1$
notI	nsp X nexp	complement of **I**

3 CONSTRAINTS

This section lists the constraints that are being used. They are simply being re-written using the matrix notation.

$$Up - \sigma * RX \geqslant 0 \tag{1a}$$
$$Um + \sigma * RX \geqslant 0 \tag{1b}$$
$$-Up + Um \leqslant 1 \tag{1c}$$
$$Up - Um - \sigma * RX \leqslant 0 \tag{1d}$$
$$-Up + Um + \sigma * RX \leqslant 0 \tag{1e}$$
$$P^T Up - Xp \geqslant 0 \tag{1f}$$
$$P^T Um - Xm \geqslant 0 \tag{1g}$$
$$X - Xp + Xm - B = 0 \tag{1h}$$
$$X * I_{nan} = INPUTS * I_{nan} \tag{1i}$$
$$(X - B) * USR = 0 \tag{1j}$$
$$B * (notI) = 0 \tag{1k}$$
$$(X + ABS) * M_{ind} \geqslant M * M_{ind} \tag{1l}$$
$$(X - ABS) * M_{ind} \leqslant M * M_{ind} \tag{1m}$$
$$R_d D - P_d D + 101 * P_{nI} Up \leqslant 100 \tag{1n}$$
$$R_d D - P_d D + 101 * P_{nI} Um \leqslant 100 \tag{1o}$$
$$\tag{1p}$$

4 CODE

The code to use the matrix formulation as an input to CPLEX [1] makes use of scipy and numpy arrays [2] to form the helper matrices. These are then converted to symbolic matrices and the objects in the obtimization are also declared as symbolic matrices. We make use of the SYMPY package [3] to do this. We write the constraints as symbolic expressions and pass it into CPLEX. The goals achieved by this code is:

- Read in the file and make the helper matrices.

- Pass variables into CPLEX
- Pass constraints as expressions to CPLEX
- Run CPLEX optimization

The first task is done using functions from scipy and numpy arrays. The arrays are be saved using `scipy.io.mmwrite` as `.mtx` files and reused. The `R` and `P` matrix are built by reading the `.sif` file. These can saved as sparse scipy matrices. The measurement matrix and the INPUT matrix are made by reading the `measurements.txt` and `inputs.txt` files and can also stored similarly using the `scipy.io.mmwrite`. thes saved arrays and matrices are read using `scipy.io.mmread` function.
For the second and the third task, we defined a class named `tocplex` that stores reads the variables and constraints to be passed to CPLEX. The functions and the class is defined in the file named `sympybasedfns.py`. The attributes of the class are:

- `tocplex.lb` Lower bounds (list of int)
- `tocplex.ub` Upper bounds (list of int)
- `tocplex.names` Names of variables (list of str)
- `tocplex.vtypes` Types of variables (list of str)
- `tocplex.obj` Object weights for each variable (list of int)
- `tocplex.NAMEdict` Dictionary mapping each variable to an id to be used by CPLEX
- `tocplex.Makevars(self,a,n, m,UB,LB,Vtype,OBJ,name=True)` To make the variables where a is a character (name of matrix) and n and m are number of rows and columns for the matrix. If name=True would produce the each element to by indexed by species name rather than species index. If true, the n should be a list.
- `tocplex.Makecons(self,expr,sens,rhs)` To make constraints. expr is the symbolic expression, sens is a str either 'E','L' or 'G' accepted by CPLEX. rhs is a number or a sympy array indicating the rhs of the expression.
- `tocplex.cons_rows` list of rows of constraints
- `tocplex.cons_rhs` list of rhs values of constraints
- `tocplex.cons_senses` list of senses of constraints
- `tocplex.cons_len` list of number of rows of constraints passed for each symbolic expression.

Global functions which are defined in the same file called by the script are:

- `make_symbolic(a,n, m,name=True)`
- `multiply_elem(X,I,makeint=True)`
- `sci2sym(sci)`
- `getsols(sol,S)`

Note that any constraint that is made should consist only of sympy matrices. A conversion from scipy to sympy matrices using `sci2sym()` before making the constraints. The variables are passed through the `tocplex.Makevars()` function that takes in the name, dimensions, bounds and the weights and variable type as input for each variable. This takes roughly 30 seconds for all the variables.The constraints are passed as symbolic expressions through the `tocplex.Makecons()` function. For example:

```
model=tocplex()
A=model.Makevars('A',2,1,100,0,'I',0)
abs=model.Makevars('abs',1,1,100,0,'I',0)
model.Makecons(X*A,'L',b)
model.Makecons(Y*A-abs,'L',0)
```

Once the `tocplex` object contains all the variables and the constraints, pass it to the `cplex` object as usual. For example:

```
m=cplex.Cplex()
m.variables.add(obj=model.obj,ub=model.ub,lb=model.lb,types=model.vtypes,names=model.names)
m.linear_constraints.add(lin_expr=model.cons_rows,senses=model.cons_senses,rhs=model.cons_rhs)
m.solve()
```

List the solution using `getsols()`

```
nsol=m.solution.pool.get_num()
sol=[ m.solution.pool.get_values(i) for i in range(nsol)]
Asol=getsols(sol,A)
```

5 TOY EXAMPLE

A toy network is created and is solved using the above formulation. It is present in the directory named 'mytest' and 'myMatrix'. `mystudyfile.py` runs Ioannis' code on the toy network and data, while the 'myMatrix' folder contains the file `sympy_test.py` which implements the matrix formulation of writing the constraints.

6 MKN1 CELL LINE DATA

We then compare how the constraints are written for the MKN1 cell line data in the folder MKN1. We find that, the results of the presolve step in solving MIP in CPLEX produces different results. This can be due to different order of the constraints when using the matrix formulation (`http://www-01.ibm.com/support/docview.wss?uid=swg21399979`). We check this by ensuring that the constraints are generated exactly the same as in Ioannis' code and this produces identical steps in solving the problem. The script running this program is `sympybased_test.py` and `sympybased_{test.py}`. The options for the ILP are read using the file `leks_options.txt` (the options are read in different orientation, contrary to that of `ilp_options.txt`) and the inputs are read using the file `leks_inputs.txt` (this is different from that of Ioannis' because I removed an entry which was confounding: KRAS had three entries for inputs where in one instance, it was 1 and another it was denoted as 'NaN'. But this can be taken care of while making the Inputs matrix. The matrices are pre-made and are named 'InputsF.mtx' (Inputs), 'Mf.mtx'(M) ,'Mindf.mtx' (M_{ind}), 'Rf.mtx' (R) and 'Pf.mtx' (P) .

The results however, are not identical. This could be due to stochastics in the MIP solver and does not have anything to do with the way the problem is formulated.

REFERENCES

[1] IBM ILOG CPLEX Optimizer. `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/index.html`.

[2] S. Chris Colbert Stefan van der Walt and Gael Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science and Engineering*, 13:22–30, 2011.

[3] SymPy Development Team. *SymPy: Python library for symbolic mathematics*, 2014. `http://www.sympy.org`.