ECOLE
POLYTECHNIQUE
DE BRUXELLES

INFO - F424 - COMBINATORIAL OPTIMIZATION : PROJECT REPORT

Project Report

Professors:

Chicosine RENAUD

Plein Frank

Sassine NAIM

Matricule 440091

16 Mai 2020

Contents

1 Intro

The kidney assignment problem is a real life situation that many patients face on a daily basis. In fact the preferred treatment for kidney failure is transplantation. However, the demand for donor kidneys is far greater than the supply. So, for a certain patient, finding a matching donor, which needs to have some tissue-type compatibility can require sometimes a tremendous job. In fact, the act of deceased donation is called kidney exchange, which allows patients with a willing but medically incompatible living donor to swap their donor with other patients. The process is generalisable to more than two pairs, forming cycles of compatible transplants. The problem now is finding the matching pairs, and cycles, of patients and donors, to maximize the compatibility, as well as undergoing certain constraints.
The aim of this assignment is to solve the kidney assignment problem via Integer Programming techniques, with the goal of maximizing the success rate of the transplantations overall.
I am first going to start by answering the theoretical questions, and then I am going to comment on the codes I wrote, and the methods I used.

2 Theoretical Questions for Part 1

2.1 1st Question

The goal here is to maximise the success rate of the transplantations overall, while also satisfying constraints.

In terms of variables, we represent each exchange in the graph by the variable x(u,v) which is equal to 1 when an exchange between u and v is happening and 0 when it is not (u and v are eventually nodes in the graph). We also need to assign a certain "cost" to each arc between two nodes. This cost represents the compatibility of the pair of nodes. It is illustrated here by a matrix (it's not necessarily a matrix but here I used the word matrix to represent a two dimensional element) w(u,v) which gives the cost of the arc between the two nodes u and v. To maximize the success rate of the transplantations overall we need to maximize the sum of w(u,v)x(u,v) on the entire set of nodes. That is our objective function.

For the first constraint, it illustrates the flow conservation principle : if an arc comes in a node, an arc needs to come out of this node. We can clearly see in the picture below, that the first term of the sum states : "all arcs coming out of u" and for the second term "all arcs getting in u", so that the sous-traction of both is null. Means that the number of arcs coming in = number of arcs getting out. In our case, since each patient cannot give more than one kidney, it is going to be a "1-1 = 0" for the entire set of nodes V.

$$\text{s.t.} \sum_{v:(u,v)\in A} x_{uv} - \sum_{v:(v,u)\in A} x_{vu} = 0, \qquad \forall u \in V$$

Figure 1: Flow conservation constraint

For the patients, it may seem unreal to give an arc, but in our case, (p_i,d_i) arcs exist, and have a cost equal to zero, so that the constraint can be applicable, in case the patient receives an arc. The (p_i,d_i) arcs are indeed essential for flow conservation.

For the third constraint, it just states that our variable x(u,v) should be binary, so equals to 1 when an arc between the two nodes (u,v) is taken, 0 otherwise.

NB : it is not really a flow conservation constraint, since here we don't have a "real" flow, it's more of an arc conservation constraint, but I will also call it Flow conservation constraint in this report for obvious simplification reasons

2.2 2nd Question

Comparing IPs, and LPs, in general, the solution quality of the linear program is at least as good as that of the integer program, because any integer program solution would also be a valid linear program solution.

Here we relax the problem by modifying the constraint on x : instead of ordering it to be binary (0 or 1), the constraints becomes : x can take any value between 0 and 1.
We can state that the relaxation provides an optimistic bound on the integer program's solution, and in case of a maximization problem, the relaxed program has a value greater than or equal to that of the original program.

But, in some cases, where the optimal solution to the linear program happens to have all variables either 0 or 1, it will also be an optimal solution to the original integer program. This is true with totally uni modular matrix specifications.

It is clear that when A, the constraint matrix is TU, the linear programming relaxation also solves IP.

In our case, the "A" matrix is the matrix of the constraints in the figure 1 above. In this case we are facing a cost network-flow problem. We know that, from the proposition 3.4 in the reference book [4], "The constraint matrix A arising in a minimum cost network flow problem is totally unimodular".

In our case, it is not really a "minimum" cost problem but since the x variable is bounded, we can adapt this proposition. Indeed the constraints matrix in our case is a flow conservation constraint, which makes it TU since a flow conservation constraint matrix always verifies the conditions of a TU that are [3] : Let A be an m by n matrix whose rows can be partitioned into two disjoint sets B and C.

- Every entry in A is 0, +1, or -1;
- Every column of A contains at most two non-zero (i.e., +1 or -1) entries;
- If two non-zero entries in a column of A have the same sign, then the row of one is in B, and the other in C;
- If two non-zero entries in a column of A have opposite signs, then the rows of both are in B, or both in C.

2.3 3rd Question

From the flow conservation constraint, and also taking into consideration that a donor can only give one kidney, a patient can only receive one kidney(in this case) and that (p_i, d_i) arcs are possible, (some may think that it's not possible because it means "a patient gives a kidney" but here we use it, but not in that way) we can say that each node has maximum one arc coming out, and maximum one arc coming into the node.

Then, we can clearly see that our graph will be composed of cycles, since a donor needs to give a kidney to a patient, a patient(i) needs to connect to its donor(i), which will also need to give a kidney (number of arcs conserved) ...

The cycles will also be disjoint, because if any node where to be in two different cycles, it means that it will have more than one arc coming out of it or getting in it, which will violate the constraints stated above (maximum one arc coming in, maximum one arc getting out, the flow conservation).

3 Theoretical Questions for Part 3

3.1 4th Question

For this IP, it is approximately the same as the IP in the first section of the project, but here we have a maximum number M of exchanges possible per cycle of transplants.

For the formulation, it is the exact same formulation as for the 1st part of the project, +1 constraint. So here we add one constraint which needs to model the fact that we want a maximum number M of exchanges possible per cycle of transplants.

This constraint, which is the (3)rd constraint, clearly states the fact that if a cycle has more than M exchanges (or 2*M arcs between donors/patients or patients/donors) the sum of the arcs in this cycle (which is the size of the cycle) should be lower than the size of the cycle - 1.

This condition clearly puts out all the cycles that have a number of exchanges bigger than M, because it will directly eliminate the cycle with the condition that the number of arcs should be lower than the size of the cycle -1.

4 Comments about the different Parts of the Project

NB : If you already read the code, or you will read my code later, sorry if it's a bit long, but I divided it into many functions, so it could be as clean and readable as possible (that's why it's taking so many lines). Also for the kep_algo.py, I coded two different methods that

I will explain later, that is why it is unusually long.

In the section below, you will find all my comments and explications about the code and the different methods that I implemented in the different parts of the project. I also tried to heavily comment my code, so it could be easily understandable.

4.1 Section 1 A Basic Model

For this first part of the project, we had to use the *pyomo* library to build a model for the formulation, as well as use the *GLPK* solver to solve it. This first model is implemented in the kep_ip.py file, where all the used functions are defined at the beginning of the program. The code first starts by scanning the given csv file, that contains the different arcs, and the number of nodes. By scanning this file we retrieve different information :

- the different arcs and their costs
- the number of nodes
- the number of compatibilities (d_i, p_j)
- and for the Section 3, the maximum number of exchanges per cycle of transplants

By retrieving this data, we can now build the set of arcs, the set of nodes as well as a dictionary that will contain each arc with it's cost value. I faced a small problem which was that the graph is a directed graph, so arc (0,0) which is donor 0 -> patient 0 is different from arc (0,0) which is patient 0 -> donor 0. Of course, python won't recognize the difference between those two tuples, so to differentiate between patients and donors, I sum n to the node number of each patient. For example, for the arc patient 0 -> donor 0, with n = 5 (n is the number of patient-donor pairs) I represent this arc with the tuple (5,0) which is different from the tuple (0,5) which represents donor 0 -> patient 0.
We then build a pyomo model, that is intuitive when comparing to the IP model we have :

- I start by defining the model as a concrete model, here I chose a concrete model so I could give the necessary data during the buildup of the model
- I then define intuitively the node_set and the arc_set as the set of arcs and nodes respectively.
- After that I define our variable X that will represent the variable x(u,v) in the model.

- I finish its building by writing the objective function that needs to be maximised as well as the different constraints that need to be verified
- I finally call the *SolveFactory* method to solve the problem, and return the optimal version of the variable X
- Once the optimal value of X calculated, I store it into a sort of a list, to be able to convert it to the formulation asked in the project brief

4.2 Section 2 Positive Cycle Elimination Procedure

For this part of the project, we had to to implement the positive cycle elimination procedure, using the Networkx package. This package is very useful when dealing with graphs, since it lets you visualise your entire graphs, determine the cycles, assign weights ...

For my code, it first starts with a definition of all the functions that I used (and built) during the code. Then comes the two main functions : *solve* and *solveBeta*. When you run the code with the kep_solve.py file, only the *solve* function will be executed, but let me explain why I wrote the two functions.

In the positive cycle elimination procedure, it is possible to have common arcs between two sets of arcs : the forward arcs and the backward arcs that we want to join. Sadly, networks.DiGraph, the package that is used here, does not allow for duplicate arcs to happen in a graph. So I had to figure out a solution for those duplicate arcs. I imagined, and implemented two different solutions for this problem. They are similar in some ways, and different in other ways, but they both give the exact, and correct result. I will explain in these subsections below the difference between each and everyone of them, and why did I choose one over the other.

4.2.1 First method : solve function

First of all, like in the kep_ip.py file, I scan the given csv file to collect the data that is needed, and with that data I start by building the set of nodes, the set of arcs, the dictionary containing the cycle costs ... I then construct a feasible solution for the given data. This feasible solution is generated with the *feasibleSolution* method that takes as input the dictionary of arcs and costs, as well as the number of (pi,di) pairs, and outputs

a feasible solution that I chose. In this case the feasible solution is just the connection of all the (di,pi) and (pi,di) as long as they follow the constraints of the formulation (flow conservation typically).

The code enters then a while loop,that will only stop when no positive cost cycle can be found in the residual graph. Inside the while loop, we build two residual graphs.
To face the double arcs problem, my first method consisted in the following :

- Build the forward arcs and backward arcs based on the feasible solution
- Build two residual graphs joining both of the set of arcs mentioned above
- For the first graph, let the common arcs be in the forward arcs, so when defining their cost, we consider them as forward arcs only
- For the second graph, let the common arcs be in the backward arcs, so when defining their cost, we consider them as backward arcs only
- For each of the two graphs mentioned above, determine the cycles in the graph, for every cycle, calculate its cost and take the cycle with the maximum cost value
- Build x(C) with the cycle found in the previous step (so basically here we should have two x(C) one for the first graph, and one for the second graph)
- Compare the two x(C) obtained from the two different graphs, and choose the one that will augment the objective value of the already determined solution x
- Once the best graph chosen, we replace the solution x with x + x(C) and we loop
- The loop will eventually stop when no positive cost cycle can be found in the residual graph

In resume, the method implemented in the solve function, consists in classifying all common arcs once as forward arcs, once as backward arcs, calculate the x(C) for both of them, and them choose the best one.

4.2.2 Second method : solveBeta function

For the solveBeta function, the beginning of the method is the same as the first method, what changes is the inside of the while loop. In this method, instead of exploring only two possibilities, we explore all of them. For example, imagine the set of forward arcs is : (1,2)

(2,3) and (4,5) only three arcs, and the set of backward arcs is (6,7) (2,3) (4,5). We have two common arcs which are (2,3) and (4,5). In the first method, what we did was build two residual graphs, the first where (2,3) and (4,5) are both taken as forward arcs, the second where both are taken as backward arcs. In this second method, we build 4 residual graphs (2^n where n is the number of common arcs between forward and backward arcs). So in this second method, we build all the possible and imaginable residual graphs. With this method, we can take into consideration every possible way of classifying the arcs into backward or forward.
For the rest of the algorithm, it is very similar to the first one, where we calculate for each possible graph, the cycles, the costs ... and then chose the x(C) that will maximize the objective value of x.

4.2.3 Comparison

I know the implementation of two methods was not necessary, but I first built the first method, than I realised that maybe the algorithm could get more accurate if we took into consideration the entire set of possible solutions. In our case, the results were the same. Both methods returned the right set of arcs to be taken, and the same optimal solution. But the second method was of course slower, since it needs to test out more than two graphs. That is why I chose the first method as the main *solve* method, so it could give you the result faster while you correct! But the solveBeta function is totally usable, but it will take much more time, specially for the large.csv file.

4.3 Section 3 Implementability

For the last part of the project, the main goal was to implement a new model, very similar to the first one, but that takes into consideration the maximum number M of exchanges possible per cycle of transplants. The model took this into account by adding the constraint below.

$$\sum_{(u,v)\in C} x_{uv} \leqslant |C| - 1, \qquad \forall C \text{ cycle of } G \text{ such that } |C| > 2M$$

Figure 2: Maximum number of exchanges

To implement this restriction in my code, I started off with the model that I already had in the first part of the project and I then implemented the algorithm given in the

project brief. Here are the different steps of the algorithm that I implemented to model this formulation :

- First, build the pyomo model, very similar to the first formulation, but here we add a pyomo set *cycle_num* that will store the number corresponding to a certain cycle. Here this number will reference to a much bigger list *p_cycle* that will store the complete cycles. For example p_cycle will contain : [[0,1,2,6,0], [7,5,3,7], [8,9,4,8]] which are three different cycles represented by their nodes, *cycle_num* will then contain [0,1,2] in reference to each cycle.

- The objective rule is the same, but we add one more constraint, this constraint should illustrate the fact that if a cycle has more than 2*M arcs (this means that it completes more than M exchanges), we should not take it. We achieve this negation by ordering that the number of arcs in the cycle should be less or equal than the length of the cycle - 1. This obviously is always false, so for each cycle that has more than M exchanges, its arcs will not be taken.

- Now that we modified accordingly the pyomo model by including a cycle set and a new constraint, we need to find the cycles in our graph and add to *cycle_num* the cycles that have a number of arcs bigger than 2*M.

- To find the different cycles in a graph, I used a Depth First Search algorithm. This algorithm, starts off on a starting node, and using the adjacency function, will go through the adjacent nodes of the starting node. It will go through the graph accordingly, using the adjacent nodes and the different given arcs, and label the visited nodes. If it stumbles on the starting node, a cycle is found. DFS was chosen due to its simplicity and its basic implementation. It may not be the fastest algorithm, but in our case here it performed well, since the number of nodes and arcs was not very high. But we could of course imagine another algorithm who could do a better job (BFS, A* ....)

4.4 Results

This section contains screenshots of the results I got for the different implementations and files.

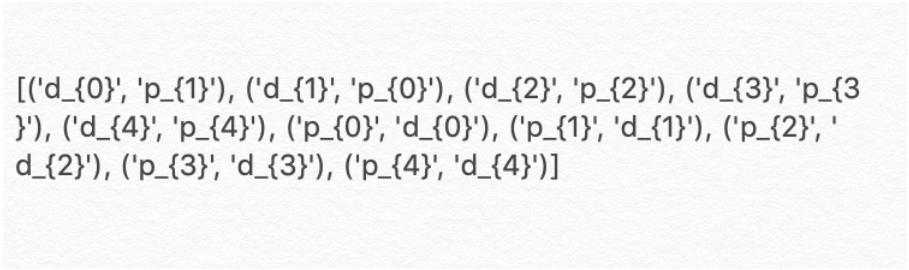

Figure 3: First implementation - small.csv : OBJ = 3.3

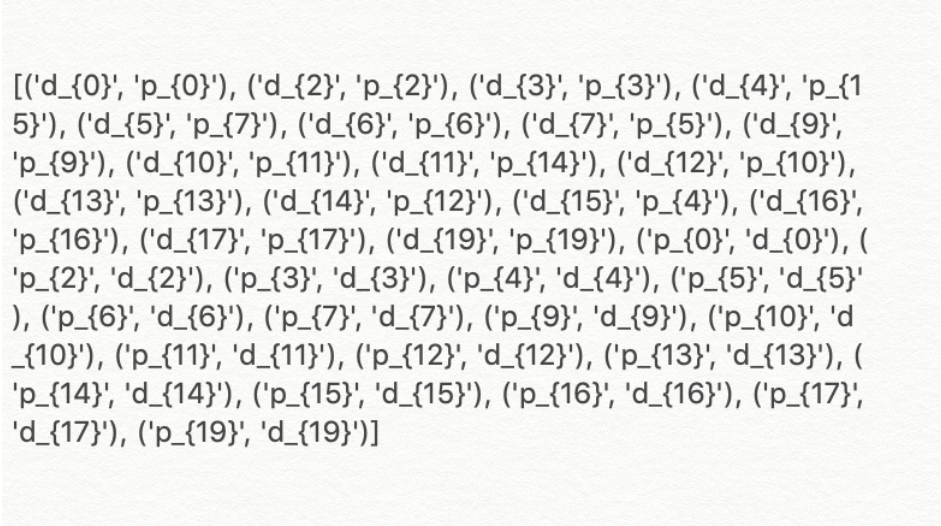

Figure 4: First implementation - normal.csv : OBJ = 12.2

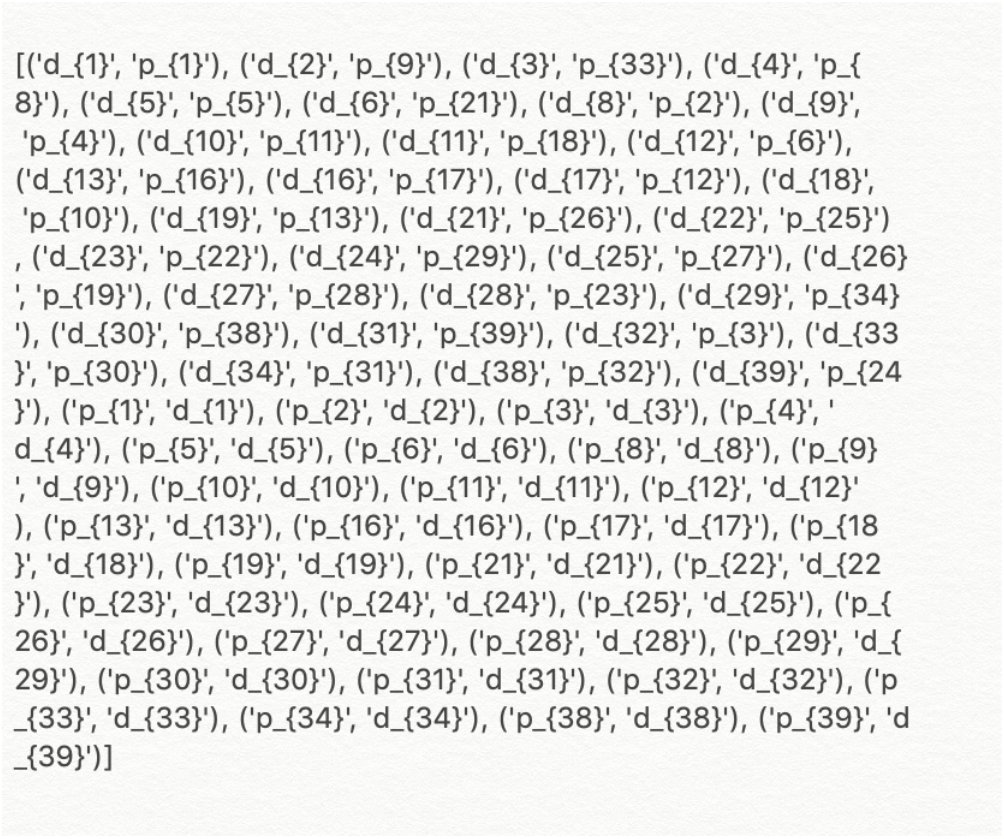

Figure 5: First implementation - large.csv : OBJ = 21.5

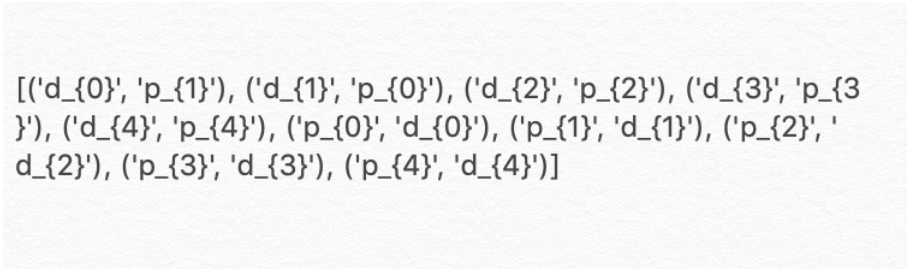

Figure 6: Positive Cycle Elimination - small.csv : OBJ = 3.3

```
[('d_{0}', 'p_{0}'), ('d_{2}', 'p_{2}'), ('d_{3}', 'p_{3}'), ('d_{4}', 'p_{1
5}'), ('d_{5}', 'p_{7}'), ('d_{6}', 'p_{6}'), ('d_{7}', 'p_{5}'), ('d_{9}',
'p_{9}'), ('d_{10}', 'p_{11}'), ('d_{11}', 'p_{14}'), ('d_{12}', 'p_{10}'),
('d_{13}', 'p_{13}'), ('d_{14}', 'p_{12}'), ('d_{15}', 'p_{4}'), ('d_{16}',
'p_{16}'), ('d_{17}', 'p_{17}'), ('d_{19}', 'p_{19}'), ('p_{0}', 'd_{0}'), (
'p_{2}', 'd_{2}'), ('p_{3}', 'd_{3}'), ('p_{4}', 'd_{4}'), ('p_{5}', 'd_{5}'
), ('p_{6}', 'd_{6}'), ('p_{7}', 'd_{7}'), ('p_{9}', 'd_{9}'), ('p_{10}', 'd
_{10}'), ('p_{11}', 'd_{11}'), ('p_{12}', 'd_{12}'), ('p_{13}', 'd_{13}'), (
'p_{14}', 'd_{14}'), ('p_{15}', 'd_{15}'), ('p_{16}', 'd_{16}'), ('p_{17}',
'd_{17}'), ('p_{19}', 'd_{19}')]
```

Figure 7: Positive Cycle Elimination - normal.csv : OBJ = 12.2

```
[('d_{1}', 'p_{1}'), ('d_{2}', 'p_{9}'), ('d_{3}', 'p_{33}'), ('d_{4}', 'p_{
8}'), ('d_{5}', 'p_{5}'), ('d_{6}', 'p_{21}'), ('d_{8}', 'p_{2}'), ('d_{9}',
 'p_{4}'), ('d_{10}', 'p_{11}'), ('d_{11}', 'p_{18}'), ('d_{12}', 'p_{6}'),
('d_{13}', 'p_{16}'), ('d_{16}', 'p_{17}'), ('d_{17}', 'p_{12}'), ('d_{18}',
 'p_{10}'), ('d_{19}', 'p_{13}'), ('d_{21}', 'p_{26}'), ('d_{22}', 'p_{25}')
, ('d_{23}', 'p_{22}'), ('d_{24}', 'p_{29}'), ('d_{25}', 'p_{27}'), ('d_{26}
', 'p_{19}'), ('d_{27}', 'p_{28}'), ('d_{28}', 'p_{23}'), ('d_{29}', 'p_{34}
'), ('d_{30}', 'p_{38}'), ('d_{31}', 'p_{39}'), ('d_{32}', 'p_{3}'), ('d_{33
}', 'p_{30}'), ('d_{34}', 'p_{31}'), ('d_{38}', 'p_{32}'), ('d_{39}', 'p_{24
}'), ('p_{1}', 'd_{1}'), ('p_{2}', 'd_{2}'), ('p_{3}', 'd_{3}'), ('p_{4}', '
d_{4}'), ('p_{5}', 'd_{5}'), ('p_{6}', 'd_{6}'), ('p_{8}', 'd_{8}'), ('p_{9}
', 'd_{9}'), ('p_{10}', 'd_{10}'), ('p_{11}', 'd_{11}'), ('p_{12}', 'd_{12}'
), ('p_{13}', 'd_{13}'), ('p_{16}', 'd_{16}'), ('p_{17}', 'd_{17}'), ('p_{18
}', 'd_{18}'), ('p_{19}', 'd_{19}'), ('p_{21}', 'd_{21}'), ('p_{22}', 'd_{22
}'), ('p_{23}', 'd_{23}'), ('p_{24}', 'd_{24}'), ('p_{25}', 'd_{25}'), ('p_{
26}', 'd_{26}'), ('p_{27}', 'd_{27}'), ('p_{28}', 'd_{28}'), ('p_{29}', 'd_{
29}'), ('p_{30}', 'd_{30}'), ('p_{31}', 'd_{31}'), ('p_{32}', 'd_{32}'), ('p
_{33}', 'd_{33}'), ('p_{34}', 'd_{34}'), ('p_{38}', 'd_{38}'), ('p_{39}', 'd
_{39}')]
```

Figure 8: Positive Cycle Elimination - large.csv : OBJ = 21.5

```
[('d_{0}', 'p_{0}'), ('d_{2}', 'p_{2}'), ('d_{3}', 'p_{3}'), ('d_{4}', 'p_{4
}'), ('p_{0}', 'd_{0}'), ('p_{2}', 'd_{2}'), ('p_{3}', 'd_{3}'), ('p_{4}', '
d_{4}')]
```

Figure 9: Part 3 - small.csv - OBJ = 3.2

```
[('d_{0}', 'p_{0}'), ('d_{2}', 'p_{2}'), ('d_{3}', 'p_{3}'), ('d_{4}', 'p_{1
5}'), ('d_{5}', 'p_{7}'), ('d_{6}', 'p_{6}'), ('d_{7}', 'p_{5}'), ('d_{9}',
'p_{9}'), ('d_{10}', 'p_{10}'), ('d_{12}', 'p_{13}'), ('d_{13}', 'p_{14}'),
('d_{14}', 'p_{12}'), ('d_{15}', 'p_{4}'), ('d_{16}', 'p_{16}'), ('d_{17}',
'p_{17}'), ('d_{19}', 'p_{19}'), ('p_{0}', 'd_{0}'), ('p_{2}', 'd_{2}'), ('p
_{3}', 'd_{3}'), ('p_{4}', 'd_{4}'), ('p_{5}', 'd_{5}'), ('p_{6}', 'd_{6}'),
 ('p_{7}', 'd_{7}'), ('p_{9}', 'd_{9}'), ('p_{10}', 'd_{10}'), ('p_{12}', 'd
_{12}'), ('p_{13}', 'd_{13}'), ('p_{14}', 'd_{14}'), ('p_{15}', 'd_{15}'), (
'p_{16}', 'd_{16}'), ('p_{17}', 'd_{17}'), ('p_{19}', 'd_{19}')]
```

Figure 10: Part 3 - normal.csv - OBJ = 11.4

```
[('d_{1}', 'p_{1}'), ('d_{2}', 'p_{9}'), ('d_{3}', 'p_{33}'), ('d_{4}', 'p_{
8}'), ('d_{5}', 'p_{5}'), ('d_{6}', 'p_{6}'), ('d_{8}', 'p_{2}'), ('d_{9}',
'p_{4}'), ('d_{10}', 'p_{11}'), ('d_{11}', 'p_{18}'), ('d_{12}', 'p_{14}'),
('d_{13}', 'p_{16}'), ('d_{14}', 'p_{17}'), ('d_{16}', 'p_{19}'), ('d_{17}',
 'p_{12}'), ('d_{18}', 'p_{10}'), ('d_{19}', 'p_{13}'), ('d_{20}', 'p_{29}')
, ('d_{22}', 'p_{27}'), ('d_{23}', 'p_{22}'), ('d_{24}', 'p_{26}'), ('d_{26}
', 'p_{24}'), ('d_{27}', 'p_{28}'), ('d_{28}', 'p_{23}'), ('d_{29}', 'p_{20}
'), ('d_{30}', 'p_{32}'), ('d_{32}', 'p_{3}'), ('d_{33}', 'p_{30}'), ('d_{34
}', 'p_{36}'), ('d_{36}', 'p_{34}'), ('p_{1}', 'd_{1}'), ('p_{2}', 'd_{2}'),
 ('p_{3}', 'd_{3}'), ('p_{4}', 'd_{4}'), ('p_{5}', 'd_{5}'), ('p_{6}', 'd_{6
}'), ('p_{8}', 'd_{8}'), ('p_{9}', 'd_{9}'), ('p_{10}', 'd_{10}'), ('p_{11}'
, 'd_{11}'), ('p_{12}', 'd_{12}'), ('p_{13}', 'd_{13}'), ('p_{14}', 'd_{14}'
), ('p_{16}', 'd_{16}'), ('p_{17}', 'd_{17}'), ('p_{18}', 'd_{18}'), ('p_{19
}', 'd_{19}'), ('p_{20}', 'd_{20}'), ('p_{22}', 'd_{22}'), ('p_{23}', 'd_{23
}'), ('p_{24}', 'd_{24}'), ('p_{26}', 'd_{26}'), ('p_{27}', 'd_{27}'), ('p_{
28}', 'd_{28}'), ('p_{29}', 'd_{29}'), ('p_{30}', 'd_{30}'), ('p_{32}', 'd_{
32}'), ('p_{33}', 'd_{33}'), ('p_{34}', 'd_{34}'), ('p_{36}', 'd_{36}')]
```

Figure 11: Part 3 - large.csv - OBJ = 19.5

References

[1] Overview of NetworkX — NetworkX 2.4 documentation.

[2] Pyomo Documentation 5.6.9 — Pyomo 5.6.9 documentation.

[3] Unimodular matrix - Wikipedia.

[4] Laurence A. Wolsey. *Integer programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, New York, 1998.