METHODS AND MODELS FOR COMBINATORIAL OPTIMIZATION SIMPLE (FOR REAL)

GABRIEL ROVESTI

1 Table of Contents

Disclaimer

This is the last ever file I wrote as of notes for courses, so I hope I matured enough having written this, for every single course I have done up to now, bachelor and master included (and hope you saw and used one, I tried my best always). So, I hope to have done something useful with these – I know that I helped many and if I helped you I am glad, even though I was directly insulted and attacked by the few people claiming to help others. I do not pretend to be exact, correct or precise, I think I am enough though, just by reading one of these files you can understand it.

All of references are present and given it's an informal file, shared between students, one does not need a complete bibliography like papers – that I also wrote in other cases – for everything. Many things can be found between the web or existing notes. But I think I did something useful, profound, motivated by the will of community and helping both you and me, enthusiastic of learning and writing (I think if you are not totally shallow to just take the credits and discredit unfairly other people you can easily see that). I am leaving this to you. In case of feedback of whatever kind, contact me no problem. Even to disagree with me, I'm always open for confrontation.

In any case: the file is organized not chronologically, but more content-wise, so you have theory first, laboratories second (which were done many times coupled with the theory in detail, but for reading sakes they are moved after the entire theory modules), so to immediately jump to content of your interest when needed.

Another thing: many things were also translated by the professor notes in Italian (particularly, for the metaheuristics part, partial in English, here complete with the class examples coming from recordings) and also the Italian notes I recovered from GitHub of this course, written by Manzoli, which are amazing in their own right and definitely useful for the most part (can recommend). If possible, I tried, in my case as always if you read (for real) at least one file, to create a complete resource (or as complete as possible) to be the only material of your reference. Slides not put by the professor were taken by me from recordings and put on MEGA/Telegram.

The course was definitely a lot to work upon (but satisfying regardless): the theory doesn't seem like it in the beginning, but as seen by the file length and contents, well, you have to give a shit about that. Also, the project, particularly for the second part, will be definitely time-consuming and something which you have to work upon for quite a good amount of time.

There is a dedicated (informal) lesson for that at the very end of the course – since the professor himself is very knowledgeable but also very long in presenting lessons, each year some topics are cut out simply because of lack of time. Just telling you, latest lessons starting from October were all the way up to the half of January – and always at least 1.20/1.30 of lesson. So yeah.

That's all folks! – fucking finally, let me say!

2 INTRODUCTION (1)

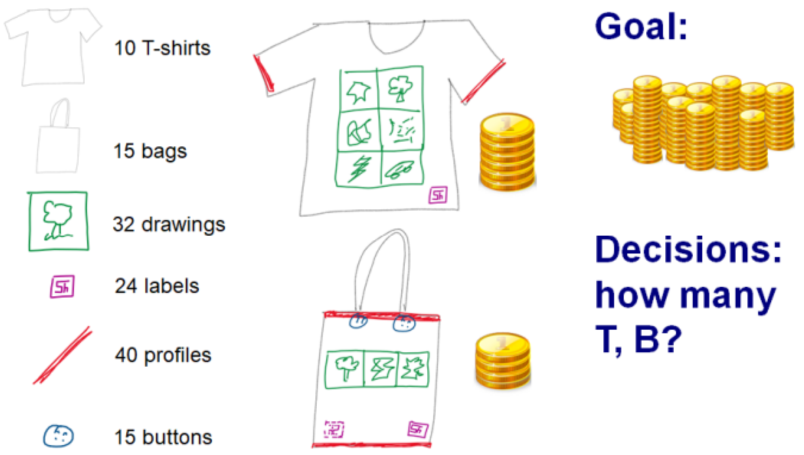
We start from the concept of combinatorial optimization: given a problem, find the best solution inside of a set. This problem often occurs inside different fields.

- There is no best solution starting from the beginning, neither does the number matter: an optimal solution has to be determined among a number of alternatives that *combinatorially explodes* – consider for example the bike sharing rebalancing problem here
- Mathematics provides tools in order to solve problems practically from the real point of view
- After this course, one is expected to have the "ability to search for, find, understand, adapt and implement state-of-the-art approaches to solve real-world combinatorial optimization problems"

Problems seem different, but in reality they are similar:

- *Logistic and transportation network*: optimal origin-destination paths, optimal pickup/delivery routes, line configuration, driver scheduling
- *Production management*: production and resource planning, job shop scheduling, optimal cutting patterns
- *Machine learning*: optimal structure and weight of neural networks, clustering algorithms
- *Data-driven decision making*: cooling schedule based on massive simulation, air traffic management based on trajectory repositories
- *Optimization on graphs and networks*: coloring, cliques, quickest paths, multicommodity flows
- *Telecommunication networks*: telecom-facility location, virtual network configuration, optimal routing
- *Complex network analysis*: community detection, maximize influence
- and many others...

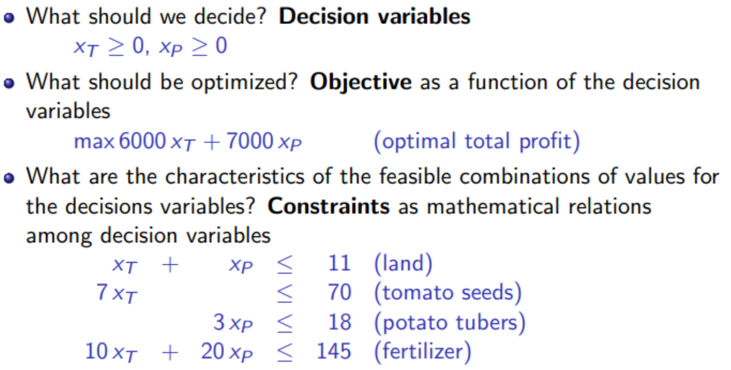
A toy problem is the following, combining constraints with the goal of earning as much as possible, with no costs since all resources are there:



The model can be solved by the Simplex algorithm, in order to iterate until no more solutions are present, in which we make inequalities in order to represent constraints and then understand the feasibility of all of them.

Given a problem, we do not go into its details; given any problem, *how to manage the combinatorial explosion of the size of the solution space using a unifying approach*?

Consider a problem:



- What should we decide? **Decision variables**

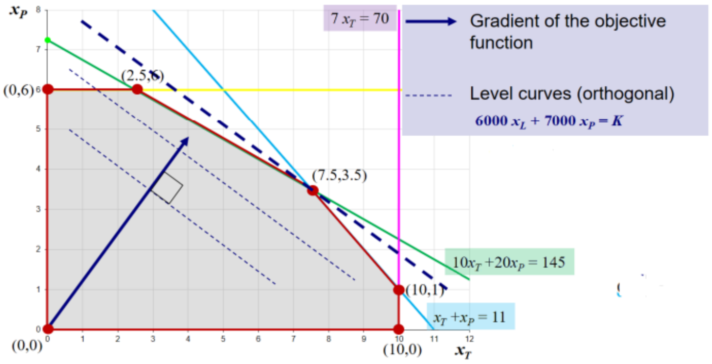
  $x_T \geq 0,\ x_P \geq 0$

- What should be optimized? **Objective** as a function of the decision variables

  $\max 6000\,x_T + 7000\,x_P$ (optimal total profit)

- What are the characteristics of the feasible combinations of values for the decisions variables? **Constraints** as mathematical relations among decision variables

$$\begin{array}{rcrcll} x_T & + & x_P & \leq & 11 & \text{(land)} \\ 7\,x_T & & & \leq & 70 & \text{(tomato seeds)} \\ & & 3\,x_P & \leq & 18 & \text{(potato tubers)} \\ 10\,x_T & + & 20\,x_P & \leq & 145 & \text{(fertilizer)} \end{array}$$



There are different exact methods to solve problems with integer variables: Cutting planes [improved geometry], branch-and-bound [implicit enumeration] (computational resources!) – they may take long computational times, since they are NP-Hard problems.

A viable choice might be focusing on *discrete* choices, so to apply heuristic methods: exact methods may be theoretically and computationally critical, heuristics still work.

Keep an eye to the course programme and general info here.

3 MODELING BY LINEAR PROGRAMMING (2)

A toy example is the following, coming from the Introduction set of slides – here, the variable are continuous – keep in mind:

"A farmer owns 11 *hectares of land where he can grow potatoes or tomatoes. Beyond the land, the available resources are:* 70 *kg of tomato seeds,* 18 *tons of potato tubers,* 145 *tons of fertilizer. The farmer knows that all his production can be sold with a profit of* 6000 *Euros per hectare of tomatoes and 7000 Euros per hectare of potatoes. Each hectare of tomatoes needs* 7 *kg seeds and* 10 *tons fertilizer. Each hectare of potatoes needs* 3 *tons of tubers and* 20 *tons fertilizer. How does the farmer divide his land in order to gain as much as possible from the available resources?"*

We will translate such model as seen above with the following ones:

- Decision variables, symbolizing the decisions to be made
- Objective, meaning what we would have to optimize
- Constraints, which is a system of inequalities useful for the solution in order to be feasible

This all works because both the constraints and the objective function are linear, and the variables are real numbers. This type of model is therefore called Linear Programming.

Note that in this case the optimal solution is on an integer vertex, but it's only the case. If the variables used can only be integers the situation becomes more complex because approximations must be made. If we have the models, we have the solutions – so, let's focus on the models.

3.1 DESCRIPTION AND FEATURES

More specifically, a mathematical programming model describes the characteristics of the optimal solution of an optimization problem by means of mathematical relations. It provides formulation and a basis for standard optimization algorithms.

- Sets: they group the elements of the system
- Parameters: the data of the problem, which represent the known quantities depending on the elements of the system
- Decision (or control) variables: the unknown quantities, on which we can act in order to find different viable solutions to the problem
- Constraints: mathematical relations that describe solution feasibility conditions (they distinguish acceptable combinations of values of the variables)
- Objective function: quantity to maximize or minimize, as a function of the decision variables

Mathematical programming models where:

- The objective function is a linear expression of the decision variables
- The constraints are a system of linear equations and/or inequalities

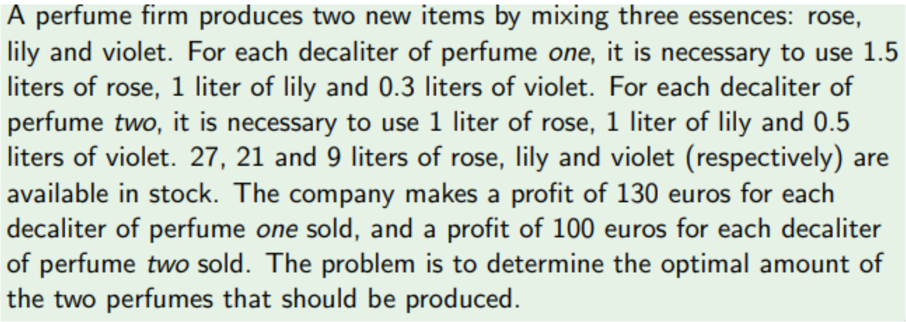
Classification of linear programming models:

- Linear Programming models (LP): all the variables can take real (ℝ) values
    - There are poly-time algorithms for these – easy
- Integer Linear Programming models (ILP): all the variables can take integer (ℤ) values only
    - These ones are NP-Hard – not easy
- Mixed Integer Linear Programming models (MILP): some variables can take real values and others can take integer values only
    - These are more difficult

3.2 OPTIMAL PRODUCTION MIX – PERFUMES

Linearity limits expressiveness but allows faster solution techniques – if we are able to linearize models, they are simpler. From now on, we will discuss modeling schemas.

Consider the following example:



A perfume firm produces two new items by mixing three essences: rose, lily and violet. For each decaliter of perfume *one*, it is necessary to use 1.5 liters of rose, 1 liter of lily and 0.3 liters of violet. For each decaliter of perfume *two*, it is necessary to use 1 liter of rose, 1 liter of lily and 0.5 liters of violet. 27, 21 and 9 liters of rose, lily and violet (respectively) are available in stock. The company makes a profit of 130 euros for each decaliter of perfume *one* sold, and a profit of 100 euros for each decaliter of perfume *two* sold. The problem is to determine the optimal amount of the two perfumes that should be produced.

The variables here are continuous, since we are deciding the decalitres of perfume, so we call them:

- x_{one}, x_{two} for the two quantities of decalitres

Since they are real, we call them: $x_{one}, x_{two} \in \mathbb{R}$.

There are constraints here, since these ones need to be respected in order for the model to be valid; for example, on the availability on the liter of rose, we have 1.5 liters of rose and one liter of rose:

$$1.5x_{one} + x_{two} \le 27$$

Off we go with the other variables, so we have:

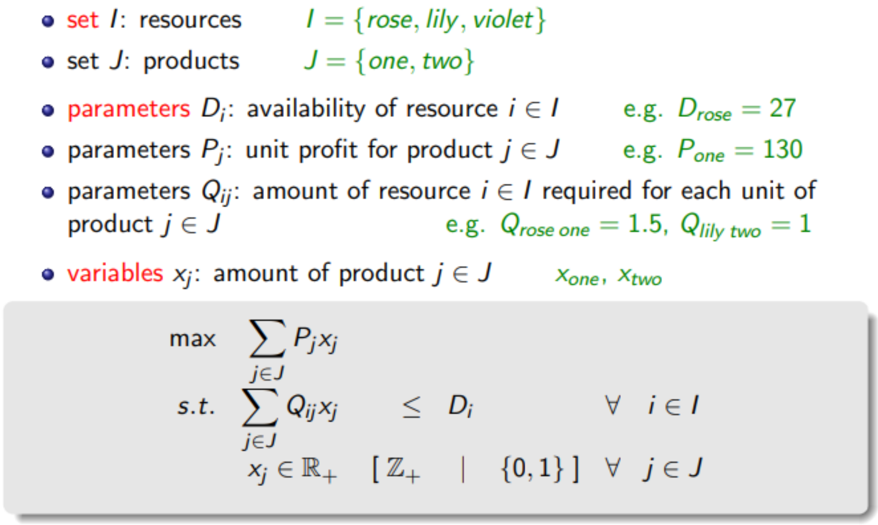
$$1.5x_{one} + x_{two} \le 27 \quad (rose)$$

$$x_{one} + x_{two} \le 21 \quad (lily)$$

$$0.3x_{one} + 0.5_{two} \le 9 \quad (violet)$$

$$x_{one}, x_{two} \ge 0 \quad (domains\ of\ variables)$$

A possible modeling schema is the following, to represent the optimal production mix:



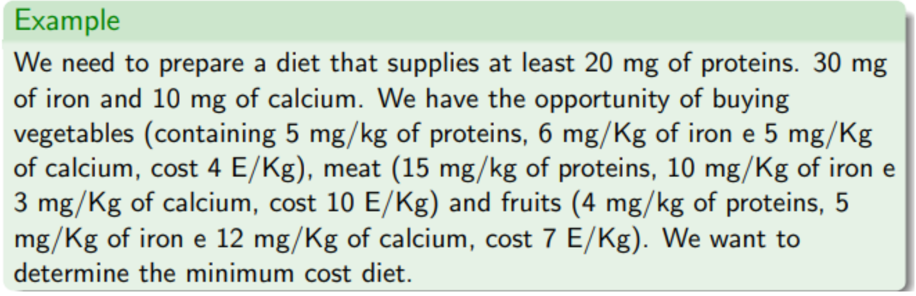
- set I: resources $I = \{rose, lily, violet\}$
- set J: products $J = \{one, two\}$
- parameters D_i: availability of resource $i \in I$ e.g. $D_{rose} = 27$
- parameters P_j: unit profit for product $j \in J$ e.g. $P_{one} = 130$
- parameters Q_{ij}: amount of resource $i \in I$ required for each unit of product $j \in J$ e.g. $Q_{rose\ one} = 1.5$, $Q_{lily\ two} = 1$
- variables x_j: amount of product $j \in J$ x_{one}, x_{two}

$$\begin{array}{lllll}
\max & \sum_{j \in J} P_j x_j & & & \\
s.t. & \sum_{j \in J} Q_{ij} x_j & \leq & D_i & \forall \ i \in I \\
 & x_j \in \mathbb{R}_+ & [\,\mathbb{Z}_+ & | \quad \{0,1\}\,] & \forall \ j \in J
\end{array}$$

A side note: if we have negative numbers in variable for constraints, this means we are creating resources, unless not specified inside of the constraints.

3.3 Minimum Cost Covering – Diet

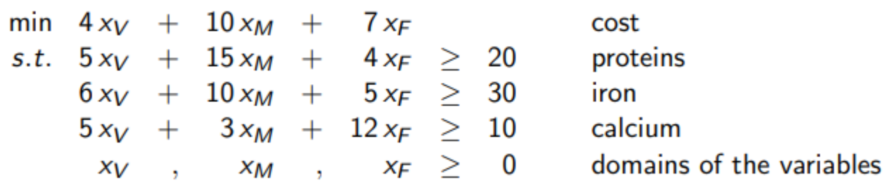
Another model called the *diet problem*:



Example

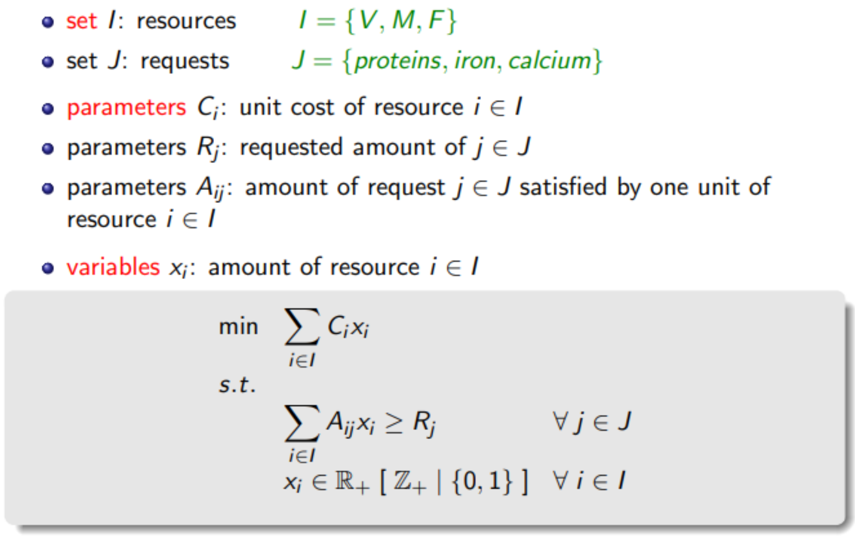
We need to prepare a diet that supplies at least 20 mg of proteins. 30 mg of iron and 10 mg of calcium. We have the opportunity of buying vegetables (containing 5 mg/kg of proteins, 6 mg/Kg of iron e 5 mg/Kg of calcium, cost 4 E/Kg), meat (15 mg/kg of proteins, 10 mg/Kg of iron e 3 mg/Kg of calcium, cost 10 E/Kg) and fruits (4 mg/kg of proteins, 5 mg/Kg of iron e 12 mg/Kg of calcium, cost 7 E/Kg). We want to determine the minimum cost diet.

We would need to minimize the cost, and the decision variable are kilos of veggies, meat and fruits.



$$\begin{array}{lrcrcrcrl}
\min & 4\,x_V & + & 10\,x_M & + & 7\,x_F & & & \text{cost} \\
s.t. & 5\,x_V & + & 15\,x_M & + & 4\,x_F & \geq & 20 & \text{proteins} \\
 & 6\,x_V & + & 10\,x_M & + & 5\,x_F & \geq & 30 & \text{iron} \\
 & 5\,x_V & + & 3\,x_M & + & 12\,x_F & \geq & 10 & \text{calcium} \\
 & x_V & , & x_M & , & x_F & \geq & 0 & \text{domains of the variables}
\end{array}$$

A more general schema for this problem is the minimum cost covering, also called infact *diet problem*.



- set I: resources $I = \{V, M, F\}$
- set J: requests $J = \{proteins, iron, calcium\}$
- parameters C_i: unit cost of resource $i \in I$
- parameters R_j: requested amount of $j \in J$
- parameters A_{ij}: amount of request $j \in J$ satisfied by one unit of resource $i \in I$
- variables x_i: amount of resource $i \in I$

$$\begin{aligned} \min \quad & \sum_{i \in I} C_i x_i \\ s.t. \quad & \\ & \sum_{i \in I} A_{ij} x_i \geq R_j \qquad && \forall\, j \in J \\ & x_i \in \mathbb{R}_+ \,[\, \mathbb{Z}_+ \mid \{0,1\} \,] && \forall\, i \in I \end{aligned}$$

It's important to create schemas inside of our mind since modeling needs to be modularized dividing the data, the model and the formulations.

Side note: $x_j \in \mathbb{R}^+[\mathbb{Z}^+ \mid \{0,1\}]\ \forall j \in J$

Means the variables xj are non-negative real numbers, with the optional additional constraint of being integers or binary.

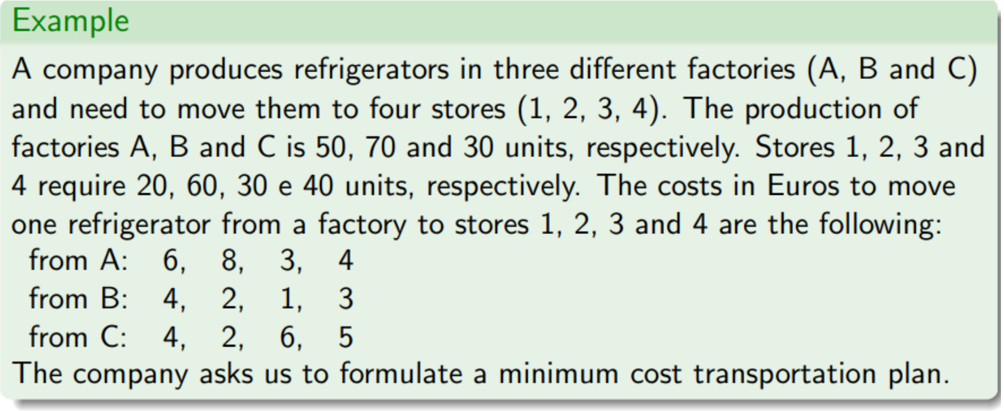
This allows the modeling schema to be flexible and cover different variants of the problem:

- Continuous variables: just $x_j \in \mathbb{R}^+$
- Integer variables: $x_j \in \mathbb{R}^+$and $\mathbb{Z}^+$
- Binary variables: $x_j \in \mathbb{R}^+$and $\{0,1\}$

The modeler can choose which constraint is appropriate when applying this schema to a specific production mix optimization problem. The square brackets succinctly show these options in the general modeling framework.

3.4 Transportation Problem

Another problem, which is called the transportation problem:



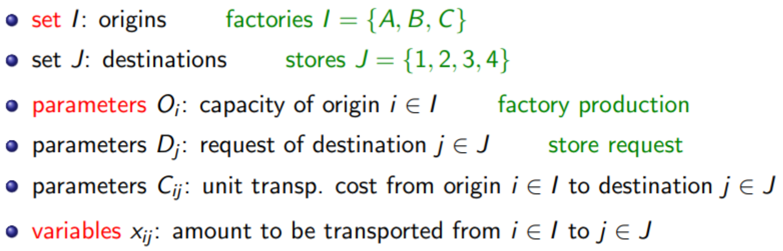
Example

A company produces refrigerators in three different factories (A, B and C) and need to move them to four stores (1, 2, 3, 4). The production of factories A, B and C is 50, 70 and 30 units, respectively. Stores 1, 2, 3 and 4 require 20, 60, 30 e 40 units, respectively. The costs in Euros to move one refrigerator from a factory to stores 1, 2, 3 and 4 are the following:

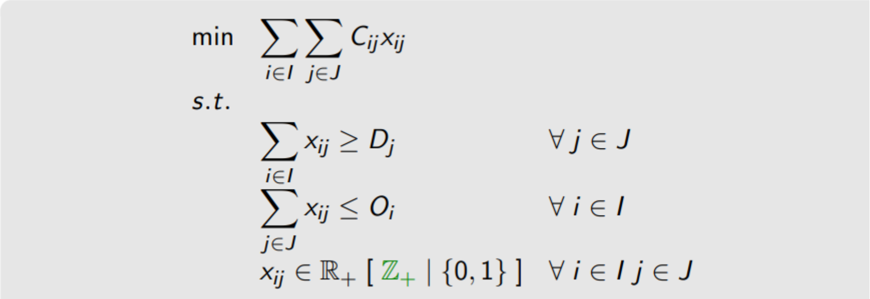
from A:	6,	8,	3,	4
from B:	4,	2,	1,	3
from C:	4,	2,	6,	5

The company asks us to formulate a minimum cost transportation plan.

A possible schema to represent this one is the transportation problem:



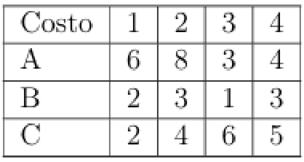
- set I: origins factories $I = \{A, B, C\}$
- set J: destinations stores $J = \{1, 2, 3, 4\}$
- parameters O_i: capacity of origin $i \in I$ factory production
- parameters D_j: request of destination $j \in J$ store request
- parameters C_{ij}: unit transp. cost from origin $i \in I$ to destination $j \in J$
- variables x_{ij}: amount to be transported from $i \in I$ to $j \in J$



$$\begin{aligned}
\min \quad & \sum_{i\in I}\sum_{j\in J} C_{ij}x_{ij} \\
s.t. \quad & \\
& \sum_{i\in I} x_{ij} \geq D_j && \forall\, j \in J \\
& \sum_{j\in J} x_{ij} \leq O_i && \forall\, i \in I \\
& x_{ij} \in \mathbb{R}_+ \; [\, \mathbb{Z}_+ \mid \{0,1\} \,] && \forall\, i \in I\; j \in J
\end{aligned}$$

To completely resolve the problem, we would have to write an example to represent cost minimization

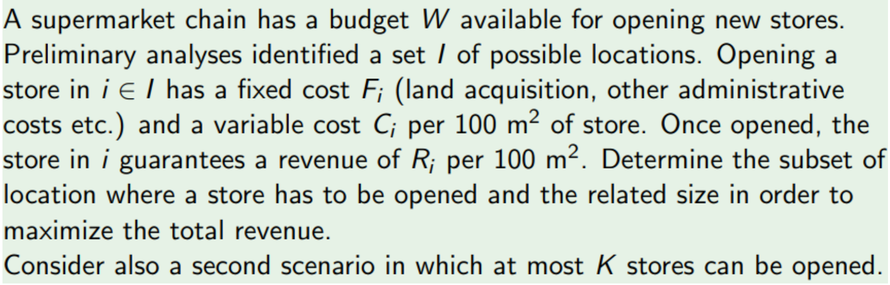
- A company produces refrigerators in three different factories (A, B, C) and has to move them to 4 warehouses (1,2,3,4). The production of the factories is 50, 70 and 20 respectively. The warehouses can contain 10, 60, 30 and 40 units
- The cost of moving refrigerators is shown in the following table:



Costo	1	2	3	4
A	6	8	3	4
B	2	3	1	3
C	2	4	6	5

3.5 Fixed Costs and Big-M Constraints

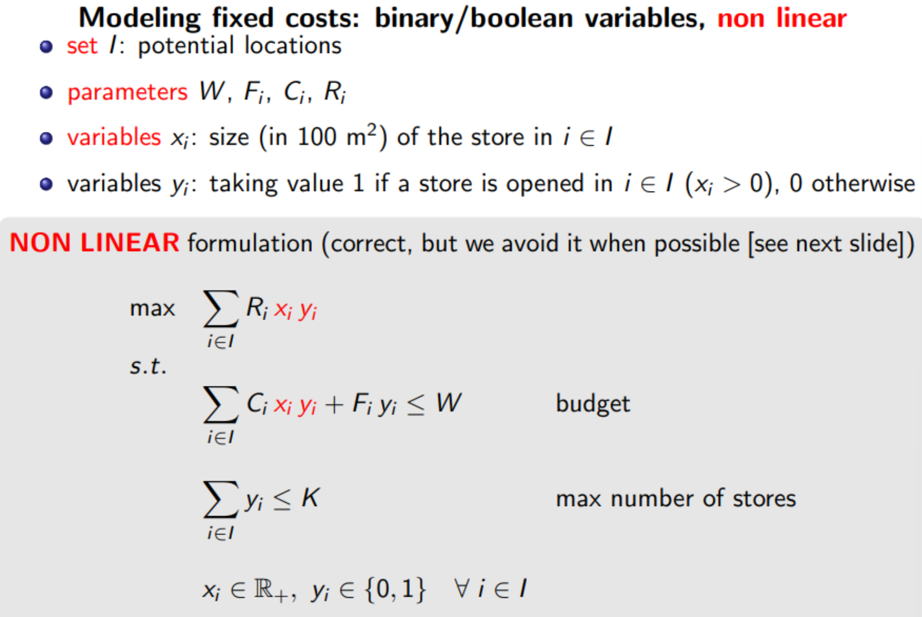
Now we introduce the notion of fixed costs:



A supermarket chain has a budget W available for opening new stores. Preliminary analyses identified a set I of possible locations. Opening a store in $i \in I$ has a fixed cost F_i (land acquisition, other administrative costs etc.) and a variable cost C_i per 100 m^2 of store. Once opened, the store in i guarantees a revenue of R_i per 100 m^2. Determine the subset of location where a store has to be opened and the related size in order to maximize the total revenue.
Consider also a second scenario in which at most K stores can be opened.

These are problems of the kind where decisions are taken about what actions to take. Each action has a fixed cost but produces some gain. The aim is to determine what actions should be taken under some constraints regarding the actions.

Modeling this problem requires a reasoning which might introduce non-linearity – we introduce for this specific reason the concept of *linear variables*:



Modeling fixed costs: binary/boolean variables, non linear

- set I: potential locations
- parameters W, F_i, C_i, R_i
- variables x_i: size (in 100 m^2) of the store in $i \in I$
- variables y_i: taking value 1 if a store is opened in $i \in I$ $(x_i > 0)$, 0 otherwise

NON LINEAR formulation (correct, but we avoid it when possible [see next slide])

$$\max \sum_{i \in I} R_i \, x_i \, y_i$$

s.t.

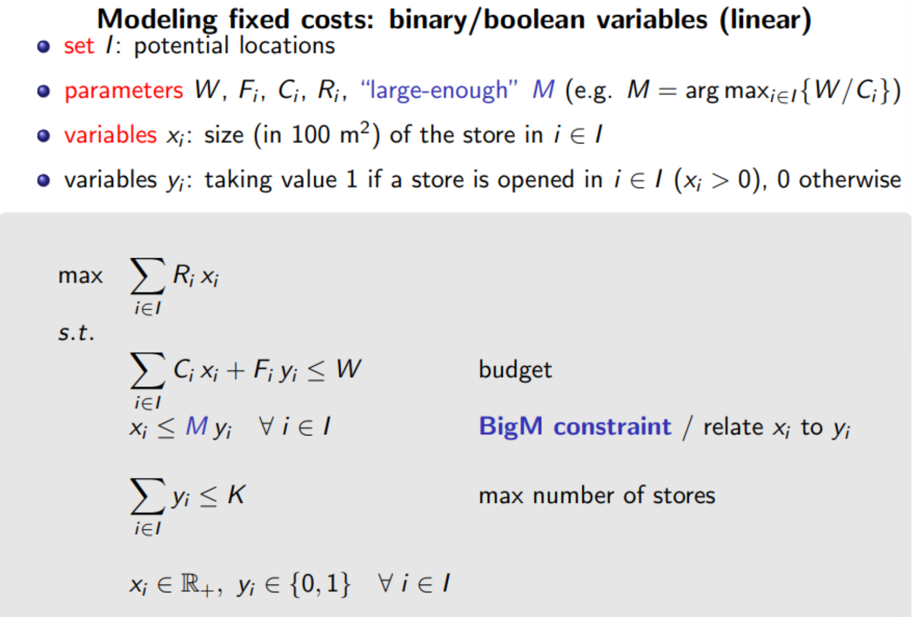
$$\sum_{i \in I} C_i \, x_i \, y_i + F_i \, y_i \le W \qquad \text{budget}$$

$$\sum_{i \in I} y_i \le K \qquad \text{max number of stores}$$

$$x_i \in \mathbb{R}_+, \; y_i \in \{0,1\} \quad \forall \, i \in I$$

We use binary variables to represent "IF" we open a store, then something happens; we want to keep constraints linear in order to have acceptable times of computation for the algorithms of solvers implemented.

Given this formulation would deviate the problem making it go into quadratic programming, we do not do this but try to linearize it. There is a link between x_i and y_i, so we try to write this expression with a constraint.



Modeling fixed costs: binary/boolean variables (linear)

- set I: potential locations
- parameters W, F_i, C_i, R_i, "large-enough" M (e.g. $M = \arg\max_{i \in I}\{W/C_i\}$)
- variables x_i: size (in 100 m^2) of the store in $i \in I$
- variables y_i: taking value 1 if a store is opened in $i \in I$ $(x_i > 0)$, 0 otherwise

$$\max \sum_{i \in I} R_i \, x_i$$

s.t.

$$\sum_{i \in I} C_i \, x_i + F_i \, y_i \le W \qquad \text{budget}$$

$$x_i \le M \, y_i \quad \forall \, i \in I \qquad \text{BigM constraint / relate } x_i \text{ to } y_i$$

$$\sum_{i \in I} y_i \le K \qquad \text{max number of stores}$$

$$x_i \in \mathbb{R}_+, \; y_i \in \{0,1\} \quad \forall \, i \in I$$

As you can see we introduce M, generally called Big-M, which is a modeling technique used in mixed-integer programming to enforce logical conditions. In this case, it's used to relate the continuous variable x_i (store size) to the binary variable y_i (whether a store is opened or not).

The Big-M value should be "large enough" to allow the maximum possible store size when a store is opened ($y_i = 1$), but also ensure that xi is forced to 0 when a store is not opened ($y_i = 0$).

This definition of M is clever because:

- It's large enough to never constrain a valid solution
- It's not unnecessarily large, which could lead to numerical issues in solvers
- It's based on the problem parameters, so it automatically adjusts if the budget or costs change

This is only then a technicality to write quadratic constraints into a linear form.

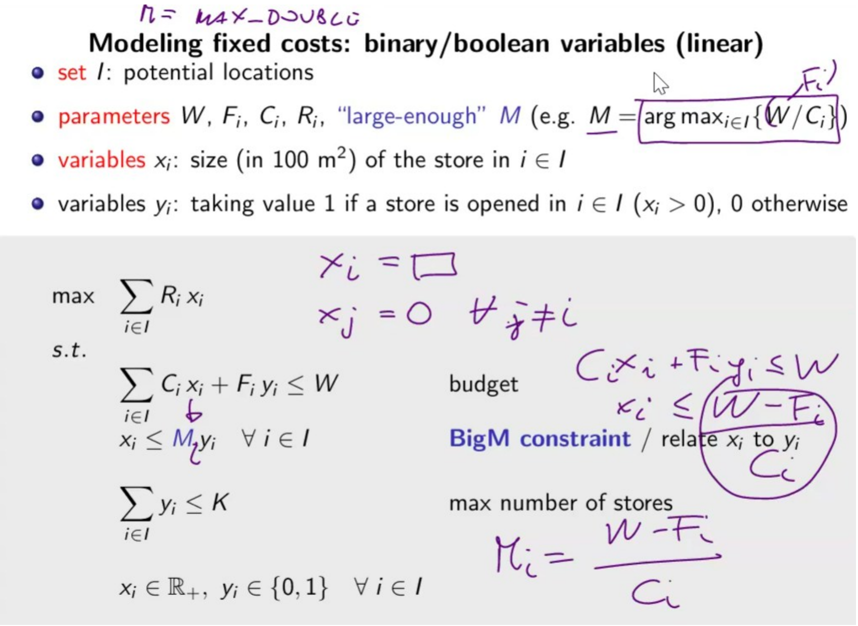
In every feasible solution of this model, x_i has no different value from:

$$x_i \leq \frac{W - F_i}{C_i}$$

The constant must be small, but not that much.

Suppose we want to open at least three stores: so, the constraint should be $\sum_{i \in I} y_i \geq 3$. This, however, is not enough since there is no enforcement relationship between the x_i and y_i. This is, once again, useful since the fixed cost notion is useful in fact for this.

Completely, just to see it under your eyes well-posed:



M = MAX_DOUBLE

Modeling fixed costs: binary/boolean variables (linear)

- set I: potential locations
- parameters W, F_i, C_i, R_i, "large-enough" M (e.g. $\underline{M} = \boxed{\arg\max_{i \in I}\{(W - F_i)/C_i\}}$)
- variables x_i: size (in 100 m^2) of the store in $i \in I$
- variables y_i: taking value 1 if a store is opened in $i \in I$ ($x_i > 0$), 0 otherwise

$$\max \sum_{i \in I} R_i x_i$$

$x_i = \square$

$x_j = 0 \quad \forall j \neq i$

s.t.

$$\sum_{i \in I} C_i x_i + F_i y_i \leq W \qquad \text{budget}$$

$C_i x_i + F_i y_i \leq W$

$x_i \leq \frac{W - F_i}{C_i}$

$$x_i \leq M_i y_i \quad \forall i \in I \qquad \text{BigM constraint / relate } x_i \text{ to } y_i$$

$$\sum_{i \in I} y_i \leq K \qquad \text{max number of stores}$$

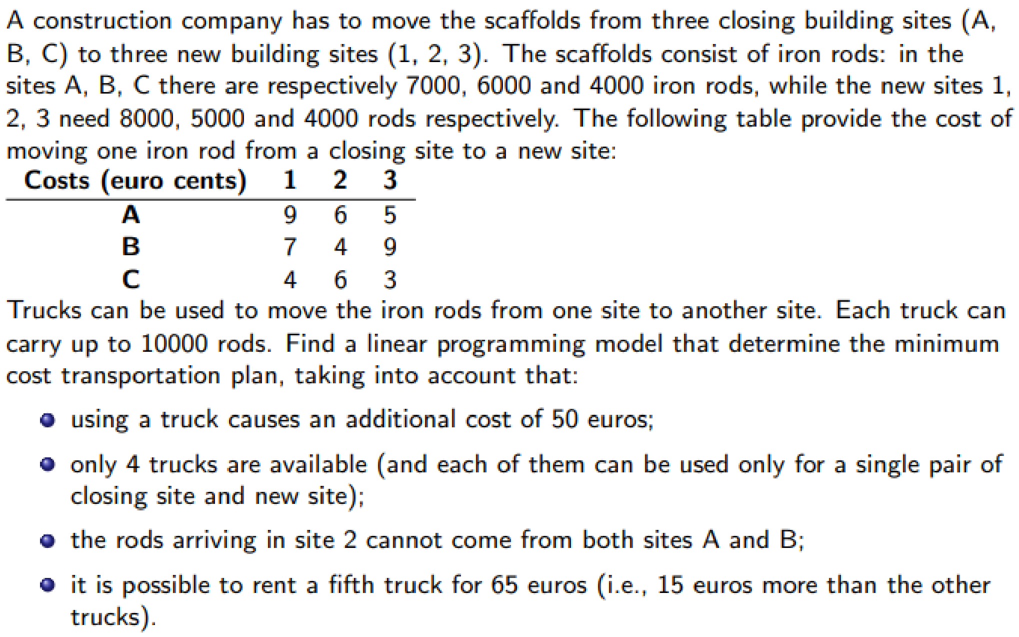
$M_i = \frac{W - F_i}{C_i}$

$$x_i \in \mathbb{R}_+, \; y_i \in \{0, 1\} \quad \forall i \in I$$

One variant of this model is that which provides for an upper limit U_i to the amount of action $i \in I$ that can be taken. In this case, the BigM constraints can be replaced with $x_i \leq U_i y_i, \forall\, i \in I$

3.6 A more complex problem – Moving scaffolds (+ 2 Variants)

We have another problem, important for lab examples – moving scaffolds:



A construction company has to move the scaffolds from three closing building sites (A, B, C) to three new building sites (1, 2, 3). The scaffolds consist of iron rods: in the sites A, B, C there are respectively 7000, 6000 and 4000 iron rods, while the new sites 1, 2, 3 need 8000, 5000 and 4000 rods respectively. The following table provide the cost of moving one iron rod from a closing site to a new site:

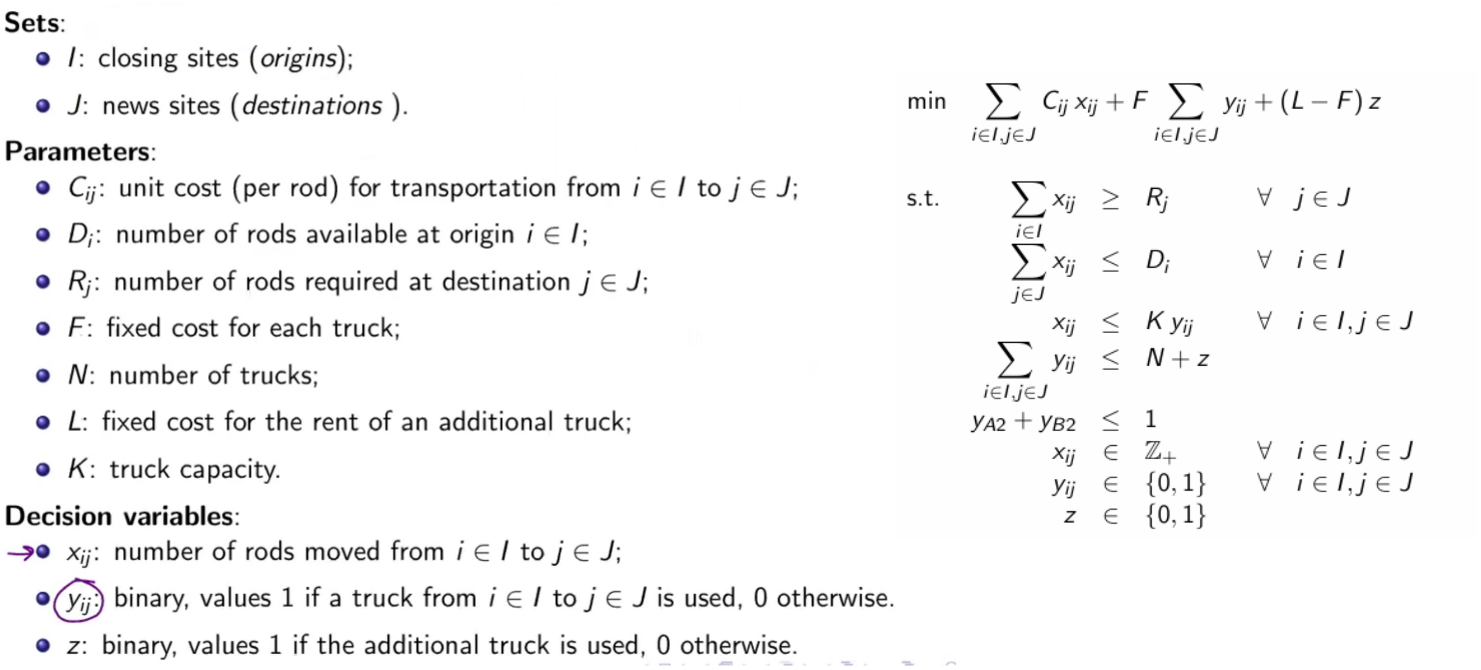
Costs (euro cents)	1	2	3
A	9	6	5
B	7	4	9
C	4	6	3

Trucks can be used to move the iron rods from one site to another site. Each truck can carry up to 10000 rods. Find a linear programming model that determine the minimum cost transportation plan, taking into account that:

- using a truck causes an additional cost of 50 euros;
- only 4 trucks are available (and each of them can be used only for a single pair of closing site and new site);
- the rods arriving in site 2 cannot come from both sites A and B;
- it is possible to rent a fifth truck for 65 euros (i.e., 15 euros more than the other trucks).

The problem is referring to the transportation schema; with these class of problems, usually it is useful to understand the similar problem(s) and then try to figure out the actual schema.

All of the elements are here:



Sets:

- I: closing sites (*origins*);
- J: news sites (*destinations*).

Parameters:

- C_{ij}: unit cost (per rod) for transportation from $i \in I$ to $j \in J$;
- D_i: number of rods available at origin $i \in I$;
- R_j: number of rods required at destination $j \in J$;
- F: fixed cost for each truck;
- N: number of trucks;
- L: fixed cost for the rent of an additional truck;
- K: truck capacity.

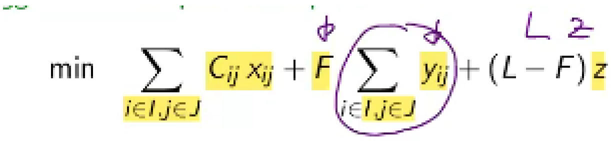
Decision variables:

- x_{ij}: number of rods moved from $i \in I$ to $j \in J$;
- y_{ij}: binary, values 1 if a truck from $i \in I$ to $j \in J$ is used, 0 otherwise.
- z: binary, values 1 if the additional truck is used, 0 otherwise.

$$
\begin{array}{lrcll}
\min & \sum_{i\in I, j\in J} C_{ij}\, x_{ij} + F \sum_{i\in I, j\in J} y_{ij} + (L-F)\, z & & & \\
\text{s.t.} & \sum_{i\in I} x_{ij} & \geq & R_j & \forall \ j \in J \\
& \sum_{j\in J} x_{ij} & \leq & D_i & \forall \ i \in I \\
& x_{ij} & \leq & K\, y_{ij} & \forall \ i \in I, j \in J \\
& \sum_{i\in I, j\in J} y_{ij} & \leq & N + z & \\
& y_{A2} + y_{B2} & \leq & 1 & \\
& x_{ij} & \in & \mathbb{Z}_+ & \forall \ i \in I, j \in J \\
& y_{ij} & \in & \{0,1\} & \forall \ i \in I, j \in J \\
& z & \in & \{0,1\} &
\end{array}
$$

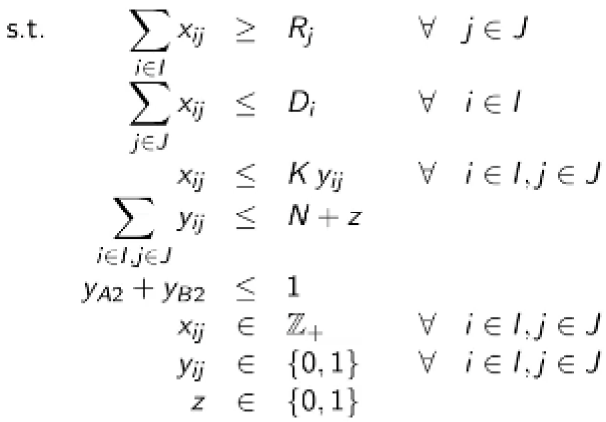
Some comments, step by step:

- We want to take the origins and destinations, considering the costs of transportation but also the material costs, which are fixed. The binary variables refer to the choice of using or not a specific truck. There is also an additional cost for a specific truck, depending on the previous choice



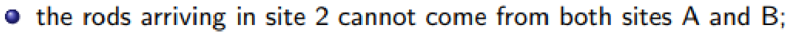
$$\min \sum_{i\in I, j\in J} C_{ij}\, x_{ij} + \overset{\phi}{F} \sum_{i\in I, j\in J} \overset{d}{y_{ij}} + (\overset{L}{L} - F)\, \overset{2}{z}$$

- Origins and destinations are to be respected and also linked, considering the truck capacity and the fact specific trucks need to be available



$$\begin{array}{llcll}
\text{s.t.} & \sum_{i\in I} x_{ij} & \geq & R_j & \forall\ j \in J \\
& \sum_{j\in J} x_{ij} & \leq & D_i & \forall\ i \in I \\
& x_{ij} & \leq & K\, y_{ij} & \forall\ i \in I, j \in J \\
& \sum_{i\in I, j\in J} y_{ij} & \leq & N + z & \\
& y_{A2} + y_{B2} & \leq & 1 & \\
& x_{ij} & \in & \mathbb{Z}_+ & \forall\ i \in I, j \in J \\
& y_{ij} & \in & \{0,1\} & \forall\ i \in I, j \in J \\
& z & \in & \{0,1\} &
\end{array}$$

We did not consider this constraint, which is in some way logical:

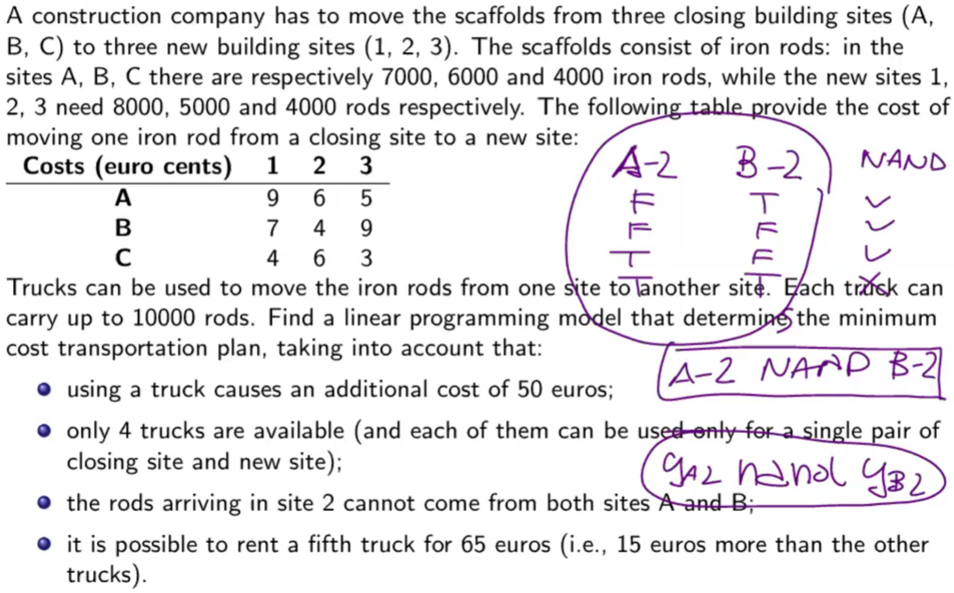


- the rods arriving in site 2 cannot come from both sites A and B;

Actually, since this is all numbers up to now (linear), we should not represent such relationship as the following (logical constraint):

$$A_2 \quad NAND \quad B_2$$

The situation would be represented by the following:



A construction company has to move the scaffolds from three closing building sites (A, B, C) to three new building sites (1, 2, 3). The scaffolds consist of iron rods: in the sites A, B, C there are respectively 7000, 6000 and 4000 iron rods, while the new sites 1, 2, 3 need 8000, 5000 and 4000 rods respectively. The following table provide the cost of moving one iron rod from a closing site to a new site:

Costs (euro cents)	1	2	3
A	9	6	5
B	7	4	9
C	4	6	3

A-2	B-2	NAND
F	T	✓
F	F	✓
T	F	✓
T	T	✗

Trucks can be used to move the iron rods from one site to another site. Each truck can carry up to 10000 rods. Find a linear programming model that determines the minimum cost transportation plan, taking into account that:

A-2 NAND B-2

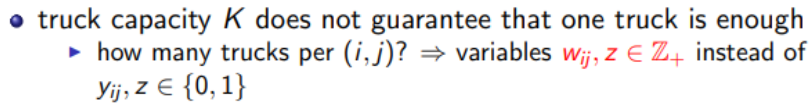
- using a truck causes an additional cost of 50 euros;
- only 4 trucks are available (and each of them can be used only for a single pair of closing site and new site);
- the rods arriving in site 2 cannot come from both sites A and B;
- it is possible to rent a fifth truck for 65 euros (i.e., 15 euros more than the other trucks).

y_{A2} nand y_{B2}

Of course, we should be able to represent such construction (since this is a Boolean operator), similar to a truth table as a combination of feasibility and infeasibility; the right way would be:

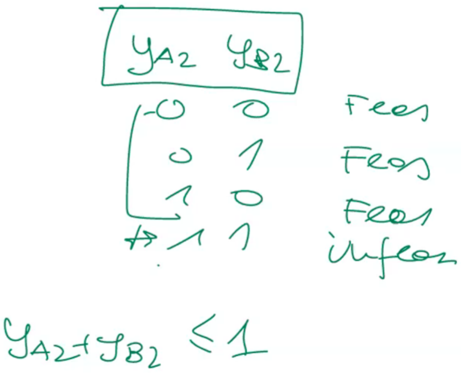
$$y_{A2} + y_{B2} \leq 1$$

However, we can construct a variant of the previous considering:



- truck capacity K does not guarantee that one truck is enough
  - how many trucks per (i,j)? $\Rightarrow$ variables $w_{ij}, z \in \mathbb{Z}_+$ instead of $y_{ij}, z \in \{0,1\}$

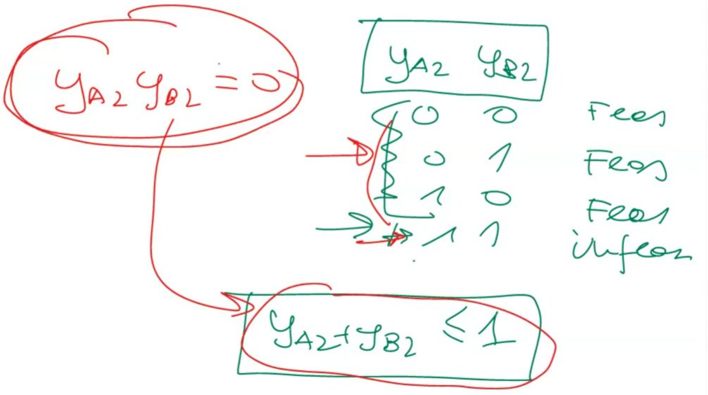
With binary variables, we would represent the situation as the following:



y_{A2}	y_{B2}	
0	0	Feas
0	1	Feas
1	0	Feas
1	1	infeas

$y_{A2}+y_{B2} \leq 1$

A constraint like the following would make the model non-linear:



$y_{A2}\, y_{B2} = 0$

y_{A2}	y_{B2}	
0	0	Feas
0	1	Feas
1	0	Feas
1	1	infeas

$y_{A2}+y_{B2} \leq 1$

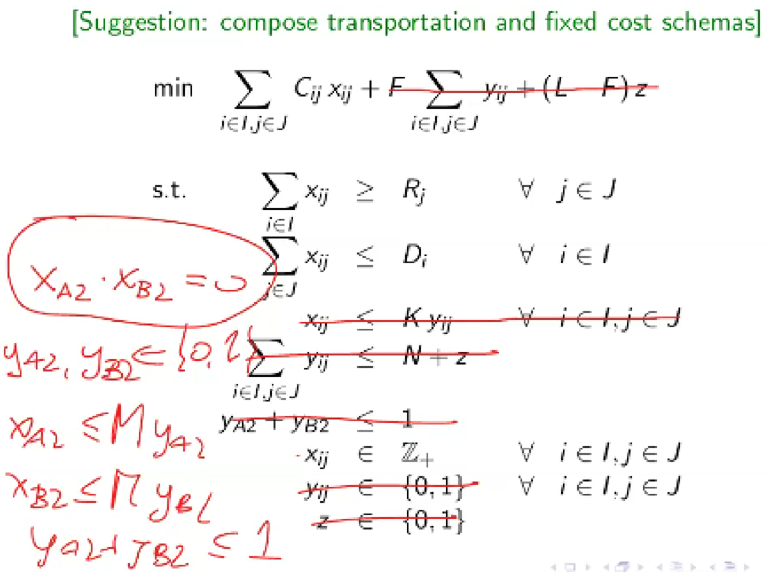
Actually, to write this, we would write something like the following, but the ceiling function is not linear:

$$\min \sum c_{ij} x_{ij} + F \sum \left\lceil \frac{x_{ij}}{k} \right\rceil$$

We link the two variables with a linear inequality, also adding an integer variable w_{ij}, writing for example a relationship between the already-existing decision variables:

$$w_{ij} \geq \frac{x_{ij}}{k}$$

This way, it becomes a linear expression. The following is the complete reasoning that brings us to these conclusions (better up to now from an algorithmic point of view):



[Suggestion: compose transportation and fixed cost schemas]

$$\min \sum_{i\in I, j\in J} C_{ij}\, x_{ij} + \cancel{F \sum_{i\in I, j\in J} y_{ij} + (L - F) z}$$

$$\begin{array}{llll}
\text{s.t.} & \sum_{i\in I} x_{ij} & \geq R_j & \forall\ j \in J \\
& \sum_{j\in J} x_{ij} & \leq D_i & \forall\ i \in I \\
& \cancel{x_{ij}} & \cancel{\leq K y_{ij}} & \cancel{\forall\ i \in I, j \in J} \\
& \cancel{\sum_{i\in I, j\in J} y_{ij}} & \cancel{\leq N + z} & \\
& \cancel{y_{A2} + y_{B2}} & \cancel{\leq 1} & \\
& x_{ij} & \in \mathbb{Z}_+ & \forall\ i \in I, j \in J \\
& \cancel{y_{ij}} & \cancel{\in \{0,1\}} & \forall\ i \in I, j \in J \\
& \cancel{z} & \cancel{\in \{0,1\}} &
\end{array}$$

$x_{A2} \cdot x_{B2} = 0$

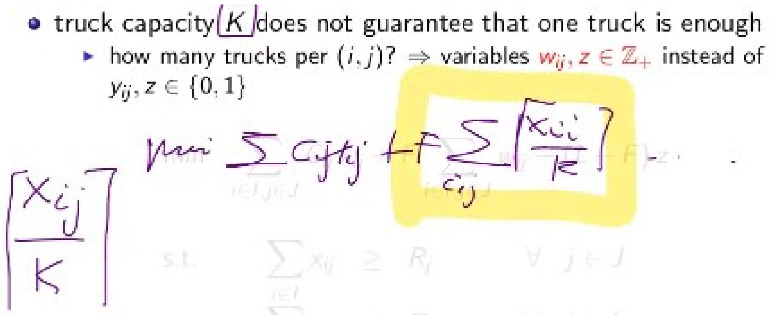
$y_{A2}, y_{B2} \in \{0,1\}$

$x_{A2} \leq M y_{A2}$

$x_{B2} \leq M y_{B2}$

$y_{A2} + y_{B2} \leq 1$

In complete form, it appears like this (moving scaffolds – variant 1 – limited truck capacity). We cannot write the following into the model, it's not linear to use the ceiling function:



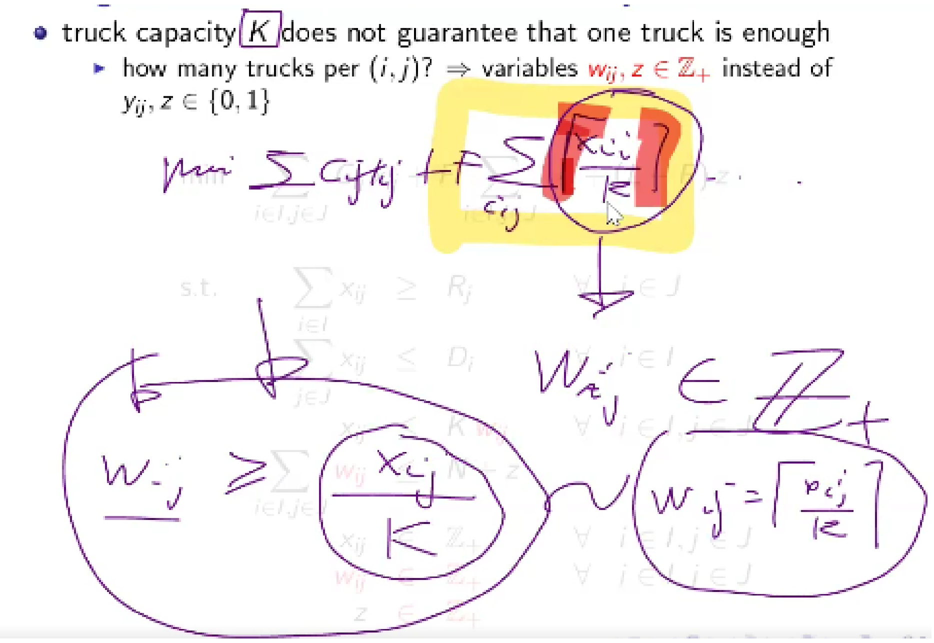
- truck capacity $\boxed{K}$ does not guarantee that one truck is enough
  - how many trucks per (i, j)? ⇒ variables $w_{ij}, z \in \mathbb{Z}_+$ instead of $y_{ij}, z \in \{0, 1\}$

$$\min \sum C_{ij} x_{ij} + F \sum_{c_{ij}} \left\lceil \frac{x_{ij}}{K} \right\rceil$$

$$\left\lceil \frac{x_{ij}}{K} \right\rceil$$

s.t. $\sum_{i\in I} x_{ij} \geq R_j \quad \forall\ j \in J$

Ceiling is an unknown; the number of needed tracks using linear constraints combines with a relation (division) numbers to use the ceiling and make it linear:



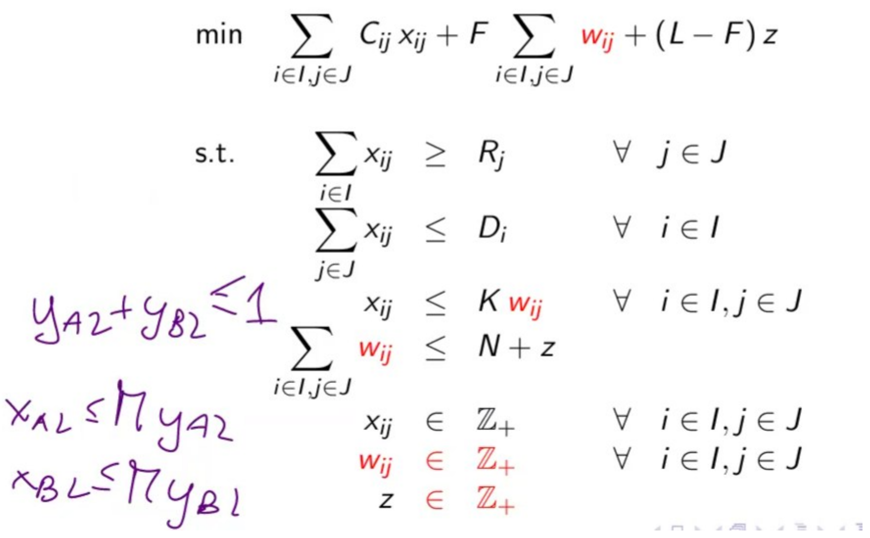
- truck capacity $\boxed{K}$ does not guarantee that one truck is enough
  - how many trucks per (i, j)? ⇒ variables $w_{ij}, z \in \mathbb{Z}_+$ instead of $y_{ij}, z \in \{0, 1\}$

$$\min \sum C_{ij} x_{ij} + F \sum_{c_{ij}} \left\lceil \frac{x_{ij}}{K} \right\rceil$$

$$w_{ij} \in \mathbb{Z}_+$$

$$w_{ij} \geq \frac{x_{ij}}{K} \sim w_{ij} = \left\lceil \frac{x_{ij}}{K} \right\rceil$$

Now, completely:



$$
\begin{aligned}
\min \quad & \sum_{i\in I, j\in J} C_{ij}\, x_{ij} + F \sum_{i\in I, j\in J} w_{ij} + (L-F)\, z \\
\text{s.t.} \quad & \sum_{i\in I} x_{ij} \ge R_j && \forall\ j \in J \\
& \sum_{j\in J} x_{ij} \le D_i && \forall\ i \in I \\
& x_{ij} \le K\, w_{ij} && \forall\ i \in I, j \in J \\
& \sum_{i\in I, j\in J} w_{ij} \le N + z \\
& x_{ij} \in \mathbb{Z}_+ && \forall\ i \in I, j \in J \\
& w_{ij} \in \mathbb{Z}_+ && \forall\ i \in I, j \in J \\
& z \in \mathbb{Z}_+
\end{aligned}
$$

$y_{A2} + y_{B2} \le 1$

$x_{AL} \le M y_{A2}$

$x_{BL} \le M y_{Bl}$

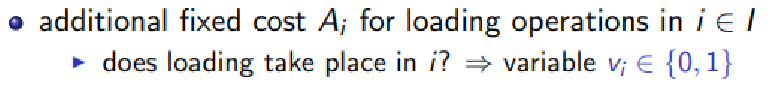
In conclusion:

A possible variant of this problem may be the addition of constraints on the maximum capacity K of trucks. Previously it was assumed that K was high enough to ensure that a truck could move everything needed. To manage this situation you need to change the variables $y_{i,j} \in \{0,1\}$ into whole variables, which represent how many trucks are needed in a given route.

As a result, some constraints also change:

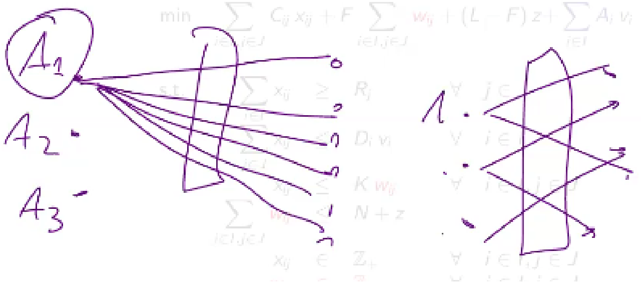
$$
\begin{aligned}
\min & \sum_{i\in I, j\in J} C_{i,j} x_{i,j} + F \sum_{i\in I, j\in J} w_{i,j} + (L-F) z \\
& \vdots \\
& x_{i,j} \le K w_{i,j} \quad \forall i \in I, j \in J
\end{aligned}
$$

Instead consider this other option, based on the following constraint (moving scaffolds – variant 2 – fixed costs to load the trucks):

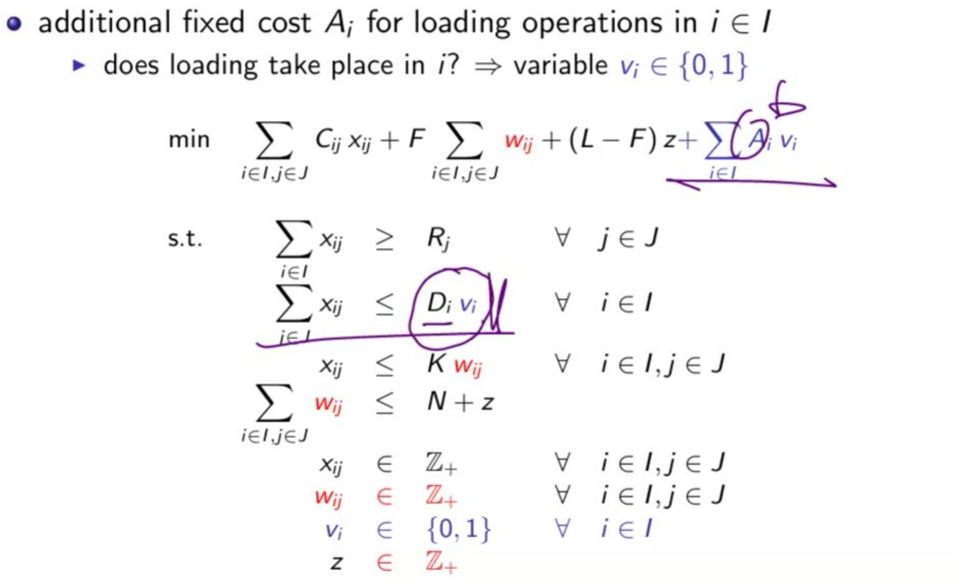


- additional fixed cost A_i for loading operations in $i \in I$
  - does loading take place in i? $\Rightarrow$ variable $v_i \in \{0,1\}$

It's better to move stuff in a mixed way, where each operation has fixed costs, which represents the decision "should I move something from I?":


A_1
A_2
A_3
1
min $\sum C_{ij} x_{ij} + F \sum w_{ij} + (L-F) z + \sum A_i v_i$

Constraints are once again linear, so everything works:

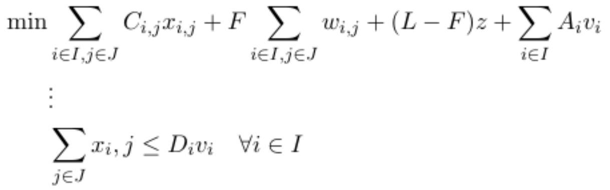


- additional fixed cost A_i for loading operations in $i \in I$
  - does loading take place in i? ⇒ variable $v_i \in \{0,1\}$

$$
\begin{array}{llll}
\min & \sum_{i\in I, j\in J} C_{ij}\, x_{ij} + F \sum_{i\in I, j\in J} w_{ij} + (L-F)\, z + \sum_{i\in I} A_i\, v_i & & \\
\text{s.t.} & \sum_{i\in I} x_{ij} \ \geq \ R_j & \forall & j \in J \\
& \sum_{j\in J} x_{ij} \ \leq \ D_i\, v_i & \forall & i \in I \\
& x_{ij} \ \leq \ K\, w_{ij} & \forall & i \in I, j \in J \\
& \sum_{i\in I, j\in J} w_{ij} \ \leq \ N + z & & \\
& x_{ij} \ \in \ \mathbb{Z}_+ & \forall & i \in I, j \in J \\
& w_{ij} \ \in \ \mathbb{Z}_+ & \forall & i \in I, j \in J \\
& v_i \ \in \ \{0,1\} & \forall & i \in I \\
& z \ \in \ \mathbb{Z}_+ & &
\end{array}
$$

So, in conclusion:

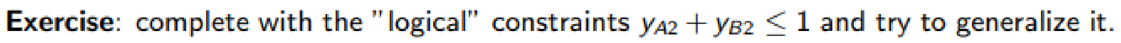
There is a fixed cost for loading the beams into I. To model this, a binary variable is used which is 1 if goods are loaded into site i. Model changes concern target function and trigger constraints for new variables $v_i \in \{0,1\}$:



$$
\begin{aligned}
\min \sum_{i\in I, j\in J} C_{i,j} x_{i,j} + F \sum_{i\in I, j\in J} w_{i,j} + (L-F)z + \sum_{i\in I} A_i v_i \\
\vdots \\
\sum_{j\in J} x_i, j \leq D_i v_i \quad \forall i \in I
\end{aligned}
$$

Sometimes modelling does not mean reinventing the wheel, but instead adopting a literature-based one and then constructing a problem upon it. Perhaps you may find it somewhere but adapt it to your problem.

Let's try to complete the following:



Exercise: complete with the "logical" constraints $y_{A2} + y_{B2} \leq 1$ and try to generalize it.

To complete the model with this constraint and generalize it, we can add:

$$y_{ij} + y_{kj} \leq 1 \quad \forall\, i, k \in I, i \neq k, \forall\, j \in J$$

This constraint ensures that for any destination j, at most one origin site can send rods there using a truck. In other words, the rods arriving at site j cannot come from both sites i and k.

The specific constraint $y_{A2} + y_{B2} \leq 1$) is just one instance of this more general constraint, ensuring rods arriving at site 2 cannot come from both sites A and B.

Adding this generalized constraint to the model gives:

$min\ \sum_{i\in I,j\in J} C_{ij} * x_{ij} +\ F * \sum_{i\in I,j\in J} y_{ij} + (L - F)z$

$s.t. \sum_{i\in I} x_{ij} \geq R_j, \forall j \in J$

$\sum_{j\in J} x_{Ij} \leq D_i \quad \forall\ i \in I$

$x_{ij} \leq K * y_{ij} \qquad \forall\ i \in I, j \in J$

$\sum_{i\in I,j\in J} y_{ij} \leq \mathbb{N} + z$

$y_{ij} + ykj \leq 1 \quad \forall\ i,k \in I, i \neq k, \forall\ j \in J$

$x_{ij} \in \mathbb{Z}^+ \qquad \forall\ i \in I, j \in J$

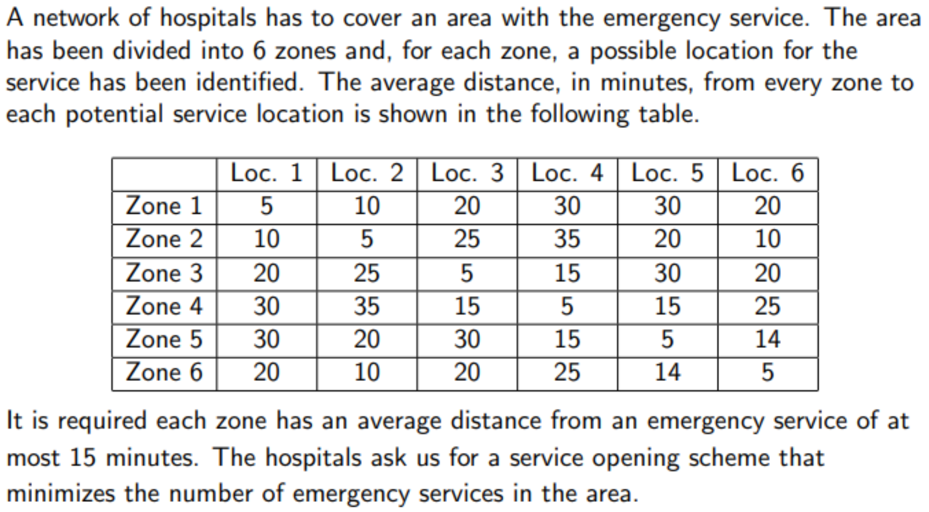
$y_{ij} \in \{0,1\} \quad \forall\ i \in I, j \in J$

$z \in \{0,1\}$

This generalized constraint captures the requirement that rods arriving at any destination cannot come from multiple origins using trucks, which is a logical extension of the specific constraint given for site 2.

3.7 Emergency Location – Minimum Cost Covering

Let's go on with the following exercise – emergency location schema:

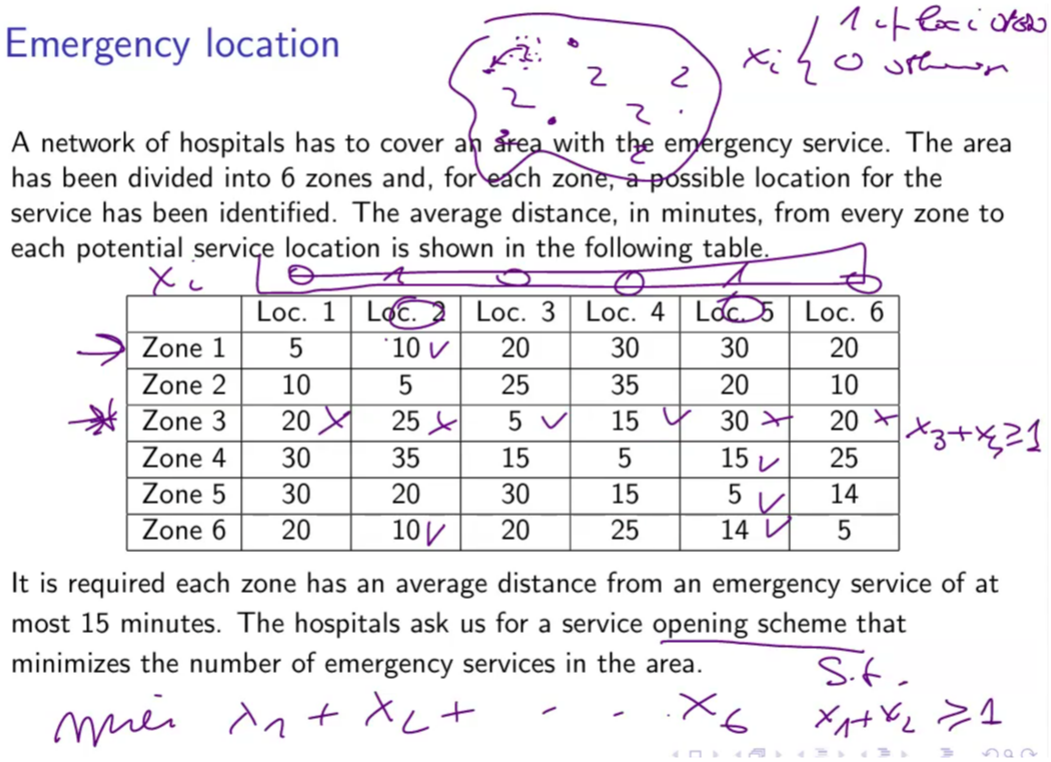


A network of hospitals has to cover an area with the emergency service. The area has been divided into 6 zones and, for each zone, a possible location for the service has been identified. The average distance, in minutes, from every zone to each potential service location is shown in the following table.

	Loc. 1	Loc. 2	Loc. 3	Loc. 4	Loc. 5	Loc. 6
Zone 1	5	10	20	30	30	20
Zone 2	10	5	25	35	20	10
Zone 3	20	25	5	15	30	20
Zone 4	30	35	15	5	15	25
Zone 5	30	20	30	15	5	14
Zone 6	20	10	20	25	14	5

It is required each zone has an average distance from an emergency service of at most 15 minutes. The hospitals ask us for a service opening scheme that minimizes the number of emergency services in the area.

The problem is not so clear; is there a way to find a feasible solution for this problem? In this way, we understand the decisions to make. Decisions are related to locations, so to open or not – using ofc binary variables and relating them all between each other:



Emergency location

$x_i \begin{cases} 1 \text{ if loc } i \text{ open} \\ 0 \text{ otherwise} \end{cases}$

A network of hospitals has to cover an area with the emergency service. The area has been divided into 6 zones and, for each zone, a possible location for the service has been identified. The average distance, in minutes, from every zone to each potential service location is shown in the following table.

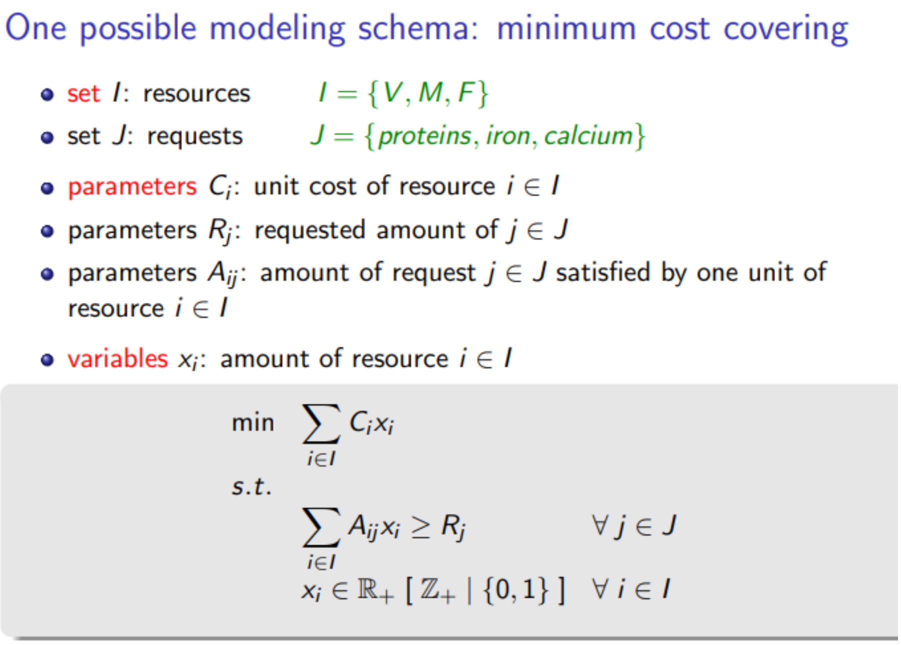
x_i	0	1	0	0	1	0
	Loc. 1	Loc. 2	Loc. 3	Loc. 4	Loc. 5	Loc. 6
Zone 1	5	10 ✓	20	30	30	20
Zone 2	10	5	25	35	20	10
Zone 3	20 ✗	25 ✗	5 ✓	15 ✓	30 ✗	20 ✗
Zone 4	30	35	15	5	15 ✓	25
Zone 5	30	20	30	15	5 ✓	14
Zone 6	20	10 ✓	20	25	14 ✓	5

$x_3 + x_4 \geq 1$

It is required each zone has an average distance from an emergency service of at most 15 minutes. The hospitals ask us for a service opening scheme that minimizes the number of emergency services in the area.

$\min \; x_1 + x_2 + \dots x_6 \qquad \text{s.t.} \quad x_1 + x_2 \geq 1$

The actual schema where this comes from is the following, since we need to arrive to the hospital within 15 minutes:

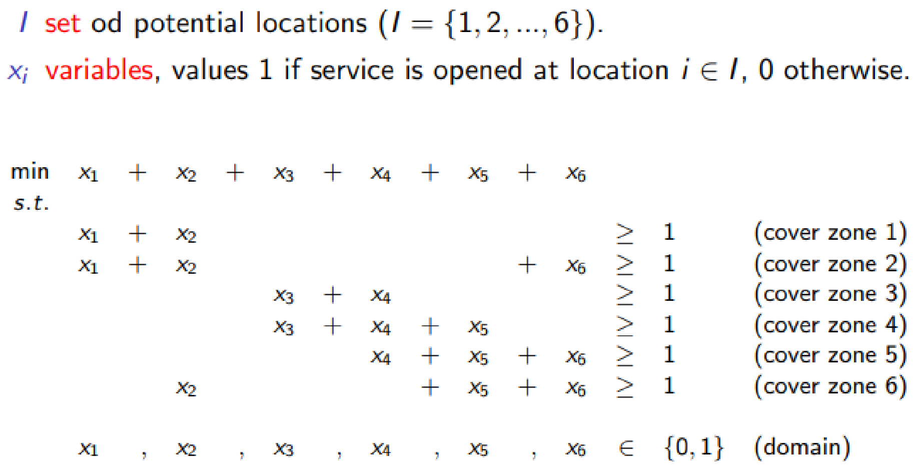


One possible modeling schema: minimum cost covering

- set I: resources $\quad I = \{V, M, F\}$
- set J: requests $\quad J = \{proteins, iron, calcium\}$
- parameters C_i: unit cost of resource $i \in I$
- parameters R_j: requested amount of $j \in J$
- parameters A_{ij}: amount of request $j \in J$ satisfied by one unit of resource $i \in I$
- variables x_i: amount of resource $i \in I$

$$\begin{aligned} \min \quad & \sum_{i \in I} C_i x_i \\ s.t. \quad & \\ & \sum_{i \in I} A_{ij} x_i \geq R_j \qquad \forall\, j \in J \\ & x_i \in \mathbb{R}_+ \; [\, \mathbb{Z}_+ \mid \{0,1\} \,] \quad \forall\, i \in I \end{aligned}$$

We see the problem model changes a bit – this is a covering schema:

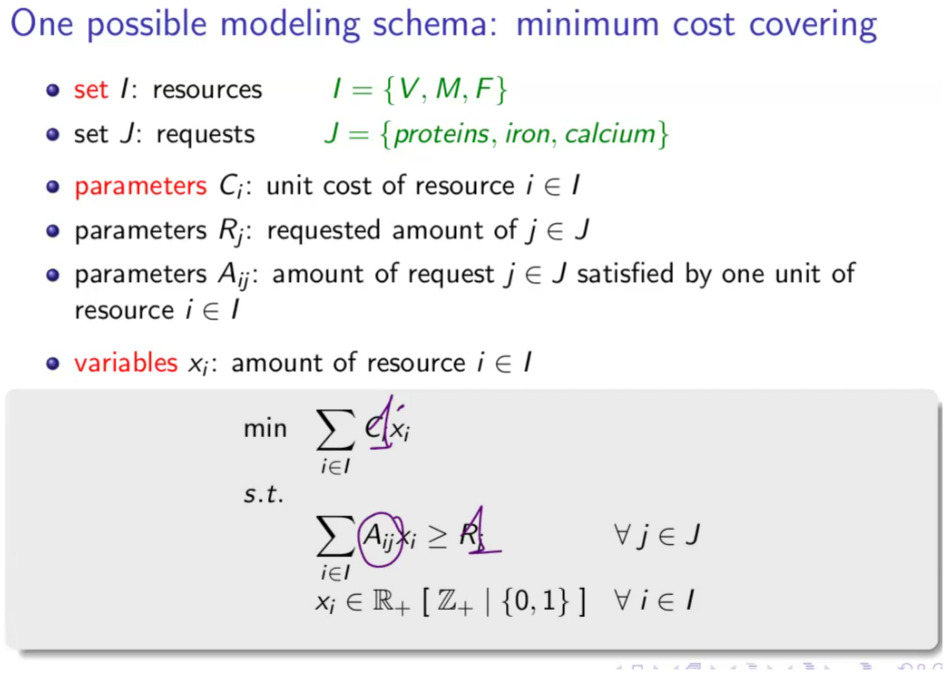


I set od potential locations ($I = \{1, 2, ..., 6\}$).

x_i variables, values 1 if service is opened at location $i \in I$, 0 otherwise.

$$\begin{array}{lcccccccccccccl}
\min & x_1 & + & x_2 & + & x_3 & + & x_4 & + & x_5 & + & x_6 & & & \\
s.t. & & & & & & & & & & & & & & \\
 & x_1 & + & x_2 & & & & & & & & & \geq & 1 & \text{(cover zone 1)} \\
 & x_1 & + & x_2 & & & & & & & + & x_6 & \geq & 1 & \text{(cover zone 2)} \\
 & & & & & x_3 & + & x_4 & & & & & \geq & 1 & \text{(cover zone 3)} \\
 & & & & & x_3 & + & x_4 & + & x_5 & & & \geq & 1 & \text{(cover zone 4)} \\
 & & & & & & & x_4 & + & x_5 & + & x_6 & \geq & 1 & \text{(cover zone 5)} \\
 & & & x_2 & & & & & + & x_5 & + & x_6 & \geq & 1 & \text{(cover zone 6)} \\
 & x_1 & , & x_2 & , & x_3 & , & x_4 & , & x_5 & , & x_6 & \in & \{0,1\} & \text{(domain)}
\end{array}$$

Is the problem similar to something we already saw? Yes, since requests are related to zones, and we want to cover locations.

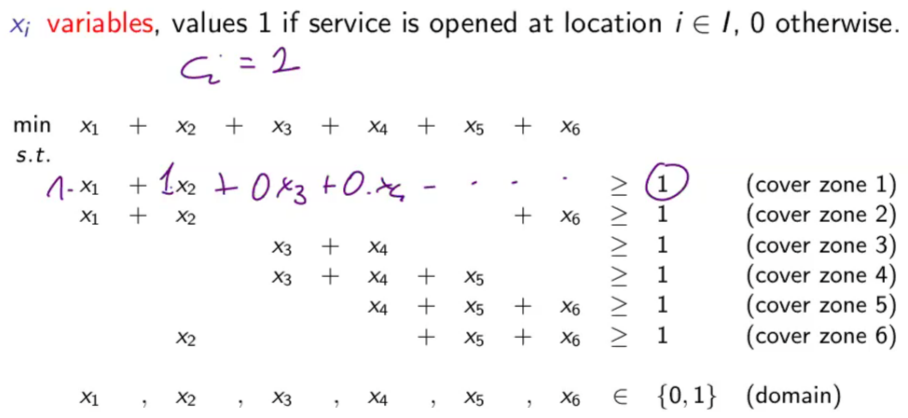


One possible modeling schema: minimum cost covering

- set I: resources $I = \{V, M, F\}$
- set J: requests $J = \{proteins, iron, calcium\}$
- parameters C_i: unit cost of resource $i \in I$
- parameters R_j: requested amount of $j \in J$
- parameters A_{ij}: amount of request $j \in J$ satisfied by one unit of resource $i \in I$
- variables x_i: amount of resource $i \in I$

$$\begin{array}{lll}
\min & \sum_{i\in I} \overset{1}{C_i} x_i & \\
s.t. & & \\
 & \sum_{i\in I} \boxed{A_{ij}} x_i \geq \overset{1}{R_j} & \forall j \in J \\
 & x_i \in \mathbb{R}_+ \; [\, \mathbb{Z}_+ \mid \{0,1\} \,] & \forall i \in I
\end{array}$$

This depends on whether we open a specific location or not:



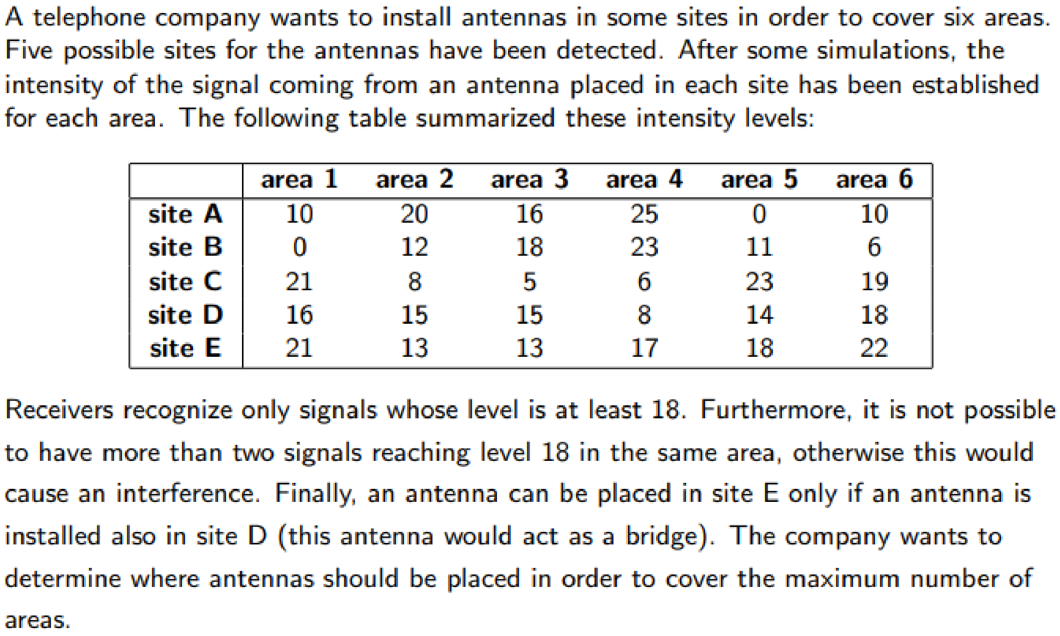
x_i variables, values 1 if service is opened at location $i \in I$, 0 otherwise.

$C_i = 1$

$$\begin{array}{lcccccccccccccl}
\min & x_1 & + & x_2 & + & x_3 & + & x_4 & + & x_5 & + & x_6 & & & \\
s.t. & & & & & & & & & & & & & & \\
 & 1\cdot x_1 & + & 1x_2 & + & 0x_3 & + & 0\cdot x_4 & - & \cdot & \cdot & \cdot & \geq & \boxed{1} & \text{(cover zone 1)} \\
 & x_1 & + & x_2 & & & & & & & + & x_6 & \geq & 1 & \text{(cover zone 2)} \\
 & & & & & x_3 & + & x_4 & & & & & \geq & 1 & \text{(cover zone 3)} \\
 & & & & & x_3 & + & x_4 & + & x_5 & & & \geq & 1 & \text{(cover zone 4)} \\
 & & & & & & & x_4 & + & x_5 & + & x_6 & \geq & 1 & \text{(cover zone 5)} \\
 & & & x_2 & & & & & + & x_5 & + & x_6 & \geq & 1 & \text{(cover zone 6)} \\
 & x_1 & , & x_2 & , & x_3 & , & x_4 & , & x_5 & , & x_6 & \in & \{0,1\} & \text{(domain)}
\end{array}$$

3.8 TLC Antennas – Minimum Cost Covering

A different problem is the following – TLC antennas location:

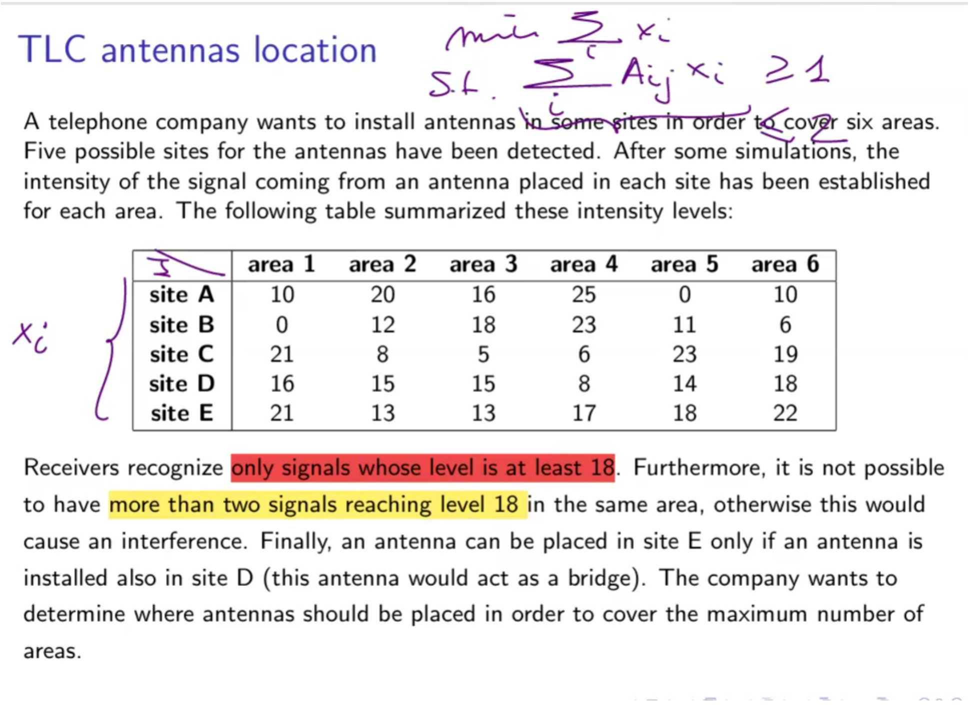


A telephone company wants to install antennas in some sites in order to cover six areas. Five possible sites for the antennas have been detected. After some simulations, the intensity of the signal coming from an antenna placed in each site has been established for each area. The following table summarized these intensity levels:

	area 1	area 2	area 3	area 4	area 5	area 6
site A	10	20	16	25	0	10
site B	0	12	18	23	11	6
site C	21	8	5	6	23	19
site D	16	15	15	8	14	18
site E	21	13	13	17	18	22

Receivers recognize only signals whose level is at least 18. Furthermore, it is not possible to have more than two signals reaching level 18 in the same area, otherwise this would cause an interference. Finally, an antenna can be placed in site E only if an antenna is installed also in site D (this antenna would act as a bridge). The company wants to determine where antennas should be placed in order to cover the maximum number of areas.

In this case, we want to cover all of the areas in which signal levels should be acceptable enough in order to cover everything. We see here how the constraints are posed:



TLC antennas location

$$\min \sum_i x_i$$

$$\text{s.t.} \sum_i A_{ij} x_i \geq 1$$

$$\leq 2$$

A telephone company wants to install antennas in some sites in order to cover six areas. Five possible sites for the antennas have been detected. After some simulations, the intensity of the signal coming from an antenna placed in each site has been established for each area. The following table summarized these intensity levels:

I	area 1	area 2	area 3	area 4	area 5	area 6
site A	10	20	16	25	0	10
site B	0	12	18	23	11	6
site C	21	8	5	6	23	19
site D	16	15	15	8	14	18
site E	21	13	13	17	18	22

x_i

Receivers recognize only signals whose level is at least 18. Furthermore, it is not possible to have more than two signals reaching level 18 in the same area, otherwise this would cause an interference. Finally, an antenna can be placed in site E only if an antenna is installed also in site D (this antenna would act as a bridge). The company wants to determine where antennas should be placed in order to cover the maximum number of areas.

The problem is similar to the previous one, although there are slightly different constraints.

There is a set I with the possible sites and a set J with the possible areas. The signal strength in area $j \in J$ from the antenna at site $i \in I$ is modelled by parameters $\sigma_{i,j}$. There is then parameter T which represents the minimum signal strength (18) and N which is the maximum number of signals that can overlap in an area (1).

- The first difference with the previous problem is the objective function, before you wanted to minimize the cost and now you want to maximize coverage
- The second difference concerns the constraints, which in this case are related to the signal overlap and the minimum threshold

The choice, and therefore the variables, is where to place an antenna, so binary xi variables are used which are worth 1 if an antenna is placed in site i. However, using only xi does not express well the objective function, Another set of binary variables is needed to indicate whether a given area j is covered (z_j).

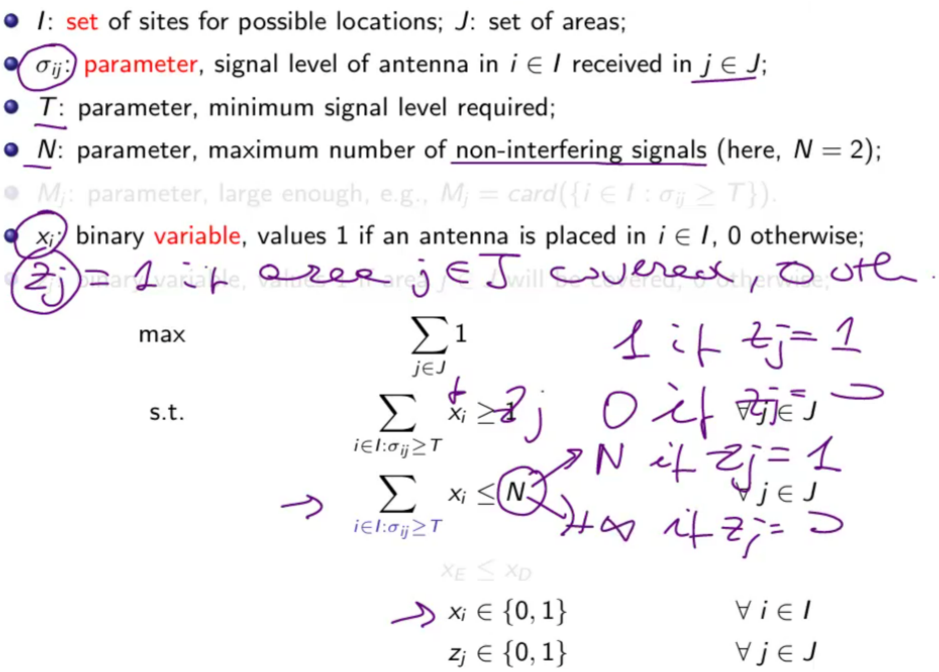
The model is therefore:

$$
\begin{aligned}
\max \quad & \sum_{j \in J} z_j \\
\text{s.t.} \quad & \sum_{i \in I, \sigma_{i,j} \geq T} x_i \geq x_j \quad \forall\, j \in J && (1) \\
& \sum_{i \in I, \sigma_{i,j} \geq T} x_i \leq N + M_j(1 - z_j) \quad \forall j \in J && (2) \\
& x_d \geq x_e && \text{vincolo sui siti E e D}
\end{aligned}
$$

Constraints (1) link variables relating to the coverage of a given area with antennas which are able to cover it. The constraints (2) require that an area be covered by at most N signals (1 for this instance of the problem).

The second part of these constraints concerns areas we are not interested in covering, that is those areas for which $z_j = 0$, and therefore if there are interferences there are no problems. This works because when $z_j = 0$, the constraint becomes $x_i \leq N + M_j$ where M_j is a number large enough to make the constraint redundant and does not go to limit the space of solutions. An optimal value for M_j is the cardinality of all sites covering area j, which is $M_j = |\{i \in I: \sigma_{i,j} \geq T\}|$

Decisions on areas should be made, considering whether we want to cover a specific area or not. We need a remodeling based on redundant constraints on a binary variable which makes us question whether the opening of specific areas or not is done:



- I: set of sites for possible locations; J: set of areas;
- σ_{ij}: parameter, signal level of antenna in $i \in I$ received in $j \in J$;
- T: parameter, minimum signal level required;
- N: parameter, maximum number of non-interfering signals (here, $N = 2$);
- M_j: parameter, large enough, e.g., $M_j = card(\{i \in I : \sigma_{ij} \geq T\})$.
- x_i: binary variable, values 1 if an antenna is placed in $i \in I$, 0 otherwise;
- z_j = 1 if area $j \in J$ covered, 0 othe.

$$\max \sum_{j \in J} 1 \quad 1 \text{ if } z_j = 1$$

$$\text{s.t.} \quad \sum_{i \in I : \sigma_{ij} \geq T} x_i \geq \cancel{1}\, z_j \quad 0 \text{ if } z_j = 0 \qquad \forall j \in J$$

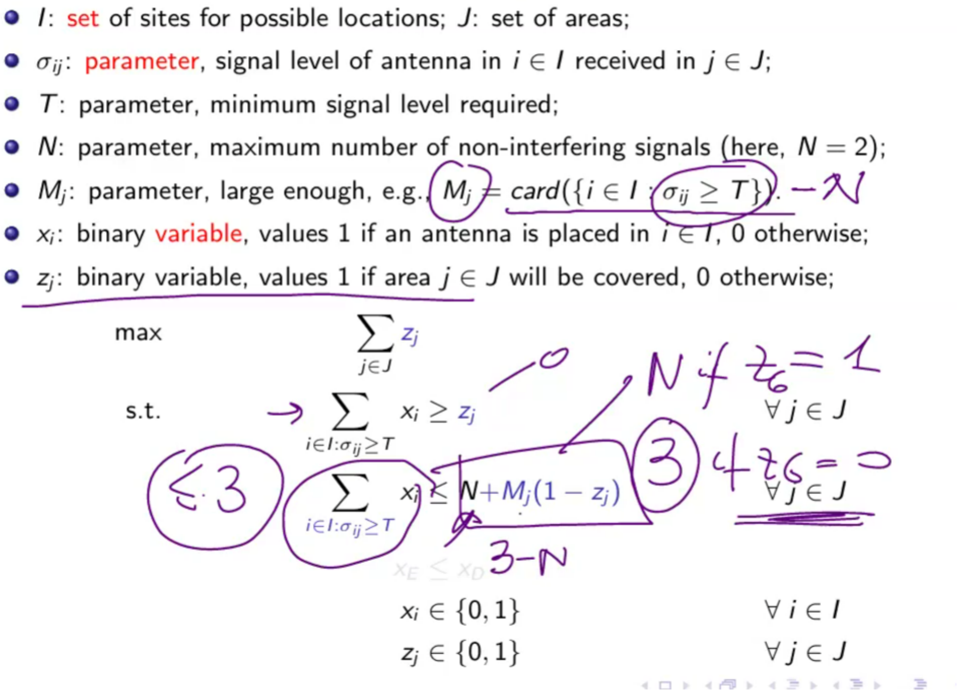
$$\rightarrow \sum_{i \in I : \sigma_{ij} \geq T} x_i \leq N \quad \begin{cases} N \text{ if } z_j = 1 \\ +\infty \text{ if } z_j = 0 \end{cases} \qquad \forall j \in J$$

$$x_E \leq x_D$$

$$\rightarrow x_i \in \{0, 1\} \qquad \forall i \in I$$

$$z_j \in \{0, 1\} \qquad \forall j \in J$$

Levels depend on the zones covered and the signals reached, whether they are installed within specific sites or not:



- I: set of sites for possible locations; J: set of areas;
- σ_{ij}: parameter, signal level of antenna in $i \in I$ received in $j \in J$;
- T: parameter, minimum signal level required;
- N: parameter, maximum number of non-interfering signals (here, $N = 2$);
- M_j: parameter, large enough, e.g., $M_j = card(\{i \in I : \sigma_{ij} \geq T\})$. $-N$
- x_i: binary variable, values 1 if an antenna is placed in $i \in I$, 0 otherwise;
- z_j: binary variable, values 1 if area $j \in J$ will be covered, 0 otherwise;

$$\max \sum_{j \in J} z_j$$

$$\text{s.t.} \rightarrow \sum_{i \in I : \sigma_{ij} \geq T} x_i \geq z_j \quad 0 \quad N \text{ if } z_6 = 1 \qquad \forall j \in J$$

$$\leq 3 \quad \sum_{i \in I : \sigma_{ij} \geq T} x_i \leq N + M_j(1 - z_j) \quad 3 \text{ if } z_6 = 0 \qquad \forall j \in J$$

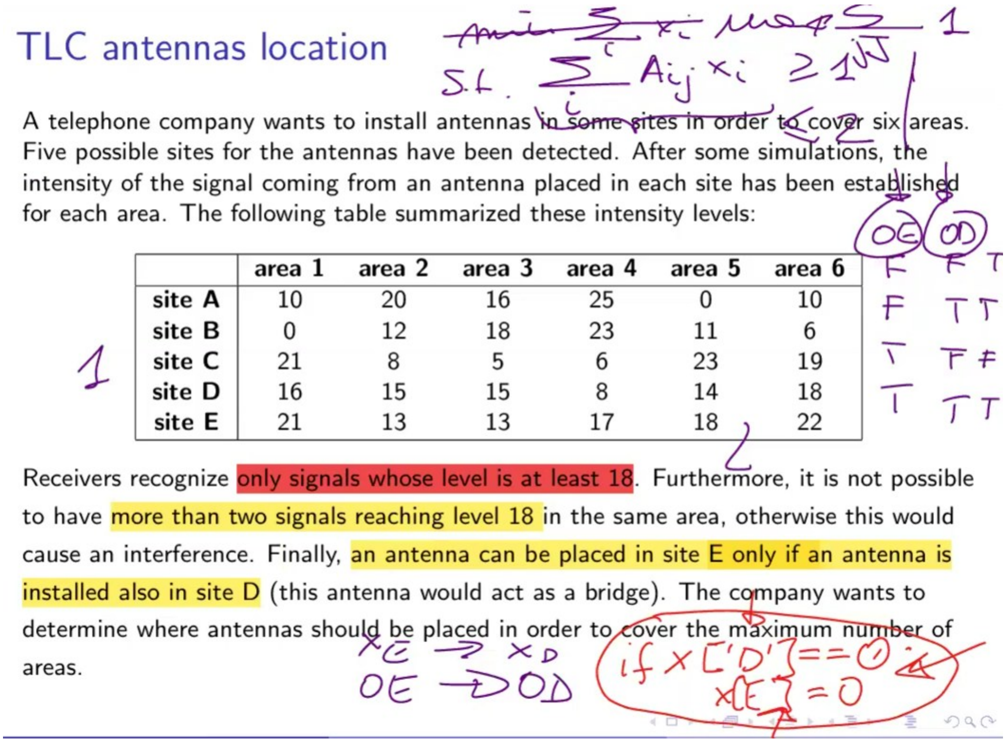
$$3 - N$$

$$x_E \leq x_D$$

$$x_i \in \{0, 1\} \qquad \forall i \in I$$

$$z_j \in \{0, 1\} \qquad \forall j \in J$$

Can we write linearly the opening conditions? Not so much, since if we cover a site, we do not cover the other one:



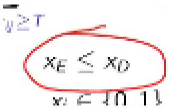
TLC antennas location

A telephone company wants to install antennas in some sites in order to cover six areas. Five possible sites for the antennas have been detected. After some simulations, the intensity of the signal coming from an antenna placed in each site has been established for each area. The following table summarized these intensity levels:

	area 1	area 2	area 3	area 4	area 5	area 6
site A	10	20	16	25	0	10
site B	0	12	18	23	11	6
site C	21	8	5	6	23	19
site D	16	15	15	8	14	18
site E	21	13	13	17	18	22

Receivers recognize only signals whose level is at least 18. Furthermore, it is not possible to have more than two signals reaching level 18 in the same area, otherwise this would cause an interference. Finally, an antenna can be placed in site E only if an antenna is installed also in site D (this antenna would act as a bridge). The company wants to determine where antennas should be placed in order to cover the maximum number of areas.

This brings to write the following constraint:



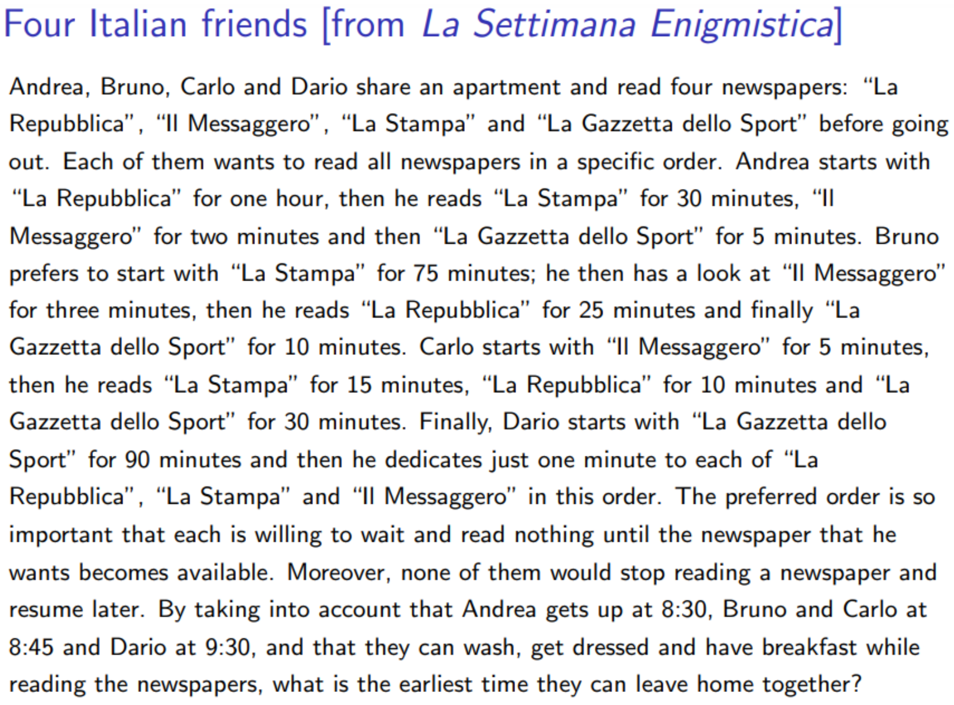
$y \geq T$

$x_E \leq x_D$

$x_i \in \{0, 1\}$

3.9 Job Scheduling Problem – Four Italian Friends

We consider a different modeling schema – newspaper reading (four Italian friends):



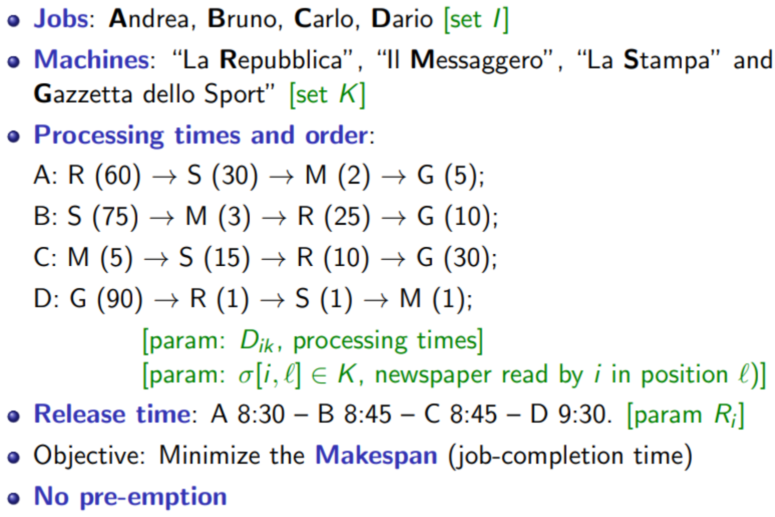
Four Italian friends [from *La Settimana Enigmistica*]

Andrea, Bruno, Carlo and Dario share an apartment and read four newspapers: "La Repubblica", "Il Messaggero", "La Stampa" and "La Gazzetta dello Sport" before going out. Each of them wants to read all newspapers in a specific order. Andrea starts with "La Repubblica" for one hour, then he reads "La Stampa" for 30 minutes, "Il Messaggero" for two minutes and then "La Gazzetta dello Sport" for 5 minutes. Bruno prefers to start with "La Stampa" for 75 minutes; he then has a look at "Il Messaggero" for three minutes, then he reads "La Repubblica" for 25 minutes and finally "La Gazzetta dello Sport" for 10 minutes. Carlo starts with "Il Messaggero" for 5 minutes, then he reads "La Stampa" for 15 minutes, "La Repubblica" for 10 minutes and "La Gazzetta dello Sport" for 30 minutes. Finally, Dario starts with "La Gazzetta dello Sport" for 90 minutes and then he dedicates just one minute to each of "La Repubblica", "La Stampa" and "Il Messaggero" in this order. The preferred order is so important that each is willing to wait and read nothing until the newspaper that he wants becomes available. Moreover, none of them would stop reading a newspaper and resume later. By taking into account that Andrea gets up at 8:30, Bruno and Carlo at 8:45 and Dario at 9:30, and that they can wash, get dressed and have breakfast while reading the newspapers, what is the earliest time they can leave home together?

In this case, we refer to a schema so called Job-Shop Scheduling Problem (JSP), which has these characteristics:

- No preemption
- Sequence constraints (specific reading order for each person)
- Release dates (different wake-up times)
- Single machine capacity (one newspaper can be read by one person at a time)
- Minimize makespan objective

The origin of the name comes from the sequence of operations on different machines (jobs), with different workshops/facilities where processing happen, processing all jobs available (makespan):



- **Jobs**: **A**ndrea, **B**runo, **C**arlo, **D**ario [set I]
- **Machines**: “La **R**epubblica”, “Il **M**essaggero”, “La **S**tampa” and “La **G**azzetta dello Sport” [set K]
- **Processing times and order**:
  A: R (60) → S (30) → M (2) → G (5);
  B: S (75) → M (3) → R (25) → G (10);
  C: M (5) → S (15) → R (10) → G (30);
  D: G (90) → R (1) → S (1) → M (1);
  [param: D_{ik}, processing times]
  [param: $\sigma[i,\ell] \in K$, newspaper read by i in position ℓ)]
- **Release time**: A 8:30 – B 8:45 – C 8:45 – D 9:30. [param R_i]
- Objective: Minimize the **Makespan** (job-completion time)
- **No pre-emption**

Let’s try to understand the model; the decisions are:

- “At what time each person starts reading the newspaper”
- Minimize the maximum value in which each person finishes
- The time when each person finishes reading must be at least as large as when each person finishes their last paper

$$\min y = (\max_{i\in I}\{h_{i,\sigma[i,|K|)} + D_{i,\sigma[I,|K|)}\}$$

At a time, people can have a conflict in which at the same time they read the same things – given it’s a decision. Now, let’s analyze each constraint:

- Linking the makespan to actual completion times
  - For each person i, y must be ≥ their start time on their last paper ($\sigma_{[i,|K|]}$) plus its duration
- No one can start their first paper ($\sigma_{[i,1]}$) before their wake-up time (R_i)
- Person i can't start their ℓ^{th} paper before finishing their (ℓ-1)th paper

$$h_{i\,\sigma[i,1]} \geq R_i \qquad \forall\, i \in I \qquad h_{ik} \geq h_{jk} + D_{jk}$$

$$h_{i\,\sigma[i,\ell]} \geq h_{i\,\sigma[i,\ell-1]} + D_{i\,\sigma[i,\ell-1]} \qquad \forall\, i \in I, \ell = 2...|K| \qquad h_{jk} \geq h_{ik} + D_{ik}$$

Given two persons and a newspaper, what is the exact order between the variables? We don’t know, specifically the order between i, j, k, that’s where the variable x_{ijk} comes from.

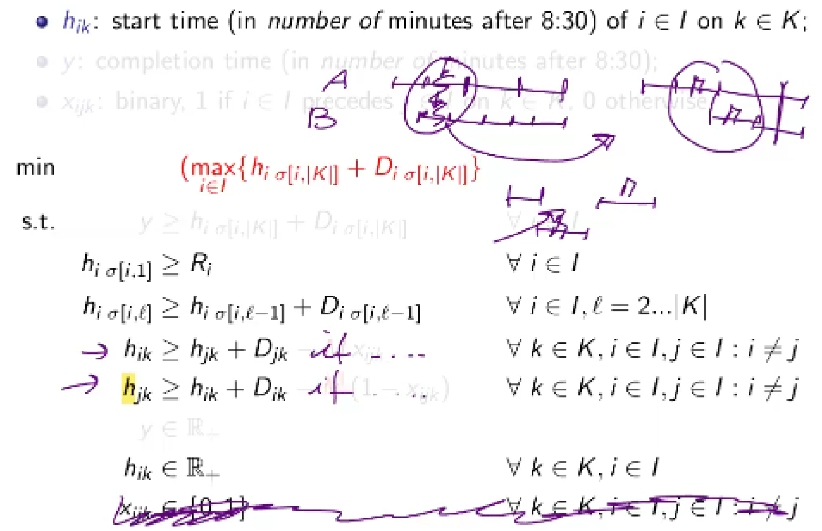
The maximum function is not a linear function, so we have to linearize it, introducing y. We talk here about disjunction constraints, which represent union between binary variables and also their intersection (e.g., different turns).

$$
\begin{array}{ll}
h_{ik} \geq h_{jk} + D_{jk} - M\, x_{ijk} & \forall\, k \in K, i \in I, j \in I : i \neq j \\
h_{jk} \geq h_{ik} + D_{ik} - M\,(1 - x_{ijk}) & \forall\, k \in K, i \in I, j \in I : i \neq j \\
y \in \mathbb{R}_+ & \\
h_{ik} \in \mathbb{R}_+ & \forall\, k \in K, i \in I \\
x_{ijk} \in \{0,1\} & \forall\, k \in K, i \in I, j \in I : i \neq j
\end{array}
$$

Their purpose is the following:

- These constraints handle the "either-or" situation where newspaper k must be read by either person i or person j at any given time
- They enforce the resource capacity constraint (one newspaper can only be read by one person at a time)

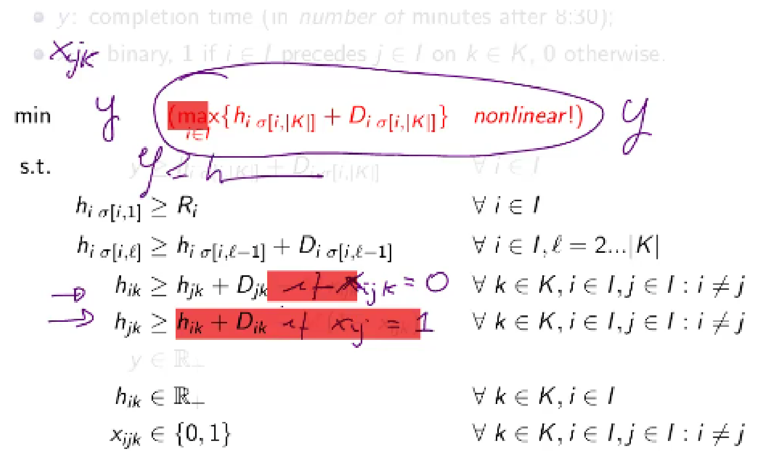
This is also represented by:



- h_{ik}: start time (in *number of* minutes after 8:30) of $i \in I$ on $k \in K$;
- y: completion time (in *number of* minutes after 8:30);
- x_{ijk}: binary, 1 if $i \in I$ precedes $j \in I$ on $k \in K$, 0 otherwise.

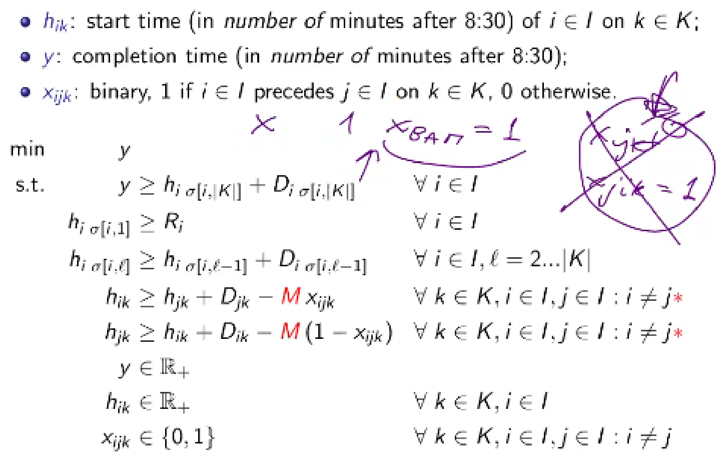
$$
\begin{array}{lll}
\min & \left(\max_{i \in I}\{h_{i\,\sigma[i,|K|]} + D_{i\,\sigma[i,|K|]}\}\right. & \\
\text{s.t.} & y \geq h_{i\,\sigma[i,|K|]} + D_{i\,\sigma[i,|K|]} & \forall\, i \in I \\
& h_{i\,\sigma[i,1]} \geq R_i & \forall\, i \in I \\
& h_{i\,\sigma[i,\ell]} \geq h_{i\,\sigma[i,\ell-1]} + D_{i\,\sigma[i,\ell-1]} & \forall\, i \in I, \ell = 2\ldots|K| \\
\rightarrow & h_{ik} \geq h_{jk} + D_{jk} \text{ if } x_{ijk} \ldots & \forall\, k \in K, i \in I, j \in I : i \neq j \\
\rightarrow & h_{jk} \geq h_{ik} + D_{ik} \text{ if } (1 - x_{ijk}) & \forall\, k \in K, i \in I, j \in I : i \neq j \\
& y \in \mathbb{R}_+ & \\
& h_{ik} \in \mathbb{R}_+ & \forall\, k \in K, i \in I \\
& \cancel{x_{ijk} \in \{0,1\}} & \cancel{\forall\, k \in K, i \in I, j \in I : i \neq j}
\end{array}
$$

The maximum makespan has to be linearized and make constraint redundant when one stops reading (introducing binary variables), choosing a specific order:



- y: completion time (in *number of* minutes after 8:30);
- x_{ijk}: binary, 1 if $i \in I$ precedes $j \in I$ on $k \in K$, 0 otherwise.

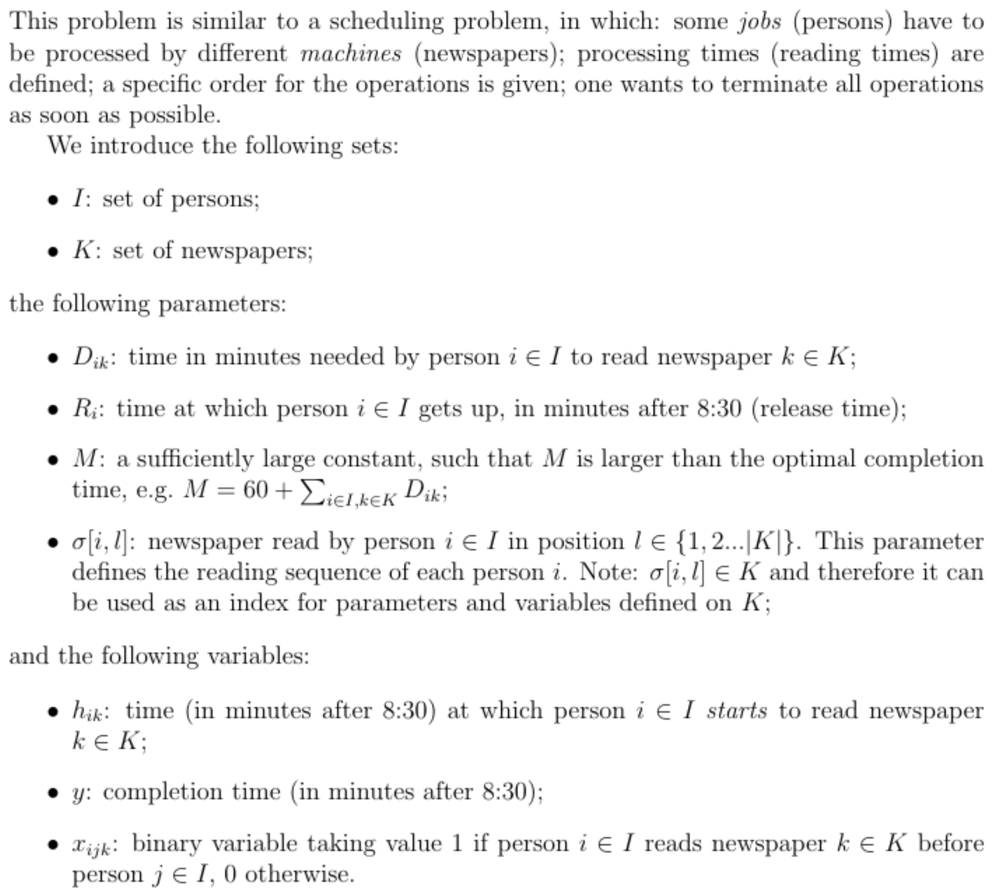
$$
\begin{array}{lll}
\min & y \quad \left(\max_{i \in I}\{h_{i\,\sigma[i,|K|]} + D_{i\,\sigma[i,|K|]}\} \quad \textit{nonlinear!}\right) \quad y & \\
\text{s.t.} & y \geq h_{i\,\sigma[i,|K|]} + D_{i\,\sigma[i,|K|]} \quad y \geq h & \forall\, i \in I \\
& h_{i\,\sigma[i,1]} \geq R_i & \forall\, i \in I \\
& h_{i\,\sigma[i,\ell]} \geq h_{i\,\sigma[i,\ell-1]} + D_{i\,\sigma[i,\ell-1]} & \forall\, i \in I, \ell = 2\ldots|K| \\
\rightarrow & h_{ik} \geq h_{jk} + D_{jk} \text{ if } x_{jk} = 0 & \forall\, k \in K, i \in I, j \in I : i \neq j \\
\rightarrow & h_{jk} \geq h_{ik} + D_{ik} \text{ if } x_{ij} = 1 & \forall\, k \in K, i \in I, j \in I : i \neq j \\
& y \in \mathbb{R}_+ & \\
& h_{ik} \in \mathbb{R}_+ & \forall\, k \in K, i \in I \\
& x_{ijk} \in \{0,1\} & \forall\, k \in K, i \in I, j \in I : i \neq j
\end{array}
$$



- h_{ik}: start time (in *number of* minutes after 8:30) of $i \in I$ on $k \in K$;
- y: completion time (in *number of* minutes after 8:30);
- x_{ijk}: binary, 1 if $i \in I$ precedes $j \in I$ on $k \in K$, 0 otherwise.

$$
\begin{array}{lll}
\min & y & \\
\text{s.t.} & y \geq h_{i\,\sigma[i,|K|]} + D_{i\,\sigma[i,|K|]} & \forall\, i \in I \\
& h_{i\,\sigma[i,1]} \geq R_i & \forall\, i \in I \\
& h_{i\,\sigma[i,\ell]} \geq h_{i\,\sigma[i,\ell-1]} + D_{i\,\sigma[i,\ell-1]} & \forall\, i \in I, \ell = 2\ldots|K| \\
& h_{ik} \geq h_{jk} + D_{jk} - M\, x_{ijk} & \forall\, k \in K, i \in I, j \in I : i \neq j* \\
& h_{jk} \geq h_{ik} + D_{ik} - M\,(1 - x_{ijk}) & \forall\, k \in K, i \in I, j \in I : i \neq j* \\
& y \in \mathbb{R}_+ & \\
& h_{ik} \in \mathbb{R}_+ & \forall\, k \in K, i \in I \\
& x_{ijk} \in \{0,1\} & \forall\, k \in K, i \in I, j \in I : i \neq j
\end{array}
$$

Completely:



This problem is similar to a scheduling problem, in which: some *jobs* (persons) have to be processed by different *machines* (newspapers); processing times (reading times) are defined; a specific order for the operations is given; one wants to terminate all operations as soon as possible.

We introduce the following sets:

- I: set of persons;
- K: set of newspapers;

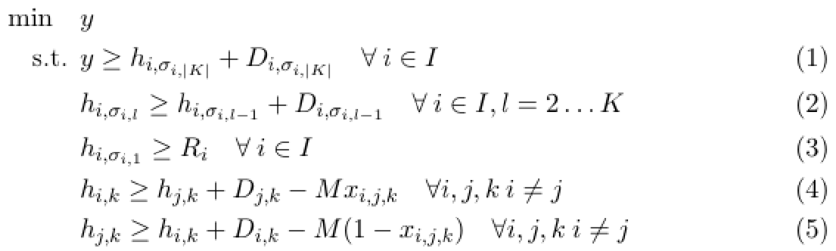
the following parameters:

- D_{ik}: time in minutes needed by person $i \in I$ to read newspaper $k \in K$;
- R_i: time at which person $i \in I$ gets up, in minutes after 8:30 (release time);
- M: a sufficiently large constant, such that M is larger than the optimal completion time, e.g. $M = 60 + \sum_{i \in I, k \in K} D_{ik}$;
- $\sigma[i, l]$: newspaper read by person $i \in I$ in position $l \in \{1, 2...|K|\}$. This parameter defines the reading sequence of each person i. Note: $\sigma[i, l] \in K$ and therefore it can be used as an index for parameters and variables defined on K;

and the following variables:

- h_{ik}: time (in minutes after 8:30) at which person $i \in I$ *starts* to read newspaper $k \in K$;
- y: completion time (in minutes after 8:30);
- x_{ijk}: binary variable taking value 1 if person $i \in I$ reads newspaper $k \in K$ before person $j \in I$, 0 otherwise.

The constraints are the following:



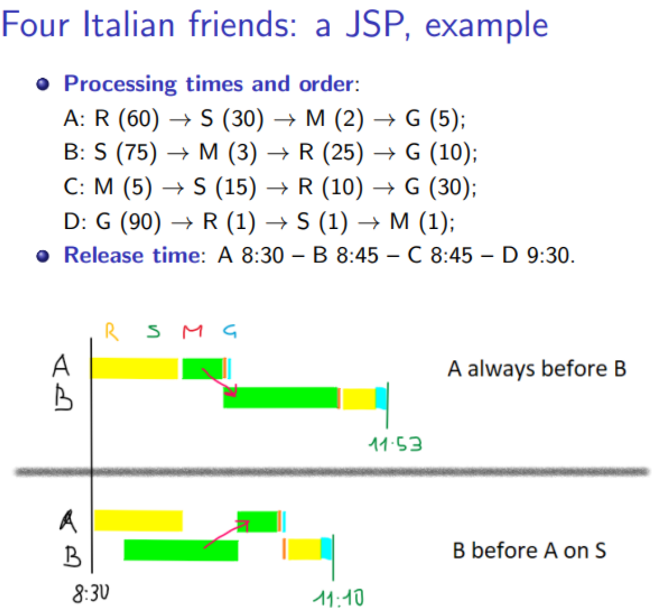
$$
\begin{aligned}
\min \quad & y \\
\text{s.t.} \quad & y \geq h_{i,\sigma_{i,|K|}} + D_{i,\sigma_{i,|K|}} \quad \forall\, i \in I & (1)\\
& h_{i,\sigma_{i,l}} \geq h_{i,\sigma_{i,l-1}} + D_{i,\sigma_{i,l-1}} \quad \forall\, i \in I, l = 2 \ldots K & (2)\\
& h_{i,\sigma_{i,1}} \geq R_i \quad \forall\, i \in I & (3)\\
& h_{i,k} \geq h_{j,k} + D_{j,k} - M x_{i,j,k} \quad \forall i, j, k\; i \neq j & (4)\\
& h_{j,k} \geq h_{i,k} + D_{i,k} - M(1 - x_{i,j,k}) \quad \forall i, j, k\; i \neq j & (5)
\end{aligned}
$$

The meaning of constraint sets is:

1. The makespan must be greater than or equal to the time of completion of each person's last activity. In short words these constraints ensure that everyone finishes reading before the y-moment.
2. Avoid overlapping two steps of the same activity. A person cannot read two newspapers at once.
3. The first step cannot be taken before the activity begins. A person cannot read while sleeping.
4. It states that if the person i reads the newspaper k before the person j, the moment when i begins reading k is any instant, while if i does not read k before j ($x_{i,j,k} = 0$), then i must start reading k after j has finished.

5. It requires that if the person i reads the newspaper k before the person j, the moment when j begins reading k is subsequent to the instant when i ends. This and the previous link are mutually exclusive.

Consider the following example:



Four Italian friends: a JSP, example

- **Processing times and order**:
  A: R (60) → S (30) → M (2) → G (5);
  B: S (75) → M (3) → R (25) → G (10);
  C: M (5) → S (15) → R (10) → G (30);
  D: G (90) → R (1) → S (1) → M (1);
- **Release time**: A 8:30 – B 8:45 – C 8:45 – D 9:30.

The Gantt charts show two different scenarios for scheduling just A and B:

1. First Scenario (11:53 completion):
   1. Shows A reading all papers before B
   2. Results in a later completion time $(11:53)$

2. Second Scenario (11:10 completion):
   1. Shows B reading Stampa before A
   2. Results in an earlier completion time $(11:10)$
   3. Demonstrates how allowing B to read Stampa before A leads to a better overall schedule

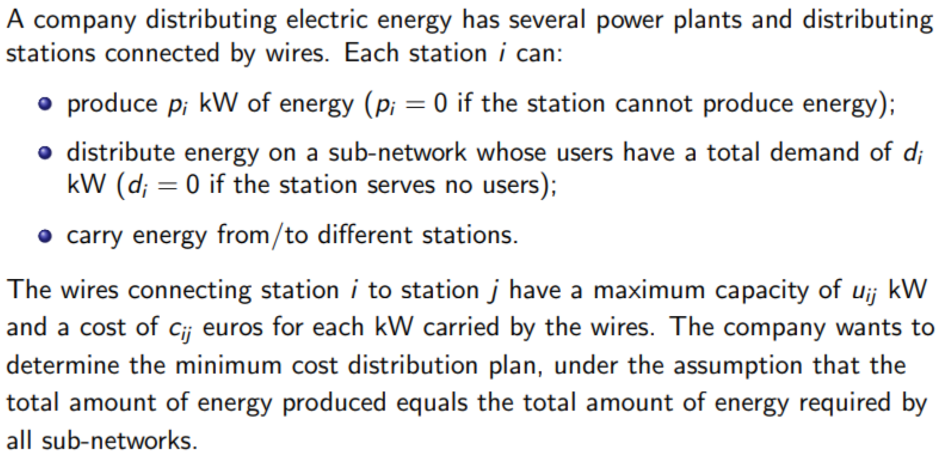
This example illustrates key JSP concepts:

- Resource conflicts (can't read same paper simultaneously)
- Sequence dependencies (must follow specific order)
- Release time constraints (can't start before wake-up)
- How different sequencing decisions affect makespan
- The importance of finding optimal ordering to minimize completion time

The two scenarios effectively demonstrate how the disjunctive constraints work in practice – either A reads before B or B before A on shared resources (newspapers), leading to different possible schedules and completion times.

3.10 ENERGY FLOW PROBLEM (SINGLE/MULTI)

We go into the details of a different problem here (energy flow problem), which can be used to describe a Network Flow Problem, specifically a Minimum Cost Network Flow Problem, where we discuss production capacities being sent between stations and units of energy which can be used:

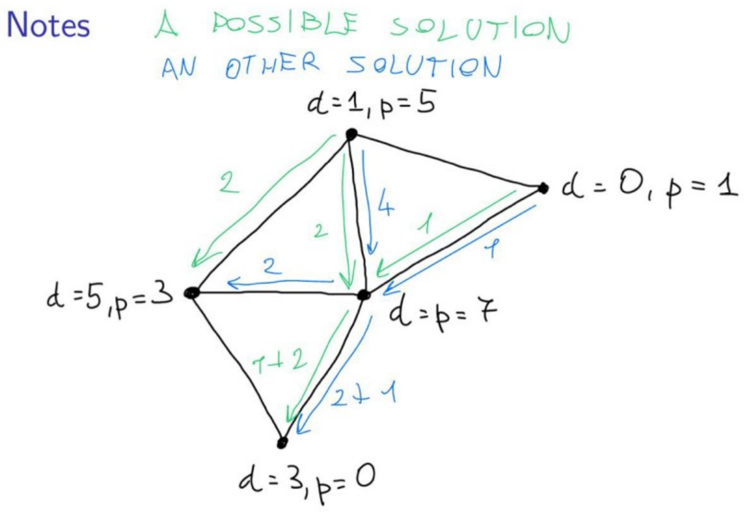


A company distributing electric energy has several power plants and distributing stations connected by wires. Each station i can:

- produce p_i kW of energy ($p_i = 0$ if the station cannot produce energy);
- distribute energy on a sub-network whose users have a total demand of d_i kW ($d_i = 0$ if the station serves no users);
- carry energy from/to different stations.

The wires connecting station i to station j have a maximum capacity of u_{ij} kW and a cost of c_{ij} euros for each kW carried by the wires. The company wants to determine the minimum cost distribution plan, under the assumption that the total amount of energy produced equals the total amount of energy required by all sub-networks.

This production network flow example shows 5 nodes with their demand (d) and production (p) values, moving the energy between places so to balance it.



To plan the distribution we have to decide how much energy is transferred from one station to another $x_{i,j}$ = amount of energy transferred from i to j.

An interesting feature of this problem is that it can be modeled as a graph $G = (N, A)$ whose nodes correspond to the energy stations and the arcs represent the connections between the various stations.

To simplify the problem modelling, it is possible to add a bv parameter for each $v \in N$ node in the network which represents the difference between the demand that the station must satisfy and the amount of energy it can produce:

- if b_v is a positive value, the demand is higher than the station's capacity and therefore energy from other stations needs to be transferred

- if b_v is a negative value, the station produces more energy than needed and therefore the excess energy must be sent to the other stations
- if $b_v = 0$, the station is self-sufficient or a transmission node because $p_v = d_v = 0$

We define the objective function:

$$\min \sum_{(i,j)\in A} c_{i,j}x_{i,j}$$

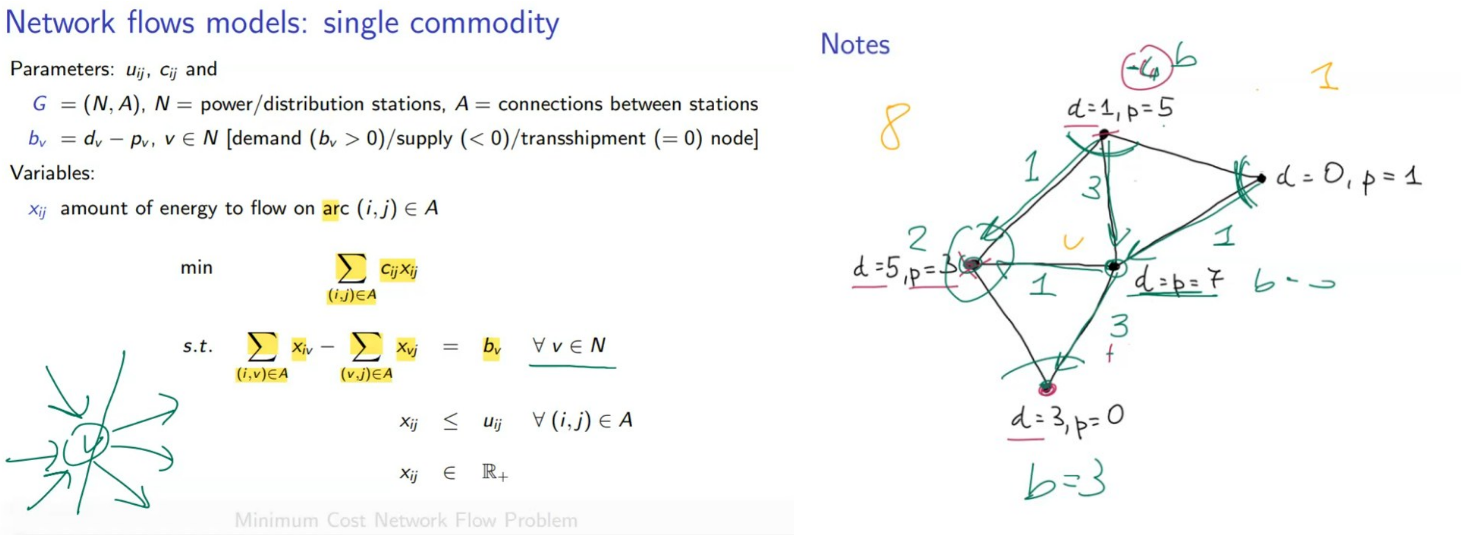
Now, pose the constraints that each node receives exactly bv units of flow (negative if they are to be removed) (node balance constraint).

$$\underbrace{\sum_{(i,v)\in A} x_{i,v}}_{\text{flusso in ingresso}} - \underbrace{\sum_{(v,j)\in A} x_{v,j}}_{\text{flusso in uscita}} = b_v \quad \forall\, v \in N$$

Finally, it is necessary to impose a limit on the capacity of cables (arc capacity constraint):

$$x_{i,j} \leq u_{i,j} \quad (i,j) \in A$$

This model has unique features, representing a minimum cost flow inside of a network: generally, we can describe the below as: "Find the cheapest way to send energy from producers (supply nodes) to consumers (demand nodes) through a network of wires (arcs), respecting capacity limits and ensuring flow balance at each station (node). Consider the flow has to be balanced between the quantity coming in and the quantity coming out.



Network flows models: single commodity

Parameters: u_{ij}, c_{ij} and

$G = (N, A)$, N = power/distribution stations, A = connections between stations

$b_v = d_v - p_v$, $v \in N$ [demand ($b_v > 0$)/supply (< 0)/transshipment ($= 0$) node]

Variables:

x_{ij} amount of energy to flow on arc $(i,j) \in A$

$$\begin{aligned} \min \quad & \sum_{(i,j)\in A} c_{ij}x_{ij} \\ s.t. \quad & \sum_{(i,v)\in A} x_{iv} - \sum_{(v,j)\in A} x_{vj} = b_v \quad \forall\, v \in N \\ & x_{ij} \leq u_{ij} \quad \forall\,(i,j) \in A \\ & x_{ij} \in \mathbb{R}_+ \end{aligned}$$

Minimum Cost Network Flow Problem

Notes

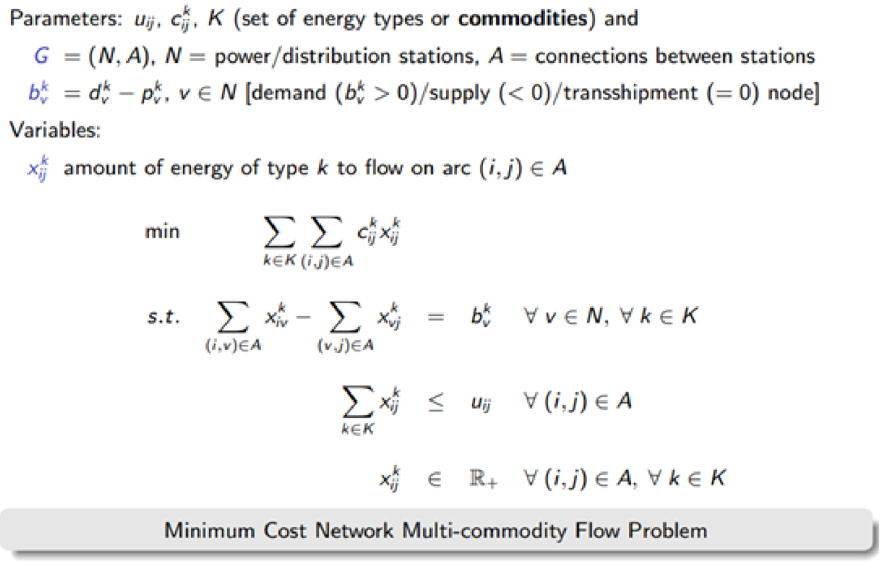
This is a single-commodity type of problem, where there is one type of flow, where each arc has one capacity constraint, which is easier to solve.

One variant of the problem is where there are each station handles various types of energy and the cost of transport depends on the type.

- The capacity of the bows is not affected by the type of energy passing through
- The solution to this problem is similar to that of the classical version, with the difference that another index is used for parameters and variables which discriminates between types of energy

This is the multi-commodity variant, where there is shared arc capacity across commodities but also flow conservation per node AND per commodity.

- This is more complex to solve because many more variables and constraints are needed since the flows depend on each other. If these were independent it would be possible to decompose this problem in $|K|$ minimum flow problems and then combine the various solutions



Parameters: u_{ij}, c_{ij}^k, K (set of energy types or **commodities**) and

$G = (N, A)$, $N =$ power/distribution stations, $A =$ connections between stations

$b_v^k = d_v^k - p_v^k$, $v \in N$ [demand ($b_v^k > 0$)/supply (< 0)/transshipment ($= 0$) node]

Variables:

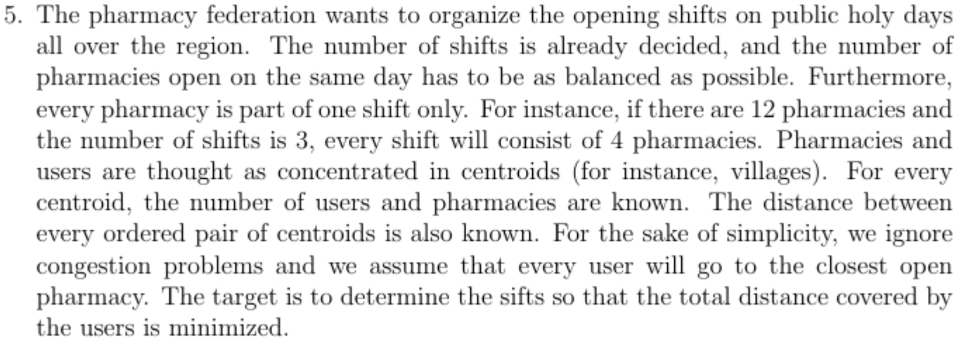
x_{ij}^k amount of energy of type k to flow on arc $(i,j) \in A$

$$\begin{aligned} \min \quad & \sum_{k \in K} \sum_{(i,j) \in A} c_{ij}^k x_{ij}^k \\ \text{s.t.} \quad & \sum_{(i,v) \in A} x_{iv}^k - \sum_{(v,j) \in A} x_{vj}^k = b_v^k \quad \forall\, v \in N,\ \forall\, k \in K \\ & \sum_{k \in K} x_{ij}^k \leq u_{ij} \quad \forall\, (i,j) \in A \\ & x_{ij}^k \in \mathbb{R}_+ \quad \forall\, (i,j) \in A,\ \forall\, k \in K \end{aligned}$$

Minimum Cost Network Multi-commodity Flow Problem

3.11 Other Models: Pharmacy Federation Turns (Variant 1 and 2)

In this section, using older notes, we want to complete the topic of modelling by including other models present inside of the notes by the professor.



5. The pharmacy federation wants to organize the opening shifts on public holy days all over the region. The number of shifts is already decided, and the number of pharmacies open on the same day has to be as balanced as possible. Furthermore, every pharmacy is part of one shift only. For instance, if there are 12 pharmacies and the number of shifts is 3, every shift will consist of 4 pharmacies. Pharmacies and users are thought as concentrated in centroids (for instance, villages). For every centroid, the number of users and pharmacies are known. The distance between every ordered pair of centroids is also known. For the sake of simplicity, we ignore congestion problems and we assume that every user will go to the closest open pharmacy. The target is to determine the sifts so that the total distance covered by the users is minimized.

In this case we want to decide which pharmacy does which shift, so that there is good coverage of the territory, assuming that people go to the nearest pharmacy. The aim is therefore to minimise the road people have to take to reach the pharmacies on duty. You certainly need a variable that specifies which pharmacy is open in which shift.

$$y_{i,k} = \begin{cases} 1 & \text{la farmacia } i \text{ è aperta nel turno } k \\ 0 & \text{altrimenti} \end{cases}$$

with $i \in P, k \in (1, \dots K)$, where P is the set of pharmacies and K is the number of turns we want to be done. To express our objective function we need other variables, because we also have to consider the distance of pharmacies, so that we can minimize it.

- There will then be set C of clients that need to be served and parameters specifying the distance $D_{j,i}$. The distance between a customer $j \in C$ and the pharmacy $i \in P$.
- However, the distance to be travelled by the user depends on which pharmacies are open in a given shift and therefore it is not advisable to use the parameter directly, since the distance varies according to the shift
- It is therefore advisable to add a variable specifying how much road the customer j must take during shift k to reach the nearest open pharmacy

$d_{jk} =$ distance between customer j and nearest pharmacy during shift k

However, it is necessary to somehow connect the variables d_{jk} with the opening/closing of pharmacies. There is therefore a way of discriminating against which pharmacy the user goes to in a given shift:

$$x_{j,i,k} = \begin{cases} 1 & \text{se } j \text{ va nella farmacia } i \text{ durante il turno } k \\ 0 & \text{altrimenti} \end{cases}$$

This way it is easy to find value for $d_{j,k}$, because the constraint is enough:

$$d_{j,k} = \sum_{i \in P} D_{j,i} x_{j,i,k} \quad \forall\, j \in C, k \in K$$

With this constraint only a distance is considered for each shift, because during a shift the customer always goes to the nearest pharmacy and then, set a j and a k, there will be only one $x_{j,i,k}$ which is 1. The solver does not know this last thing and therefore it is necessary to add the appropriate constraints:

$$\sum_{i \in P} x_{j,i,k} = 1 \quad \forall\, j \in C, k \in K$$

There is still no requirement that each pharmacy should work exactly one shift, which can simply be added with a sum on the $y_{i,k}$:

$$\sum_{k=1}^{K} y_{i,k} = 1 \quad \forall\, i \in P$$

To complete the model it remains to connect the x with the y, because obviously a customer cannot go in a closed pharmacy.

$$x_{j,i,k} \leq y_{j,k} \quad \forall\, i, j, k$$

It also remains to be required that each shift be balanced, that is, there should always be a similar number of open pharmacies:

$$\left\lfloor \frac{|P|}{K} \right\rfloor \leq \sum_{i \in P} y_{i,k} \leq \left\lceil \frac{|P|}{K} \right\rceil \quad \forall\, k$$

We then specify variables domains:

$$\begin{aligned} y_{i,k} &\in \{0,1\} \\ x_{j,i,k} &\in \{0,1\} \\ d_{j,k} &\in \mathbb{R} \end{aligned}$$

Too bad there’s a problem. With the current constraints we have expressed that for each shift a customer always goes to the same pharmacy and that that pharmacy must be open, but it is not specified that the customer goes to the nearest pharmacy.

- Actually this is not a problem, because it is during the optimization process that the various distances are set to be minimized
- This is because the objective of a model is to describe the characteristics of a solution, while it is the solver who is looking for the optimal solution performs minimization
- In fact, a solution that sends a customer to a different pharmacy from the open one closest to him is still an acceptable solution, but it is certainly not great and therefore discarded

Some observations:

- Once an optimal solution has been found for this problem, it can be observed that by swapping the order of turns obtained, another optimal solution is obtained with a different order
- This is caused by the fact that once the pharmacies are chosen which are open in the various shifts, the order in which the shifts are carried out is indifferent, thus obtaining a symmetrical solution
- The presence of these symmetries is typically a problem because it can lead to a combinatorial explosion of solutions
- The origin of these symmetries is typically caused by the model, in this case the problem stems from the fact that "a name" is given to the turns and it is not always possible to re-model the problem so that there are no symmetries

An alternate version for this problem formulation is the following one.

Since we have a set of pharmacies P and each pharmacy only does one shift, we can see a shift as a subset of P.

The choice of shifts becomes a choice of which subsets to select from the set of 2^P parts. This choice can be modelled with a binary variable.

$$x_J = \begin{cases} 1 & \text{se il sottoinsieme } J \text{ è un turno} \\ 0 & \text{altrimenti} \end{cases} \qquad \forall\, J \subset P, J \in 2^P$$

With this variable there is no symmetry as the variable is directly related to the turn it represents. The minimization to be done then becomes (j represents the customers, J the turn).

$$\min \sum_{J \in 2^P} \sum_{j \in C} D_{j,J} x_J$$

In the objective function, the variable $d_{j,k}$ no longer appears, but a parameter $D_{j,J}$ appears, because in the previous formulation the composition of the various shifts was variable and consequently the distance also changed according to the composition of the shift, With this new model I know a priori which pharmacies belong to a certain shift and therefore for each shift and for each client I can pre-calculate the minimum distance.

There are other constraints that need to be re-formulated. To specify that you are exactly K turns, just add up the x_J.

$$\sum_{J \in 2^P} x_J = K$$

It is also necessary to impose the constraint that each pharmacy should work exactly one shift, because at the moment the same pharmacy can appear in several shifts (subsets). In this case, an additional parameter is needed to specify whether a pharmacy is on a certain shift.

$$A_{i,J} = \begin{cases} 1 & \text{se } i \in J \\ 0 & \text{altriment} \end{cases} \quad \forall\, J \in 2^P$$

Note that it is a parameter and not a variable because it is a value that can be pre-calculated when the set of parts is constructed. With these parameters it is easy to establish the constraint that a pharmacy should only take one shift.

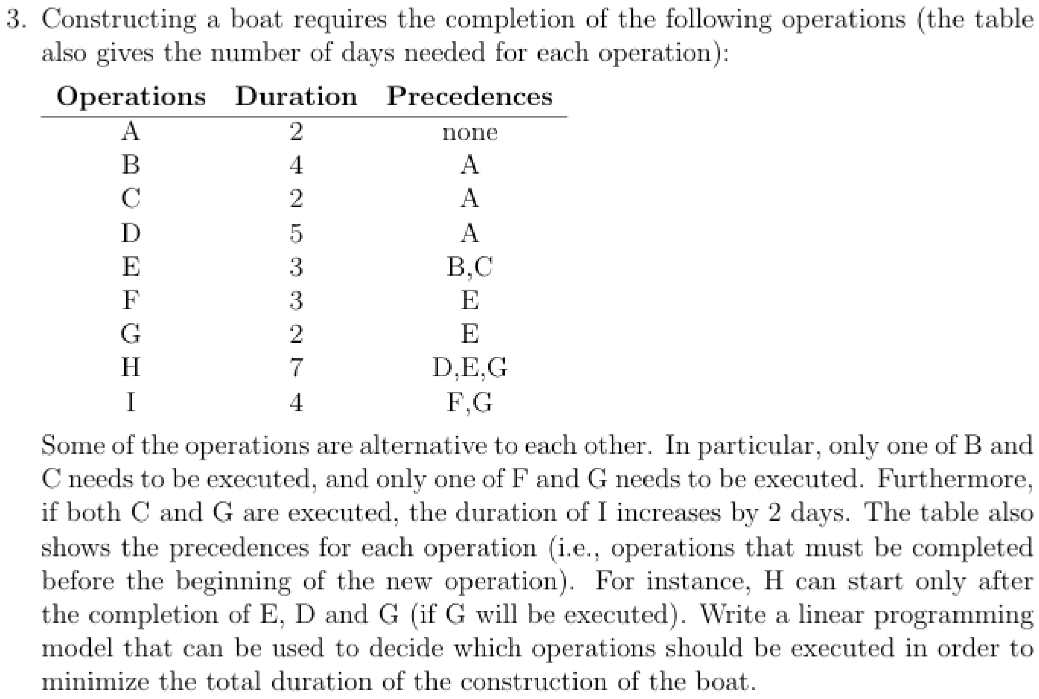
$$\sum_{J \in 2^P} A_{i,J} x_J = 1 \quad \forall\, i \in P$$

It remains to shape the fact that shifts must be balanced, but to do this we do not need new constraints. In fact it is sufficient to consider, instead of the whole set of parts 2^P, a subset G composed only by the subsets of P that have similar cardinality.

$$G = \left\{ x \mid x \in 2^P, \left\lfloor \frac{|P|}{K} \right\rfloor \leq |\, x \,| \leq \left\lceil \frac{|P|}{K} \right\rceil \right\}$$

This model has no symmetries and is quite simple, however it suffers from a big problem: if there are 100 pharmacies, the calculation of the set of parts of P and parameters can take too long because of the exponential growth of the cardinality of the set of parts.

3.12 Other Models: Boat Construction



3. Constructing a boat requires the completion of the following operations (the table also gives the number of days needed for each operation):

Operations	Duration	Precedences
A	2	none
B	4	A
C	2	A
D	5	A
E	3	B,C
F	3	E
G	2	E
H	7	D,E,G
I	4	F,G

Some of the operations are alternative to each other. In particular, only one of B and C needs to be executed, and only one of F and G needs to be executed. Furthermore, if both C and G are executed, the duration of I increases by 2 days. The table also shows the precedences for each operation (i.e., operations that must be completed before the beginning of the new operation). For instance, H can start only after the completion of E, D and G (if G will be executed). Write a linear programming model that can be used to decide which operations should be executed in order to minimize the total duration of the construction of the boat.

First, let me explain the key elements of the problem:

1. We have 9 operations (A through I) with given durations and precedence relationships
2. There are two pairs of alternative operations:
   - Either B or C must be executed (not both)
   - Either F or G must be executed (not both)
3. Special condition: If both C and G are executed, operation I takes 2 days longer

Let's see now the key components of this model:

Variables:

- t_i: Completion time of operation i
- y_i: Binary variable for alternative operations (B, C, F, G)
- y_{CG}: Binary variable that tracks if both C and G are executed
- z: Overall completion time (objective to minimize)

Constraints:

1. Precedence relationships (e.g., $t_B \geq t_A + d_B$)
2. Mutually exclusive operations ($y_B + y_C = 1$ and $y_F + y_G = 1$)
3. Special condition for I's duration when C and G are both selected
4. All operations complete by time z

To find the *optimal solution*, we need to:

1. *Determine* which alternative operations to select
2. *Schedule* the selected operations to minimize total duration

The optimal solution to this problem would be:

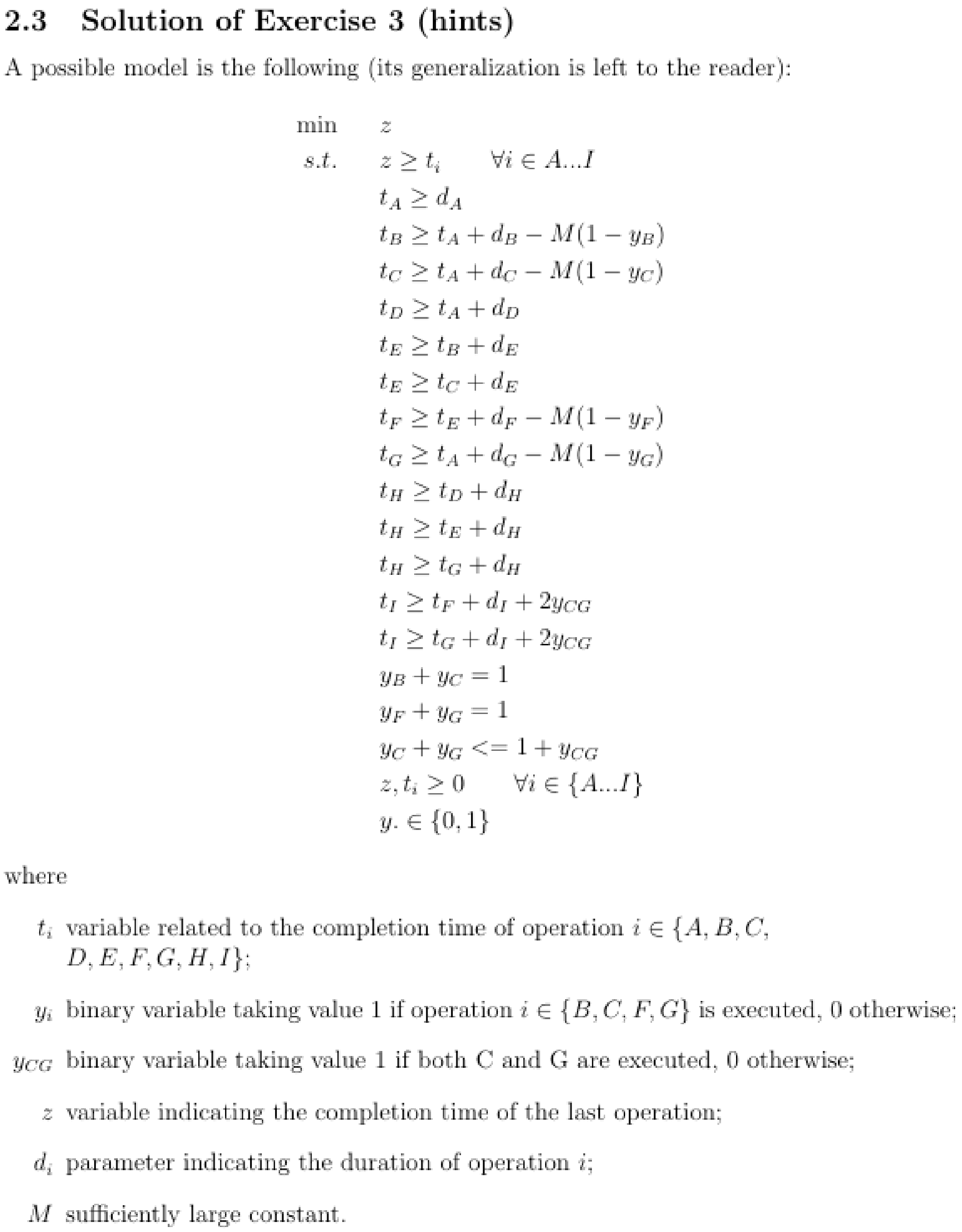
1. Select operation B over C (better for precedences)
2. Select operation F over G (shorter path to completion)
3. Schedule operations in this order: $A\ (0-2) B\ (2-6) D\ (2-7) E\ (6-9) F\ (9-12) H\ (12-19) I\ (19-23)$
4. Total duration = 23 days

This is optimal because:

- Choosing B (4 days) over C (2 days) allows for better parallel execution with D
- Choosing F over G avoids the 2-day penalty on operation I
- The critical path is $A \rightarrow B \rightarrow E \rightarrow F \rightarrow H \rightarrow I$

The following MILP formulation:

- Minimizes overall completion time
- Ensures precedence relationships are respected
- Handles alternative operations through binary variables
- Captures the duration increase for operation I when C and G are both selected



2.3 Solution of Exercise 3 (hints)

A possible model is the following (its generalization is left to the reader):

$$
\begin{array}{ll}
\min & z \\
s.t. & z \geq t_i \qquad \forall i \in A...I \\
& t_A \geq d_A \\
& t_B \geq t_A + d_B - M(1 - y_B) \\
& t_C \geq t_A + d_C - M(1 - y_C) \\
& t_D \geq t_A + d_D \\
& t_E \geq t_B + d_E \\
& t_E \geq t_C + d_E \\
& t_F \geq t_E + d_F - M(1 - y_F) \\
& t_G \geq t_A + d_G - M(1 - y_G) \\
& t_H \geq t_D + d_H \\
& t_H \geq t_E + d_H \\
& t_H \geq t_G + d_H \\
& t_I \geq t_F + d_I + 2y_{CG} \\
& t_I \geq t_G + d_I + 2y_{CG} \\
& y_B + y_C = 1 \\
& y_F + y_G = 1 \\
& y_C + y_G <= 1 + y_{CG} \\
& z, t_i \geq 0 \qquad \forall i \in \{A...I\} \\
& y. \in \{0, 1\}
\end{array}
$$

where

t_i variable related to the completion time of operation $i \in \{A, B, C, D, E, F, G, H, I\}$;

y_i binary variable taking value 1 if operation $i \in \{B, C, F, G\}$ is executed, 0 otherwise;

y_{CG} binary variable taking value 1 if both C and G are executed, 0 otherwise;

z variable indicating the completion time of the last operation;

d_i parameter indicating the duration of operation i;

M sufficiently large constant.

A more general representation is left here for the reader.

Sets

- T: set of tasks
- $P \subseteq T\ T$: precedence relationships
- $A = \{A_1, \ldots, A_\mathrm{k}\}$: groups of alternative tasks where each $A_\mathrm{i} \subseteq T$
- $I = \{(S, t, \delta)\}$: task interactions where:
  - $S \subseteq T$: set of interacting tasks
  - $t \in T$: affected task
  - δ: duration increase

Parameters

- d_{i}: duration of task i
- M: large constant

Variables

- $t_{\mathrm{i}} \geq 0$: completion time of task i
- $y_{\mathrm{i}} \in \{0,1\}$: 1 if task i is selected
- $w_{s} \in \{0,1\}$: 1 if all tasks in set S are selected
- $z \ \geq \ 0$: project makespan

Objective: $min\ z$

Subject to:

Time constraints

- Precedence relationships: $t_{\mathrm{j}} \geq t_{\mathrm{i}} + d_{\mathrm{j}} - M(1 \ - \ y_{\mathrm{j}}), \forall(i,j) \ \in \ P$
- Project completion: $z \geq t_{\mathrm{i}}, \forall i \in T$

Selection constraints

- Alternative tasks: $\Sigma_{j\in A_{\mathrm{i}}} y_{\mathrm{j}} = 1, \forall A_{\mathrm{i}} \in A$
- Task execution control: $t_{\mathrm{i}} \leq M * y_{\mathrm{i}}, \forall i \in T$

Interaction constraints

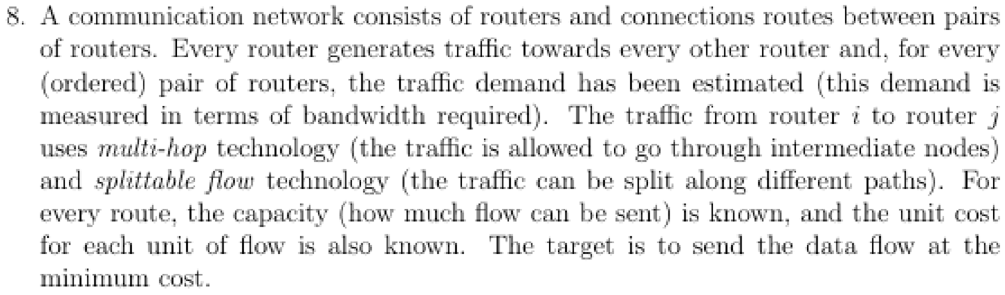
- Detecting task combinations:
    - $\Sigma_{i\in S}\, y_{\mathrm{i}} \ - |S| + 1 \ \leq \ w_{\mathrm{s}}, \quad \forall(S,t,\delta) \in I$
    - $w_{\mathrm{s}} \ \leq \ \left(\frac{1}{|S|}\right) \cdot \Sigma_{(i\in S)}\, y_{\mathrm{i}} \qquad \forall(S,t,\delta) \in I$
- Duration adjustments: $t_{\mathrm{t}} \geq t_{\mathrm{j}} + d_{\mathrm{t}} + \sigma * w_{\mathrm{s}} \quad \forall(S,t,\delta) \ \in \ I, \forall j: (j,t) \ \in \ P$

Example application (boat construction):

- Tasks B/C are alternatives: $y_B + y_C = 1$
- Tasks F/G are alternatives: $y_F + y_G = 1$
- Duration increase for I when C and G selected: $w_{CG} \geq y_C + y_G - 1\ t_I \geq t_F + 4 + 2w_{CG}\ t_I \geq t_G \ + 4 + 2w_{CG}$

This formulation emphasizes the time-based aspects while maintaining the logical requirements of task selection and interaction. It provides a clear structure that can be extended to handle additional practical considerations like resource constraints.

3.13 Other Models: Router Communication Network



8. A communication network consists of routers and connections routes between pairs of routers. Every router generates traffic towards every other router and, for every (ordered) pair of routers, the traffic demand has been estimated (this demand is measured in terms of bandwidth required). The traffic from router i to router j uses *multi-hop* technology (the traffic is allowed to go through intermediate nodes) and *splittable flow* technology (the traffic can be split along different paths). For every route, the capacity (how much flow can be sent) is known, and the unit cost for each unit of flow is also known. The target is to send the data flow at the minimum cost.

Sets

- N: set of routers (nodes)
- A: set of possible connections between routers (arcs)
- K: set of commodities, where each $k \in K$ represents a traffic demand from $o(k)$ to $d(k)$

Parameters

- $r(k)$: traffic demand for commodity $k \in K$
- u_{ij}: capacity of arc $(i, j) \in A$
- c_{ij}: unit cost for flow on arc $(i, j) \in A$
- b^k_i: node balance for commodity k at node i, where:
  - $b_i^k = \begin{cases} -r(k), & if\ i = o(k) \\ +r(k), & if\ i = d(k) \end{cases}$

Variables

- $x^k_{ij} \geq 0$: flow of commodity k on arc $(i, j) \in A$

Objective Function

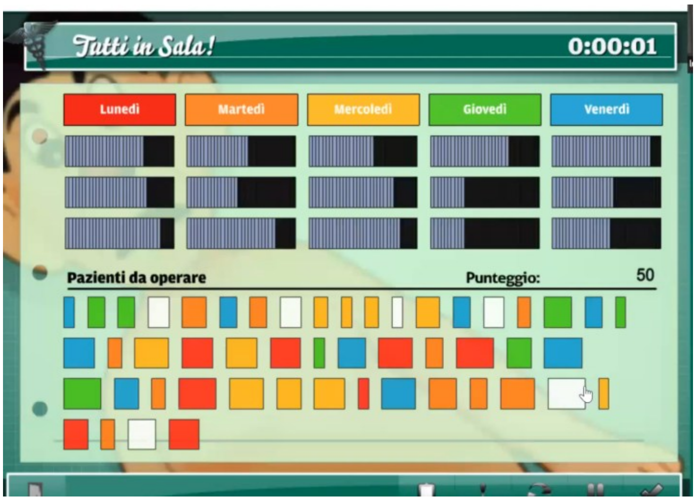
$$min\ \Sigma_{(i,j)\in A, \Sigma k \in K}\ c_{ij} x_{ij}^k$$

Constraints

- Flow Conservation: For each node $i \in N$ and commodity $k \in K$ → $\sum_{i,j \in A} x_{ij}^k - \sum_{(j,i)\in A} x_{ji}^k = b_i^k$
- Capacity Constraints: For each arc $(i, j) \in A$ → $\sum_{k \in K} x_{ij}^k \leq u_{ij}$
- Non-negativity: For each arc $(i, j) \in A$ and commodity $k \in K$ → $x_{ij}^k \geq 0$

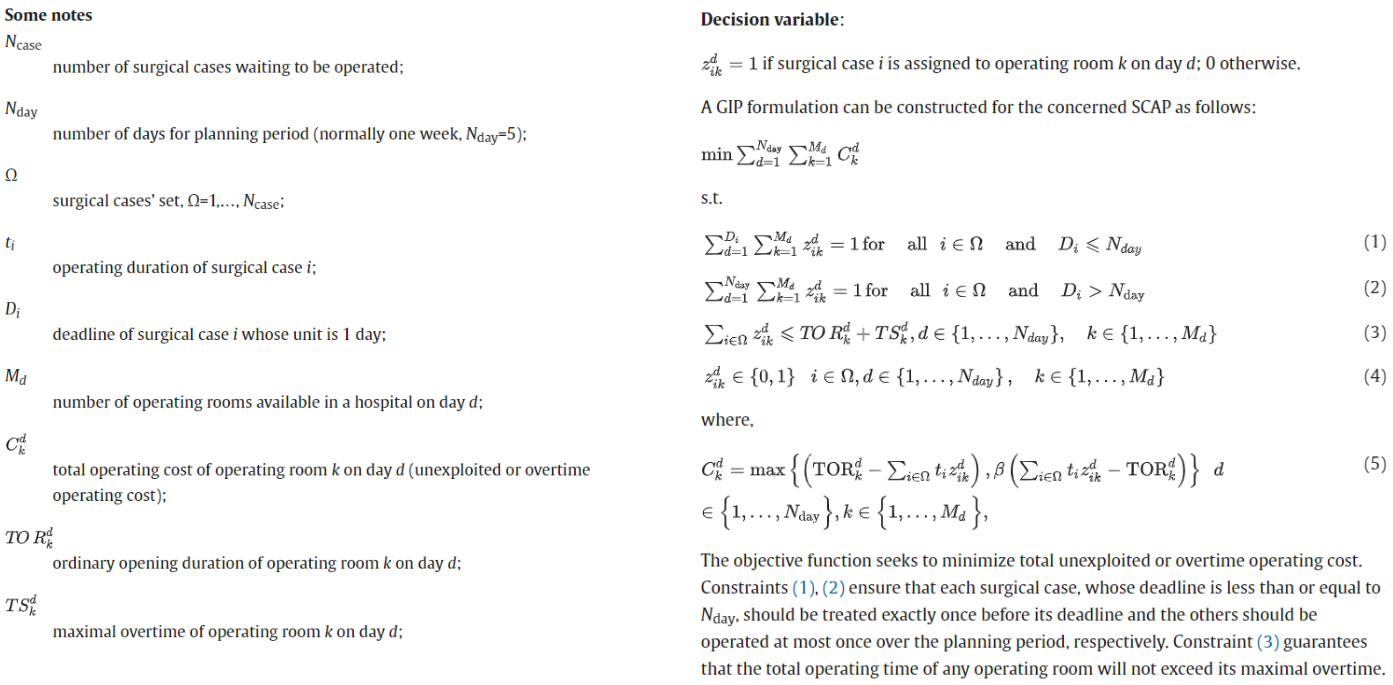
4 METAHEURISTICS (3)

Let’s start by considering a different example: a flash game, used to assign surgical operations, according to the availability of surgery rooms (which are three); each day of the week has grey zones, which is the actual time available for that room and that day. We are tasked to solve this problem, considering the priority is by color (red: most urgent, then orange, yellow, green, blue, white) – doing as much operations as possible.



Let’s try to first think about strategies and then write the actual algorithms – given the optimization problem, we find the most similar problem, creating a mathematical model. For example, we are going to look out for papers, keeping out for the actual content and where their publications come from.

- For example, a good paper to use here would be “Solving surgical cases assignment problem by a branch-and-price approach”, which having a read seems the most similar to this problem
- We also see the paper implementation of the modeling schema (below) thought to be correct and then the professor implementation (next page)



Some notes

N_{case}
number of surgical cases waiting to be operated;

N_{day}
number of days for planning period (normally one week, N_{day}=5);

Ω
surgical cases' set, Ω=1,..., N_{case};

t_i
operating duration of surgical case i;

D_i
deadline of surgical case i whose unit is 1 day;

M_d
number of operating rooms available in a hospital on day d;

C_k^d
total operating cost of operating room k on day d (unexploited or overtime operating cost);

TOR_k^d
ordinary opening duration of operating room k on day d;

TS_k^d
maximal overtime of operating room k on day d;

Decision variable:

$z_{ik}^d = 1$ if surgical case i is assigned to operating room k on day d; 0 otherwise.

A GIP formulation can be constructed for the concerned SCAP as follows:

$$\min \sum_{d=1}^{N_{\text{day}}} \sum_{k=1}^{M_d} C_k^d$$

s.t.

$$\sum_{d=1}^{D_i} \sum_{k=1}^{M_d} z_{ik}^d = 1 \text{ for} \quad \text{all} \quad i \in \Omega \quad \text{and} \quad D_i \leqslant N_{day} \tag{1}$$

$$\sum_{d=1}^{N_{\text{day}}} \sum_{k=1}^{M_d} z_{ik}^d = 1 \text{ for} \quad \text{all} \quad i \in \Omega \quad \text{and} \quad D_i > N_{\text{day}} \tag{2}$$

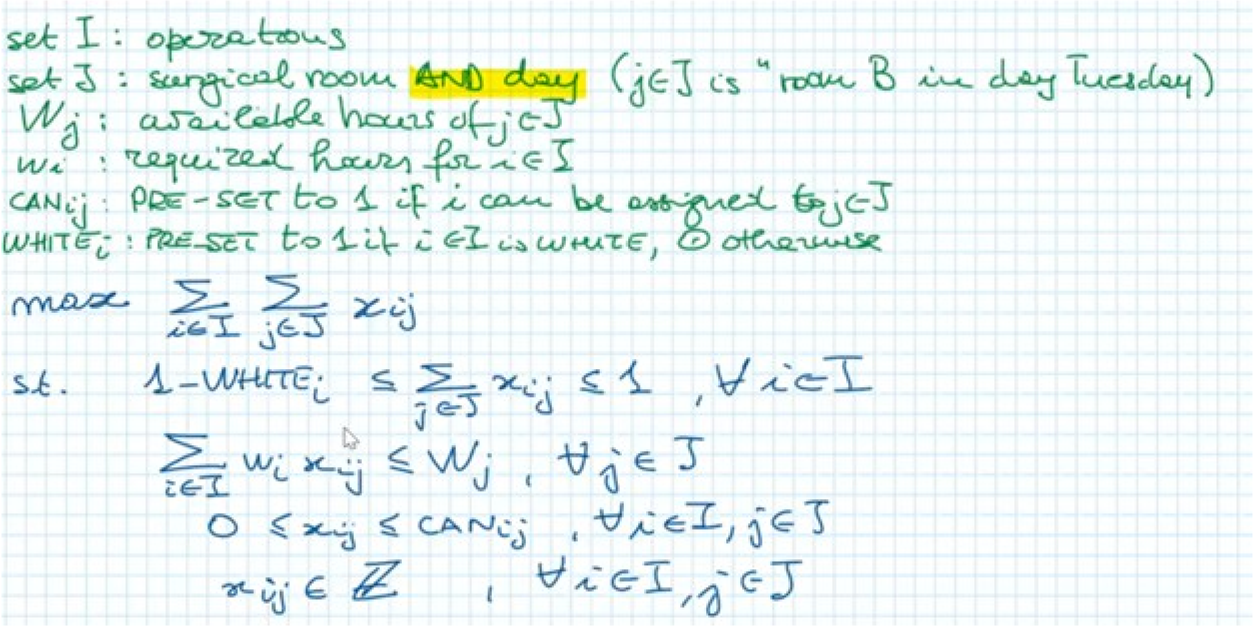
$$\sum_{i\in\Omega} z_{ik}^d \leqslant TOR_k^d + TS_k^d, d \in \{1,\ldots,N_{day}\}, \quad k \in \{1,\ldots,M_d\} \tag{3}$$

$$z_{ik}^d \in \{0,1\} \quad i \in \Omega, d \in \{1,\ldots,N_{day}\}, \quad k \in \{1,\ldots,M_d\} \tag{4}$$

where,

$$C_k^d = \max\left\{\left(\text{TOR}_k^d - \sum_{i\in\Omega} t_i z_{ik}^d\right), \beta\left(\sum_{i\in\Omega} t_i z_{ik}^d - \text{TOR}_k^d\right)\right\} \ d \in \left\{1,\ldots,N_{\text{day}}\right\}, k \in \left\{1,\ldots,M_d\right\}, \tag{5}$$

The objective function seeks to minimize total unexploited or overtime operating cost. Constraints (1), (2) ensure that each surgical case, whose deadline is less than or equal to N_{day}, should be treated exactly once before its deadline and the others should be operated at most once over the planning period, respectively. Constraint (3) guarantees that the total operating time of any operating room will not exceed its maximal overtime.



set I : operations

set J : surgical room AND day ($j \in J$ is "room B in day Tuesday)

W_j : available hours of $j \in J$

w_i : required hours for $i \in I$

CAN_{ij} : PRE-SET to 1 if i can be assigned to $j \in J$

$WHITE_i$: PRE-SET to 1 if $i \in I$ is WHITE, 0 otherwise

$$\max \sum_{i \in I} \sum_{j \in J} x_{ij}$$

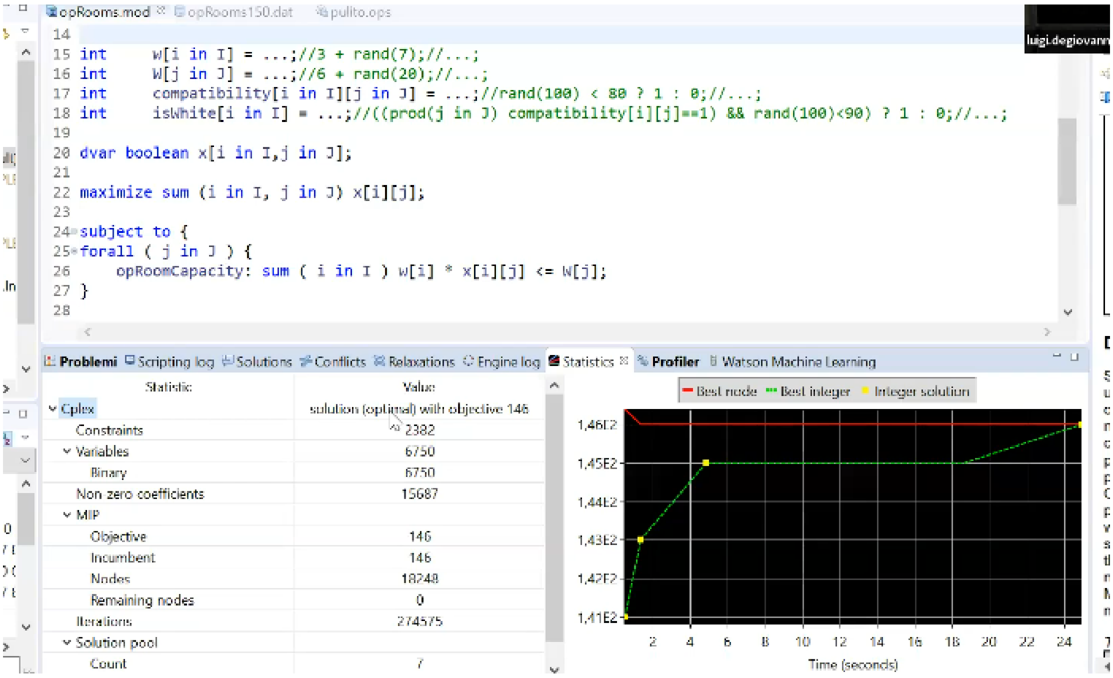
$$\text{s.t.} \quad 1 - WHITE_i \le \sum_{j \in J} x_{ij} \le 1 \quad , \forall i \in I$$

$$\sum_{i \in I} w_i x_{ij} \le W_j \quad , \forall j \in J$$

$$0 \le x_{ij} \le CAN_{ij} \quad , \forall i \in I, j \in J$$

$$x_{ij} \in \mathbb{Z} \quad , \forall i \in I, j \in J$$

Then, from here, a practical implementation has to come out, for example this one in CPLEX:



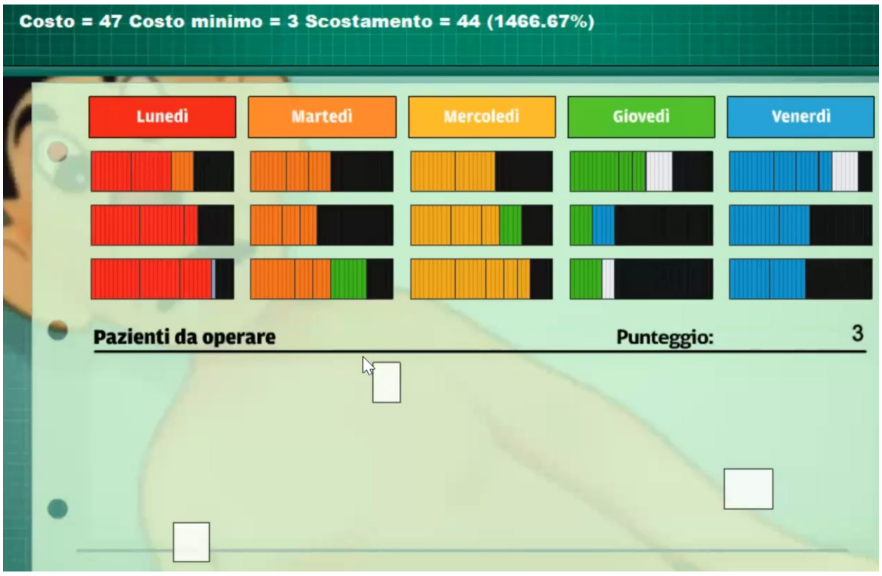
```
int     w[i in I] = ...;//3 + rand(7);//...;
int     W[j in J] = ...;//6 + rand(20);//...;
int     compatibility[i in I][j in J] = ...;//rand(100) < 80 ? 1 : 0;//...;
int     isWhite[i in I] = ...;//((prod(j in J) compatibility[i][j]==1) && rand(100)<90) ? 1 : 0;//...;

dvar boolean x[i in I,j in J];

maximize sum (i in I, j in J) x[i][j];

subject to {
forall ( j in J ) {
    opRoomCapacity: sum ( i in I ) w[i] * x[i][j] <= W[j];
}
```

Statistic	Value
Cplex	solution (optimal) with objective 146
Constraints	2382
Variables	6750
Binary	6750
Non zero coefficients	15687
MIP	
Objective	146
Incumbent	146
Nodes	18248
Remaining nodes	0
Iterations	274575
Solution pool	
Count	7

Coming back to our problem, we are trying to use something which can optimize the decisions, which will become useful later for the actual definition of this chapter:

- For example, let's try to write an algorithm to have priorities assigned according to how they possibly fit, one by one
    - This may not work, but computationally it's fast
- Another idea would be to try to assign the priorities randomly, given it takes basically no time, and then getting all of the solution
    - Randomness can come into play, trying to design some kind of fitting algorithms
    - For example (remember Operating Systems? – worst fit/first fit/best fit – see here)

A solution might be found by applying the best fit algorithm:



Costo = 47 Costo minimo = 3 Scostamento = 44 (1466.67%)

Another idea is perturbing the system a bit in order to obtain better and better solutions, adopting this as a rule to refine and improve progressively what we get as output, once we do not need to find the optimal solution.

4.1 CLASSIFICATION OF METHODS

We have a class of algorithms not trying to guarantee the solution optimality designed to provide "good" solutions, not the "optimal" ones (which required further overhead considering parts in the computation not needed to find the optimal solutions). So:

- Exact methods: devised to provide a *provably optimal* solution
- Heuristic methods: provides "*good*" solution with *no optimality guarantee*

Consider also:

- Sometimes the exact solution is mandatory
- Always try to devise an exact approach first!

When do we use heuristics?

- To formulate an exact model is unpractical or impossible
- Need for just "good" solution using "reasonable" resources"
  - Limited amount of time to provide a solution (running time)
    - E.g., quick scenario evaluation in interactive Decision Support Systems
    - E.g., real time system/NP-Hard problems
  - Limited amount of computational resources (memory, CPUs, hardware)
  - Limited amount of time to develop an effective solution
    - E.g., off-the-shelf solvers cannot effectively solve an available formulation)
  - Limited amount of economic resources to develop a solution algorithm
    - E.g., costs for analysers and developers) or run it (e.g., costs for solver licenses, new hw etc.)

- Just estimates of the problem parameters are available (and we do not want to deal with uncertainty using robust or stochastic optimization…)

Warning: NP-hard problem $\neq$ heuristics!

In some cases it's better to spend resources in order to get to better data or create models in order complex problems – so, mathematical models make their jobs in order to take better solutions.

The following is one (among many) possible classifications for the problems:

- Specific heuristics
  - Exploits unique features of the problem at hand
  - May encode the current "manual" solution, good practice
  - May be "the first reasonable algorithm that come to our mind"

- General heuristic approaches (algorithmic "*templates*")
  - Constructive heuristics
  - Simplified exact procedures
  - Meta-heuristics (algorithmic improvement schemes)
    - They define components and interactions so to find good solutions
  - Approximation algorithms
    - Approximation guarantee to have a specific distance factor from a solution
  - Hyper-heuristics
    - They operate at the boundary between Operations Research and Artificial Intelligence so to find solution linking pieces of other algorithms

Some papers of reference here (to give a better representation):

- C. Blum and A. Roli, "Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison", ACM Computer Surveys 35:3, 2003 (p. 268-308) K. Sorensen – here
- "Metaheuristics – the metaphor exposed", International Transactions in Operational Research (22), 2015 (p. 3-18) – here

Let's go into deeper detail of the general approaches.

4.2 Constructive Heuristics

These heuristics look for a solution from the empty one, going to iteratively add elements to it, trying to limit back-tracking. The criterion by which the element to be added is chosen is called the expansion criterion. The simplest heuristic is the greedy heuristic, which at each step chooses the element that is best at that time. In particular:

- Build a solution only using input data incrementally selecting a subset of alternatives
  - Start from an empty solution, adding iteratively elements with no backtracking
- Expansion criterion (no backtracking)
  - Make local optimal choices at each step, which may not lead to globally optimal solutions

4.3 GREEDY ALGORITHMS

Between constructive heuristics, let's consider first the greedy algorithms:

They adopt local expansion criterion, because the choice is made considering the best for the current state of the solution.

The generic scheme is:

```
1. Initialize solution $S$.
2. For each choice to be made:
    a) Make the best choice for the current context, considering the
       constraints of the problem.
```

One needs a criterion to add the most feasible element at a time, using a greedy (*myopic*) vision given *on what we have at the moment*, using a *strongly local criterion*: things will be added iteratively according to the need, applying sorting rules according to *dispatching rules*.

- They are particularly easy to implement
- Time of computation is reduced, and they are used in blocks for more complex algorithms

In some cases, greedy algorithms exploit an *ordering* of elements (dispatching rule): the elements that define the solution are considered in that order and eventually inserted into the solution.

- Generally, the sorting criteria used involve associating each choice with a "*score*" that indicates the goodness of the move, trying to reward at each iteration the move that appears to be the most promising
- The score information can be computed once and for all at the beginning of execution based on the input data (*pre-sorting*)
- Often, however, the same heuristic algorithm provides better results if the element sorting criterion is dynamically updated to consider the choices made previously; of course, the continuous updating of element scores will result in an increase in the computation time required by the algorithm itself

To try to get different, and possibly better, solutions using the same procedure, one can iterate the algorithm using a different sorting each time, obtained by a *randomization of the dispatching rule*.

- For example, the score could be corrected with a random component, so as to have the possibility of choosing, at each step, not the best element, but a "good enough" element: in this way one could make the algorithm less myopic and save some elements for later steps, when the choices become more critical. Or, at each step, one could consider the random choice among the best n residual elements

Generally, greedy algorithms are of the primal type, that is, they make choices that always respect all constraints (starting from *an empty solution*). There are, however, also dual versions of such algorithms, applied to problems for which it may be difficult to determine a feasible solution: these start from unfeasible solutions and try to construct a feasible solution, making choices aimed at reducing the degree of unfeasibility, trying not to make the value of the solution much worse.

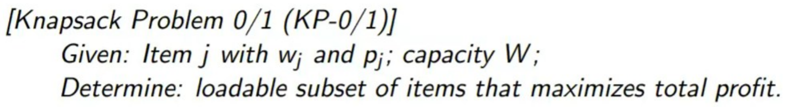
4.4 Exact method algorithms and Simplification of Exact Procedures

Other things to report:

- Exact method algorithms exploit the LP model of the problem and use *continuous relaxation* in order to define score and expand the solution, so to find the best solution at each expansion iteration, when fixed the element variables
    - Generally, the computational time is greater than greedy algorithms, with greater solution quality given they are globally optimal

- Simplification of exact procedures, taking decisions with greedy criteria but using an exact schema, for example after a certain time limit of a number of nodes
    - A variant seen here is the *beam search*

We want to start from "simple" examples, like the knapsack problem. An example of this is the classical knapsack algorithm (KP 0/1), in you need to pack a set of items, with given values and sizes (such as weights or volumes), into a container with a maximum capacity (aka "can we put it or not?").

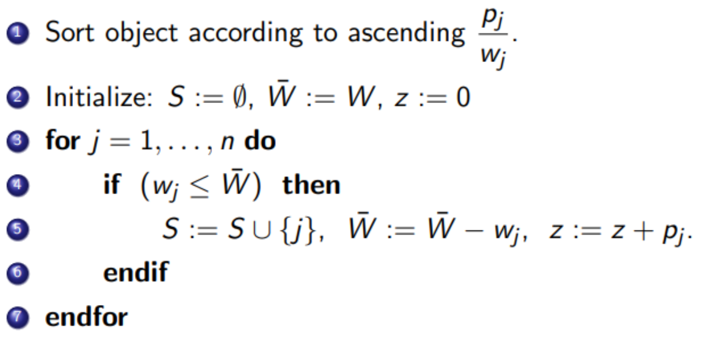


[Knapsack Problem 0/1 (KP-0/1)]
Given: Item j with w_j and p_j; capacity W;
Determine: loadable subset of items that maximizes total profit.

Greedy rules would be:

- Selecting the *smallest weight*
- Selecting the *higher profits*

An algorithm would be this one – privileging higher-profit and lower-weighted values:



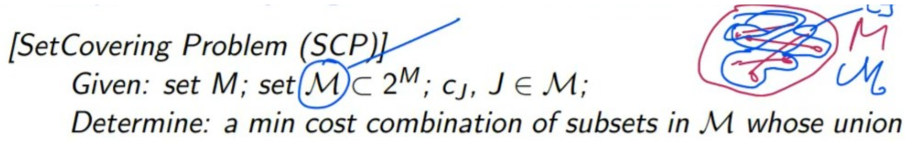
```
Sort object according to ascending $\frac{p_j}{w_j}$.
Initialize: $S := \emptyset$, $\bar{W} := W$, $z := 0$
for $j = 1, \dots, n$ do
    if $(w_j \leq \bar{W})$ then
        $S := S \cup \{j\}$, $\bar{W} := \bar{W} - w_j$, $z := z + p_j$.
    endif
endfor
```

The order is based on a *score assigned to each element*, which is *static* (dispatching rule). The logic of the algorithm is this one:

```
1. Sort items in descending order of $\frac{p_j}{w_j}$
2. Initialize: $S := \emptyset$, $remaining\_capacity := W$
3. For each item $j$ in sorted order:
   if $w_j \leq remaining\_capacity$ then
       $S := S \cup \{j\}$
       $remaining\_capacity := remaining\_capacity - w_j$
```

Note how the expansion criterion is static (the ratio) and can be evaluated once and for all at the beginning of the algorithm.

Now, we are exploring the Set Covering Problem (SCP), which selects a minimum cost combination of subsets whose union equals M (covers all elements in M). Here, the solution is built with a subset at a time.



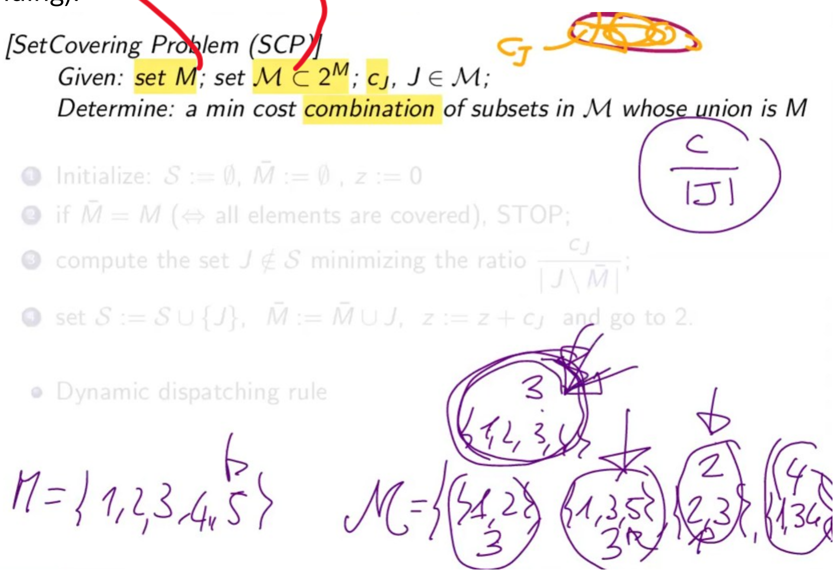
[SetCovering Problem (SCP)]

Given: set M; set $\mathcal{M} \subset 2^M$; c_J, $J \in \mathcal{M}$;

Determine: a min cost combination of subsets in $\mathcal{M}$ whose union is M

In the set covering problem, a subset is "good" if it has low cost and covers many elements (among those yet uncovered). Thus, the basic idea of the greedy algorithm is to compute the score of each subset not yet included in the solution as a function of cost and the number of additional elements covered.

We select subsets between M and calligraphic M (subset) so to select the best order (no matter if ascending or descending):



[SetCovering Problem (SCP)]

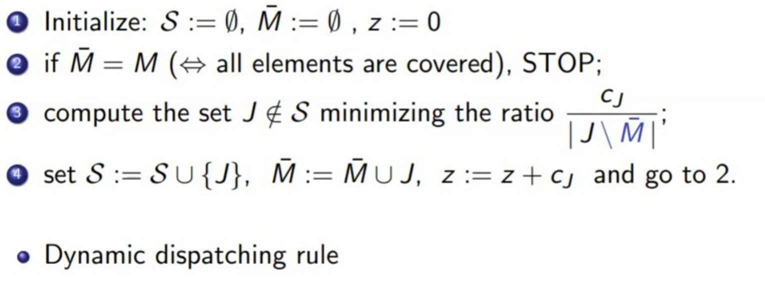
Given: set M; set $\mathcal{M} \subset 2^M$; c_J, $J \in \mathcal{M}$;

Determine: a min cost combination of subsets in $\mathcal{M}$ whose union is M

1. Initialize: $\mathcal{S} := \emptyset$, $\bar{M} := \emptyset$, $z := 0$
2. if $\bar{M} = M$ ($\Leftrightarrow$ all elements are covered), STOP;
3. compute the set $J \notin \mathcal{S}$ minimizing the ratio $\frac{c_J}{|J \setminus \bar{M}|}$;
4. set $\mathcal{S} := \mathcal{S} \cup \{J\}$, $\bar{M} := \bar{M} \cup J$, $z := z + c_J$ and go to 2.

- Dynamic dispatching rule

The score assigned to a subset not only depends on the size, but it strictly depends on the iterations. This is the greedy algorithm for KP 0/1:



1. Initialize: $\mathcal{S} := \emptyset$, $\bar{M} := \emptyset$, $z := 0$
2. if $\bar{M} = M$ ($\Leftrightarrow$ all elements are covered), STOP;
3. compute the set $J \notin \mathcal{S}$ minimizing the ratio $\frac{c_J}{|J \setminus \bar{M}|}$;
4. set $\mathcal{S} := \mathcal{S} \cup \{J\}$, $\bar{M} := \bar{M} \cup J$, $z := z + c_J$ and go to 2.

- Dynamic dispatching rule

Note how, in this case, score evaluation is dynamic, being related not only to the subset under consideration, but also to the choices previously made according to the expansion criterion, which changes the number of additional elements covered.

Question: Is it better to have a static/dynamic dispatching rule?

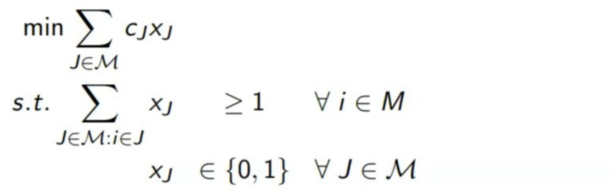
Answer: The choice between static and dynamic dispatching rules is a tradeoff between efficiency and solution quality.

- Static rules are evaluated once upfront, making them faster and simpler but less adaptive
- Dynamic rules update before each choice, providing better solutions by considering the partial solution state, but require more computation time
- Choose static for speed and simplicity, or dynamic for better quality when computational resources allow for this choice
- The best choice ultimately depends on your specific requirements for solution quality versus computational efficiency

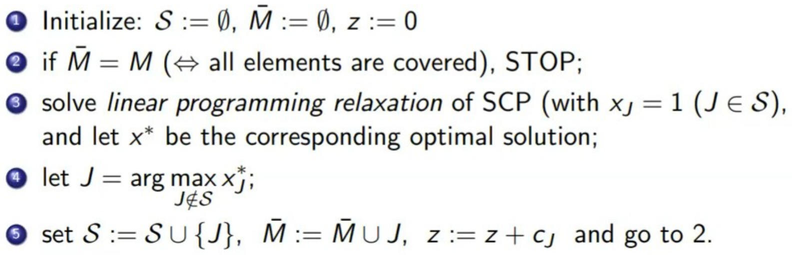
In general:

- Better to use dynamic rules – they are efficient, but they may be costly from an efficiency point of view, since it exploits partial solutions (up to that point)
- We have no current view of what will happen in the future – can we build a "look-ahead" dispatching rule?

One way would be to integrate into heuristics some exact methods. For example, let's consider a greedy algorithm for the SCP through an exact method, where we use continuous relaxation:



$$\begin{aligned} \min \sum_{J \in \mathcal{M}} c_J x_J & \\ s.t. \sum_{J \in \mathcal{M}: i \in J} x_J & \geq 1 \quad \forall\, i \in M \\ x_J & \in \{0,1\} \quad \forall\, J \in \mathcal{M} \end{aligned}$$



```
1 Initialize: $\mathcal{S} := \emptyset$, $\bar{M} := \emptyset$, $z := 0$
2 if $\bar{M} = M$ ($\Leftrightarrow$ all elements are covered), STOP;
3 solve linear programming relaxation of SCP (with $x_J = 1$ ($J \in \mathcal{S}$),
  and let $x^*$ be the corresponding optimal solution;
4 let $J = \arg\max_{J \notin \mathcal{S}} x_J^*$;
5 set $\mathcal{S} := \mathcal{S} \cup \{J\}$, $\bar{M} := \bar{M} \cup J$, $z := z + c_J$ and go to 2.
```

The key idea is about how we use LP relaxation to score/choose the next element:

1. Instead of forcing x_j to be binary (0 or 1), we relax it to be continuous ($0 \leq x_j \leq 1$). This makes the problem much easier and faster to solve

2. When we solve this relaxed LP:

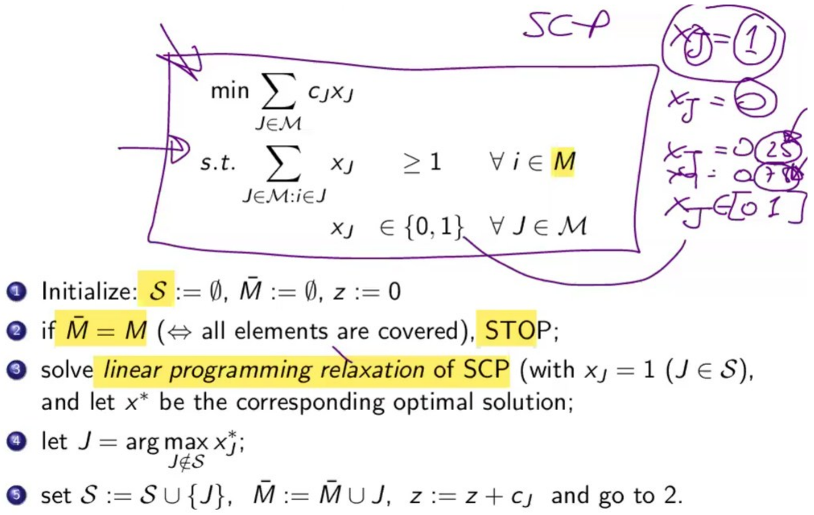
   - Already selected elements (in set S) have $x_j = 1$ fixed
   - All other variables can take any value between 0 and 1
   - This gives us fractional values for unselected elements

3. The scoring strategy:

   - The LP solution x^* gives us fractional values for each unselected element
   - We choose the element with highest fractional value ($argmax\ x_j^*$)

  - This value indicates how "important" the LP solver thinks that element is for an optimal solution

4. The intuition:
   - Elements with higher x^* values are considered more valuable by the LP solver
   - These values consider the global problem structure
   - It's like the LP is giving us a hint about which elements would be good to select next



$$\min \sum_{J \in \mathcal{M}} c_J x_J$$

$$\text{s.t.} \sum_{J \in \mathcal{M} : i \in J} x_J \geq 1 \quad \forall i \in M$$

$$x_J \in \{0,1\} \quad \forall J \in \mathcal{M}$$

1. Initialize: $\mathcal{S} := \emptyset$, $\bar{M} := \emptyset$, $z := 0$
2. if $\bar{M} = M$ ($\Leftrightarrow$ all elements are covered), STOP;
3. solve *linear programming relaxation* of SCP (with $x_J = 1$ ($J \in \mathcal{S}$), and let x^* be the corresponding optimal solution;
4. let $J = \arg\max_{J \notin \mathcal{S}} x^*_J$;
5. set $\mathcal{S} := \mathcal{S} \cup \{J\}$, $\bar{M} := \bar{M} \cup J$, $z := z + c_J$ and go to 2.

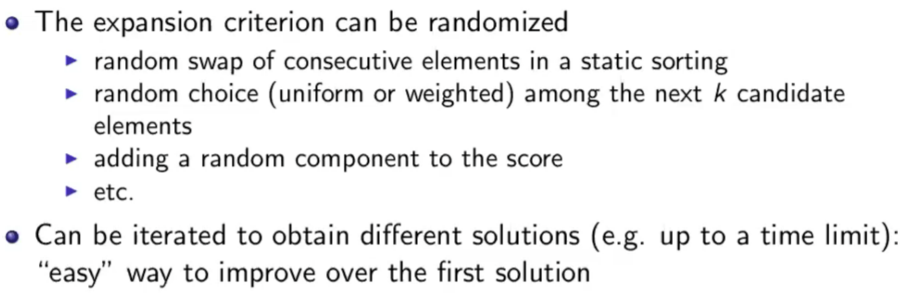
Even with simple constructive heuristics, understanding sorting and everything is important in fact. This last algorithm, most likely and in general instance conditions, works better.

An idea would be integrating exact solution methods inside of heuristics, specifically:

- Expansion criterion based on solving a sub-problem to optimality (once or at each expansion)
- Example: best (locally optimal!) element to add by MILP
- Example: locally good element to add by LP relaxation of MILP
- Normally longer running times but better final solution
- "Less greedy": solving the sub-problem involves all (remaining) decisions variables (global optimality)

Remark: having a mathematical model is useful, even if the model does not directly solve the problem.

We talk about <u>random constructive heuristics</u> algorithms:



- The expansion criterion can be randomized
  - random swap of consecutive elements in a static sorting
  - random choice (uniform or weighted) among the next k candidate elements
  - adding a random component to the score
  - etc.
- Can be iterated to obtain different solutions (e.g. up to a time limit): "easy" way to improve over the first solution

The randomic choice is not totally random but may be useful in order to make the computation faster.

An idea can be to *simplify exact procedures*, for example:

- Run CPLEX on a MILP model for a limited amount of time
- Simplify an enumeration scheme
    - Select only a limited subset of alternatives, e.g., Beam Search

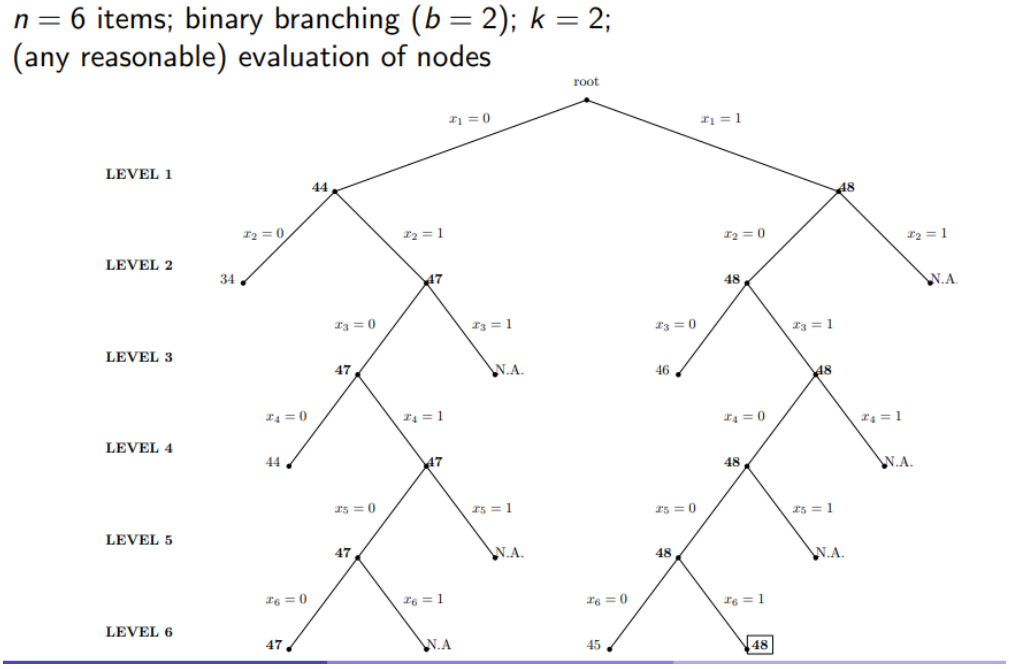
A constructive simplification might be the following one, considering a search tree for the knapsack problem, considering 6 items, doing binary branching ($b = 2$): at each node, we branch by setting a variable to 0 or 1. This is a way to implement a *bruteforce* approach.

The tree exploration works level by level:

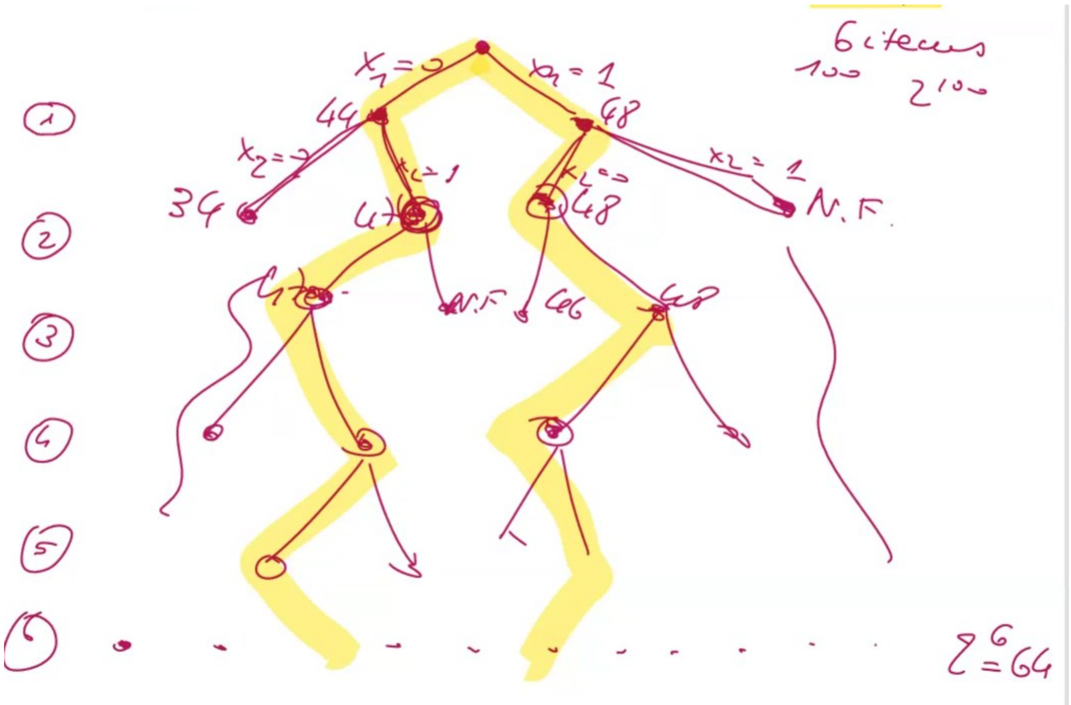
1. At Level 1: branches on x_1 $(0 - 1)$
2. At Level 2: branches on x_2 $(0 - 1)$
3. Each subsequent level fixes another variable

At each level, only the $k = 2$ best nodes are kept for further exploration, based on their evaluation values. This reduces the search space compared to full enumeration while potentially maintaining satisfactory solution quality.

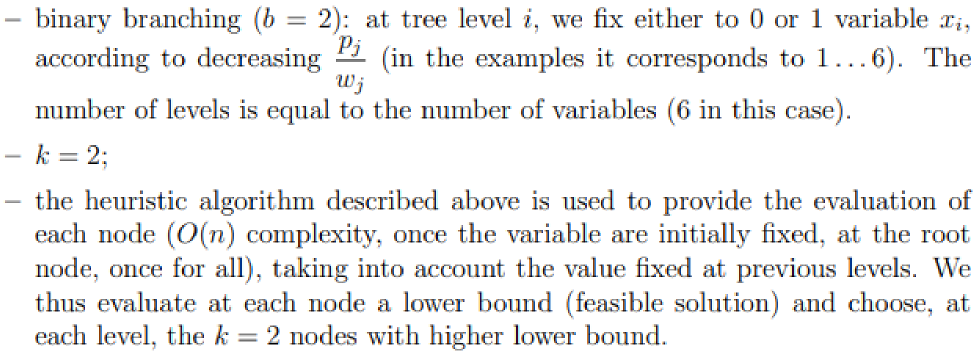
The following is the complete example (basically, it's a bruteforce approach with a search tree; "put Yes or put No", up to the last level) – with a few items it's tractable, otherwise it can become exponential on running time to explore all of the leaves.


$n = 6$ items; binary branching $(b = 2)$; $k = 2$;
(any reasonable) evaluation of nodes

A heuristic will try to explore *a part* of this tree, since otherwise it would be an exponential explosion; like a *beam*, we would like to explore a part of this tree, exploring at a time the best nodes, evaluating the nodes and selecting only one part, so to stabilize the nodes chosen (constants × size of the problem).

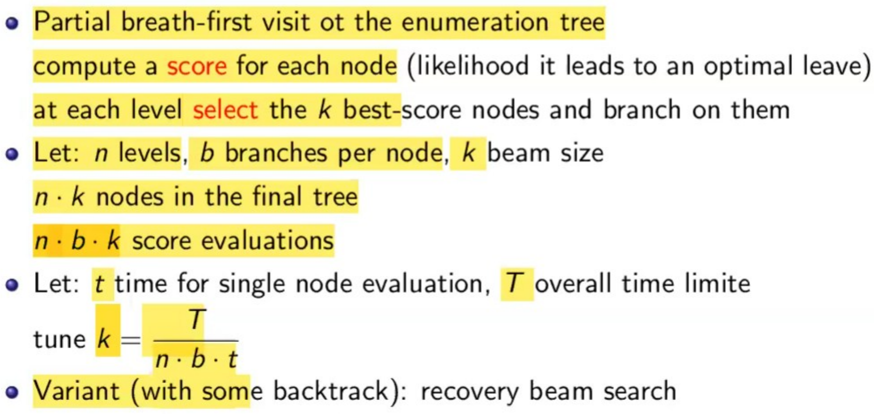


At each level, the heuristic might select nodes to be developed or not – the boxed value 48 at the bottom right indicates a feasible solution was found on that branch. Other branches were either pruned (N.A.) or had worse evaluations.



- binary branching ($b = 2$): at tree level i, we fix either to 0 or 1 variable x_i, according to decreasing $\frac{p_j}{w_j}$ (in the examples it corresponds to $1 \ldots 6$). The number of levels is equal to the number of variables (6 in this case).
- $k = 2$;
- the heuristic algorithm described above is used to provide the evaluation of each node ($O(n)$ complexity, once the variable are initially fixed, at the root node, once for all), taking into account the value fixed at previous levels. We thus evaluate at each node a lower bound (feasible solution) and choose, at each level, the $k = 2$ nodes with higher lower bound.

With this algorithm, we consider each position and examine the N sequences so far, so to consider all of the probabilities and combinations of the positions – this is known to be a fast algorithm, since it does a systematic expansion of the most promising nodes within a constrained set.



- Partial breath-first visit ot the enumeration tree
  compute a score for each node (likelihood it leads to an optimal leave)
  at each level select the k best-score nodes and branch on them
- Let: n levels, b branches per node, k beam size
  $n \cdot k$ nodes in the final tree
  $n \cdot b \cdot k$ score evaluations
- Let: t time for single node evaluation, T overall time limite
  tune $k = \frac{T}{n \cdot b \cdot t}$
- Variant (with some backtrack): recovery beam search

This variant with higher guarantees of finding good solutions is known as beam search, consisting of simplifying the [branch-and-bound algorithm](#) through a partial breadth-first visit to the tree. For each node, all b potential children nodes are generated but, for each level, at most k child nodes are developed (i.e., branched on) (where k is a parameter to be calibrated according to the computational time available).

- The choice of the k subproblems to be developed is usually made by associating with each potential child node a prior assessment of the goodness of the solutions contained in the corresponding subtree (e.g., but not limited to, the bound, or a quick assessment of a possible solution of the subtree through a greedy completion procedure, or their weighted sum etc.) and taking the k most promising child nodes of the current level current one

- In this way combinatorial explosion is avoided: at each level k nodes will be kept (at most) and the branch-and-bound tree is reduced to a bundle (=beam) of $n - k$ nodes (if n is the number of levels in the tree) thus guaranteeing polynomial complexity, if polynomial is the procedure for evaluating each node. Height of the tree remains infact polynomial

Note that if n is the number of levels in the tree (related to the size of the problem), b is the number of child nodes of the generic node and k the size of the bundle will be evaluated $O(n * k * b)$ nodes. Eventually there will be at most k leaf nodes corresponding to solutions from which the best one is chosen.

- From the above formula, fixing k, the number of nodes is known, and being able to estimate the time required to perform the evaluation of a node, one can predetermine the time total execution time of a beam search
- Or, if the maximum amount of time available is known, it is possible to size k to the maximum value that will allow the search for all nodes to be completed in the predetermined time

In its basic form, the *beam search technique does not involve backtracking* (it is not possible to backtrack once choices of nodes to be developed have been made): for this reason, it has been included in this section on constructive heuristics, although the fact that the various components have to be defined specifically for each problem within a well-defined framework makes this technique comparable to a metaheuristic.

- Indeed, the boundary (as we have already mentioned) is blurred and, for beam search, there are variants, such as recovery beam search where one handles backtracking, allowing, if one realizes that some subtree at a certain level is not “promising,” to go back to an earlier level

With this method, nodes which are computed are known in advance.

The example shows a knapsack problem with:

- $n = 6$ variables (tree levels)
- $b = 2$ branches per node (binary variables)
- $k = 2$ best nodes kept per level
- Node scores are based on relaxation values
- Infeasible/dominated solutions marked as N.A.
- Solution found at bottom-right with value 48

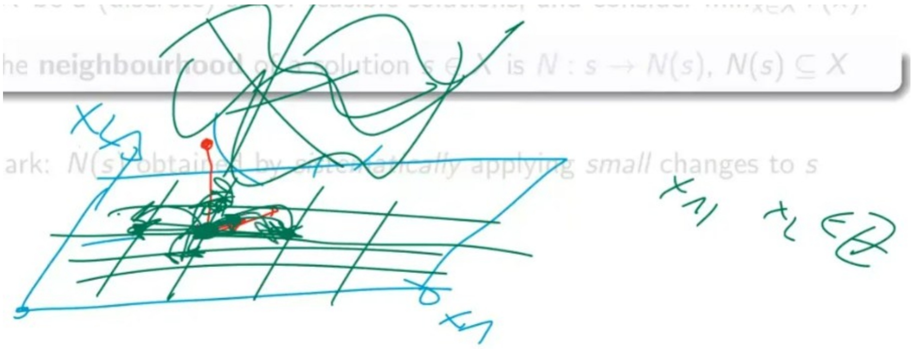
The variant "*recovery beam search*" adds limited backtracking capability to potentially improve solution quality while maintaining reasonable computational effort.

4.5 Neighborhood and Local Search

How to improve a solution?

- In *continuous* optimization: use *gradient* (to see the idea, look here)
  - Done to compute the exact direction of improvement
  - Following smooth path to the optimum, using derivatives (directions)
  - This is not directly applicable because of the nature of the objective function
- In *combinatorial* optimization: *explore nearby* solutions ("neighborhood")
  - Moves through discrete "jumps", checking nearby solutions

This image illustrates the concept of gradient vs. neighborhood search in optimization:


he neighbourhood of a solution s ∈ X is N : s → N(s), N(s) ⊆ X
ark: N(s) obtained by systematically applying small changes to s
x₂
x_N
x_i ∈ ℤ
x₁

The grid lines represent a discrete solution space X, while the curved lines show continuous paths that would be followed by gradient-based methods. Let's explain the key elements of this visual example:

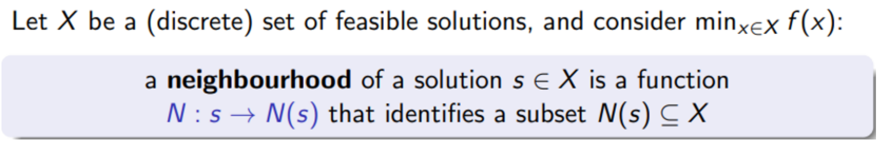
1. Red dot: Current solution s
2. Grid intersections: Feasible discrete solutions
3. Curved green lines: Continuous optimization paths (gradient)
4. Blue grid: Discrete solution space where neighborhood search operates

The image emphasizes that while continuous optimization can follow smooth paths (green curves), combinatorial optimization must "jump" between discrete points on the grid through neighborhood moves.

- The *neighborhood* $N(s)$ of the red point would be nearby grid intersections, typically those reachable through single moves in the discrete space, rather than following the continuous curves
- This visualizes why gradient methods don't work for combinatorial problems – we must explore discrete neighbors rather than follow continuous improvement directions

The basic idea of neighborhood search is to define an initial solution (current solution) and try to improve it by exploring an (appropriately defined) neighborhood of this solution. If the optimization on the current solution's surroundings produces an improving solution the procedure is repeated starting, as the current solution, from the newly determined solution.

Let’s give a more formal definition:



Let X be a (discrete) set of feasible solutions, and consider $\min_{x \in X} f(x)$:

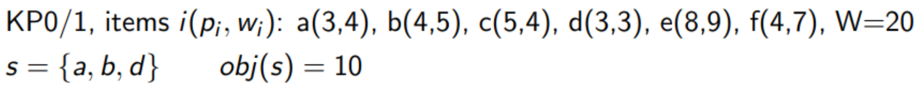
a **neighbourhood** of a solution $s \in X$ is a function $N : s \to N(s)$ that identifies a subset $N(s) \subseteq X$

Remark:

- $N(s)$ obtained by systematically applying slight changes to s
- A change is also called *move*: we from from s to a neighbor solution
- The move also identifies the applied rule, i.e., the neighborhood function

Consider the problem applied here – given:

- Set of items i with profits p_i and weights w_i
- Knapsack capacity (weight) $W = 20$
- Items: $a(3{,}4), b(4{,}5), c(5{,}4), d(3{,}3), e(8{,}9), f(4{,}7)$ where (p_i, w_i) represents (profit, weight)



KP0/1, items $i(p_i, w_i)$: a(3,4), b(4,5), c(5,4), d(3,3), e(8,9), f(4,7), W=20

$s = \{a, b, d\}$ $obj(s) = 10$

First Neighborhood $N(s)$

Current solution $s = \{a, b, d\}$ with $obj(s) = 10$

$$N(s) = \{t \subseteq X \mid t = s + i, i \in X \setminus s \; or \; t = s - i, i \in s\}$$

where:

- X is set of all feasible solutions
- $s + i$ means adding item i to solution s
- $s - i$ means removing item i from solution s

Neighbors:

$t_1 = \{b, d\} \quad obj(t_1) = 7 \quad [removed\ a]$

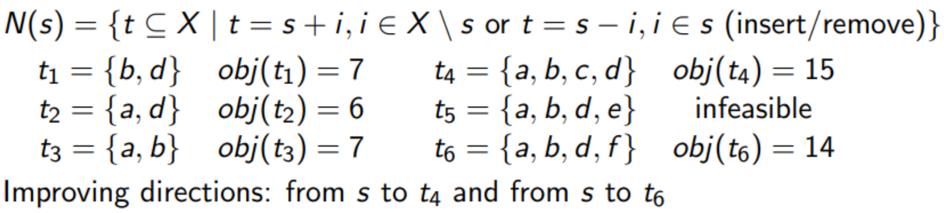
$t_2 = \{a, d\} \quad obj(t_2) = 6 \quad [removed\ b]$

$t_3 = \{a, b\} \quad obj(t_3) = 7 \quad [removed\ d]$

$t_4 = \{a, b, c, d\} \; obj(t_4) = 15 \quad [added\ c]$

$t_5 = \{a, b, d, e\} \; infeasible \quad [added\ e]$

$t_6 = \{a, b, d, f\} \; obj(t_6) = 14 \quad [added\ f]$



$N(s) = \{t \subseteq X \mid t = s + i, i \in X \setminus s$ or $t = s - i, i \in s$ (insert/remove)$\}$

$t_1 = \{b, d\}$	$obj(t_1) = 7$	$t_4 = \{a, b, c, d\}$	$obj(t_4) = 15$
$t_2 = \{a, d\}$	$obj(t_2) = 6$	$t_5 = \{a, b, d, e\}$	infeasible
$t_3 = \{a, b\}$	$obj(t_3) = 7$	$t_6 = \{a, b, d, f\}$	$obj(t_6) = 14$

Improving directions: from s to t_4 and from s to t_6

Second Neighborhood $N'(s)$:

$$N'(s) = \{t \subseteq X \mid t = s + i - j, i \in X, j \in s\}$$

where $s + i - j$ means swapping item j in s with item i not in s

Neighbors:

$t_1 \;=\; \{c, b, d\} \quad obj(t_1) \;=\; 12 \quad [swapped\ a \rightarrow c]$

$t_2 \;=\; \{e, b, d\} \quad obj(t_2) \;=\; 15 \quad [swapped\ a \rightarrow e]$

$t_3 \;=\; \{f, b, d\} \quad obj(t_3) \;=\; 11 \quad [swapped\ a \rightarrow f]$

$t_4 \;=\; \{a, c, d\} \quad obj(_4) \;=\; 11 \quad [swapped\ b \rightarrow c]$

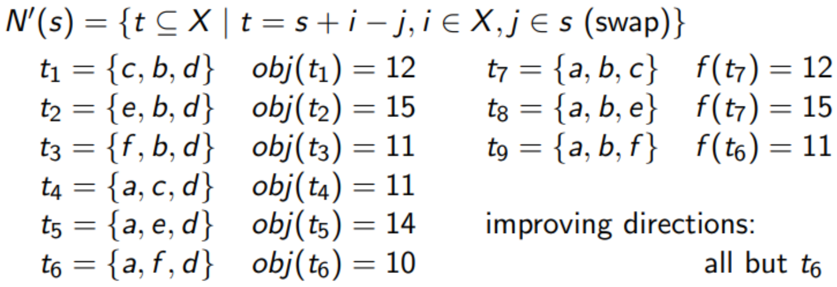
$t_5 \;=\; \{a, e, d\} \quad obj(t_5) \;=\; 14 \quad [swapped\ b \rightarrow e]$

$t_6 \;=\; \{a, f, d\} \quad obj(t_6) \;=\; 10 \quad [swapped\ b \rightarrow f]$

$t_7 \;=\; \{a, b, c\} \quad obj(t_7) \;=\; 12 \quad [swapped\ d \rightarrow c]$

$t_8 \;=\; \{a, b, e\} \quad obj(t_8) \;=\; 15 \quad [swapped\ d \rightarrow e]$

$t_9 \;=\; \{a, b, f\} \quad obj(t_9) \;=\; 11 \quad [swapped\ d \rightarrow f]$



$N'(s) = \{t \subseteq X \mid t = s + i - j, i \in X, j \in s \text{ (swap)}\}$

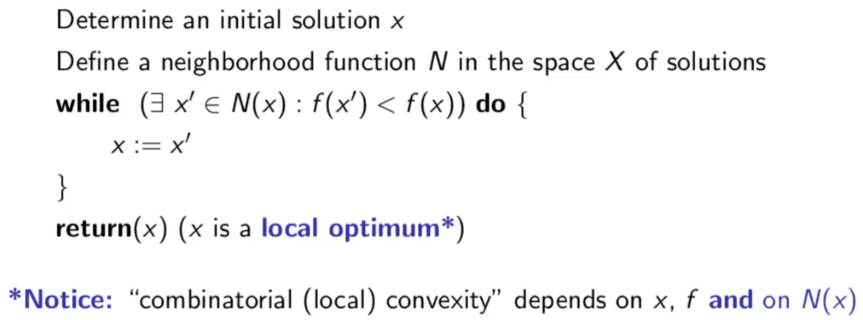
$t_1 = \{c, b, d\}$	$obj(t_1) = 12$	$t_7 = \{a, b, c\}$	$f(t_7) = 12$
$t_2 = \{e, b, d\}$	$obj(t_2) = 15$	$t_8 = \{a, b, e\}$	$f(t_7) = 15$
$t_3 = \{f, b, d\}$	$obj(t_3) = 11$	$t_9 = \{a, b, f\}$	$f(t_6) = 11$
$t_4 = \{a, c, d\}$	$obj(t_4) = 11$		
$t_5 = \{a, e, d\}$	$obj(t_5) = 14$	improving directions:	
$t_6 = \{a, f, d\}$	$obj(t_6) = 10$		all but t_6

The basic idea of the meta-heuristic known as neighborhood search is the following: start from an initial solution (current solution) x and try to improve it by exploring a suitable neighbourhood of x. If the neighbourhood contains a solution better than x, then iterate the process considering x' as the new current solution.

The simplest version of the neighbourhood search is the local search (LS): the algorithm stops when the neighbourhood of the current solution contains no improving solutions, so that the current solution is a *local optimum*.

4.6 LOCAL SEARCH SCHEME

The *basic Local Search (LS) scheme* is the following:



```
Determine an initial solution $x$
Define a neighborhood function $N$ in the space $X$ of solutions
while $(\exists\, x' \in N(x) : f(x') < f(x))$ do {
    $x := x'$
}
return($x$) ($x$ is a local optimum*)
```

*Notice: "combinatorial (local) convexity" depends on x, f and on $N(x)$

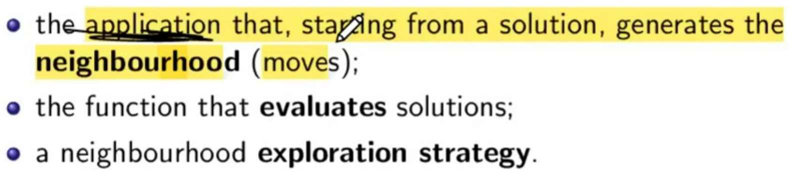
The scheme is extremely simple and general. To obtain an algorithm for a specific problem the following components should be specialized:

- A method to find an *initial solution*
- A *solution representation* (model/representation), which is the base for the following elements (which is a formulation to be used inside of the implementation)
- The application that, starting from a solution, generates the *neighbourhood* (moves)
- The function that *evaluates* solutions
- A neighbourhood *exploration strategy*

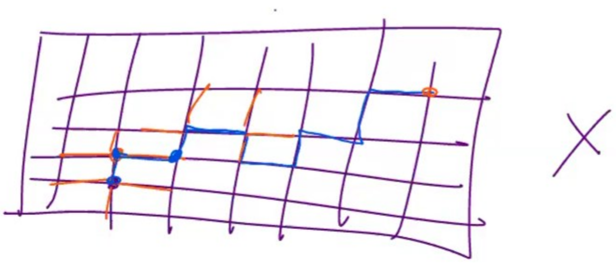
The "hidden components" in Local Search (LS) are crucial because they significantly impact the algorithm's effectiveness. Each is important because of many factors:

- Quality affects the starting point of search
- Can influence final solution quality
- Trade-off between quick random start vs. good heuristic solution
- Affects diversification (multiple random starts) vs. intensification (good starts)

This is visible by the following:



- the application that, starting from a solution, generates the **neighbourhood** (moves);
- the function that **evaluates** solutions;
- a neighbourhood **exploration strategy**.



4.7 Initial Solution and Solution Representation

There are different approaches to generating initial solutions for Local Search algorithms and their implications:

- Random choice of a solution
- From current practice – using existing solutions from real-world
- (Fast) heuristics
- Randomized heuristics
- No theoretical preference: better initial solutions may lead to worst local optima
    - Starting with a high-quality solution doesn't guarantee finding the global optimum
    - Could get stuck in local optima near the starting point
    - May miss better solutions in other regions of the solution space
- Random or randomized + multistart
    - Better exploration of solution space – different starting solutions
    - Increased chance of finding global optimum
    - Helps avoid getting stuck in specific regions

The easiest way to get a starting solution is *to generate one randomly*. Or if the problem is derived from a *real case*, there may be a currently used solution that can be *used as a starting point*.

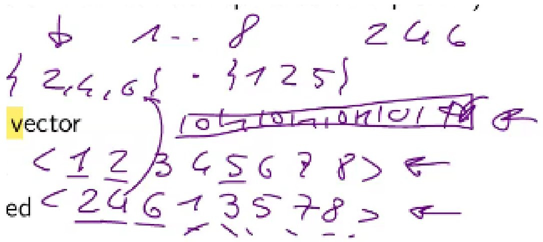
Another idea is to start with a good solution obtained by fast heuristics.

- There is no theoretically better choice anyway, so there is a trade-off between the time invested in finding a good starting solution or the time invested in finding the optimum.
- Of course, there is always the risk of getting stuck in a local maximum
- If a randomly generated starting solution is chosen, it is possible to repeat the local search several times in order to find multiple local optimum solutions, in the hope that one of them will be better than the others or a global optimum solution

The solution representation encodes the features of the solutions as to provide the "concrete" support for the operations that allow us exploring the solution space. As we will see, different solution representations may be adopted for the same problem, which influences the design of the remaining LS elements.

For the Knapsack Problem (KP 0/1), possible representations include:

1. List of loaded items
2. Characteristic binary vector indicating selected items
3. Ordered sequence of items



Decoding may be needed to translate the representation into an actual solution. For KP 0/1:

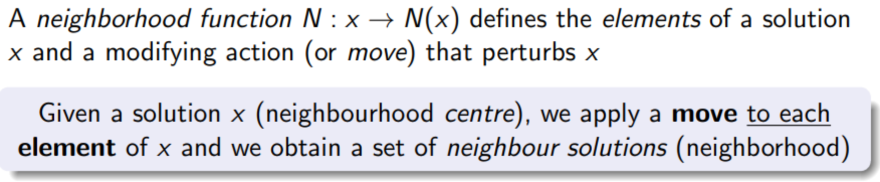
- List and vector representations have immediate decoding
- For ordered sequences, items are loaded in the given order until the knapsack is full

The way in which the solution is represented is important because it affects the definition of the neighborhood and the shape of the search space.

- By representing the solution we do not mean using a vector rather than a list, but at a more abstract level
- For example, for the backpack problem, it is possible to represent the solution with a binary vector, where a 1 indicates that the item was placed in the backpack, or as a stack of objects representing the various items
- Depending on the encoding, *decoding* may also be necessary to obtain a result that people can understand (e.g., pop/push operations for K/P 0-1)

4.8 Neighborhood Representation: Starting Solution and Representation

The neighborhood function N defines how to perturb a solution x to generate its set of neighbor solutions $N(x)$.



A *neighborhood function* $N : x \to N(x)$ defines the *elements* of a solution x and a modifying action (or *move*) that perturbs x

Given a solution x (neighbourhood *centre*), we apply a **move** to each **element** of x and we obtain a set of *neighbour solutions* (neighborhood)

For example, adding one item in the backpack or removing another. Typically, these are slight changes, so the size of the neighborhood is small to make it quick to evaluate.

- However, there is a trade-off, because as the *neighborhood size* increases the probability of converging to a local optimum decreases, but the complexity/time for generation/evaluation increases
- Therefore, the *complexity* of the evaluation algorithm must also be considered, because it must be run on all solutions in the neighborhood

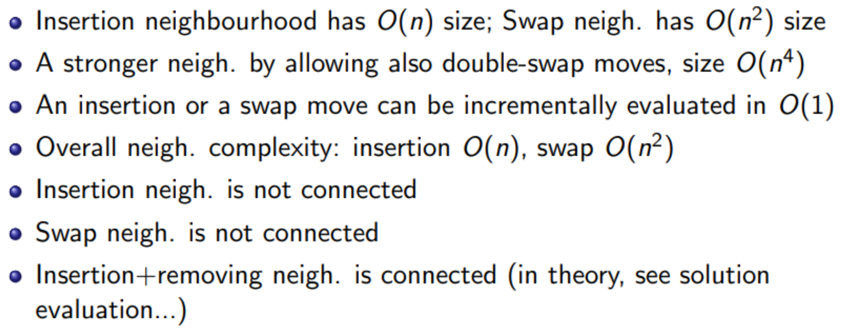
There are key properties for the neighborhood to be considered in its design:

- *Neighbourhood size*: *number* of neighbor solutions
- *Evaluation complexity*: time to evaluate one neighbor (incremental is faster)
- *Neighbourhood strength*: reach good local optima (may depend on convexity of space)
    - Good chance of producing excellent local solutions, this is because if you have a strong neighborhood it is easier to achieve a good solution even from a bad one
- *Connection*: any solution reachable by a move sequence
    - Given two feasible solutions it is always possible to find a sequence of moves that allows you to move from one solution to the other – done at priori

For KP/0-1, the addition neighborhood is clearly disjointed (reachable solutions are only those that contain the objects in the starting solution).

The second neighborhood is also disjointed (only solutions with as many objects in the knapsack as the starting solution are reachable). A connected neighborhood would be one that includes two types of moves: addition of an object in the knapsack and elimination of an object from the knapsack (note, however, that, in a local search context, elimination moves would never be selected because they are non-enhancing).

The following one is an example for the KP 0-1:



- Insertion neighbourhood has $O(n)$ size; Swap neigh. has $O(n^2)$ size
- A stronger neigh. by allowing also double-swap moves, size $O(n^4)$
- An insertion or a swap move can be incrementally evaluated in $O(1)$
- Overall neigh. complexity: insertion $O(n)$, swap $O(n^2)$
- Insertion neigh. is not connected
- Swap neigh. is not connected
- Insertion+removing neigh. is connected (in theory, see solution evaluation...)

For the KP problem we can use three representations:

- A *list* with elements included in the backpack
- A *Boolean characteristic vector* with as many values as there are total elements (conventionally n). A 1 indicates that the element is in the backpack (0 otherwise)
- A *stack* with the ordered sequence of element elements

Considering the addition of an element and the swap as moves, we do not get connected neighborhoods, because it may be that to move from one solution to a better one we need to make a not-so-great move.

But there is also a practical problem, implementing insertion/swap in a list or vector is simple, but on the stack (or ordered list), the implementation may be more complex and in some cases may not be possible.

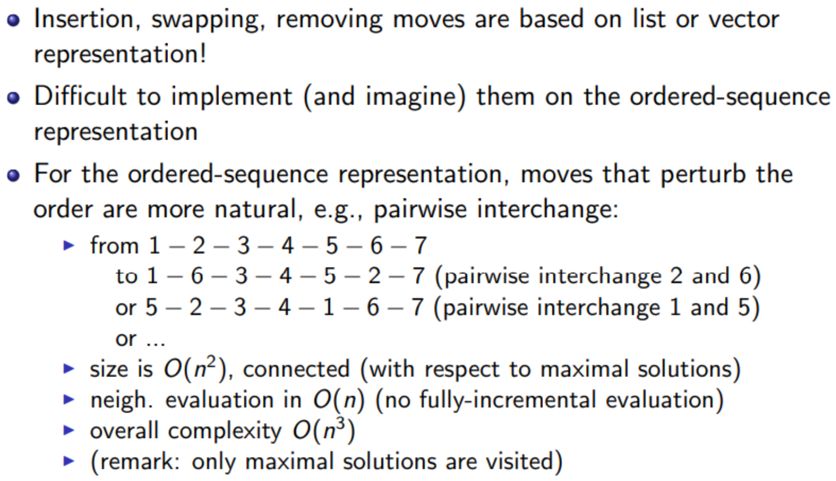
The way the neighbourhood is devised and designed strongly depends on the way solutions are represented – solution representation is important!

- All the moves we have previously described for KP-0/1 comes from the first representation (insert or remove are list operations). The same moves can be easily adapted to the second representation (characteristic vector): flipping a 0 to 1 (insertion), flipping a 1 to 0 (removing), swapping a 0 and a 1 (pairwise swap)

- The third representation (ordered item list) yield different move definitions, since a neighbour solution is given by a different order. A move may be swapping the position of two items in the sequence: for example, if $n = 7$ and the centre solution is $1-2-3-4-5-6-7$, neighbour solutions are $1 - 6 - 3 - 4 - 5 - 2 - 7$ (swap 2 and 6) or $5-2-3-4-1-6-7$ (swap 1 and 5)

- The size of this neighbourhood is $O(n^2)$ and it is connected (with respect to maximal solutions, that is, the ones where no further items can be included preserving feasibility)

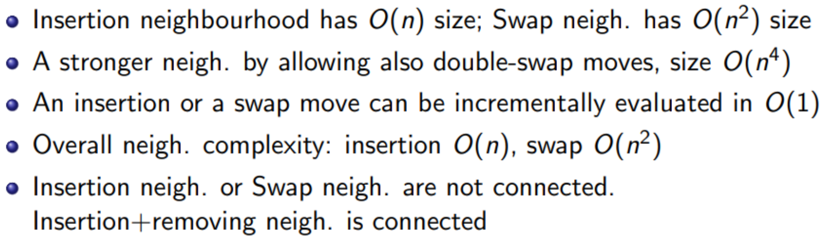
That is summarized by the following image:



- Insertion, swapping, removing moves are based on list or vector representation!
- Difficult to implement (and imagine) them on the ordered-sequence representation
- For the ordered-sequence representation, moves that perturb the order are more natural, e.g., pairwise interchange:
  - from $1-2-3-4-5-6-7$
    to $1-6-3-4-5-2-7$ (pairwise interchange 2 and 6)
    or $5-2-3-4-1-6-7$ (pairwise interchange 1 and 5)
    or ...
  - size is $O(n^2)$, connected (with respect to maximal solutions)
  - neigh. evaluation in $O(n)$ (no fully-incremental evaluation)
  - overall complexity $O(n^3)$
  - (remark: only maximal solutions are visited)

The *insertion* neighborhood is *not connected*, as it only reaches solutions with more items than the center. The *swap* neighborhood is also *not connected*, as the number of items cannot change.

A *connected neighborhood* is *insertion+removing*, as any subset can be reached by adding/removing items. However, removing moves would not improve the objective in a straightforward LS implementation (see solution evaluation).



- Insertion neighbourhood has $O(n)$ size; Swap neigh. has $O(n^2)$ size
- A stronger neigh. by allowing also double-swap moves, size $O(n^4)$
- An insertion or a swap move can be incrementally evaluated in $O(1)$
- Overall neigh. complexity: insertion $O(n)$, swap $O(n^2)$
- Insertion neigh. or Swap neigh. are not connected.
  Insertion+removing neigh. is connected

4.9 COMPLEXITY AND EVALUATION FUNCTION OF SOLUTIONS

Another important aspect to consider in the design of the neighborhood is related to the efficiency of its exploration, that is, the evaluation of the solutions that are part of it.

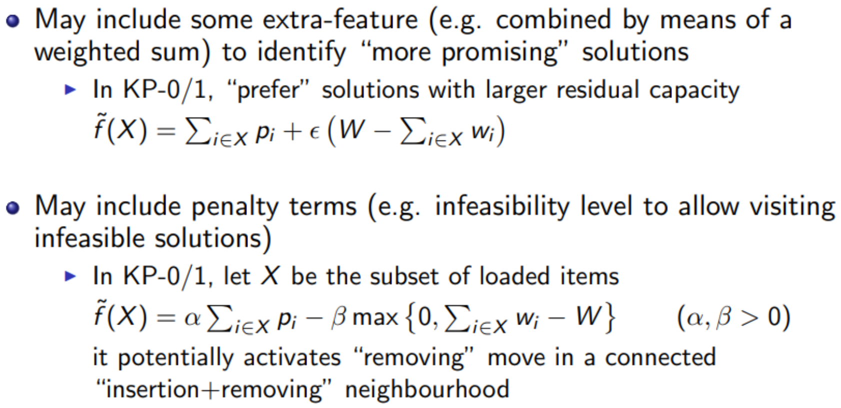
- In fact, one of the factors that determine the success of techniques based on neighborhood search is the ability to evaluate many solutions very quickly
- The time to explore a neighborhood depends not only, as we have seen, on the size, but also on the computational complexity of evaluating a single neighborhood

In this regard, it is always important to consider the possibility of *incremental evaluation* that takes advantage of the information from the neighborhood center.

- The possibility of efficient incremental evaluation is related to the *degree of perturbation* introduced by a move: for this reason, there is a tendency to favor neighborhoods determined by simple moves (often less strong but quick to evaluate), as opposed to moves that result in significant perturbations (neighborhoods that are stronger but, in addition to having larger sizes, require less efficient evaluation)
- In total, neighborhood complexity is given by the product of neighborhood size times the evaluation complexity of each neighbor

The solution evaluation function is used to compare neighbors to the center solution (and between each other) – normally, the objective function – basically, how "good" they are.

In KP-0/1, the evaluation function may:



- May include some extra-feature (e.g. combined by means of a weighted sum) to identify "more promising" solutions
  - In KP-0/1, "prefer" solutions with larger residual capacity $\tilde{f}(X) = \sum_{i \in X} p_i + \epsilon \left(W - \sum_{i \in X} w_i\right)$
- May include penalty terms (e.g. infeasibility level to allow visiting infeasible solutions)
  - In KP-0/1, let X be the subset of loaded items $\tilde{f}(X) = \alpha \sum_{i \in X} p_i - \beta \max\left\{0, \sum_{i \in X} w_i - W\right\} \qquad (\alpha, \beta > 0)$ it potentially activates "removing" move in a connected "insertion+removing" neighbourhood

4.10 Exploring strategies and LS application to TSP

The basic LS scheme depicted above makes the search go on if the neighbourhood of the current solution contains an improving solution. The choice of which improving neighbour solution to select is not unique and depends on the exploration strategy. The common alternatives for exploration are:

- First improvement: as soon as the *first improving neighbour* is generated, it *is selected as the next current solution*
  - Notice that the final results (e.g., the local minimum found, or the running times for a single move) depend on the order in which neighbour solutions are explored
  - In order to reduce running times, we may adopt a heuristic order, to give priorities to the moves that are more likely to yield an improvement
  - A random order may be used instead, so that different repetitions of the local search (starting from the same initial solution) may lead to different local optima

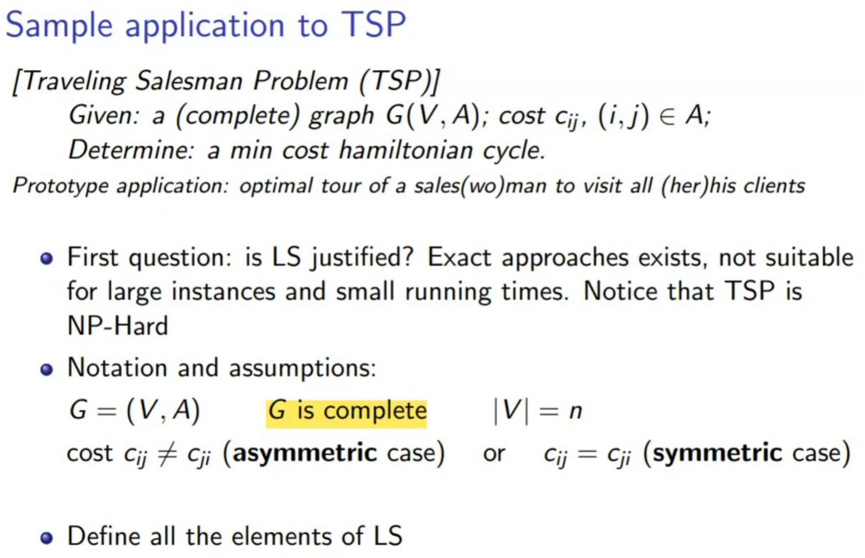
- Granularization: *apply a filter* (deterministic rules, a pre-trained classifier) *to exclude part of the neighbours* (machine learning)

- Steepest descent or best improvement: *all the neighbourhood is explored and evaluated*, and the next move is determined by the *best one*

Alternative techniques are possible, some of which, to incorporate randomness into the algorithm, determine the k neighborhood solutions that guarantee the highest improvements and randomly choose one of these solutions as the next current solution.

- Repeated execution of such an implementation of local search allows finding excellent different locales from which the best one is chosen
- Another possibility is to store some of the best unvisited neighbors and use them, at the end of the first descent, as the initial solutions of a new local search that could lead to excellent different locales

Consider a classical problem: the Traveling Salesman Problem (TSP), which considers an Hamiltonian cycle – cycle that visits each vertex exactly once. Up to a reasonable size, solving the problem is feasible, but sometimes we need approximate solutions to get heuristics and solve a specific problem (since it is NP-Hard, otherwise, algorithms have an exponential complexity).



Sample application to TSP

[Traveling Salesman Problem (TSP)]
Given: a (complete) graph $G(V, A)$; cost c_{ij}, $(i, j) \in A$;
Determine: a min cost hamiltonian cycle.
Prototype application: optimal tour of a sales(wo)man to visit all (her)his clients

- First question: is LS justified? Exact approaches exists, not suitable for large instances and small running times. Notice that TSP is NP-Hard
- Notation and assumptions:
  $G = (V, A)$ G is complete $|V| = n$
  cost $c_{ij} \neq c_{ji}$ (**asymmetric** case) or $c_{ij} = c_{ji}$ (**symmetric** case)
- Define all the elements of LS

TSP is an NP-hard problem, and as early as 100 nodes, CPLEX is struggling to find a solution optimal with the exact approach (note: important for Assignment 1!)

In addition to starting from a random solution, obtained by considering a random sequence of visiting graph nodes, there are several constructive heuristics for TSP, among which we mention the following.

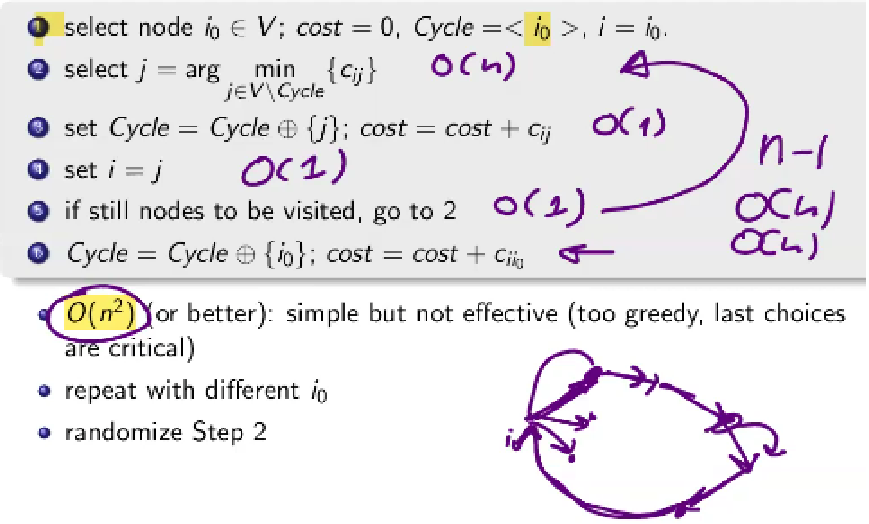
The Nearest Neighbor (NN) (also, Nearest Node in slides) heuristic provides a straightforward way to construct an initial solution:

- Start from a random node i_0
- Iteratively select the closest unvisited node until all nodes are visited
- Complete the cycle by returning to i_0
- Complexity is $O(n^2)$

The NN heuristic is:

- Simple to implement but not amazingly effective
- Greedy in nature, which can be problematic since the last choices become critical
- Can be improved by:
  - Running multiple times with different starting nodes i_0
  - Randomizing the node selection in Step 2

After the initial solution, we improve things by heuristics and local search:



```
1 select node $i_0 \in V$; $cost = 0$, $Cycle =< i_0 >$, $i = i_0$.
2 select $j = \arg\min_{j \in V \setminus Cycle} \{c_{ij}\}$   $O(n)$
3 set $Cycle = Cycle \oplus \{j\}$; $cost = cost + c_{ij}$   $O(1)$
4 set $i = j$   $O(1)$
5 if still nodes to be visited, go to 2   $O(1)$
6 $Cycle = Cycle \oplus \{i_0\}$; $cost = cost + c_{ii_0}$
```

$n-1$

$O(n)$

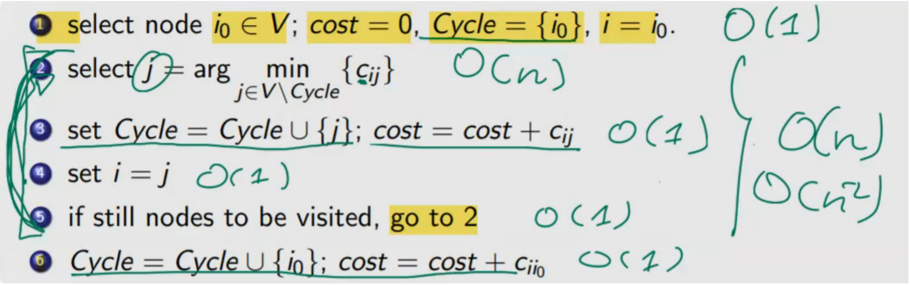
$O(n)$

- $O(n^2)$ (or better): simple but not effective (too greedy, last choices are critical)
- repeat with different i_0
- randomize Step 2

The algorithm is simple and of low complexity (improvable with particular data structures) but has mediocre performance.

- In particular, the loop degrades as it is constructed, since the initial choices tend to leave out the most disadvantaged nodes, not considering that one must return to the starting node
- To try to improve performance, with still poor effects, or to obtain different starting solutions for the local search, one could run the algorithm n times, starting from the n different nodes in the graph (choosing i_0 at step 1); or one could randomize the choice of the nearest node at each step (choosing, at step 2 randomly among the k nodes closest to i)

The complexity increases as long as we continue augmenting the number of nodes, creating larger instances of the problem – nevertheless, extremely fast and good idea for an initial solution.



```
1 select node $i_0 \in V$; $cost = 0$, $Cycle = \{i_0\}$, $i = i_0$.   $O(1)$
2 select $j = \arg\min_{j \in V \setminus Cycle} \{c_{ij}\}$   $O(n)$
3 set $Cycle = Cycle \cup \{j\}$; $cost = cost + c_{ij}$   $O(1)$
4 set $i = j$   $O(1)$
5 if still nodes to be visited, go to 2   $O(1)$
6 $Cycle = Cycle \cup \{i_0\}$; $cost = cost + c_{ii_0}$   $O(1)$
```

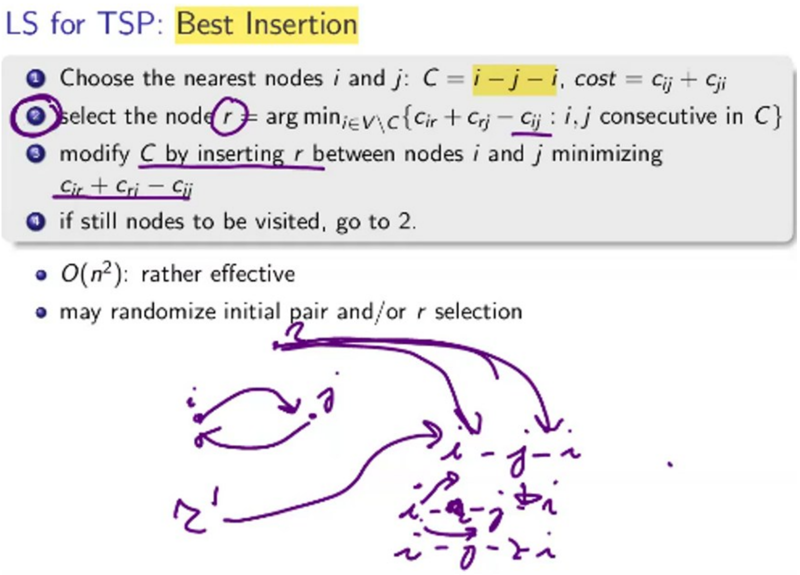
$O(n)$

$O(n^2)$

The use of Local Search for TSP is justified because:

- TSP is NP-Hard
- Exact approaches exist but are not suitable for:
  - Large instances
  - Small running time requirements
- LS provides a good trade-off between solution quality and computational effort

Another heuristic is to Best Insertion: find two closest nodes and then calculate the insertion cost between each consecutive pair, choosing the best inserting it in the best position – insertion algorithms add new points between existing points on a tour as it grows:



LS for TSP: Best Insertion

1. Choose the nearest nodes i and j: $C = i - j - i$, $cost = c_{ij} + c_{ji}$
2. select the node $r = \arg\min_{i \in V \setminus C}\{c_{ir} + c_{rj} - c_{ij} : i, j \text{ consecutive in } C\}$
3. modify C by inserting r between nodes i and j minimizing $c_{ir} + c_{ri} - c_{ij}$
4. if still nodes to be visited, go to 2.

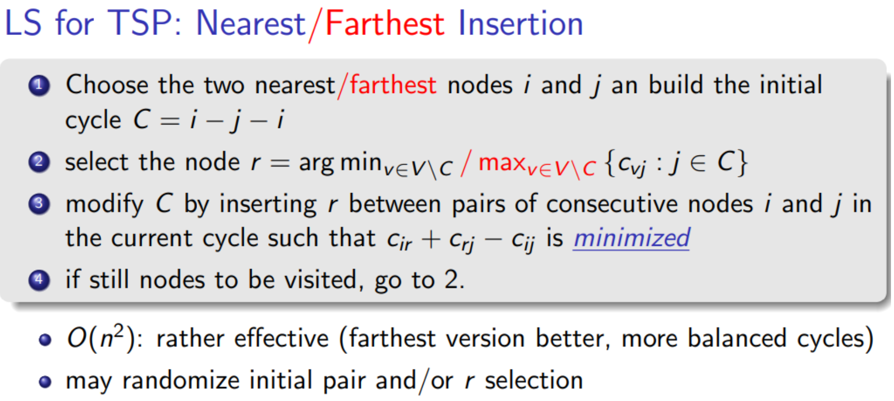
- $O(n^2)$: rather effective
- may randomize initial pair and/or r selection

2
i
j
r'
i - j - i
i - a - j - r - i
i - j - r - i

To consider that the path must close, this heuristic starts from a loop of length two and inserts, at each step, a node into the cycle, with an expansion criterion that selects the node nearest to/farther away from the cycle. We first start on the Nearest Insertion.

The initial cycle is obtained by selecting the i nodes i and j such that c_{ij} is the minimum/maximum: the initial cost is then $c_{ij} + c_{ji}$. At each iteration, if C is the set of nodes in the current cycle, we select the node $r = argmin_{k \in V \setminus C}\{c_{k,j} : j \in C\}$.

The Farthest Insertion works better (keeping the complexity to circa $O(n^3)$ since the unlucky choices or the farthest actually are effective (since cycle is balanced and not degraded after latest insertions). The goal is not to choose the minimum cost; if one gives probability to the farthest nodes and then construct to make choices starting from safe nodes.



LS for TSP: Nearest/Farthest Insertion

1. Choose the two nearest/farthest nodes i and j an build the initial cycle $C = i - j - i$
2. select the node $r = \arg\min_{v \in V \setminus C} / \max_{v \in V \setminus C} \{c_{vj} : j \in C\}$
3. modify C by inserting r between pairs of consecutive nodes i and j in the current cycle such that $c_{ir} + c_{rj} - c_{ij}$ is *minimized*
4. if still nodes to be visited, go to 2.

- $O(n^2)$: rather effective (farthest version better, more balanced cycles)
- may randomize initial pair and/or r selection

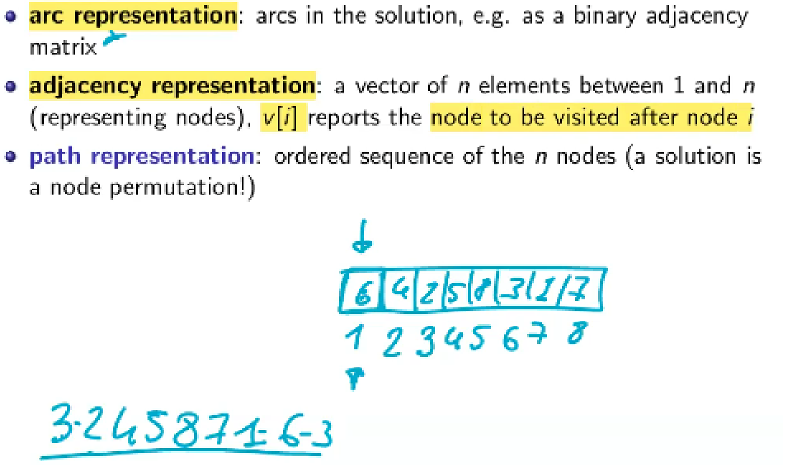
Consider also that we are not talking about Christofides algorithm, which works *only* from a *theoretical* point of view (have a read here if you don't know). Here, you have no performance guarantee since this works specifically in the worst case with a complexity of $O(n^4)$. In the worst case the tour is no longer than $\frac{3}{2}$ the length of the optimum tour (twice the cost of the optimal solution) – so, both heuristics and exact methods with smaller complexity work better than this one.

Have a look here in case if you are interested in many other heuristics and different representations.

For the TSP representation of the solution, we have separate ways:

- *Arc representation*: arcs in the solution, e.g., as a binary adjacency matrix, containing the binary matrix $N \times N$ containing the arcs being traversed (1 if $M(i,j)$ is part of the solution)
- *Adjacency representation*: a vector of n elements between 1 and n (representing nodes), $v[i]$ reports the node to be visited after node i
- *Path representation*: ordered sequence of the n nodes (a solution is a node permutation!)

Each position of the array is devoted to a specific node and in the representation we represent the sequences and respective position; the disadvantage comes from the fact positions do not represent information, but the order of visiting. Of course, it's easy and intuitive, that's why it is used generally.



- **arc representation**: arcs in the solution, e.g. as a binary adjacency matrix
- **adjacency representation**: a vector of n elements between 1 and n (representing nodes), $v[i]$ reports the node to be visited after node i
- **path representation**: ordered sequence of the n nodes (a solution is a node permutation!)

6	4	2	5	8	3	1	7
1	2	3	4	5	6	7	8

3-245871-6-3

A good representation might be using *arcs*, considering for example the representation here.

Continuing, we will mainly refer to *path* representation, which corresponds to the most natural way of representing the TSP solution and, together with adjacency representation, enjoys the property that any vector of nodes (without repetitions) represents an admissible solution (while, in the first case, not all matrices represent a tour!). Decoding the path representation is straightforward: just construct the loop following the order given in the vector.

What is the minimal way to modify the solution (removing/adding arcs), what is the way to rebuild the solution to construct an Hamiltonian cycle? Rebuild the same graph, removing X arcs and add the corresponding X arcs.

- Classically, the neighborhood for the TSP is obtained by exchanging k arcs in the solution with k arcs not belonging to the solution
- In order to obtain feasible solutions, it is necessary that the arcs in the solution are not consecutive in the starting cycle; moreover, once the k arcs to be eliminated are defined, one can explicitly define the k arcs to be introduced into the solution to form a new cycle

- This type of neighborhood is called k-opt, and was introduced by Lin-Kernighan in 1973

The k-opt neighborhood structure for TSP involves:

- Remove k edges from current tour
- Reconnect the tour with k new edges to form a valid cycle
- Must maintain tour validity

Examples shown:

- 2-opt: Removes 2 edges and reconnects with 2 new edges
- 3-opt: Removes 3 edges and reconnects with 3 new edges

Implementation details:

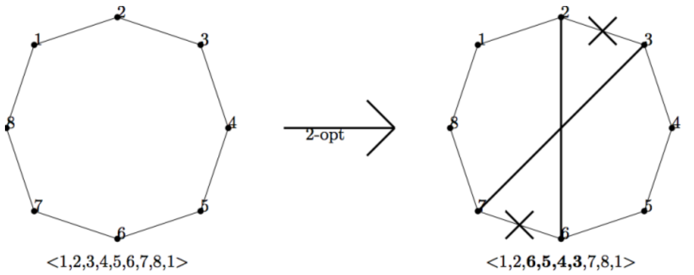
- For 2-opt: Effectively reverses a subsequence between cut points
- Original: $\langle 1,2,3,4,5,6,7,8,1\rangle$
- After 2-opt: $\langle 1,2,6,5,4,3,7,8,1\rangle$
- After 3-opt: $\langle 1,2,7,6,3,4,5,8,1\rangle$

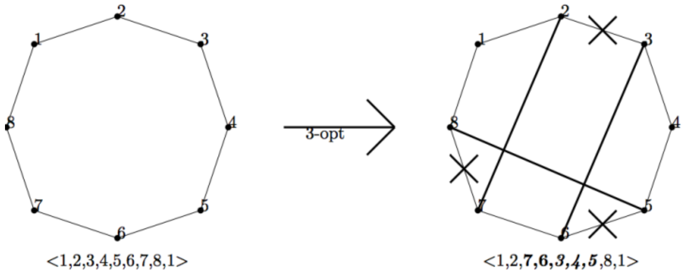
Properties:

- Neighborhood size: $O(n^k)$
- $k = 2$ gives superior results for most instances
- $k = 3$ provides marginal improvements
- $k > 3$ rarely worth computational cost
- Consider there at most 6 ways to change and construct Hamiltonian cycles

This forms the basis for effective local search improvements for TSP solutions.

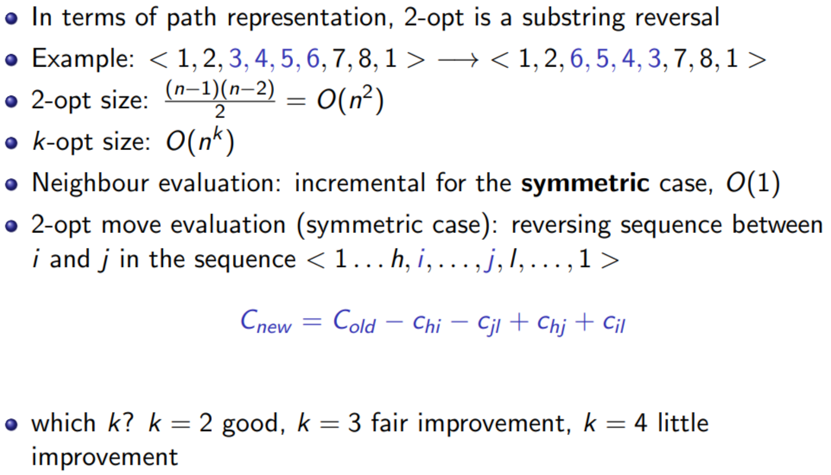
- The k-exchange moves can be defined directly as operations on the vector of path representation
- For example, the 2-opt neighborhood is obtained by defining any pair (i, j) of nodes and reversing the sub-sequence of nodes between i and j (substring reversal): in the example, $i = 3, j = 6$ and the sequence $3, 4, 5, 6$ is replaced by $6, 5, 4, 3$; another possible neighbor is obtained for $i = 3$ and $j = 7$, leading to the sequence $\langle 1, 2, 7, 6, 5, 4, 3, 8, 1\rangle$ and so on





For TSP, *2-opt moves involves reversing a substring between two cut points*, considering all of the possible pairs and moves, incrementally evaluating all of the possible combinations of nodes (constant time, since removing/adding takes the same time) – the same happens for *the 3-opt, for each triplet of non-consecutive arcs*.

About the complexity of the evaluation of each neighbor, it is a matter of subtracting from the value of the central solution the cost of the k eliminated arcs and adding the cost of the k added arcs, which can be done in constant time $O(1)$: we thus have an extremely efficient incremental evaluation, at the only additional "cost" of storing, for the center of the neighborhood, the value of the solution.



- In terms of path representation, 2-opt is a substring reversal
- Example: $< 1, 2, 3, 4, 5, 6, 7, 8, 1 > \longrightarrow < 1, 2, 6, 5, 4, 3, 7, 8, 1 >$
- 2-opt size: $\frac{(n-1)(n-2)}{2} = O(n^2)$
- k-opt size: $O(n^k)$
- Neighbour evaluation: incremental for the **symmetric** case, $O(1)$
- 2-opt move evaluation (symmetric case): reversing sequence between i and j in the sequence $< 1 \ldots h, i, \ldots, j, l, \ldots, 1 >$

$$C_{new} = C_{old} - c_{hi} - c_{jl} + c_{hj} + c_{il}$$

- which k? $k = 2$ good, $k = 3$ fair improvement, $k = 4$ little improvement

The removal operation takes constant time since we are subtracting from time to time the arcs. What is the best choice of the parameter "k"?

- $k = 2$: Good balance of improvement vs complexity
- $k = 3$: Moderate additional improvement but higher complexity $O(n^3)$
- $k = 4$: Minimal improvement for computational cost $O(n^4)$

No specific reason to adopt special choices, since we can simply use objective function (total cycle cost), with no need for special evaluation functions or penalties/modifications:

- Neighbours evaluated by the objective function (cost of the related cycle) – take first better solution found
- Steepest descent (or first improvement) – evaluate all neighbors

As implied by the previous arguments, solutions are evaluated with the mere objective function: in effect, each permutation of nodes is an eligible solution sufficiently differentiated, in terms of cost, from its other neighbors, and it is not considered useful to introduce penalties or other components.

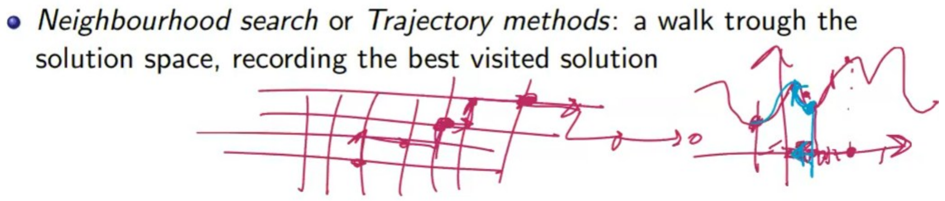
4.11 Neighborhood Search/Trajectory Methods

Local search is a good trade-off between simplicity and efficiency, but there is a risk that it *gets stuck on a local optimum*. A strategy is therefore needed to dodge these excellent ones. Note that if the function to be optimized is convex, there is no such problem because every local optimum is also global. Typically this does not occur in optimization problems. Some ideas might be to do random restarts, change the neighborhood size, randomize exploration, or perform backtracks. But these ways only allow restarts once embedded.

We have already mentioned *implementation tricks that try to escape from the local optimum*, among which we mention, for summary:

- *Random Multistart*: this is the simplest technique, which consists of repeating the local search with different initial solutions, randomly generated or with randomized heuristics
- *Dynamic Neighborhood Modification* (since the definition of local minimum also depends on the neighborhood): for example, in TSP, we start with a 2-opt neighborhood and, if no improving 2-opt neighbors are found, we switch to a 3-opt neighborhood
    - Advanced techniques such as *Variable Neighborhood Descent (VND)* or *Variable Neighborhood Search* (which are actually much more complex and are outside the scope of this course) are based on this observation
- *Randomize* the exploration strategy by randomly choosing among k neighboring enhancers
- *Introduce backtrack mechanisms*, based on the *memory* of some feasible alternative choices of neighbors that can be considered later, once a local minimum is reached by other means

An alternative approach is trajectory methods, which continue the exploration of the solution space even after ending up in a local optimum. This requires admitting moves that lead to a worse solution. With this strategy there is a risk of ending up in a loop, using an already explored solution.



- *Neighbourhood search* or *Trajectory methods*: a walk trough the solution space, recording the best visited solution

Accepting non-improving moves can be a way to try to escape the local optimum, in order to *avoid loops*. Here are the key strategies for avoiding loops in trajectory-based search:

To avoid this is possible:

- Accept only *better solutions*
    - For example: *Hill Climbing*
- *Randomly explore space* without exploiting information about the problem
    - For example: *Simulated Annealing*
- *Keeping track of solutions already been encountered*, exploiting the problem structure
    - For example: *Tabu Search*

Note that finishing several times on the same solution may be acceptable. However, in this case it is necessary to avoid choosing the same previously chosen neighbor again.

4.12 Tabu Search

Tabu search (TS) (created by Fred Glover in 1989) is a metaheuristic that relies on memory to avoid cycling by preventing certain "tabu" moves, which happens maintaining a tabu list for *forbidden* and *allowed* solutions, determining *how much they will stay forbidden (the already visited solutions)*.

A Tabu List, a list containing the last t solutions visited $T(k) \coloneqq \{x^{k-1}, x^{k-2}, \dots, x^{k-t}\}$.

- By doing so at iteration k, cycles of length $\leq t$ are avoided, where t is a parameter that needs to be calibrated. The limitation given by the parameter t is there to limit memory consumption and to make the search on the list faster. Neighborhood generation is now done by a function $N(x,k)$ that also takes iteration into account to avoid tabu

- To increase efficiency and take up less space, one may choose to keep track of the *moves* made (or some other feature) instead of the solution, because it may be that evaluating the equality of two solutions is too expensive or because a single solution requires too much memory

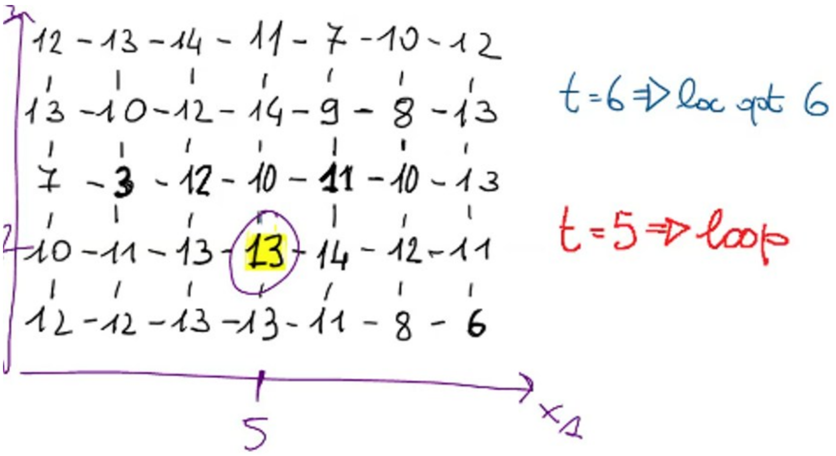
- There is a disadvantage to keeping track of moves, however, because solutions that have not yet been visited may be excluded (also seeing if a neighbor is in the tabu list)

A simple example can clarify the basic idea of tabu search:

- Suppose we are in a local minimum x and $y \in N(x)$ is the best neighbor (even if worse than x) according to the neighborhood N used
- If we agree to move to y, at the next iteration, an improving solution in the neighborhood $N(y)$ will surely be x and it is quite likely that x is chosen as the next current solution
- This triggers a cycle between x and y, from which one could escape simply by remembering that x is an already visited solution and preventing (making "tabu") its selection

If we accept non-improving moves, we are in trouble since we might get stuck in a loop at any given moment! The local optimum can be the attracting point.

We are inside of a discrete space, which represents coordinates like the following:



12 - 13 - 14 - 11 - 7 - 10 - 12

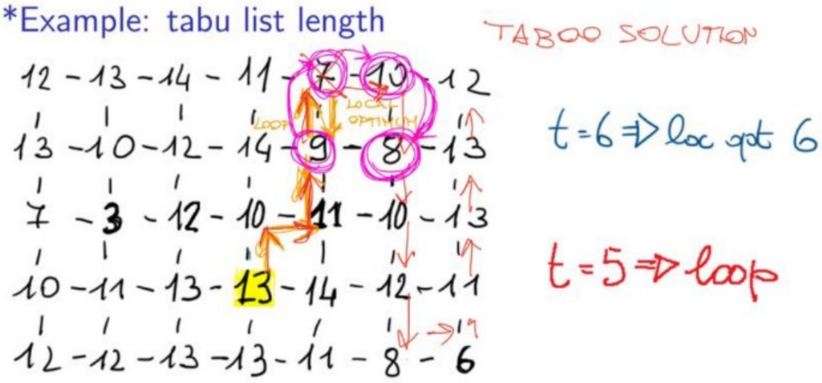
13 - 10 - 12 - 14 - 9 - 8 - 13

7 - 3 - 12 - 10 - 11 - 10 - 13

10 - 11 - 13 - 13 - 14 - 12 - 11

12 - 12 - 13 - 13 - 11 - 8 - 6

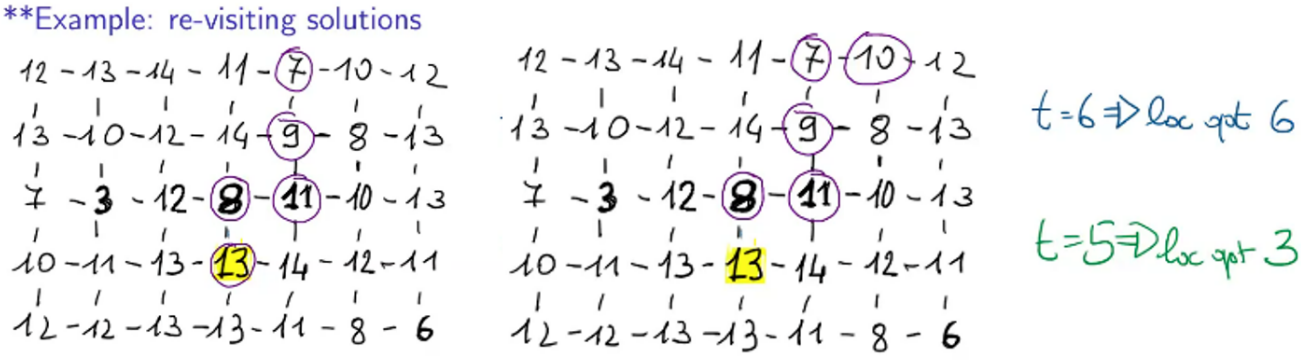
If we start from 13, we look at the neighbors and we choose 10; the third solution is the minimum with respect to 10, which is 11 and continuing like that. The complete path is the following one.



It is an innovative idea to accept non-improving moves, but in the local search scheme that we have, we eventually are going to loop between good solutions. Above, this is represented by the above pink cycle. Local search has local optimum as point of stopping, but this is too simple; it will loop somehow, depending on the inputs.

If I store inside of the tabu list all of the solutions visited, it is impossible to loop; at some point the tabu list length will be so long that it becomes impossible to compare all of the viable solutions.

Let's consider a second example with the length of the tabu list with length 5, with the following representation (left is the beginning one, right one deletes a previous node since length of list is 5):



Of course, storing solutions means coming back to those nodes, but the tabu list memorizes them in some way (not coming back to nodes, but performing the same moves with a different tabu list). The best solution going on with the walk, the best solution is 3; if all solutions were stored, everything would have been "tabu", so keeping a shorter list is definitely useful to perform again the same moves in an ordered way.

It’s important also to verify the membership of a certain neighbor in the tabu list, which depends from the length of the list and the complexity of the comparison to check.

- Let us clarify this point by considering the case of the TSP with neighborhood 2-opt: instead of storing the last t Hamiltonian cycles visited, one can store t pairs of arcs subject to deletion in the last selected moves: if we choose to delete the arcs (i, j) and (h, l) and, consequently, add the arcs (i, h) and (j, l), these arcs will not be exchanged for the next t moves. Or one could decide to make all moves involving nodes i, j, h and l tabu

- Indeed, in this context, to cycle does not simply mean to return to a certain solution, but to cyclically retrace a certain trajectory in the search space. Thus, even if one were to return to a solution already visited, the important thing is to continue on a different path, which is possible if one inhibits moves (or salient features) recently considered and therefore contained in the tabu list

For example, in solving the TSP using 2-opt moves, completing what said above (which reverse parts of the tour):

- Instead of storing complete tours in the Tabu List, we store the pairs of arcs that were added by recent moves
- This prevents immediately reversing those moves, which would undo progress
- However, if reversing a tabu move would create a tour shorter than any found so far, the aspiration criterion allows it

Another reason for limiting the memory of visited solutions is that, if one makes many neighbors tabu (think particularly of the case of prohibition on features of solutions), after a certain number of steps one risks greatly depleting the “legal” neighborhood, preventing proper exploration of solutions.

- For this reason, the length t of the tabu list (called <u>tabu tenure</u>) is a critical parameter that needs to be sized appropriately:
    - Too low = makes the tabu list too short and the risk of cycling remains
    - Too high = the tabu list is too long and as seen, there is a risk of constraining the search too much (losing potentially good neighbors) even though by now one has moved far enough away from a certain solution or local minimum to make it unlikely to cycle

This is where <u>aspiration criteria</u> come in. They can be defined that if they are met they will surpass the tabu rule and let the solution visit anyway. For example, as aspiration criteria one can use “the tabu solution has the best objective function value among all those visited so far”, *overruling* them.

- They provide a way to override the Tabu List restrictions when a promising solution is found
- The most common aspiration criterion is allowing a tabu move if it leads to a solution better than the best found so far. This makes intuitive sense – even if a move is tabu, if it leads to the best solution yet, we should allow it

There are different <u>stopping criteria</u>, since the one used by local search (find improving neighbors) is not applicable anymore. It’s a combination of (all parameters with * should be calibrated):

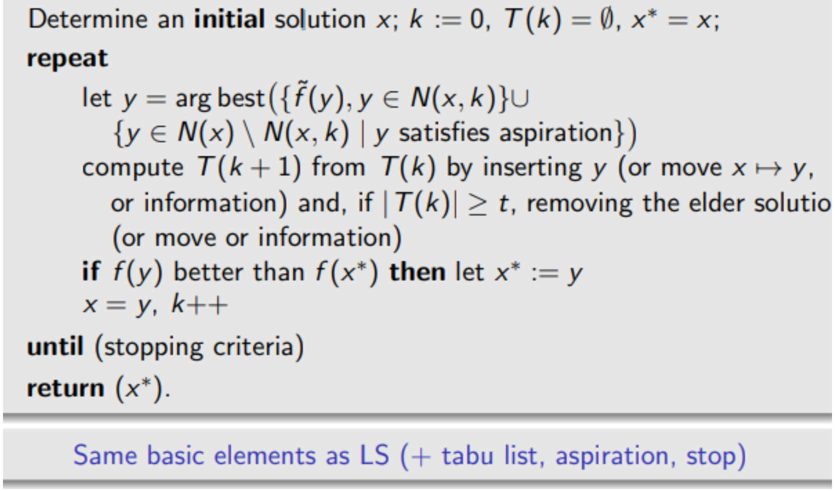
- Maximum number of iterations, or maximum time limit *
- Maximum number of NOT (locally or globally) IMPROVING iterations *
- A solution is found satisfying an optimality or “acceptability” certificate, if available…

- Empty neighbourhood and no overruling (no aspiration criteria to apply)
    - Perhaps t is too long (too high as parameter – length is the list size)
    - Perhaps visit non-feasible solutions (e.g., COP – Constrained Optimization Problem) with many constraints): modifying evaluation function, alternate dual/primal search

The last criterion is peculiar to TS and could occur for strongly constrained problems, where the number of admissible neighbors is very small. The presence of the additional restriction of the tabu list makes the connection characteristics of the neighbors even more critical, and therefore techniques that allow one to proceed in the exploration of infeasible solutions and then return to feasible solutions (one speaks in these cases of *granular tabu search*).

The basic TS scheme is the best improvement scheme (steepest descent) – this means it evaluates ALL neighbors (both non-tabu and those satisfying aspiration criteria) and selects the best one as the next solution. In this case, the tabu search is considered *reactive*, since next exploration depends on the ones done before.

- In contrast, a first improvement strategy would modify this approach significantly. Instead of evaluating every neighbor, it would accept the first neighbor it finds that improves the current solution
- The search would stop examining neighbors as soon as an improving solution is found



```
Determine an initial solution $x$; $k := 0$, $T(k) = \emptyset$, $x^* = x$;
repeat
    let $y = \arg\text{best}(\{\tilde{f}(y), y \in N(x,k)\} \cup$
      $\{y \in N(x) \setminus N(x,k) \mid y$ satisfies aspiration$\})$
    compute $T(k+1)$ from $T(k)$ by inserting $y$ (or move $x \mapsto y$,
      or information) and, if $|T(k)| \geq t$, removing the elder solutio
      (or move or information)
    if $f(y)$ better than $f(x^*)$ then let $x^* := y$
    $x = y$, $k$++
until (stopping criteria)
return $(x^*)$.
```

Same basic elements as LS (+ tabu list, aspiration, stop)

Point is: we store information only to avoid specific moves.

Having defined the essential ingredients, we schematize the basic tabu search as follows:

1. Generate an initial solution x and set $k := 0, T(k) = \emptyset, x *= x$
2. Generate the neighborhood $N(x)$
3. Choose the solution y that optimizes the evaluation criterion $\tilde{f}(y)$ among all solutions in $N(x,k)$ or among solutions in $N(x) \setminus N(x,k)$ that satisfy some aspiration criterion
4. Obtain $T(k+1)$ from $T(k)$ by fitting y (either the move $x \rightarrow y$ or some characteristic of y) and, if $|T(k)| \geq t$, eliminating the "oldest" solution (or move or feature)
5. If $f(y)$ is better than $f(x^*)$, place $x^* := y$
6. If a stopping criterion is not met, place $k = k + 1, x = y$ and return to step 2
7. Return (x^*)

The scheme is very simple and follows local search, modified with the additional ingredients such as tabu list, aspiration criteria, and stopping criteria. Therefore, in addition to what has been said above for the specific components of tabu search, all the expedients design arrangements already discussed for local search about:

- Determination of an initial solution
- Representation of the solution
- Definition of the neighborhood
    - With relative complexity and the possibility of incremental evaluation
- Solution evaluation function (which could be different from the objective function f
    - As evidenced by the use of $\tilde{f}$ in the presented scheme

Regarding the exploration strategies note how, as a base, a steepest descent strategy is used, although nothing prohibits, to speed up the search, the adoption of first improvement strategies. In addition, the possible presence of aspiration criteria necessitates the evaluation of all neighbors, even tabu ones, which could be avoided (to increase efficiency) if such criteria were not used.

The professor shows an example of code on TS, on which different input sizes are tested:

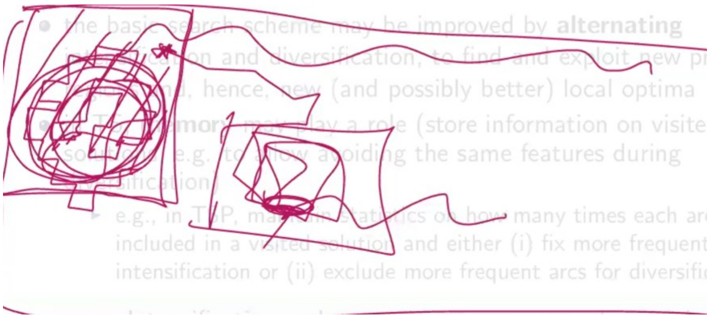
- We only have an empirical analysis to conduct to understand the exact number and it has to be conducted on different instances
- The right size strictly depends on the size of the problem

After a while, in metaheuristics it is useful to start from somewhere else and multistart again.

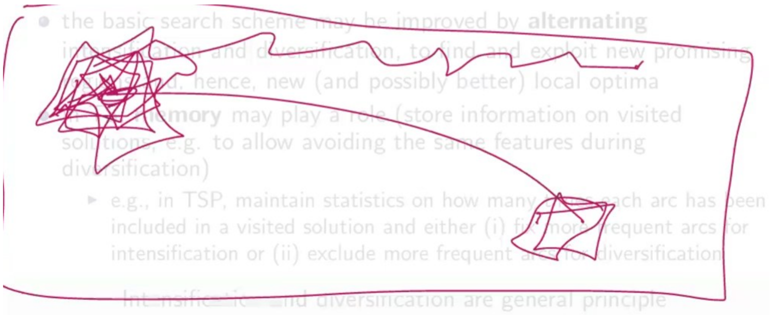
The basic scheme described above allows the development of algorithms that generally provide good performance. These can be further improved, crucially for applications, by systematically extending the use of exploration memory to alternate phases of search intensification and diversification.

- *Intensification* consists of deep exploration of certain areas of the search space that seem promising: for example, we focus on solutions that possess certain characteristics, or on solutions that are relatively "similar" to each other (left image)

- *Diversification*, on the other hand, consists of trying to identify little-visited areas of the solutions space, with the aim of identifying promising new areas on which to intensify research: e.g., the selection of solutions with different characteristics from the best current solution (right image)



- the basic search scheme may be improved by **alternating** intensification and diversification, to find and exploit new pr
regions and, hence, new (and possibly better) local optima
- in TS, memory may play a role (store information on visite
solutions, e.g. to allow avoiding the same features during
diversification)
    - e.g., in TSP, maintain statistics on how many times each ar
included in a visited solution and either (i) fix more frequent
intensification or (ii) exclude more frequent arcs for diversifi



- the basic search scheme may be improved by **alternating** intensification and diversification, to find and exploit new promising regions and, hence, new (and possibly better) local optima
- in TS, memory may play a role (store information on visited solutions, e.g. to allow avoiding the same features during diversification)
    - e.g., in TSP, maintain statistics on how many times each arc has been included in a visited solution and either (i) fix more frequent arcs for intensification or (ii) exclude more frequent arcs for diversification
- Intensification and diversification are general principle

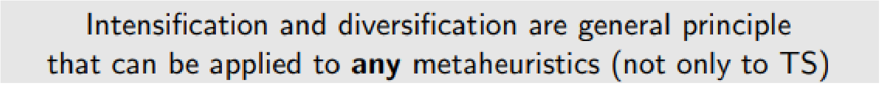
The alternating phases of intensification and diversification are intended to orient the search efficiently toward finding different local minima and, therefore, of globally better solutions. Intensification and diversification can be applied to all metaheuristics, and their exhaustive exposition is beyond the scope of our purposes: we limit ourselves here to providing some ideas on how these can be implemented In the particular context of tabu search.

The balance between these phases is crucial, since after a while you might not find any better solutions:

- *Too much intensification*: *Gets stuck* in local optima
- *Too much diversification*: *Random walk* without finding good solutions
- *Right balance*: *Thoroughly explores* areas while maintaining *ability to escape* when needed

This principle extends beyond Tabu Search to other metaheuristics (better seen up next):

- *Genetic Algorithms*: Population diversity vs. selective pressure
- *Simulated Annealing*: Temperature control (high = diversification, low = intensification)
- *Ant Colony*: Pheromone concentration vs. evaporation rates



Intensification and diversification are general principle
that can be applied to **any** metaheuristics (not only to TS)

Possible diversification techniques include the following:

- Use, within the same Tabu Search algorithm, different contours. E.g. example, for TSP, if a stopping criterion occurs with a 2-opt neighborhood, the search can continue with a 3-opt neighborhood, until an improving solution is found.
    - In general, several neighborhoods can be defined that allow for solutions that are more or less distant (dissimilar) from the center solution, and priority criteria are established in the exploration of these neighborhoods
    - Each neighborhood is associated with a tabu list, the management of which is completely independent of that of the other tabu lists

- Modify the neighborhood evaluation function, rewarding solutions that deviate, in terms of features, from the current one

- At the end of an intensification step, consider the best obtained solution x and construct a new starting solution that is as different from x (complementary) as possible, so as to search, through further intensification, for a better solution starting from a point in a different area of the search space

- A more refined way (and consistent with the principles of tabu search, which is based on the systematic use of memory) is the introduction of a *long-term memory term*, which collects information about the exploration history
    - Indeed, the tabu list, in its basic definition, represents a short-term memory (recency-based memory, one stores a few recent moves), used to direct local search in order to make the probability of cycling negligible following acceptance of deteriorating moves

- Through long-term memory, new and different directions can be given to local search
- For example, statistics can be collected on the features that are more or less explored (because they are more or less present in the solutions gradually selected in constructing the trajectory in the search space) and based on these statistics, incentivize (e.g., by rewarding in the evaluation function) the selection of solutions carrying features little explored

A simple way to achieve alternating intensification and diversification phases in tabu search contexts is to *dynamically manage the length of the tabu list* (parameter t, tabu tenure), which, therefore, no longer has the mere function of avoiding the cycles potentially triggered by the acceptance of worsening moves. In particular, in the intensification phases, t is held at low values (the minimum t_0 value that prevents cycling).

- If the best available solution x^* is not updated for a given number of iterations, the value of t increases, while it decreases again (to the limiting value t_0) when x^* is updated

- Note that as t increases, the number of solutions, moves or features increase tabu and, as a result, there is a tendency to accept solutions that are sufficiently different from the last explored and, ultimately, to move quickly to areas of the space of the solutions other than the current one, thus achieving diversification

Many metaheuristics are inspired by nature: have a read to “Metaheuristics—the metaphor exposed”, a paper present here: https://onlinelibrary.wiley.com/doi/pdf/10.1111/itor.12001

Consider the following example:

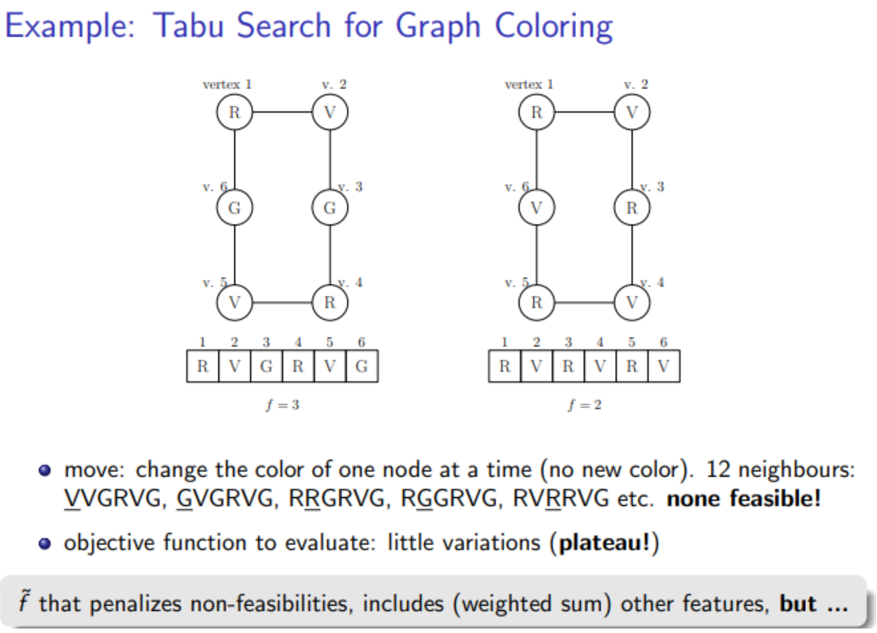
- In the Graph Coloring problem, we want to assign colors to the vertices of a graph such that no two adjacent vertices have the same color (using k colors – k-coloring)
- The goal is often to minimize the total number of colors used while ensuring a valid coloring

Given an undirected graph $G = (N, E)$, the problem of coloring of a graph is to determine the color number of G, denoted by $\gamma(G)$, i.e., the minimum number of colors required to color G and a relative assignment of colors to each vertex (typical application is the coloring of maps).

- First, it is necessary to define the basic components of local search: we focus here on the representation of the solution, the definition of the neighborhood and the evaluation function (a starting solution could trivially be obtained by coloring all nodes with different colors, although there are several possible heuristics)

- In fact, as we shall see, the problem is rather constrained and presents problems inherent in the connection of neighborhoods (already defined as the possibility, given a starting solution, of reaching any solution through a succession of moves), which is particularly relevant in the case of applying a tabu search (since the tabu list tends to impoverish neighborhoods even further)

We want to understand how to move inside of this graph, changing the color of one node at a time (make a walk to find local minimums). Keep in mind the logic of the tabu search explores the search space by moving from one solution to another one inside of the "neighborhood" – here, it includes all colorings that can be obtained by changing the color of a single node.


Example: Tabu Search for Graph Coloring

- move: change the color of one node at a time (no new color). 12 neighbours: VVGRVG, GVGRVG, RRGRVG, RGGRVG, RVRRVG etc. **none feasible!**
- objective function to evaluate: little variations (**plateau!**)

$\tilde{f}$ that penalizes non-feasibilities, includes (weighted sum) other features, **but ...**

The *solution representation* consists of a vector of length $n = |V|$, with an element for each vertex carrying the color assigned to that vertex. Some examples are given in Figure 3, with a 3-color (objective function $f = 3$) and a 2-coloring ($f = 2$). Note how not all color assignments are permissible, since they may violate constraints on the coloring of adjacent vertices.

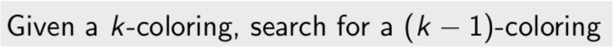
- A first *definition of neighborhood* could be derived from the moves that change the color of one node at a time, trying not to increase the number of colors used in the center solution of the neighborhood: this involves generating a neighbor for each node and for each of the colors used by other nodes in the center coloring
- In the example, starting from the 3-coloring in previous figure, you would get 2 neighbors for each node, for a total of 12 neighbors: VVGRVG, GVGRVG, RRGRVG, RGGRVG, RVRRVG etc.
- Already from this small example it can be seen that none of the neighbors are eligible, making evident the poorly connected characteristics of the defined neighborhood. We therefore have two alternatives: change neighbors or admit the transition for ineligible solutions

Before choosing between the two alternatives, let us consider some observations about the function of evaluating the solutions.

- The natural choice for the solution evaluation function would be the objective function to be minimized, that is, the number of colors used by the proposed coloring. In fact, the color number of a graph tends to be low and, in any case, not much lower than the value of the starting solution that can be obtained by heuristics

- It follows that many feasible solutions use the same number of colors and, therefore, the search space is extremely flat, that is, many neighboring solutions, regardless of the chosen neighborhood, have the same value of the objective function, configuring plateaus – snapshots capturing all relevant information about where the algorithm currently stands

- Consequently, the tabu search will follow a fairly random trajectory in the search space, since a very large number of neighbors represent equally desirable directions, thus risking visiting many equivalent solutions before finding, just as randomly, a solution with a lower number of colors (if a stopping criterion does not intervene first)


Given a k-coloring, search for a $(k-1)$-coloring

When faced with a graph coloring problem, we start with valid k-coloring and aim to find a solution using $k-1$ colors. Instead of directly searching for this reduced coloring, we transform the problem to minimize constraint violations.

- The process begins by eliminating one color, reassigning its vertices to the remaining colors. This creates an invalid coloring that we then refine. To evaluate solutions, we count edges where both endpoints share the same color (monochromatic edges). A solution becomes valid when no monochromatic edges remain

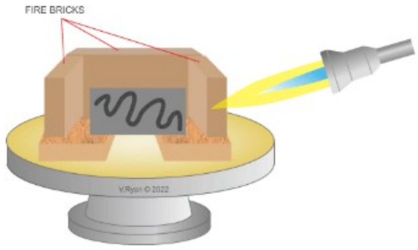
- The search moves through the solution space by changing vertex colors one at a time, focusing specifically on vertices involved in monochromatic edges. To prevent cycling, we maintain a tabu list of recent vertex-color combinations that cannot be immediately reversed

- When we eliminate all monochromatic edges, we've found a valid coloring with $k-1$ colors. At this point, we can attempt to reduce the number of colors further by repeating the process with the new solution

4.13 SIMULATED ANNEALING

Simulated Annealing (SA) is a metaheuristic search algorithm that works by drawing an analogy to the physical annealing process.

When metals are heated and then cooled slowly (annealing – here a visual example), their atoms initially have high energy and move freely, then gradually settle into a low-energy crystalline structure. If cooled properly, this results in a strong, stable configuration (cooling schedule). This physical process inspired the optimization algorithm.



The algorithm works by iteratively exploring solutions while allowing occasional "worse" moves, with their acceptance probability controlled by a temperature parameter. As the temperature decreases, the algorithm becomes more selective about accepting worse solutions.

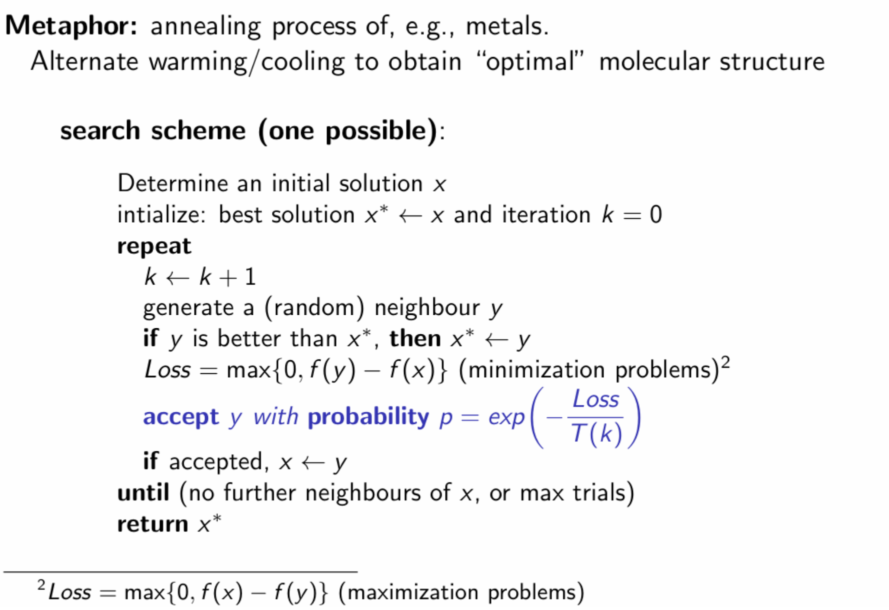
The search process follows these steps:

- First, it starts with an initial solution and temperature
  - At each iteration, it generates a random neighbor solution. If this neighbor is better than the current best solution, it's automatically accepted and becomes the new best
  - If it's worse, it may still be accepted with a probability that depends on two factors: how much worse the solution is (the "Loss") and the current temperature $T(k)$

- The probability of accepting a worse solution is calculated using $p = exp(-\frac{Loss}{T(k)})$
    - This means that small deteriorations and/or high temperatures lead to higher acceptance probabilities
    - As the temperature decreases according to a cooling schedule, the algorithm becomes less likely to accept worse solutions, gradually focusing on improving moves

- The cooling schedule is crucial for the algorithm's performance
    - It typically starts with a high temperature where worse solutions are readily accepted, allowing broad exploration of the solution space
    - The temperature then gradually decreases, making the algorithm more selective and focusing on local improvements
    - The schedule is defined by parameters including the initial temperature, number of iterations at each temperature, temperature decrease rate, and minimum temperature

The following is the SA scheme:



Metaphor: annealing process of, e.g., metals.
Alternate warming/cooling to obtain "optimal" molecular structure

search scheme (one possible):

```
Determine an initial solution $x$
intialize: best solution $x^* \leftarrow x$ and iteration $k = 0$
repeat
  $k \leftarrow k + 1$
  generate a (random) neighbour $y$
  if $y$ is better than $x^*$, then $x^* \leftarrow y$
  $Loss = \max\{0, f(y) - f(x)\}$ (minimization problems)[2]
  accept $y$ with probability $p = exp\left(-\frac{Loss}{T(k)}\right)$
  if accepted, $x \leftarrow y$
until (no further neighbours of $x$, or max trials)
return $x^*$
```

[2] $Loss = \max\{0, f(x) - f(y)\}$ (maximization problems)

The parameter $T(k)$ represents the temperature of the process; the higher it is, the more likely it is to accept worse solutions. As the execution progresses the temperature drops. At the theoretical level it is possible to prove that under certain assumptions (a very long cooling time) this method succeeds in converging to a global optimum. But on a practical level there are meta-heuristics that work better.

4.14 Population-based heuristics

There are metaheuristics that, on the other hand, maintain a population of solutions, that is, a set of several solutions, and, at each iteration, combine these solutions together to obtain a new population.

- The idea is that, through appropriate recombination operators, better solutions can be obtained than current ones
- These methods are called population based and, in many cases, are inspired by natural mechanisms, assuming a tendency of nature to organize itself into structures that are "optimized"

Population-based meta-heuristic algorithms are a class of approaches that search *near-optimal solutions* by maintaining a *set of candidate solutions* and using population characteristics to guide the search iteratively.

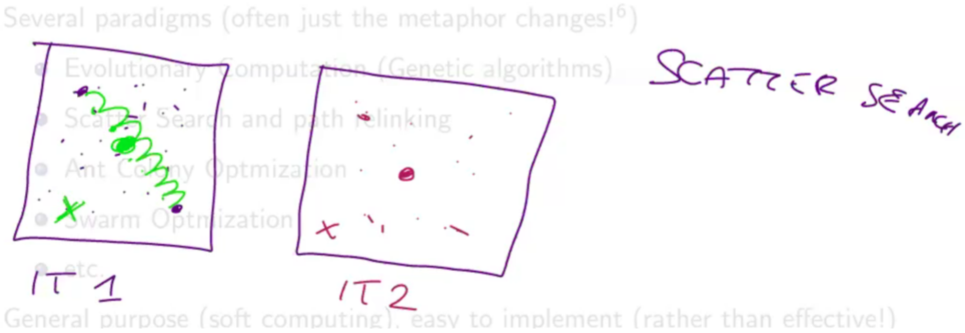
In recent years there have been many studies in this area, moreover with strong interdisciplinary connotations that have led to the definition of different optimization paradigms, for example Evolutionary Computation, Scatter Search e path relinking, Ant Colony Optimization, Swarm Optimization etc. – see "Metaheuristics – the metaphor exposed".

- The basic principles of genetic algorithms were established by Holland in 1975, and were inspired by Darwin's evolutionary theories, published in 1859. Paraphrasing (with much license) these theories, we can see individuals, in their different evolutionary stages, as "solutions" that are increasingly adapted to the environment in which they live and liken the evolution of a population of individuals to some process of "optimization"

- Individuals (*parents*) combine with each other (reproduction) to generate new and different individuals (*offspring*) that become part of subsequent populations (generations); participation in reproductive processes is more likely for those individuals most adapted to the environment, according to the principles of "natural" selection (natural selection) and "survival of the fittest" (survival of the fittest)

- Genetic algorithms attempt to simulate the evolutionary process by matching to each individual a solution, and to the level of adaptation to the environment a fitness, that is, a quantitative measure of the quality of the solution itself, thus trying to make solutions of increasingly high quality survive

This is the base of the scatter search, so to get a better population at each step, which becomes on average better in the solution space and find better solutions:



Several paradigms (often just the metaphor changes!6)

- Evolutionary Computation (Genetic algorithms)
- Scatter Search and path relinking
- Ant Colony Optmization
- Swarm Optimization
- etc.

General purpose (soft computing), easy to implement (rather than effective!)

Summarizing some other methods here, which are inspired by biology, so to replace each time different solutions from an optimization point of view. There are different ways to do so:

- *Scatter search* maintains a small, diverse reference set of high-quality solutions. It creates novel solutions by systematically combining subsets of the reference solutions and improving them with local search

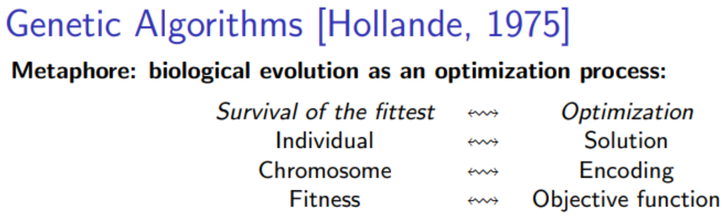
- *Ant colony optimization* is based on how ants use pheromone trails to find efficient routes. Artificial ants construct solutions guided by pheromone information from past searches spread by ants and heuristic information about good decisions. Pheromone is updated to reinforce promising solution components

- Particle swarm optimization moves a swarm of particles through the search space. Particles are attracted to their own best solution and the swarm's best overall. Velocity and position updates balance exploring new areas with exploiting good regions found so far

For the above techniques consider the summary present in the paper – Sorensen's "Metaheuristics – the metaphor exposed"

4.15 Genetic Algorithms: Schema, Encoding, Operators

At its heart, a genetic algorithm frames optimization as an evolutionary process, like how biological organisms evolve and adapt over generations. The idea is that, by mimicking the key drivers of natural evolution (selection pressure, recombination of genetic material, and random mutation), we can "evolve" initially random solutions into highly optimized ones tailored to our problem (survival of the fittest – so, the stronger/better survive).



Genetic Algorithms [Hollande, 1975]

Metaphore: biological evolution as an optimization process:

Survival of the fittest	⇝	*Optimization*
Individual	⇝	Solution
Chromosome	⇝	Encoding
Fitness	⇝	Objective function

- To start, we have to present the potential solutions to our problem in a way that allows the evolutionary mechanisms to operate. Typically this means encoding solutions as "chromosomes" - essentially strings of genes (bits, numbers, or symbols) that capture the key variables or decisions.
- This chromosomal encoding is like the DNA of our candidate solutions. Just as biological DNA encodes the traits of an organism, our artificial chromosomes encode the parameters of a solution. This mapping between the encoding and solution space is a key design step

Genetic algorithms start with an initial population of solutions (the individuals in biological systems) and iteratively evolve them.

- At each iteration, the solutions are evaluated (fitness, level of adaptation to the environment) and, based on this evaluation, a few of them are selected (selection principle), favoring the solutions (parents) with higher fitness (survival of the fittest)
- The selected solutions are recombined (reproduction) to generate new solutions (offspring) that tend to transmit the (good) characteristics of the parent solutions into subsequent generations

The process is articulated as follows:

1. *Coding* of solutions of the specific problem
2. Creation of an initial set of solutions (*initial population*)
3. Repeat, until a stopping criterion is met
    1. *Select* pairs (or groups) of solutions (parent)
    2. *Recombine* parents by generating new solutions (offspring)
    3. Evaluates the *fitness* of the new solutions
    4. *Replace* the population, using the new solutions
4. Return the best generated solution

As with all metaheuristics, this is a very general scheme that must be specialized for different problems. The starting point is the encoding of the solutions based on which the different genetic operators must be defined, mainly:

- Methods for generating an appropriate set of solutions from the initial population
- Function that evaluates the fitness of each solution
- Recombination operators
- Generational transition operators

Genetic operators are based on a *genetic representation* of the solution that encodes the characteristics of a solution.

- This representation corresponds to the chromosome of biological individuals, to the point that we often speak indifferently of solution, individual or chromosome. Still continuing with the analogy, each chromosome is obtained as a sequence (string) of genes
- Each gene is usually associated with a decision variable of the problem and assumes one of the possible values for that variable: depending on the different values actually assumed by the different genes, a different chromosome is obtained and, therefore, a different solution
- To go from the chromosome to the solution, the following a *decoding* is required (which could be immediate) to get a solution in the COP

The following are different representations examples:

- KP/0-1 problem. Binary encoding can be used, associating each object with an order number from 1 to n (number of objects) and using a gene for each object that can take the values 0 or 1. Decoding is immediate: gene i is worth 1 if and only if object i is in the knapsack. An example of a chromosome for $n = 10$ is as follows (1,4,5,9 in the knapsack):

1	0	0	1	1	0	0	0	1	0

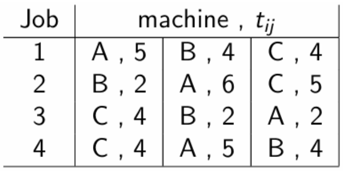
- TSP problem. If n is the number of cities, we use n genes that can take on, each, a value associated with a city. The gene at position i indicates the city to be visited at position i in the Hamiltonian cycle. The chromosome is thus a string (sequence) of cities whose decoding is immediate, directly indicating the order of visitation, the permutation of cities (corresponds to path representation). An example with 10 cities encoded from 0 to 9 is as follows:

3	2	6	1	8	0	4	7	1	5

  which indicates the Hamiltonian cycle $3 \rightarrow 2 \rightarrow 6 \rightarrow 1 \rightarrow 8 \rightarrow 0 \rightarrow 4 \rightarrow 7 \rightarrow 1 \rightarrow 5$

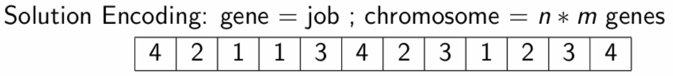
- A job scheduling problem. They are given n jobs to executed on m machines. Each job consists of an ordered sequence of k tasks.
  - Task j of job i is to be executed on a given machine, committing it continuously and exclusively for a time t_{ij}
  - It is fixed the order of execution of the tasks of the same job, and each task must wait until the previous task is finished. It is desired to determine the order of execution of different tasks on different machines so as to minimize the completion time of the jobs
  - The problem is a generalization of the newspaper reading problem presented in the first part of the course: each boy corresponds to a job, each newspaper to a machine, and a boy's reading of a newspaper corresponds to a task.

- One possible encoding uses a sequence of $n \times k$ genes. Each gene can take values between 1 and n. For example: let there be a problem with 4 jobs, each with 3 tasks to be executed on machines A, B and C. The sequence of tasks and completion times are given in the following table.



Job	machine , t_{ij}		
1	A , 5	B , 4	C , 4
2	B , 2	A , 6	C , 5
3	C , 4	B , 2	A , 2
4	C , 4	A , 5	B , 4

One possible chromosome is as follows:

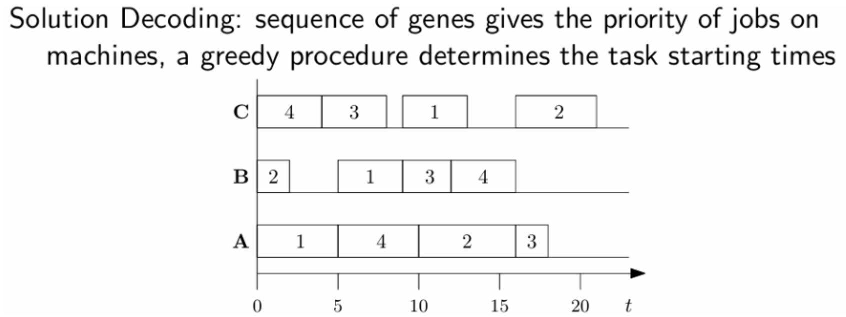


Solution Encoding: gene = job ; chromosome = $n * m$ genes

4	2	1	1	3	4	2	3	1	2	3	4

Note how there is no need to indicate the number of tasks in the chromosome, the sequence of tasks being fixed in the same job.

- In this case, decoding is not straightforward and requires the use of a heuristic, for example, the following one, of linear complexity: run through the sequence of genes and let i be the job indicated by the current gene; consider the first task j of job i not yet considered and schedule it on the corresponding machine as soon as possible (the machine must be free, and the task previous task terminated)
- In practice, the order of genes indicates the priority of each task on the machines (which, in fact, is the decision variable of the problem)
- The value of the objective function is obtained by considering the time when the last task ends. In accordance with this heuristic, the previous chromosome corresponds to the solution shown in the Gantt diagram which has 21 as value of the o.f.



Solution Decoding: sequence of genes gives the priority of jobs on machines, a greedy procedure determines the task starting times

Each element of the solution (decision variable) becomes a gene, so to be used inside of the COP (Combinatorial Optimization Problem).

- Encoding is important and affects following design steps (like solution representation in neighbourhood search)
- Decoding to transform a chromosome (or individual) into a solution of the COP (in the cases above it is straightforward)

All of these algorithms have *genetic operators*, in order to make these mechanisms work. We need a lot of individuals, generated in a random way, in order to have an initial population diversified enough to have a rich genetic heritage:

- To accelerate the general convergence of the method and not simply leave to chance the task of discovering some good features that we would like to include in the solution, one can introduce into the initial population some *solutions generated with heuristics* (constructive or a fast local search) possibly *randomized*, to obtain a variety of good individuals with several good features

- It is important, however, that the number of such solutions be limited, so as not to affect too much the characteristics of the solutions that will be generated in subsequent iterations, causing them to converge, yes quickly, but toward individuals that resemble the starting individuals (obtaining, probably, some local minima) preventing the exploration of individuals with different and, perhaps, better ones

4.16 Fitness Function and Genetic Operators

It is important that the initial set be as diverse as possible and can be generated using other randomized meta-heuristics. The focus on diversification is very important.

- The <u>fitness function</u> is import to give a *quantitative* measure of the fitness (idoneità – being suitable) of individuals guides the processes of selection of individuals, so that, from generation to generation, it is tending to make "survive" individuals with greater fitness, thus passing from one generation to another their genetic makeup and therefore characteristics
- Since we are interested in obtaining optimal values of the objective function, we usually link the fitness function to the value of the objective function (or to its inverse measure for minimum problems)

For these reasons, we may want to use diverse variants of the o.f. so penalize non-feasible solutions, similar to the current optimum, too much distant from the current optimum, etc.

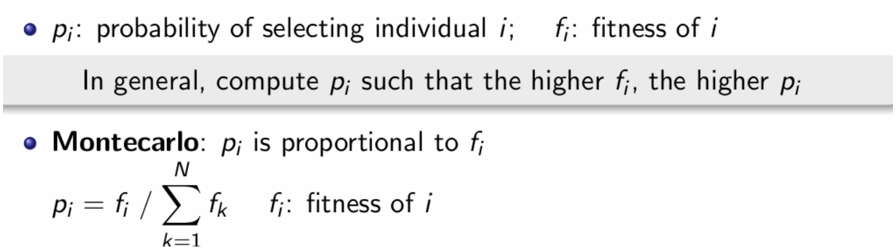
<u>Selection</u> should give a greater chance to the best solutions (*fittest*), but also the worst ones must have the possibility of being chosen, so as to avoid too fast convergence (*prematurely*!) – because *they might contain good features!*

- If only the "best" individuals were selected, the algorithm could converge prematurely towards good local points, because after a few iterations all the individuals would tend to be similar to the best individuals in the initial population, preventing the possibility of discovering individuals with different and, perhaps, better characteristics
- For this reason, selection is again based on probabilistic basis: individuals with a healthy fitness have a higher probability of being selected for subsequent recombinations

Once this principle is established, there are several ways to achieve it, for example:

- *Mode 1*: one parent pair (or generally a group of one/more individuals) is selected at a time
- *Mode 2*: a subset of the current population (*mating pool*) is selected on fitness basis and the individuals in this subset will be used (fished) by recombination operators

The first method to achieve fitness-driven selection is the *Monte Carlo method*, whereby the probability of selecting an individual is simply proportional to his or her fitness score:



- p_i: probability of selecting individual i; f_i: fitness of i

In general, compute p_i such that the higher f_i, the higher p_i

- **Montecarlo**: p_i is proportional to f_i

$$p_i = f_i \, / \sum_{k=1}^{N} f_k \qquad f_i\text{: fitness of } i$$

In this way:

- Especially when combined with the first selection method mentioned, one could excessively privilege the best individuals, especially in the presence of one or a few *superindividuals* with a fitness value much higher than that of the others

- Such individuals tend to be selected very frequently, generating so-called offspring similar to them and, again, in a few iterations the population could converge towards individuals not too dissimilar from superindividuals (local minimum)

The literature suggests various methods to overcome this disadvantage, including:

- *Linear ranking*: individuals are sorted by increasing fitness and selected in proportion to their position (ranking) in the order.
  - This cancels out the effect of fitness values, which could be very different from each other, while only the position of each other is considered
  - More precisely, if σ_i is the position of the individual in the system, one has

$$p_i = \frac{2\sigma_i}{N(N+1)} \quad \left[= \text{constant} \cdot \sigma_i \ (\text{linear in } \sigma_i) \right]$$

- *n-tournament*: in order to select one individual, first select a small subset of n individuals uniformly in the population, then select the best individual in the subset

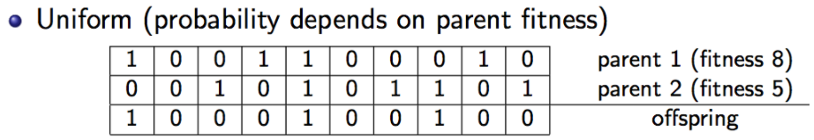
The recombination operators (crossover) act on one or more individuals generating one or more children that "resemble" their parents: they are therefore individuals different from their parents, but which combine their characteristics.

- Usually, the number of parents is greater than or equal to two and often exactly two, in analogy with most natural reproductive processes
- This is precisely because, using only one individual, it would tend to make a copy of the parent, not having the basic mechanisms to obtain different solutions

From n parents it is possible to obtain m different but similar children. One way you can generate children is to choose genes from various parents, giving more choice to the best parent's genes (uniform). An alternative is to inherit genes from "block" (k-cut-point) parents.

- Uniform crossover: from two parents, a child is generated. The genes of the child are copied by the first parent with probability p and by the second parent with probability $(1 - p)$. Usually, $p = 0.5$ is used, or calculated in a way proportional to fitness (so that the child resembles more closely the parent with higher fitness)

The following is an example of uniform crossover on a binary chromosome:

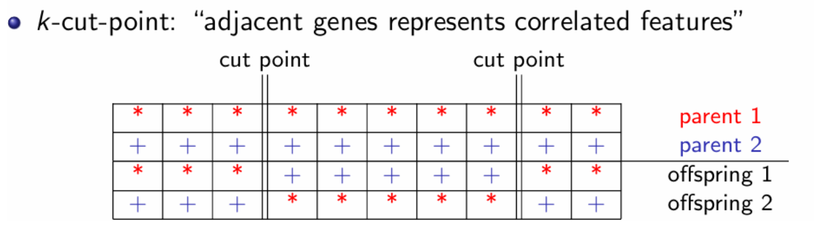


- Uniform (probability depends on parent fitness)

1	0	0	1	1	0	0	0	1	0	parent 1 (fitness 8)
0	0	1	0	1	0	1	1	0	1	parent 2 (fitness 5)
1	0	0	0	1	0	0	1	0	0	offspring

- <u>K-cut-point crossover</u>: it assumes that neighboring genes control related characteristics, so that for children to preserve parental characteristics, *blocks* of contiguous genes must be passed.
    - In practice two parents are considered and k cut points, $k \geq 1$ (k cut-point crossover) are defined *randomly*
    - Then you get a first child by copying the blocks defined by the cut points alternately from the first and second parent, and in a complementary way you get a second child

Below we give an example of 1 cut-point crossover:



- k-cut-point: "adjacent genes represents correlated features"

		cut point					cut point			
*	*	*	*	*	*	*	*	*	*	parent 1
+	+	+	+	+	+	+	+	+	+	parent 2
*	*	*	+	+	+	+	+	*	*	offspring 1
+	+	+	*	*	*	*	*	+	+	offspring 2

Crossover provides the basic mechanism for generating new and different individuals. To make it more effective, one can integrate it with the following steps.

A key to the evolutionary process are random <u>mutations</u>, which must also be encoded within the algorithm during or immediately after the reproduction process.

- The mutation is replicated by *randomly* modifying some genes of the new generation. This prevents a <u>genetic drift</u>, which is to say a population in which all individuals have the same value for some genes – that's why we introduce mutation to complement crossover
- This reintroduces the diversity of genes and slows down population convergence. You can then use a larger mutation to further diversify the population

Example: possible mutation operator on a binary chromosome. Each of the n genes of a solution x is considered separately and modified with probability p_m:

```
for $i := 1$ to $n$ do
  $p := rand(0,1)$;
  if $p \leq p_m$ then $x_i := 1 - x_i$ else $x_i := x_i$;
next $i$
```

The mutation operator also has the important function of counteracting the premature convergence of the population, which is a situation in which all individuals of the population are similar to each other.

- This could easily happen, despite the attention in the selection of parents, precisely because the crossover tends to generate children that resemble parents and, therefore, with the progress of iterations that still favor the best individuals, they are alike

- The same situation could occur in the initial population, although the generation of the initial population must take care of the diversification of individuals

In any case, it is possible that no individual of the current population possesses characteristics which are desirable for optimality of the solution. The mutation operator is then used to introduce, in a random manner, characteristics not possessed by the current population.

- Finally, by dynamically controlling the parameters regulating mutation probability, steps of diversification could be implemented: it is a question of introducing measures of the population's convergence towards certain chromosomes or gene blocks, and, above a certain threshold, increase the probability of mutation (which usually assumes very low values, of the order of 10^3), to modify the individuals generated and obtain different solution
- The mutation operator mimics the natural reproduction mechanism mutation, which occurs during chromosome crossover, introducing more or less "lucky" characteristics in terms of adaptation to the environment

In nature, the level of adaptation to the environment depends not only on an individual's genetic background but also on experiences, which allow further development of the genetic potential and increase the ability to survive and enter into reproductive processes: for example, children are sent to school.

- A similar mechanism can be simulated, complementing the recombination operators, through a *local search*: starting from the generated child, a local search algorithm is applied, and the child is replaced by the corresponding local minimum
- In this case it is important to find a compromise between the quality of the solutions and the computational effort
- Therefore, as a rule, local search operators are preferred not systematically applied to all children but only to a selection (random or fitness-driven) of few individuals in the population

The crossover and mutation operators could generate chromosomes corresponding to unacceptable solutions (unfeasible offspring): think, trivially, of a binary-coded knapsack problem. There are several ways to manage the presence of chromosomes corresponding to non-eligible solutions, including:

1. *Reject* infeasible solutions
    1. The method is not widely used, as several attempts may be required before a qualified chromosome can be generated (by chance)

2. *Accept* the presence of infeasible chromosomes in the population
    1. As mentioned, these chromosomes may contain desirable features and used by recombination operators, could lead to good solutions
    2. Therefore, these are allowed in the population, but appropriately penalized by the fitness function, in relation to the degree of inadmissibility

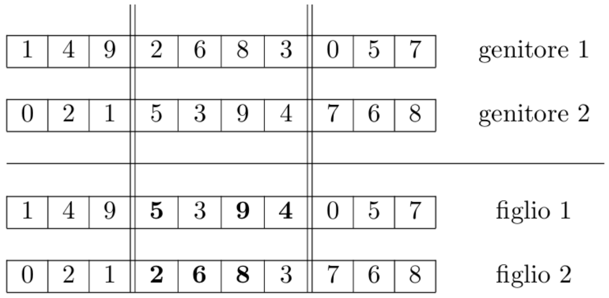
3. "*Repair*" the infeasible chromosomes
    1. It is a question of applying repair techniques, specific to each problem, which implement a forced mutation of a chromosome, transforming it into a feasible solution

Example: KP/0-1 with binary coding. Given a chromosome that has the capacity of the backpack, one by one the objects in the reverse order of the ratio prize/weight are eliminated, until you get an eligible chromosome that enters in the population.

4. *Design of encoding* and/or operators that automatically guarantee the eligibility of generated chromosomes
    1. This is typically the best solution, when it can be implemented without excessive computational overhead

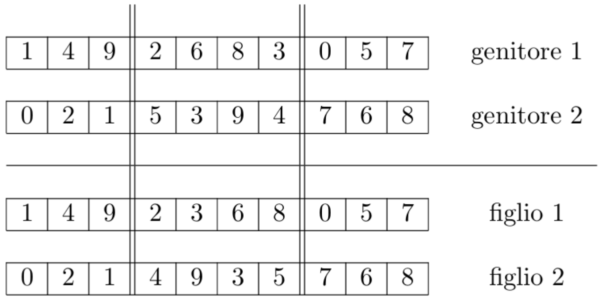
(In all of the following screens: genitore = parent / figlio = child)

Example: TSP. As we have seen, a possible coding is given by the positional chromosome corresponding to the path representation. To make a chromosome feasible, it is necessary and sufficient that all genes are all different between each other. This feature, for example, may be destroyed by a uniform crossover or from a cut/point crossover, as seen in the following example with 10 cities:



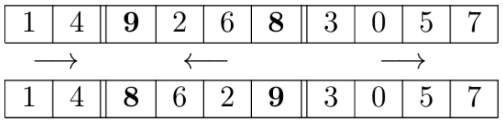
1	4	9	2	6	8	3	0	5	7	genitore 1
0	2	1	5	3	9	4	7	6	8	genitore 2
1	4	9	**5**	3	**9**	**4**	0	5	7	figlio 1
0	2	1	**2**	**6**	**8**	3	7	6	8	figlio 2

The operator can be modified to preserve the eligibility of children, obtaining the order crossover: defined the two cut points, child 1 (Ref. 2) returns the external parts of parent 1 (Ref. 2); the remaining genes are obtained by copying the missing cities in the order they appear in parent 2 (Ref. 1):



1	4	9	2	6	8	3	0	5	7	genitore 1
0	2	1	5	3	9	4	7	6	8	genitore 2
1	4	9	2	3	6	8	0	5	7	figlio 1
0	2	1	4	9	3	5	7	6	8	figlio 2

Similarly, to prevent the mutation from affecting the acceptability, define the mutation by substring reversal: two points of the sequence are randomly generated and the subsequence between the two points is reversed (corresponds to a 2-opt move):



1	4	**9**	2	6	**8**	3	0	5	7
→			←				→		
1	4	**8**	6	2	**9**	3	0	5	7

Note that the order crossover and inversion mutation operators can be used in all cases where the solution is obtainable as element permutation (think of the case of KP/0-1 with encoding/decoding obtained as an ordered sequence of objects).

4.17 Population Management

For each iteration, the new population is obtained by considering the previous iteration's population and the generated offspring. Clearly, if you simply add the new individuals, the population will grow exponentially and therefore population management policies are needed.

- Usually, the number of individuals in the various iterations is kept constant, controlled by an N parameter. There are no shortage of cases where this number is dynamically varied (for example, higher to diversify the research and lower to intensify)

Once R new individuals are generated through recombination (could be R), the basic population management policies are as follows:

- *Generational replacement*: $R = N$ new individuals are generated, replacing the N old ones (mimics biological systems)
- *Steady state*: unlike the previous one, it replaces only a minimal number of elements from the previous generation, selected with fitness-driven criteria (they are tending, on a probabilistic basis, to be replaced by the worst individuals)
- *Elitism*: as generational replacement, but some (few units) of the individuals with greater fitness than the previous population are maintained
- *Best individuals*: the current population is maintained with the best N individuals among the $N + R$. The selection may be deterministic or probabilistic (select, with the Montecarlo method, N individuals among the $N + R$, with probability proportional to fitness)

In practice, mixed techniques are often used. In addition, as we have seen, one of the characteristics to be preserved in the population is still sufficient diversification.

- Therefore, to avoid a premature convergence of the method, acceptance of a new individual in the population could be made conditional on an assessment of how different this individual is from the others, for example using the Hamming distance as a measure of diversity (it may be useless, especially in the phases of diversification, to insert an individual into the population whose chromosome is exactly the same as another present one)
- In any case, a dynamically managed "diversity" threshold could be used to implement intensification and diversification phases

Some examples of *stopping criteria* might be:

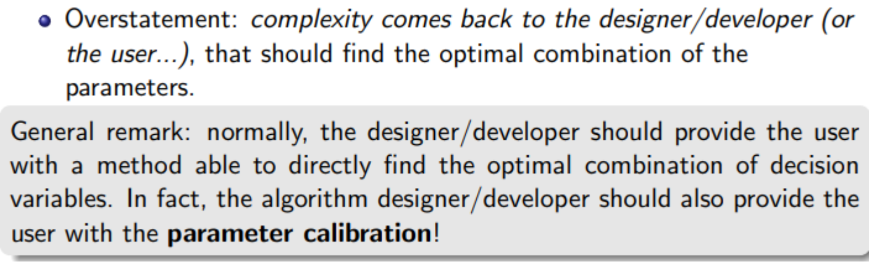
- Time limit (maximum execution time)
- Maximum number of iterations (or generations)
- Number of (not improving) iterations (=generations): stops when the latest improving individual in the o.f. was found many generations before
- Population convergence: all individuals are similar to each other (pathology: not well designed or calibrated) – convergence measures might be similar chromosomes or low fitness variance

4.18 Observations on Genetic Algorithms: Calibration & Performance

Genetic algorithms are very general, but they fall into the category of "soft" methods, since they cannot exploit the specific problem properties: we are not thinking about the problem to solve, but the population to evolve. They have many parameters which impact performance and *probability* of their application, since many parameters are not deterministic (= need to be *calibrated*).

- On the advantages side, genetic algorithms are remarkably versatile and robust. Their primary strength lies in their adaptability - they require only two basic components to function: an encoding scheme for solutions and a fitness function to evaluate them
- However, this apparent simplicity masks a significant challenge: the need for extensive parameter calibration. They are not so controllable, since parameter calibration (=finding standard values working on all instances of the same problem) is difficult
- These parameters include population size, mutation rates, crossover probabilities, and selection criteria, among others
- This phase is very important but often left to the user, repeating the same runs – even the single run is fast, the user spends much time

Genetic algorithms are in the class of *weak methods* or *soft computing* (exploit little or no knowledge of the specific problem) – only components of the problem are encoding/decoding of the chromosomes and fitness evaluation, but other components exploit standard implementations.



- Overstatement: *complexity comes back to the designer/developer (or the user...)*, that should find the optimal combination of the parameters.

General remark: normally, the designer/developer should provide the user with a method able to directly find the optimal combination of decision variables. In fact, the algorithm designer/developer should also provide the user with the **parameter calibration**!

A final note goes to the *importance of alternating*, in genetic algorithms as in all metaheuristics, *phases of intensification and diversification*. These can be implemented in the different genetic operators, as described above and summarized below by way of examples:

- Dynamically varying the probability of mutation
- Introducing appropriate penalties in fitness, to penalize or encourage (with dynamic weights, which is and variables during iterations) individuals with certain characteristics
- Linking the likelihood of acceptance of a new individual not only to fitness, but also to his degree of diversity from the remaining individuals
- Increasing the number of individuals subject to local research after their generation in order to intensify it

When evaluating an optimization algorithm, several critical factors must be considered. The implementation choices and the determination of parameters are factors that contribute to determining the performance of an algorithm, and both must be carefully carried out.

- *Simplicity of implementation*, considering the resources (economic, time, personnel) available
- *Computation time*, and the computational efficiency of the algorithm, considering the actual time available to find solutions
- *Quality of the solutions obtained*, that is the "goodness" (or effectiveness) of the algorithm

- *Algorithms with probabilistic components*, the robustness or reliability of the algorithm (*reliability*), the ability to produce good solutions in every run, regardless of the particular random choices

We may have an experimental analysis, which is *empirical*:

1. *Implementation* of the algorithm

2. *Selection of an appropriate set of instances* (specific cases) of the problem
    1. The instances can be real, and/or randomly generated, and/or standard benchmarks provided by literature
    2. The choice of sample depends on the purposes of the evaluation:
        1. For example, if we want to see the behavior of the algorithm in a specific company, it will be appropriate to consider many real instances
        2. If we want to demonstrate that our algorithm is better than others in general, «it is necessary to refer to standard benchmarks»

        3. If we want to test robustness, it will be appropriate to include in the sample several randomly generated instances

3. *The tests are carried out, recording for each execution the evaluation* of the solutions produced and the required calculation times. In the case of parameters (almost always for metaheuristics), it is advisable to preface a calibration step of the parameters themselves (as described below) and use the same parameter definition for all tests.
    1. Furthermore, if the algorithm is not deterministic, a reasonable number of executions on each instance must be considered and average performance values or more accurate statistics, including robust, evaluated

4. *Comparison of the results obtained*: the objective function is compared with the value of the optimal solution (when this is known) or with bound values or with the performance of alternative algorithms, obtaining relative measurements of goodness. Similarly, relative estimates of the time taken for calculation can be made

This approach is always practicable and fairly simple, at least conceptually, and often the one practiced, even if the conclusions drawn from it are not generally valid, since the analysis depends strongly on the instances considered.

Another approach is the probabilistic analysis, which is based on the concept of the average instance of the problem, expressed as a distribution of probability over the class of all possible instances.

- The execution time and the value of the solution are treated as random variables, the tendency of which is studied, generally to tend the dimensions of the instances to infinity according to a certain distribution of probability (usually uniform)
- This approach has strong theoretical foundations but is often impractical (only possible for very simple algorithms)
- Furthermore, the extent of the real case approach is not well known, as the actual distribution of probabilities for data may be unknown or too complicated to be treated analytically

Another case is the worst case analysis, which is based on the determination of the maximum deviation (absolute and relative) that the solution produced by the algorithm can have compared to the optimal solution.

- The analysis is conducted with respect to the worst-case conditions for the algorithm. The result obtained is very strong and of great value (algorithms with guaranteed performance), although it may be difficult to derive it
- Also, often, the resulting indications are very pessimistic compared to the average behavior of the algorithm

Particularly important for the exercise is the parameter calibration (or estimation), which begins with a recognition that the process must be completed before any algorithm deployment, and the resulting parameter settings should be applicable across all instances of the problem.

- This requires a systematic approach using a sufficiently large and representative set of problem instances, which are to be pre-deployed (always choose the same parameter setting), to be then estimated for every instance, justifying the rule calibration so to obtain the right settings.
- It is to emphasize that it is a good rule to fix once and for all the values (absolute or functions), and not adapt them to each individual instance, otherwise you risk spending time "optimizing parameters" to optimize a single problem.

Parameter calibration techniques have recently become the subject of research, and they range from black-box techniques to identify the parameters that guarantee the best performance (*black box optimization*), to automatic adaptation of parameters (*adaptivity*), involving interdisciplinary domains such as artificial intelligence. Here, however, we limit ourselves to mention the standard techniques, simple to implement.

- It is essentially a matter of carrying out repeated tests with different sets of parameters on a small set of instances (test instances), so as to make the time for the tests reasonable
  - The tests are evaluated with the criteria seen above, in order to choose the set of parameters that experimentally guarantee the best performance on the test instances
  - The parameters to be calibrated are generally few for metaheuristics with trajectory, while they tend to be many for population-based metaheuristics

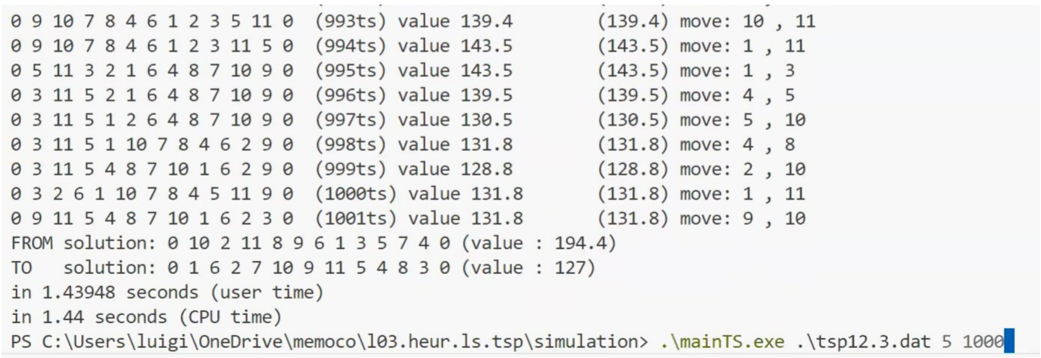
- Obviously, the parameters interact with each other in determining performance, so that the difficulty of calibration grows exponentially with the number of parameters
  - In addition, a factor that complicates calibration further and the presence of random components, which make it more difficult to interpret the influence on performance of a parameter variation, since the performance itself could simply depend on the case
  - For example, the calibration of genetic algorithms could be the real critical step in their use, while their implementation can be relatively simple

The process follows a three-phase methodology using distinct data subsets (sufficiently large and representative):

- Select an instance subset for training (= training set)
    - Computational experiments and testing different parameters configurations
- Extensive runs on the training set
    - Verify parameters effectiveness to ensure training instances work
- Select an instance subset for validation (= validation set)
- Performance analysis to select better parameters
- Take interaction among parameters into account
- Stochastic components make the calibration harder
- Select an instance subset for test (= test set)
- Runs with the estimated parameter to evaluate the "final" performance

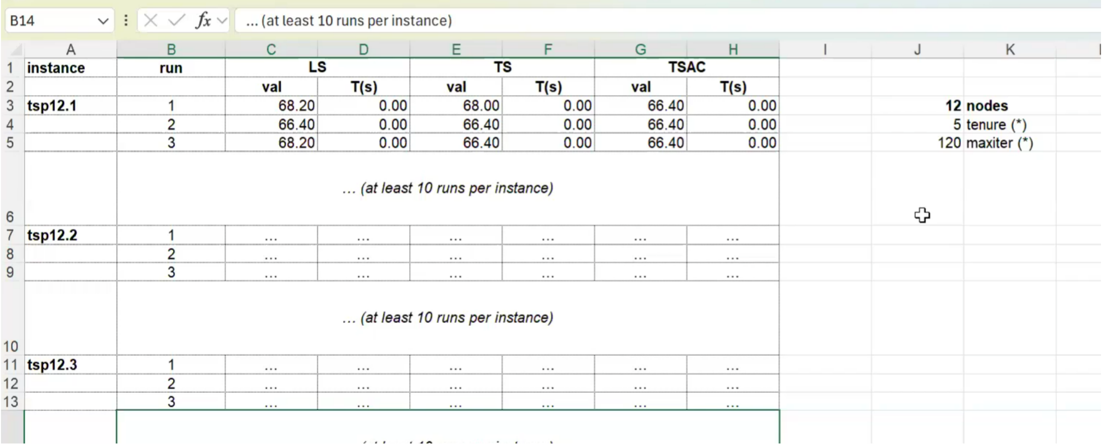
The professor shows us an algorithm for which multiple runs obtain different values, but he says, do not spend that much time on tuning, rather spend your mind on the actual problem resolution. You want to obtain a specific tradeoff between complexity and time.

The parameter setting should depend on some easy feature of the instance, so look at the instance, for example:



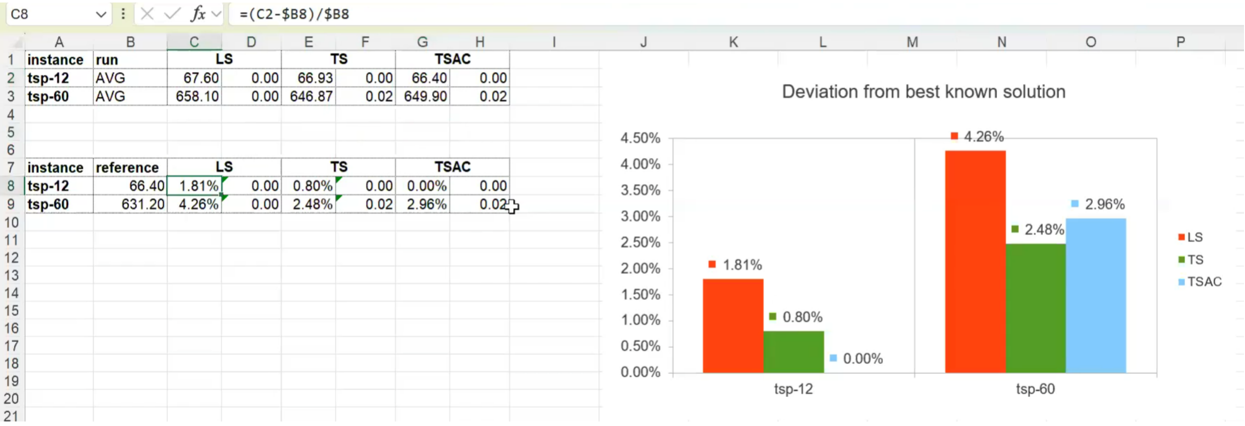
```
0 9 10 7 8 4 6 1 2 3 5 11 0  (993ts) value 139.4       (139.4) move: 10 , 11
0 9 10 7 8 4 6 1 2 3 11 5 0  (994ts) value 143.5       (143.5) move: 1 , 11
0 5 11 3 2 1 6 4 8 7 10 9 0  (995ts) value 143.5       (143.5) move: 1 , 3
0 3 11 5 2 1 6 4 8 7 10 9 0  (996ts) value 139.5       (139.5) move: 4 , 5
0 3 11 5 1 2 6 4 8 7 10 9 0  (997ts) value 130.5       (130.5) move: 5 , 10
0 3 11 5 1 10 7 8 4 6 2 9 0  (998ts) value 131.8       (131.8) move: 4 , 8
0 3 11 5 4 8 7 10 1 6 2 9 0  (999ts) value 128.8       (128.8) move: 2 , 10
0 3 2 6 1 10 7 8 4 5 11 9 0  (1000ts) value 131.8      (131.8) move: 1 , 11
0 9 11 5 4 8 7 10 1 6 2 3 0  (1001ts) value 131.8      (131.8) move: 9 , 10
FROM solution: 0 10 2 11 8 9 6 1 3 5 7 4 0 (value : 194.4)
TO   solution: 0 1 6 2 7 10 9 11 5 4 8 3 0 (value : 127)
in 1.43948 seconds (user time)
in 1.44 seconds (CPU time)
PS C:\Users\luigi\OneDrive\memoco\l03.heur.ls.tsp\simulation> .\mainTS.exe .\tsp12.3.dat 5 1000
```

For example, considering a tabu search, a tabu search with aspiration criteria and local search, one gets the average performance of TSP on a specified number of values:



B14
... (at least 10 runs per instance)

	A	B	C	D	E	F	G	H	I	J	K
1	instance	run	LS		TS		TSAC				
2			val	T(s)	val	T(s)	val	T(s)			
3	tsp12.1	1	68.20	0.00	68.00	0.00	66.40	0.00		12	nodes
4		2	66.40	0.00	66.40	0.00	66.40	0.00		5	tenure (*)
5		3	68.20	0.00	66.40	0.00	66.40	0.00		120	maxiter (*)
6		... (at least 10 runs per instance)									
7	tsp12.2	1	...	...	...	...	...	...			
8		2	...	...	...	...	...	...			
9		3	...	...	...	...	...	...			
10		... (at least 10 runs per instance)									
11	tsp12.3	1	...	...	...	...	...	...			
12		2	...	...	...	...	...	...			
13		3	...	...	...	...	...	...			



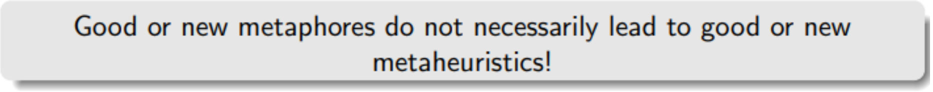
C8 =(C2-$B8)/$B8

	A	B	C	D	E	F	G	H
1	instance	run	LS		TS		TSAC	
2	tsp-12	AVG	67.60	0.00	66.93	0.00	66.40	0.00
3	tsp-60	AVG	658.10	0.00	646.87	0.02	649.90	0.02
4								
5								
6								
7	instance	reference	LS		TS		TSAC	
8	tsp-12	66.40	1.81%	0.00	0.80%	0.00	0.00%	0.00
9	tsp-60	631.20	4.26%	0.00	2.48%	0.02	2.96%	0.02

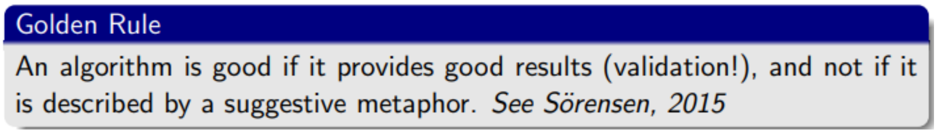
Deviation from best known solution

There are a lot of metaphor-based algorithms; what matters the most is the results part!

- Recent literature proposed a true tsunami of “novel” metaheuristic methods, most of them based on a metaphor of some natural or manufactured process: the behavior of any species (bees, wasps, monkeys, apes, birds etc.), the flow of water, musicians playing together etc.
- Actually, the basic principles are often not novel, but the same as for trajectory or population based methods



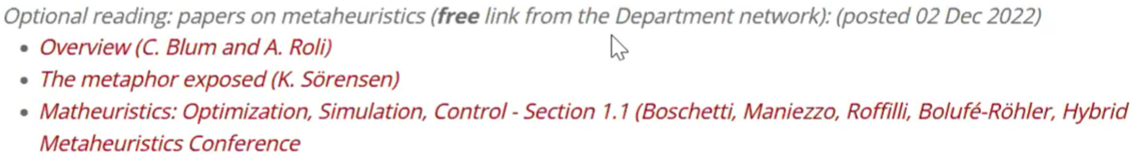
Good or new metaphores do not necessarily lead to good or new metaheuristics!



Golden Rule

An algorithm is good if it provides good results (validation!), and not if it is described by a suggestive metaphor. *See Sörensen, 2015*

Inside the Moodle, you will find these references, to be used in general both for this part and the second assignment:



*Optional reading: papers on metaheuristics (**free** link from the Department network): (posted 02 Dec 2022)*

- *Overview (C. Blum and A. Roli)*
- *The metaphor exposed (K. Sörensen)*
- *Matheuristics: Optimization, Simulation, Control - Section 1.1 (Boschetti, Maniezzo, Roffilli, Bolufé-Röhler, Hybrid Metaheuristics Conference*

We’ll be coming back to the hybrid metaheuristics in the last part of the course unit (last lesson), based on math or data driven optimization techniques (basically, the last 3-4 slides of this set).

4.19 Hybrid Metaheuristics

In these notes, only some of the possible metaheuristics for combinatorial optimization have been given.

- The approach is understood in a much more flexible sense, and those proposed are only suggestions for design choices that must be adapted and questioned according to the particular problem to be solved.
- In this sense we can interpret the development of hybrid metaheuristics in recent years, which seek to combine the merits of different algorithmic schemes.

- Hybridization may take place at various levels and according to different schemes, and their treatment would require a more in depth study outside the scope of the course and refer to specific texts.

We provide below some examples of possible hybridizations, which usually give rise to more powerful algorithmic schemes:

- One common hybridization approach combines population-based methods with trajectory methods
    - For instance, genetic algorithms can be used to identify promising regions of the solution space, while local search techniques provide intensification within these regions
    - A practical example involves using genetic algorithms to generate initial solutions that are then explored more thoroughly using Tabu Search

- Different metaheuristics can also be combined directly
    - For example, Tabu Search principles can be integrated with Simulated Annealing by incorporating probabilistic acceptance criteria into the Tabu Search framework, or by adding memory structures to Simulated Annealing algorithms

- *Matheuristics* represent a particularly interesting class of hybrid methods that combine mathematical programming with heuristics.
    - This is currently a hot research area with several promising directions. These include construction heuristics driven by mathematical models, exact methods for exploring large neighborhoods, and heuristics that provide bounds for exact methods
    - Common frameworks in this area include Local Branching and Kernel Search

- Another emerging trend is *data-driven optimization*, where machine learning and artificial intelligence techniques are integrated into optimization methods
    - This includes using ML for parameter tuning, AI for detecting promising search regions, and various guided search techniques like ML-guided granular search

- Real-world applications demonstrate the effectiveness of hybrid approaches
    - For instance, in pickup and delivery problems, two-level local search combines Tabu Search for intensification with Variable Neighborhood Search for diversification, enhanced by randomization
    - Traffic flow management problems benefit from data-driven matheuristics that utilize historical trajectory data and components determined through data analytics. Electric vehicle sharing systems employ combinations of mathematical models with various heuristic approaches, including partition heuristics and neighborhood search.

The key advantage of hybrid approaches lies in their ability to combine the strengths of different methods while compensating for their individual weaknesses. This makes them particularly valuable for complex real-world optimization problems where single approaches may struggle to provide satisfactory solutions.

Side note: see here for the complete ending of this course. This subsection was quoted there just to complete the file.

Written by Gabriel R.

5 Linear Programming & Simplex Method (4)

(Note: see for this part here for a more complete thing and here for the course part – both Italian)

Initially we saw how the solutions of a whole linear programming problem are located on the vertices of the eligible region and how it was possible to find a solution in graphical way.

With the simplex method, similar considerations are made, but at an algebraic level, so that they can be generalized to cases using more than two variables.

The simplex method is an algorithm for solving linear programming (LP) problems in standard form:

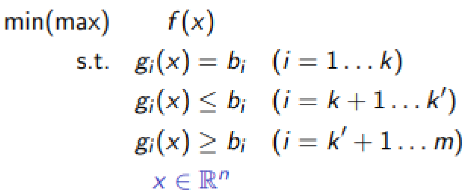
$$min\ c^T x\ subject\ to: Ax \leq b, x \geq 0$$

where:

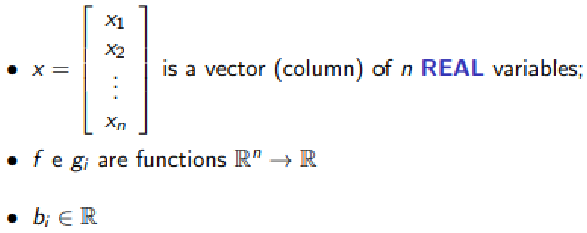
- $A \in \mathbb{R}^{m\times n}$ is the constraint matrix
- $b \in \mathbb{R}^{m}$ is the right-hand side vector
- $c \in \mathbb{R}^{n}$ is the cost vector
- $x \in \mathbb{R}^{n}$ is the variables vector

5.1 Definition and General Notations

A general mathematical programming model takes the form that follows:

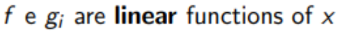


$$\begin{array}{lll} \min(\max) & f(x) & \\ \text{s.t.} & g_i(x) = b_i & (i = 1 \dots k) \\ & g_i(x) \leq b_i & (i = k+1 \dots k') \\ & g_i(x) \geq b_i & (i = k'+1 \dots m) \\ & x \in \mathbb{R}^n & \end{array}$$

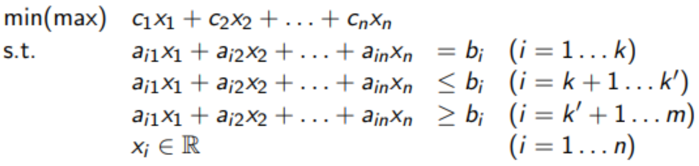


- $x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$ is a vector (column) of n **REAL** variables;
- f e g_i are functions $\mathbb{R}^n \rightarrow \mathbb{R}$
- $b_i \in \mathbb{R}$

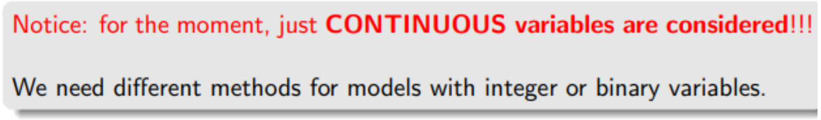
A linear programming model requires that both the objective function $f(x)$ and all constraint functions $g_i(x)$ must be linear functions of the variables. This means they take the specific form:



f e g_i are **linear** functions of x



$$\begin{array}{llll} \min(\max) & c_1x_1 + c_2x_2 + \dots + c_nx_n & & \\ \text{s.t.} & a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in}x_n & = b_i & (i = 1 \dots k) \\ & a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in}x_n & \leq b_i & (i = k+1 \dots k') \\ & a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in}x_n & \geq b_i & (i = k'+1 \dots m) \\ & x_i \in \mathbb{R} & & (i = 1 \dots n) \end{array}$$



Notice: for the moment, just **CONTINUOUS variables are considered**!!!

We need different methods for models with integer or binary variables.

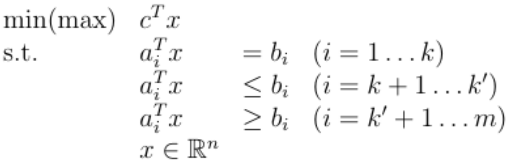
For problems requiring integer or binary variables, different solution methods are needed beyond standard linear programming techniques.

- The linearity requirement is significant because it allows for specialized solution methods like the simplex algorithm. Linear functions have properties that make optimization more tractable compared to general nonlinear functions
- However, this also means that any nonlinear relationships in the real problem must either be approximated linearly or handled through different optimization techniques

In the simplex method:

- We use only continuous variables
- There are NO strict equalities
- The objective function is obtained by the scalar product of the two vectors c and x

More compactly, we can write the problem in this way:



$$\begin{array}{lll} \min(\max) & c^T x & \\ \text{s.t.} & a_i^T x = b_i & (i = 1 \dots k) \\ & a_i^T x \leq b_i & (i = k+1 \dots k') \\ & a_i^T x \geq b_i & (i = k'+1 \dots m) \\ & x \in \mathbb{R}^n & \end{array}$$

An LP model has three possible outcomes, and the resolution process aims to determine which one applies:

- A *feasible solution* is any point x in an n-dimensional real space ($\mathbb{R}^n$) that satisfies all constraints in the model
- The *feasible region* comprises all such points
- An optimal solution x is a feasible solution that optimizes (maximizes or minimizes) the value of the objective function among all feasible solutions

$$c^T x^* \leq (\geq)\ c^T x, \forall x \in \mathbb{R}^n, x\ feasible$$

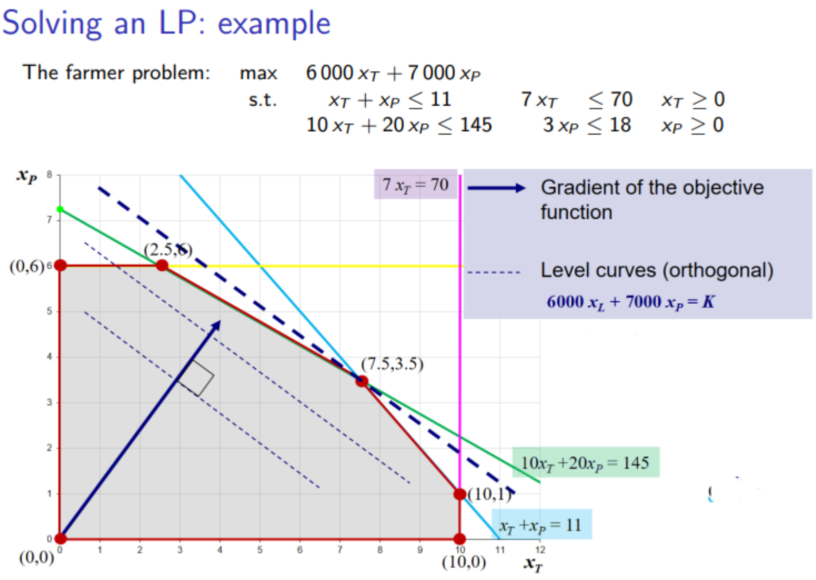
Not always a PL problem is an optimal solution. In fact, it can be shown that each PL problem always satisfies only one of the following 3 cases:

1. *Unfeasible*: the feasible region is empty
2. *Unlimited*: it is possible to find feasible solutions that make decrease (or increase for maximum problems) the value of the objective function as you like
3. *Admits an optimal solution*: there is at least one acceptable solution which optimizes the objective function (and the optimum value of the objective function is limited)

Solve a LP problem by recognizing one of the three cases mentioned and giving, in case 3, an optimal solution and the corresponding value of the objective function.

5.2 Geometry of Linear Programming

Consider the following example:



Solving an LP: example

The farmer problem: max $6\,000\,x_T + 7\,000\,x_P$

s.t. $x_T + x_P \leq 11$ $\quad 7\,x_T \leq 70$ $\quad x_T \geq 0$

$10\,x_T + 20\,x_P \leq 145$ $\quad 3\,x_P \leq 18$ $\quad x_P \geq 0$

The farmer's problem is a maximization LP with two variables (x_T for tomatoes and x_P for potatoes):

$$Maximize: 6000xT + 7000xP$$

The feasible region, shown in gray on the graph, is bounded by several lines representing the constraints. The key points defining this region are:

- Origin (0,0)
- Point (0,6) on the potato tubers constraint
- Point (2.5,6) where potato tubers and fertilizer constraints intersect
- Point (7.5,3.5) where fertilizer and tomato seeds constraints intersect
- Point (10,1) where land and fertilizer constraints intersect
- Point (10,0) on the x-axis

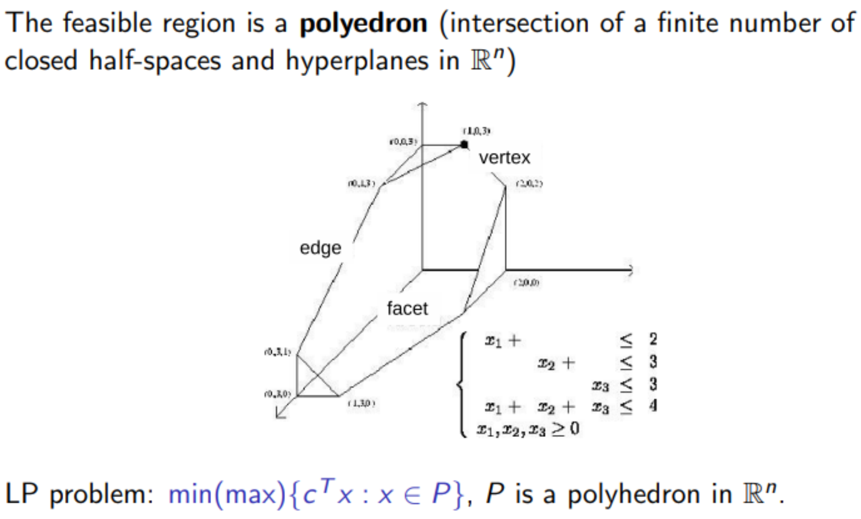
The objective function is represented by the blue arrow showing its *gradient* (direction of steepest increase). The dashed parallel lines are level curves of the objective function, perpendicular to the gradient.

- To maximize the objective function, we move in the direction of the gradient until we reach the furthest possible point in the feasible region. This occurs at the point (7.5,3.5), where the fertilizer constraint ($10x_T + 20x_P = 145$) intersects with the tomato seeds constraint ($7x_T = 70$)

- The optimal solution is therefore: $x_T = 7.5$ hectares for tomatoes $x_P = 3.5$ hectares for potatoes

- The maximum profit can be calculated by plugging these values into the objective function: $6000(7.5) + 7000(3.5) = 45{,}000 + 24{,}500 = 69{,}500$ euros

Geometrically, a solution is a point in the n-dimensional space and the feasible region is a convex polyhedron inside of the same space:



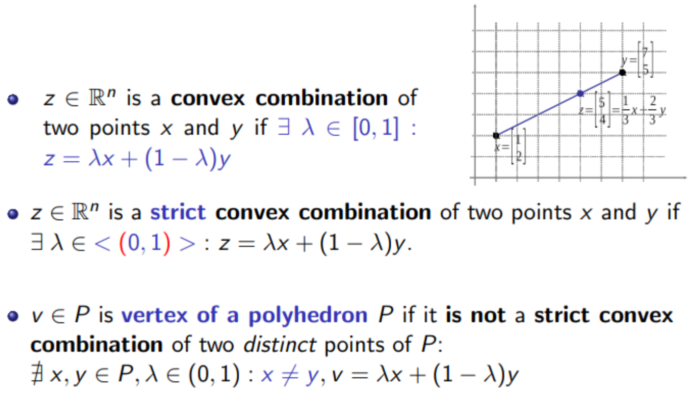
The feasible region is a **polyedron** (intersection of a finite number of closed half-spaces and hyperplanes in $\mathbb{R}^n$)

$$\begin{cases} x_1 + & & & \leq & 2 \\ & x_2 + & & \leq & 3 \\ & & x_3 & \leq & 3 \\ x_1 + & x_2 + & x_3 & \leq & 4 \\ x_1, x_2, x_3 \geq 0 \end{cases}$$

LP problem: $\min(\max)\{c^T x : x \in P\}$, P is a polyhedron in $\mathbb{R}^n$.

A polyhedron is a geometric object formed by the intersection of a finite number of closed half-spaces and hyperplanes in n-dimensional real space. In the context of linear programming, this represents our feasible region – the set of all points that satisfy our constraints.

Points on the polyhedron are to be represented graphically by vertices, which represent geometrically the convex combination on the plane (if they are convex, they are inside of the feasible region):

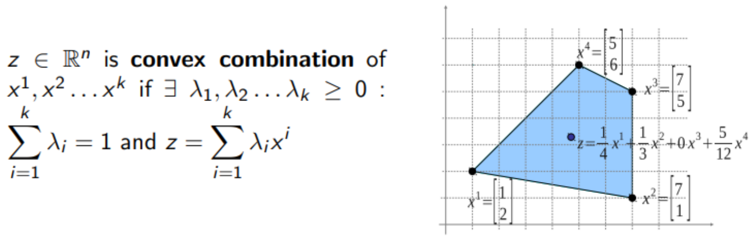


- $z \in \mathbb{R}^n$ is a **convex combination** of two points x and y if $\exists\ \lambda \in [0,1]$: $z = \lambda x + (1-\lambda)y$

$y=\begin{bmatrix}7\\5\end{bmatrix}$
$z=\begin{bmatrix}5\\4\end{bmatrix}=\frac{1}{3}x+\frac{2}{3}y$
$x=\begin{bmatrix}1\\2\end{bmatrix}$

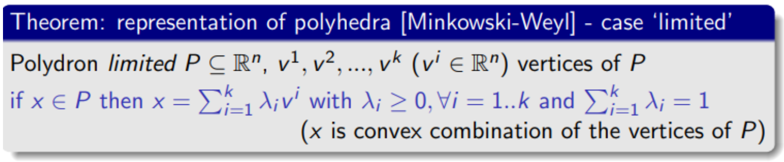
- $z \in \mathbb{R}^n$ is a **strict convex combination** of two points x and y if $\exists\ \lambda \in <(0,1)> : z = \lambda x + (1-\lambda)y$.
- $v \in P$ is **vertex of a polyhedron** P if it **is not** a **strict convex combination** of two *distinct* points of P: $\nexists\, x, y \in P, \lambda \in (0,1) : x \neq y, v = \lambda x + (1-\lambda)y$

The following brings us to another theorem:



$z \in \mathbb{R}^n$ is **convex combination** of $x^1, x^2 \ldots x^k$ if $\exists\ \lambda_1, \lambda_2 \ldots \lambda_k \geq 0$: $\sum_{i=1}^{k} \lambda_i = 1$ and $z = \sum_{i=1}^{k} \lambda_i x^i$

$x^4=\begin{bmatrix}5\\6\end{bmatrix}$
$x^3=\begin{bmatrix}7\\5\end{bmatrix}$
$z=\frac{1}{4}x^1+\frac{1}{3}x^2+0x^3+\frac{5}{12}x^4$
$x^1=\begin{bmatrix}1\\2\end{bmatrix}$
$x^2=\begin{bmatrix}7\\1\end{bmatrix}$



Theorem: representation of polyhedra [Minkowski-Weyl] - case 'limited'

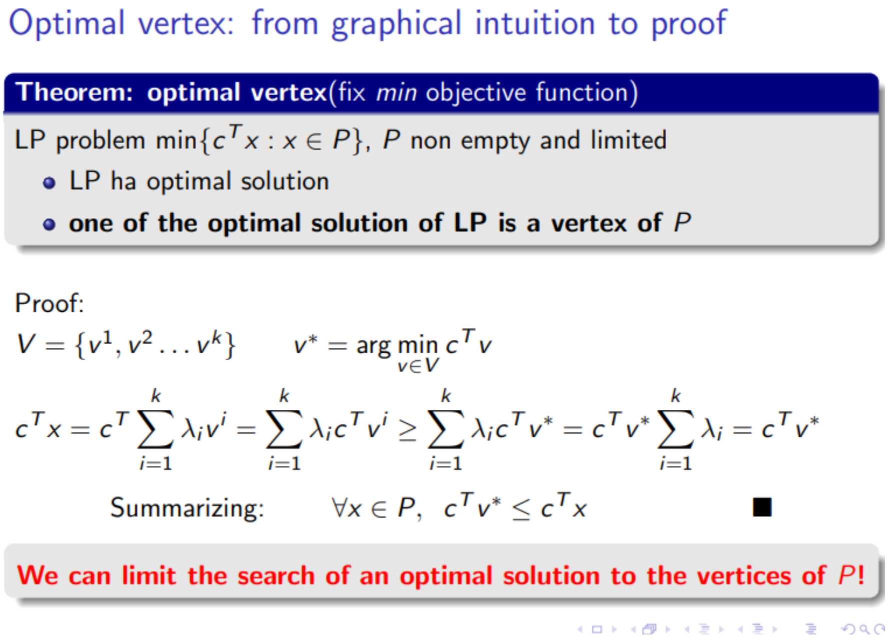
Polydron *limited* $P \subseteq \mathbb{R}^n$, $v^1, v^2, \ldots, v^k$ ($v^i \in \mathbb{R}^n$) vertices of P

if $x \in P$ then $x = \sum_{i=1}^{k} \lambda_i v^i$ with $\lambda_i \geq 0, \forall i = 1..k$ and $\sum_{i=1}^{k} \lambda_i = 1$

(x is convex combination of the vertices of P)

For the Minkowski-Weyl theorem, the convex combination of all the vertices of a polyhedron allows us to represent all the points belonging to the polyhedron.

So, for the optimal vertex theorem, if a LP can be represented by a polyhedron P, then there is at least one optimal solution and one of these is on a vertex. This is an important result because we can limit the search for the optimal solution on the vertices of P and not on the whole space.



Optimal vertex: from graphical intuition to proof

Theorem: optimal vertex(fix *min* objective function)

LP problem $\min\{c^T x : x \in P\}$, P non empty and limited

- LP ha optimal solution
- **one of the optimal solution of LP is a vertex of** P

Proof:

$V = \{v^1, v^2 \dots v^k\} \qquad v^* = \arg\min_{v \in V} c^T v$

$$c^T x = c^T \sum_{i=1}^{k} \lambda_i v^i = \sum_{i=1}^{k} \lambda_i c^T v^i \geq \sum_{i=1}^{k} \lambda_i c^T v^* = c^T v^* \sum_{i=1}^{k} \lambda_i = c^T v^*$$

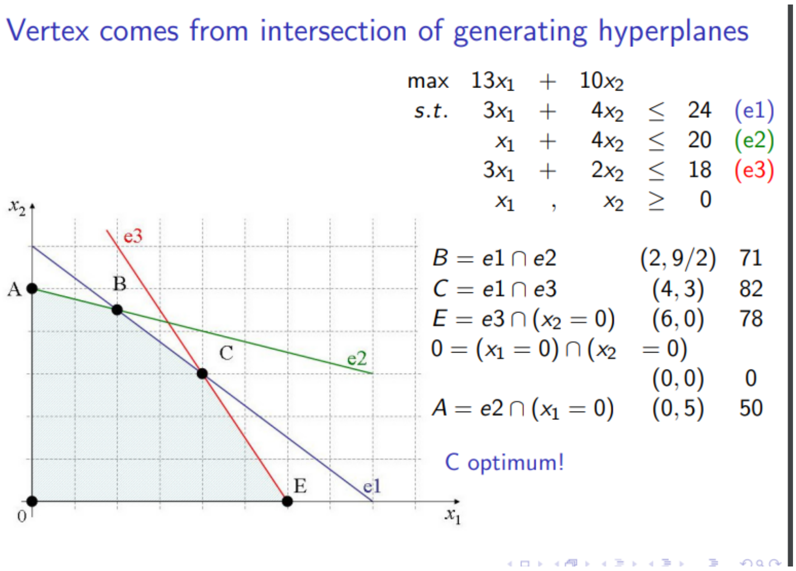
Summarizing: $\forall x \in P, \;\; c^T v^* \leq c^T x$ ■

We can limit the search of an optimal solution to the vertices of P**!**

So basically:

- This theorem has profound practical implications: when solving a linear programming problem, we can restrict our search to vertices of the feasible region rather than considering all points in P
- The theorem transforms what appears to be an infinite search problem into a finite one, though the number of vertices may still be exponentially large

Now we consider how the vertices problem intersection of hyperplanes arises:



Vertex comes from intersection of generating hyperplanes

$$\begin{array}{llllll} \max & 13x_1 & + & 10x_2 & & \\ s.t. & 3x_1 & + & 4x_2 & \leq 24 & (e1) \\ & x_1 & + & 4x_2 & \leq 20 & (e2) \\ & 3x_1 & + & 2x_2 & \leq 18 & (e3) \\ & x_1 & , & x_2 & \geq 0 & \end{array}$$

$B = e1 \cap e2$	$(2, 9/2)$	71
$C = e1 \cap e3$	$(4, 3)$	82
$E = e3 \cap (x_2 = 0)$	$(6, 0)$	78
$0 = (x_1 = 0) \cap (x_2 = 0)$	$(0, 0)$	0
$A = e2 \cap (x_1 = 0)$	$(0, 5)$	50

C optimum!

Each vertex in the feasible region is created by the intersection of exactly two constraints (hyperplanes in this 2D case). The image shows several key vertices:

- Point B occurs at the intersection of constraints e1 and e2, giving coordinates (2, 9/2) with objective value 71
- Point C forms where e1 and e3 meet, at coordinates (4, 3) with objective value 82
- Point E comes from e3 intersecting $x_2 = 0$, at (6, 0) with value 78
- Point A results from e2 meeting $x_1 = 0$, at (0, 5) with value 50
- The origin (0, 0) is formed by $x_1 = 0$ intersecting $x_2 = 0$, with value 0

In this case, vertex C at (4, 3) provides the optimal solution with the highest objective value of 82. This aligns with the theory that an optimal solution will occur at a vertex formed by intersecting hyperplanes.

Understanding how vertices form from constraint intersections is crucial because:

1. It helps visualize how the feasible region is bounded
2. It provides a systematic way to identify candidate optimal solutions
3. It forms the theoretical foundation for algorithms like simplex that move between adjacent vertices

We want to transform inequalities in equalities (we have some *gap*, which are to be called *slack* variables). In this case, there is a specific algebraic representation of vertices:

In our linear programming problem, we start with inequalities: $3x_1 + 4x_2 \leq 24$ $x_1 + 4x_2 \leq 20$ $3x_1 + 2x_2 \leq 18$

To convert these into equations, we introduce slack variables (s_1, s_2, s_3) that represent the "gap" between the left and right sides of each inequality:

$3x_1 + 4x_2 + s_1 = 24$ $x_1 + 4x_2 + s_2 = 20$ $3x_1 + 2x_2 + s_3 = 18$

These slack variables must be non-negative, as they represent the amount by which each constraint is not tight.

There is some *degree of freedom* – in this system, we have:

- Variables total (x_1, x_2, s_1, s_2, s_3)
- 3 equations; this gives us 5 - 3 = 2 degrees of freedom, meaning we can set any two variables to zero and solve for the remaining three variables to potentially find a vertex.

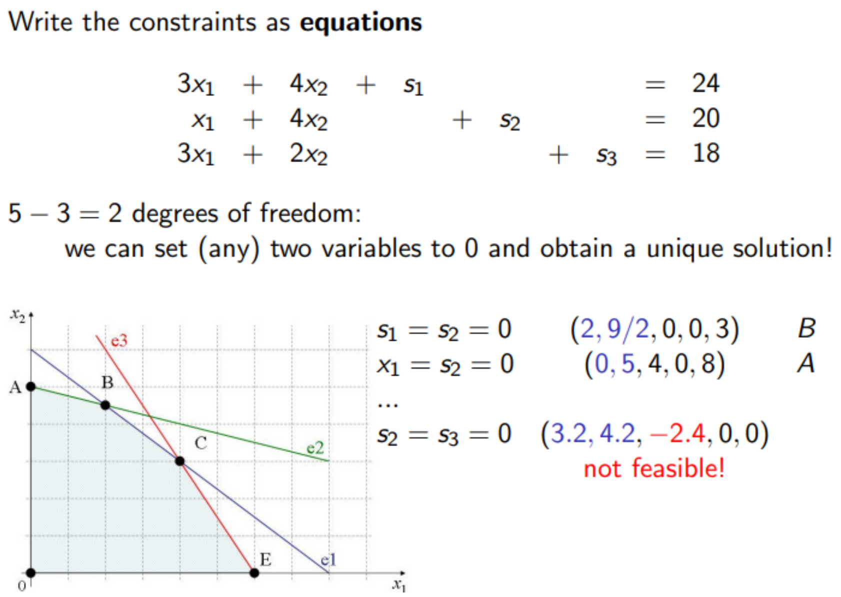
To find a vertex algebraically:

1. Select any two variables to set to zero
2. Solve the resulting system of three equations in three unknowns
3. Verify the solution is feasible (all variables non-negative)

For example:

- Setting $s_1 = s_2 = 0$ gives vertex B at (2, 9/2) with $s_3 = 3$
- Setting $x_1 = s_2 = 0$ gives vertex A at (0, 5) with other variables $s_1 = 4$, $s_3 = 8$
- Setting $s_2 = s_3 = 0$ gives point (3.2, 4.2) with $s_1 = -2.4$, which is not feasible because $s_1 < 0$

This algebraic approach provides a systematic way to identify all vertices of the feasible region by examining different combinations of variables set to zero.



Write the constraints as **equations**

$$\begin{array}{rcrcrcrcrcl} 3x_1 & + & 4x_2 & + & s_1 & & & & & = & 24 \\ x_1 & + & 4x_2 & & & + & s_2 & & & = & 20 \\ 3x_1 & + & 2x_2 & & & & & + & s_3 & = & 18 \end{array}$$

$5 - 3 = 2$ degrees of freedom:
we can set (any) two variables to 0 and obtain a unique solution!

$s_1 = s_2 = 0$	$(2, 9/2, 0, 0, 3)$	B
$x_1 = s_2 = 0$	$(0, 5, 4, 0, 8)$	A
$\dots$		
$s_2 = s_3 = 0$	$(3.2, 4.2, -2.4, 0, 0)$ not feasible!	

Whenever we have a negative slack, the problem is not feasible (not inside of the feasible region).

- Note that this solution corresponds to the vertex B. Infact, place s1 = s2 = 0 means, from a geometrical point of view, saturate the constraints (e1) and (e2): the solution will then be found at the intersection of the corresponding lines. Another particular solution can be obtained by fixing at 0 the variables x1 and s2, which leads to the solution x1 = 0, x2 = 5, s1 = 4, s2 = 0, s3 = 8, corresponding to the vertex A.
- We therefore feel that, among the infinite (∞^{5-3}) solutions of the system of equations equates to the constraints of the problem, there are some particular ones: these solutions are obtained by setting a suitable number of variables to 0 and correspond to vertices of the eligible region.
- Note that the variables to be set at 0 must be appropriately chosen. For example, if x1 = s1 = 0, we get the solution x1 = 0, x2 = 6, s1 = 0, s2 = 4, s3 = 6 which does not correspond to a vertex of the polyhedron: the solution obtained is in fact inadmissible since s2 < 0 indicates that the constraint (e2) is violated.

We try to generalise these observations.

- The first step is to write the constraints of a PL problem in a convenient way as a system of linear equations
- The second step is to manipulate the system of equations in order to derive solutions corresponding to vertices of the allowable polyhedron.

We then introduce the standard form for a PL problem and recall some notations and properties of the linear algebra.

To create a generic approach to use, we consider the standard form for LP problems, with all variables ≥ 0, so to find easily the unfeasible form, with all constraints as equalities:

$$\begin{array}{lllr} \min & c_1x_1 + c_2x_2 + \ldots + c_nx_n & & \\ \text{s.t.} & a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n & = b_i & (i = 1 \ldots m) \\ & x_i \in \mathbb{R}_+ & & (i = 1 \ldots n) \end{array}$$

- **minimizing** objective function (if not, multiply by -1);
- variables ≥ 0; (if not, substitution)
- all constraints are equalities; ($+/-$ slack/surplus variables)
- $b_i \geq 0$. (if not, multiply by -1)

Here:

- The objective function is minimization and without additive or multiplicative constants (multiply the maximizing functions by -1; additive constants can be neglected; positive multiplicative constants may be overlooked, the negative multiplicative constants can be eliminated by changing the direction of optimization)
- All variables are positive or nil (if and where there are substitutions of variables for the free or negative variables)
- All constraints are equations (add a positive slack variable for the $\leq$ constraints and subtract a positive surplus variable for the $\geq$ constraints)
- The known terms b_i are all positive or null (multiply by -1 the constraints with negative constant term)

This allows, without loss of generality (wlog) to solve whatever PL problem via systems of linear equations.

Consider the following example of the standard form, where we use what described above via whatever PL problem using linear equations systems:

$$\begin{array}{ll} \max & 5(-3x_1 + 5x_2 - 7x_3) + 34 \\ s.t. & -2x_1 + 7x_2 + 6x_3 - 2x_1 \leq 5 \\ & -3x_1 + x_3 + 12 \geq 13 \\ & x_1 + x_2 \leq -2 \\ & x_1 \leq 0 \\ & x_2 \geq 0 \end{array}$$

$$\begin{array}{ll} \hat{x}_1 = -x_1 & (\hat{x}_1 \geq 0) \\ x_3 = x_3^+ - x_3^- & (x_3^+ \geq 0 \ , \ x_3^- \geq 0) \end{array}$$

$$\begin{array}{ll} \min & -3\hat{x}_1 - 5x_2 + 7x_3^+ - 7x_3^- \\ s.t. & 4\hat{x}_1 + 7x_2 + 6x_3^+ - 6x_3^- + s_1 = 5 \\ & 3\hat{x}_1 + x_3^+ - x_3^- - s_2 = 1 \\ & \hat{x}_1 - x_2 - s_3 = 2 \\ & \hat{x}_1 \geq 0 \ , \ x_2 \geq 0 \ , \ x_3^+ \geq 0 \ , \ x_3^- \geq 0 \ , \ s_1 \geq 0 \ , \ s_2 \geq 0 \ , \ s_3 \geq 0. \end{array}$$

Here's how we transform the original problem:

Step 1: *Convert Maximization to Minimization*

The objective max $5(-3x_1 + 5x_2 - 7x_3) + 34$ becomes: min $-5(-3x_1 + 5x_2 - 7x_3) - 34$, which simplifies to: min $-15x_1 + 25x_2 - 35x_3 - 34$

Step 2: Unrestricted Variables Handle

For $x_1 \leq 0$: Replace x_1 with $-\hat{x}_1$ where $\hat{x}_1 \geq 0$

For unrestricted x_3: Replace with $x_3 = x_3^+ - x_3^-$ where $x_3^+, x_3^- \geq 0$

Step 3: Convert Inequalities to Equations

Add slack variables (s_1, s_2, s_3) to convert inequalities into equations:

- For ≤ constraints: Add slack variable
- For ≥ constraints: Subtract slack variable

Step 4: The Final Standard Form

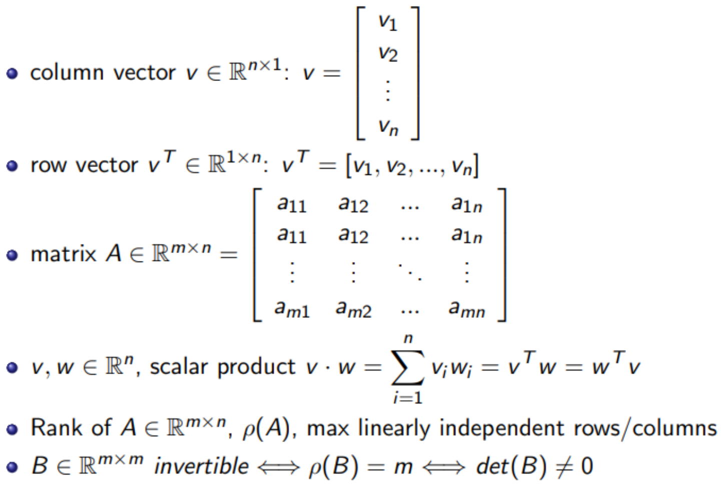
Objective: min $-3\hat{x}_1 - 5x_2 + 7x_3^+ - 7x_3^-$

Subject to: $4\hat{x}_1 + 7x_2 + 6x_3^+ - 6x_3^- + s_1 = 5$ $3\hat{x}_1 + x_3^+ - x_3^- - s_2 = 1$ $\hat{x}_1 - x_2 - s_3 = 2$

Non-negativity: $\hat{x}_1, x_2, x_3^+, x_3^-, s_1, s_2, s_3 \geq 0$

This standard form ensures all variables are non-negative and all constraints are equations, making it suitable for solution methods like the simplex algorithm.

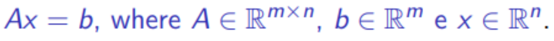
Now, some recalls of linear algebra:



- column vector $v \in \mathbb{R}^{n \times 1}$: $v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$
- row vector $v^T \in \mathbb{R}^{1 \times n}$: $v^T = [v_1, v_2, \dots, v_n]$
- matrix $A \in \mathbb{R}^{m \times n} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$
- $v, w \in \mathbb{R}^n$, scalar product $v \cdot w = \sum_{i=1}^{n} v_i w_i = v^T w = w^T v$
- Rank of $A \in \mathbb{R}^{m \times n}$, $\rho(A)$, max linearly independent rows/columns
- $B \in \mathbb{R}^{m \times m}$ *invertible* $\iff \rho(B) = m \iff det(B) \neq 0$

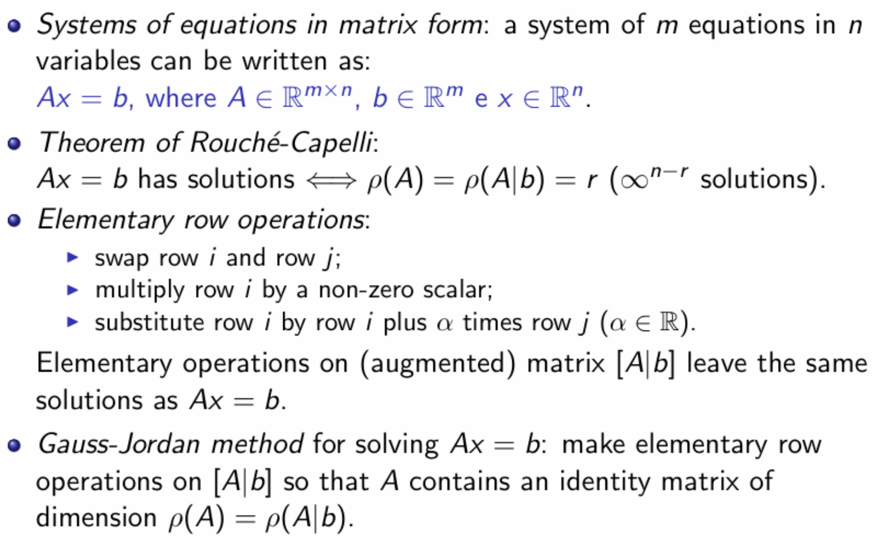
So, basically a system of m linear equations in n variables can be written in matrix form as $Ax = b$, where:

- A is an $m \times n$ matrix containing the coefficients
- x is an n-dimensional vector of variables
- b is an m-dimensional vector of right-hand side values



$Ax = b$, where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ e $x \in \mathbb{R}^n$.

There are different ways on which we calculate solutions for systems of linear equations as seen below:



- *Systems of equations in matrix form*: a system of m equations in n variables can be written as:
  $Ax = b$, where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ e $x \in \mathbb{R}^n$.
- *Theorem of Rouché-Capelli*:
  $Ax = b$ has solutions $\iff \rho(A) = \rho(A|b) = r$ (∞^{n-r} solutions).
- *Elementary row operations*:
  - swap row i and row j;
  - multiply row i by a non-zero scalar;
  - substitute row i by row i plus α times row j ($\alpha \in \mathbb{R}$).

  Elementary operations on (augmented) matrix $[A|b]$ leave the same solutions as $Ax = b$.
- *Gauss-Jordan method* for solving $Ax = b$: make elementary row operations on $[A|b]$ so that A contains an identity matrix of dimension $\rho(A) = \rho(A|b)$.

One way to solve a system of linear equations refers to the concept of base which is present when the matrix has maximum rank (square submatrix $B \in \mathbb{R}^{m \times m}$), obtained by taking m linearly independent columns from the matrix. Having the determinant not null, we could rewrite the system as:

$$Ax = b$$

$$[B|F]\begin{bmatrix} x_B \\ x_F \end{bmatrix} = b$$

$$Bx_B + Fx_F = b$$

We can find the variables present in the basis with:

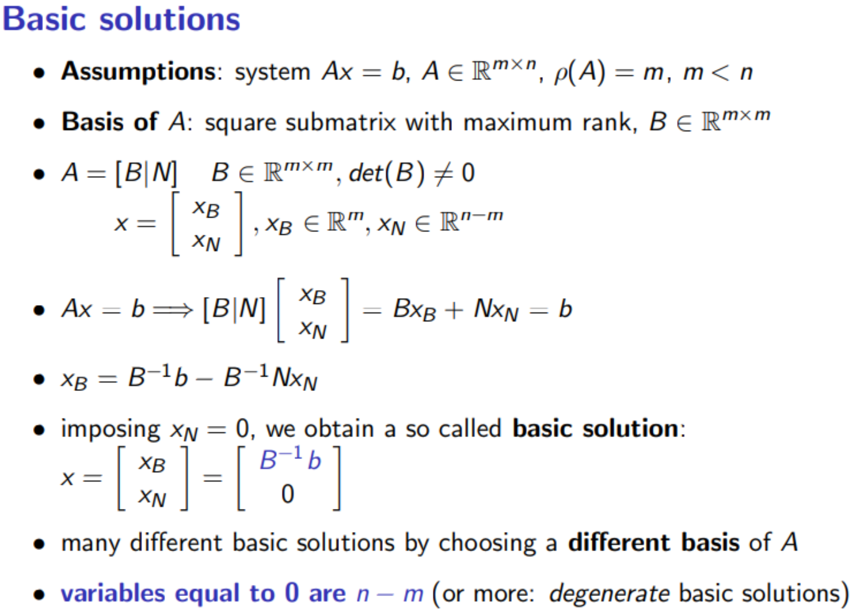
$$x_B = B^{-1}b - B^{-1}Fx_F$$

The variables outside of the base are set to 0 (x_F), we get a basic solution. In a basic solution at least $n - m$ variables are equal to 0 (if more, the basis becomes *degenerate*).

We get the values of what's present in base (x_B) finding a feasible solution when coming back to the original constraints.

$$x_B = B^{-1}b \geq 0$$

Since vertices and basic solutions correspond ($Ax = b \Leftrightarrow x\ is\ a\ P\ vertex$), the solving the linear system brings a polyhedron vertex; to use different bases, one only needs to *change the variables fixed to* 0. Everything said up to know is summarized below:

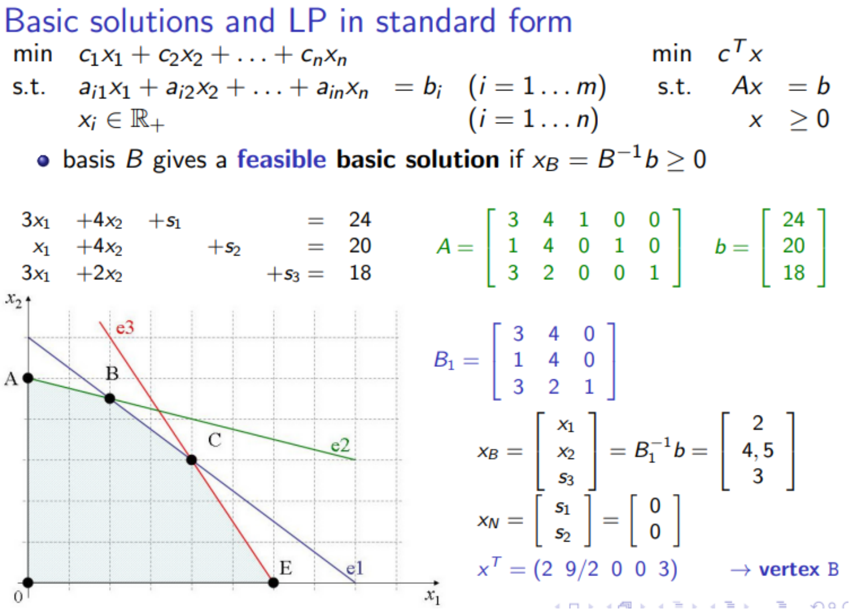


Basic solutions

- **Assumptions**: system $Ax = b$, $A \in \mathbb{R}^{m \times n}$, $\rho(A) = m$, $m < n$
- **Basis of** A: square submatrix with maximum rank, $B \in \mathbb{R}^{m \times m}$
- $A = [B|N]$ $\quad B \in \mathbb{R}^{m \times m}$, $det(B) \neq 0$
  $$x = \begin{bmatrix} x_B \\ x_N \end{bmatrix}, x_B \in \mathbb{R}^m, x_N \in \mathbb{R}^{n-m}$$
- $Ax = b \Longrightarrow [B|N] \begin{bmatrix} x_B \\ x_N \end{bmatrix} = Bx_B + Nx_N = b$
- $x_B = B^{-1}b - B^{-1}Nx_N$
- imposing $x_N = 0$, we obtain a so called **basic solution**:
  $$x = \begin{bmatrix} x_B \\ x_N \end{bmatrix} = \begin{bmatrix} B^{-1}b \\ 0 \end{bmatrix}$$
- many different basic solutions by choosing a **different basis** of A
- **variables equal to 0 are** $n - m$ (or more: *degenerate* basic solutions)

All of the non-basic variables are set to 0 → basic solution.

In a linear program in standard form, we seek to minimize $c^{\mathrm{T}}x$ subject to $Ax = b$ and $x \geq 0$. A basic solution becomes feasible when all basic variables are non-negative. Let's look at an example:



Basic solutions and LP in standard form

$$\begin{array}{llll} \min & c_1x_1 + c_2x_2 + \ldots + c_nx_n & & \\ \text{s.t.} & a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n & = b_i & (i = 1 \ldots m) \\ & x_i \in \mathbb{R}_+ & & (i = 1 \ldots n) \end{array} \qquad \begin{array}{lll} \min & c^Tx & \\ \text{s.t.} & Ax & = b \\ & x & \geq 0 \end{array}$$

- basis B gives a **feasible basic solution** if $x_B = B^{-1}b \geq 0$

$$\begin{array}{rrrrrrl} 3x_1 & +4x_2 & +s_1 & & & = & 24 \\ x_1 & +4x_2 & & +s_2 & & = & 20 \\ 3x_1 & +2x_2 & & & +s_3 & = & 18 \end{array} \qquad A = \begin{bmatrix} 3 & 4 & 1 & 0 & 0 \\ 1 & 4 & 0 & 1 & 0 \\ 3 & 2 & 0 & 0 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 24 \\ 20 \\ 18 \end{bmatrix}$$

$$B_1 = \begin{bmatrix} 3 & 4 & 0 \\ 1 & 4 & 0 \\ 3 & 2 & 1 \end{bmatrix}$$

$$x_B = \begin{bmatrix} x_1 \\ x_2 \\ s_3 \end{bmatrix} = B_1^{-1}b = \begin{bmatrix} 2 \\ 4,5 \\ 3 \end{bmatrix}$$

$$x_N = \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$x^T = (2\ \ 9/2\ \ 0\ \ 0\ \ 3)$ → **vertex** B

Looking at the example provided:

- We have three equations with five variables (x_1, x_2, s_1, s_2, s_3), where s_1, s_2, s_3 are slack variables: $3x_1 + 4x_2 + s_1 = 24$ $x_1 + 4x_2 + s_2 = 20$ $3x_1 + 2x_2 + s_3 = 18$

- Coefficient matrix A is shown as a 3×5 matrix containing both the original coefficients and the identity matrix corresponding to slack variables

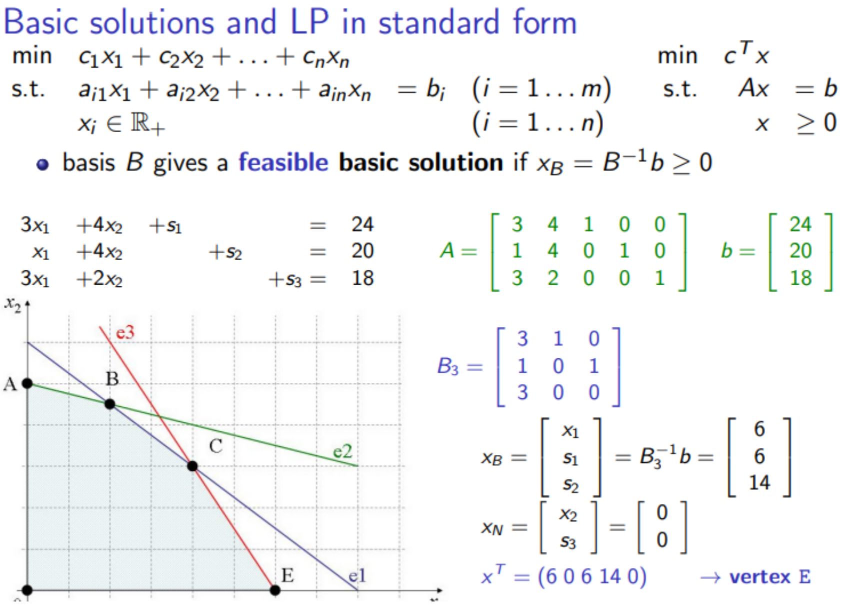
- To find a basic solution, we select a basis matrix B_1 composed of three columns from A. In this example, B_1 is formed by columns 1, 2, and 5 of A, corresponding to variables x_1, x_2, and s_3

- The basic variables xB are computed as B_1^{-1}b: xB = $[x_1, x_2, s_3]^T$ = $[2, 4.5, 3]^T$

- The non-basic variables xN are set to zero: xN = $[s_1, s_2]^T$ = $[0, 0]^T$

- This gives us the complete solution vector: x = [2, 4.5, 0, 0, 3]

This basic solution is feasible because all components are non-negative. Geometrically, this solution corresponds to vertex B in the feasible region shown in the graph. The same continues for one another iteration:



Basic solutions and LP in standard form

$$\begin{array}{lllll} \min & c_1x_1 + c_2x_2 + \ldots + c_nx_n & & & \\ \text{s.t.} & a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n & = b_i & (i = 1 \ldots m) \\ & x_i \in \mathbb{R}_+ & & (i = 1 \ldots n) \end{array} \qquad \begin{array}{lll} \min & c^Tx & \\ \text{s.t.} & Ax & = b \\ & x & \geq 0 \end{array}$$

- basis B gives a **feasible basic solution** if $x_B = B^{-1}b \geq 0$

$$\begin{array}{rrrrrl} 3x_1 & +4x_2 & +s_1 & & & = 24 \\ x_1 & +4x_2 & & +s_2 & & = 20 \\ 3x_1 & +2x_2 & & & +s_3 & = 18 \end{array} \qquad A = \begin{bmatrix} 3 & 4 & 1 & 0 & 0 \\ 1 & 4 & 0 & 1 & 0 \\ 3 & 2 & 0 & 0 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 24 \\ 20 \\ 18 \end{bmatrix}$$

$$B_3 = \begin{bmatrix} 3 & 1 & 0 \\ 1 & 0 & 1 \\ 3 & 0 & 0 \end{bmatrix}$$

$$x_B = \begin{bmatrix} x_1 \\ s_1 \\ s_2 \end{bmatrix} = B_3^{-1}b = \begin{bmatrix} 6 \\ 6 \\ 14 \end{bmatrix}$$

$$x_N = \begin{bmatrix} x_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$x^T = (6\ 0\ 6\ 14\ 0)$ → vertex E

In this example, we can see that the chosen basis B_4 leads to a basic solution that is not feasible.

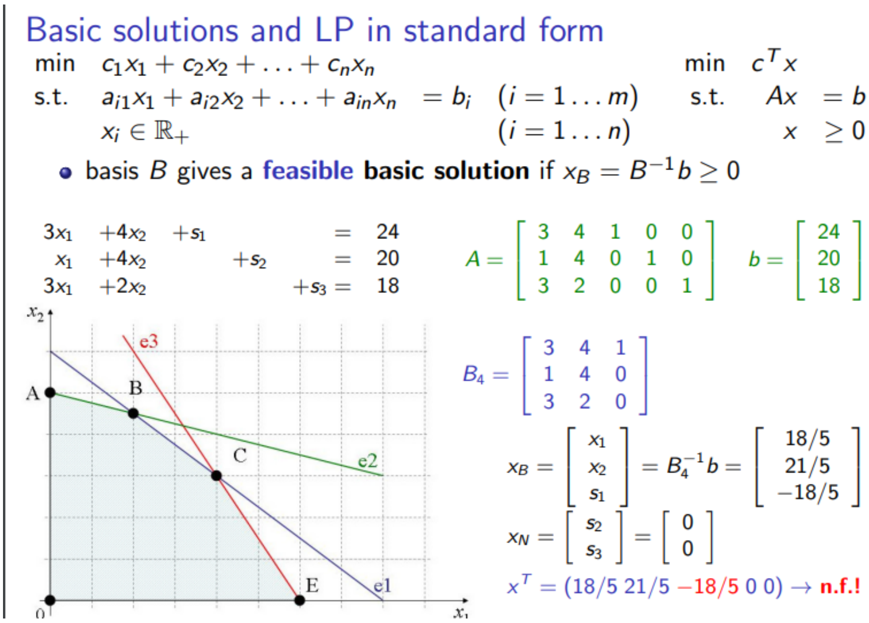
When we calculate xB = B_4^{-1}b, we get:

- x_1 = 18/5
- x_2 = 21/5
- s_1 = -18/5

The issue lies with the value of s_1. In linear programming, all variables (including slack variables) must be non-negative due to the constraint x ≥ 0 in standard form. However, s_1 = -18/5 is negative, violating this non-negativity requirement.

This illustrates an important principle in linear programming: while a basis B may be mathematically valid (in that B is invertible and we can compute B^{-1}b), the resulting basic solution is only feasible if all components of xB = B^{-1}b are non-negative. When any component is negative, as in this case, we say the basic solution is infeasible.

Geometrically, this means that while the intersection of the chosen constraints does define a point in space (18/5, 21/5), this point lies outside the feasible region of our linear program because it violates the non-negativity requirement for slack variables.



Basic solutions and LP in standard form

$$\begin{array}{llll} \min & c_1x_1 + c_2x_2 + \ldots + c_nx_n & & \\ \text{s.t.} & a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n & = b_i & (i = 1 \ldots m) \\ & x_i \in \mathbb{R}_+ & & (i = 1 \ldots n) \end{array} \qquad \begin{array}{lll} \min & c^T x & \\ \text{s.t.} & Ax & = b \\ & x & \geq 0 \end{array}$$

- basis B gives a **feasible basic solution** if $x_B = B^{-1}b \geq 0$

$$\begin{array}{llllll} 3x_1 & +4x_2 & +s_1 & & & = 24 \\ x_1 & +4x_2 & & +s_2 & & = 20 \\ 3x_1 & +2x_2 & & & +s_3 & = 18 \end{array} \qquad A = \begin{bmatrix} 3 & 4 & 1 & 0 & 0 \\ 1 & 4 & 0 & 1 & 0 \\ 3 & 2 & 0 & 0 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 24 \\ 20 \\ 18 \end{bmatrix}$$

$$B_4 = \begin{bmatrix} 3 & 4 & 1 \\ 1 & 4 & 0 \\ 3 & 2 & 0 \end{bmatrix}$$

$$x_B = \begin{bmatrix} x_1 \\ x_2 \\ s_1 \end{bmatrix} = B_4^{-1}b = \begin{bmatrix} 18/5 \\ 21/5 \\ -18/5 \end{bmatrix}$$

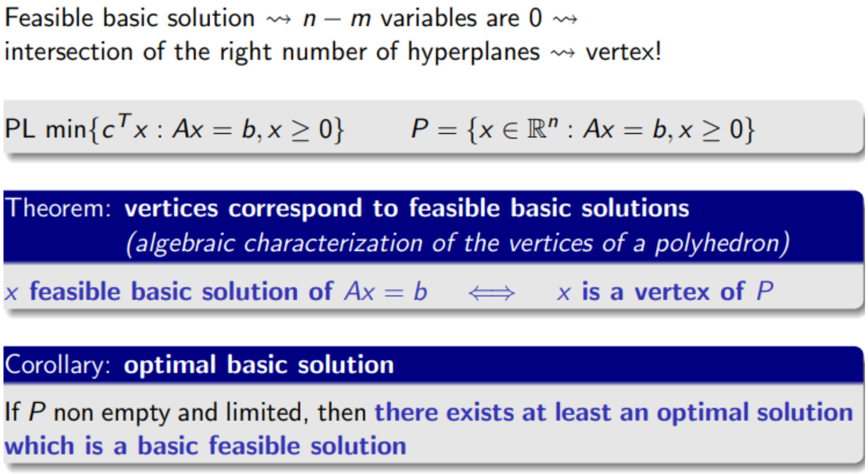
$$x_N = \begin{bmatrix} s_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$x^T = (18/5\ 21/5\ -18/5\ 0\ 0) \rightarrow$ **n.f.!**

This relationship is captured in a key theorem that provides an algebraic characterization of polyhedron vertices. This equivalence has several important implications.

- First, it connects the geometric concept of vertices (intersections of the right number of hyperplanes) with the algebraic concept of basic feasible solutions (where $n - m$ variables are zero). This provides two complementary ways to understand and work with optimal solutions.

- Second, this relationship leads to a crucial corollary about optimal solutions: if the feasible region P is non-empty and bounded, then there exists at least one optimal solution that is a basic feasible solution. This corollary is fundamental to linear programming because it tells us we can restrict our search for optimal solutions to the vertices of the feasible region.



Feasible basic solution $\rightsquigarrow$ $n - m$ variables are 0 $\rightsquigarrow$
intersection of the right number of hyperplanes $\rightsquigarrow$ vertex!

PL $\min\{c^T x : Ax = b, x \geq 0\}$ $\quad P = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$

Theorem: **vertices correspond to feasible basic solutions**
(algebraic characterization of the vertices of a polyhedron)

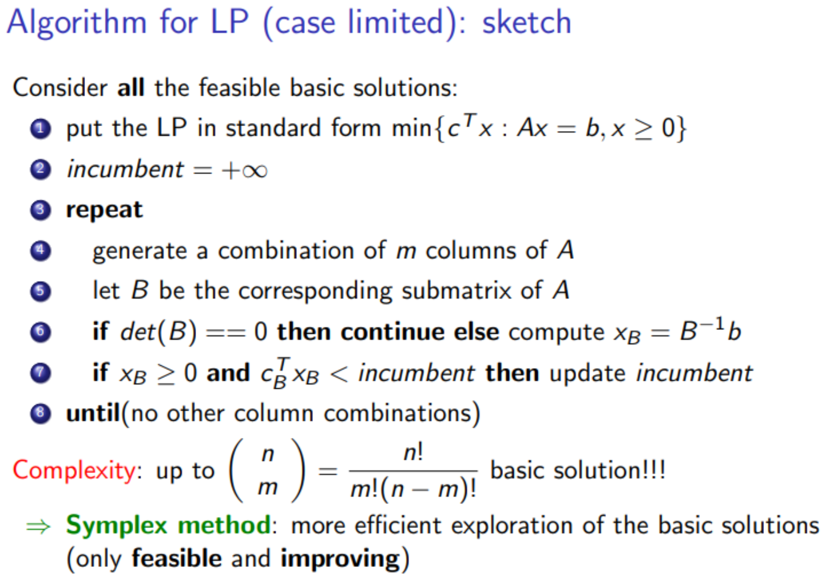
x feasible basic solution of $Ax = b$ $\iff$ x is a vertex of P

Corollary: **optimal basic solution**

If P non empty and limited, then **there exists at least an optimal solution which is a basic feasible solution**

5.3 Simplex Basic Algorithm and Example

The following algorithm explores all possible basic feasible solutions, which can be done efficiently using this approach. The complexity is up to exponentiality, but the Simplex method provides a more efficient way to explore the feasible solutions, considering only the improving ones.



Algorithm for LP (case limited): sketch

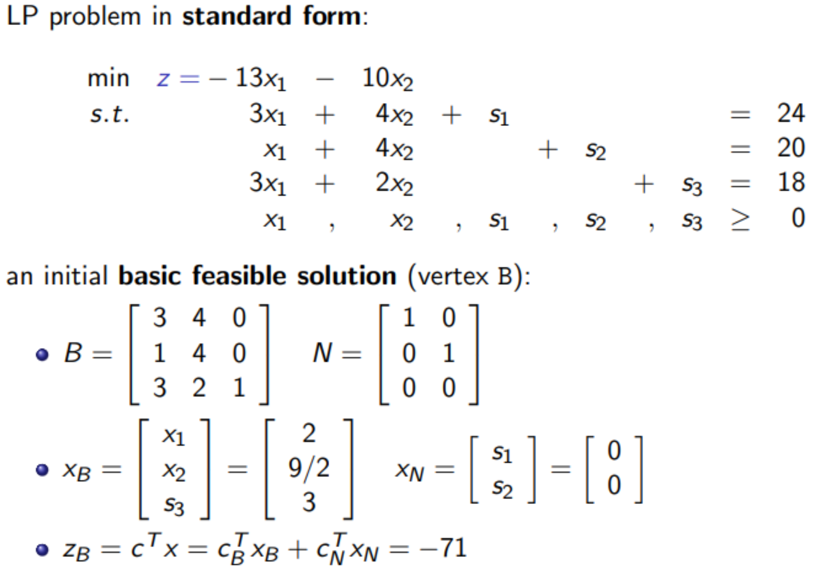
Consider **all** the feasible basic solutions:

```
put the LP in standard form $\min\{c^T x : Ax = b, x \geq 0\}$
$incumbent = +\infty$
repeat
    generate a combination of $m$ columns of $A$
    let $B$ be the corresponding submatrix of $A$
    if $det(B) == 0$ then continue else compute $x_B = B^{-1}b$
    if $x_B \geq 0$ and $c_B^T x_B < incumbent$ then update $incumbent$
until(no other column combinations)
```

Complexity: up to $\left(\begin{array}{c} n \\ m \end{array}\right) = \frac{n!}{m!(n-m)!}$ basic solution!!!

⇒ **Symplex method**: more efficient exploration of the basic solutions (only **feasible** and **improving**)

To exploit this idea is to change the basic variables (take a column inside of the basis and exchange columns between each basis):



LP problem in **standard form**:

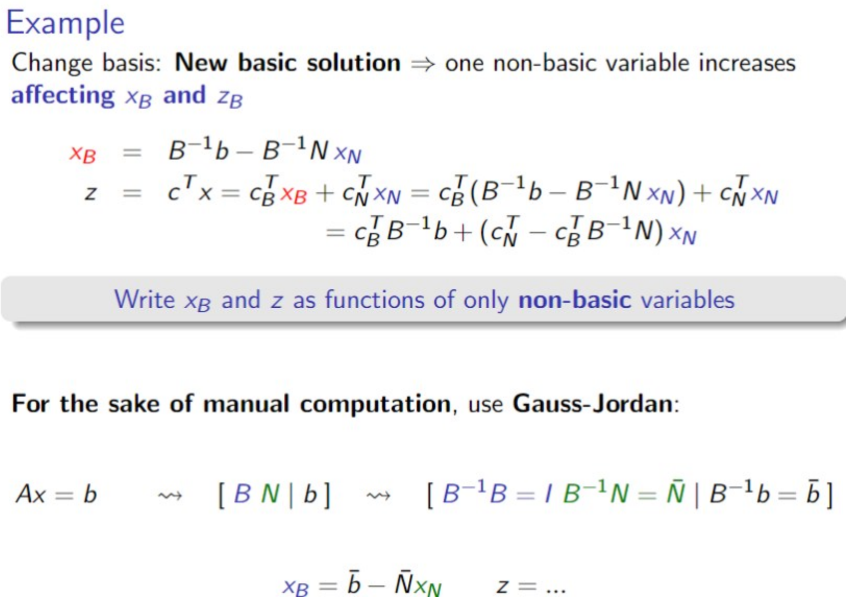
$$\begin{array}{lrcrcrcrcrcl}
\min & z = -13x_1 & - & 10x_2 & & & & & & & & \\
s.t. & 3x_1 & + & 4x_2 & + & s_1 & & & & & = & 24 \\
 & x_1 & + & 4x_2 & & & + & s_2 & & & = & 20 \\
 & 3x_1 & + & 2x_2 & & & & & + & s_3 & = & 18 \\
 & x_1 & , & x_2 & , & s_1 & , & s_2 & , & s_3 & \geq & 0
\end{array}$$

an initial **basic feasible solution** (vertex B):

- $B = \begin{bmatrix} 3 & 4 & 0 \\ 1 & 4 & 0 \\ 3 & 2 & 1 \end{bmatrix} \quad N = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$
- $x_B = \begin{bmatrix} x_1 \\ x_2 \\ s_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 9/2 \\ 3 \end{bmatrix} \quad x_N = \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
- $z_B = c^T x = c_B^T x_B + c_N^T x_N = -71$

When the basis changes, one non-basic variable increases, affecting the values of the basic variables and the objective function value. The objective function contains only non-basic variables, and base variables are expressed only in terms of non-basic variables.

This can be expressed mathematically as follows:



Example

Change basis: **New basic solution** ⇒ one non-basic variable increases **affecting x_B and z_B**

$$\begin{aligned} x_B &= B^{-1}b - B^{-1}N\,x_N \\ z &= c^Tx = c_B^Tx_B + c_N^Tx_N = c_B^T(B^{-1}b - B^{-1}N\,x_N) + c_N^Tx_N \\ &\qquad\qquad\qquad\qquad = c_B^TB^{-1}b + (c_N^T - c_B^TB^{-1}N)\,x_N \end{aligned}$$

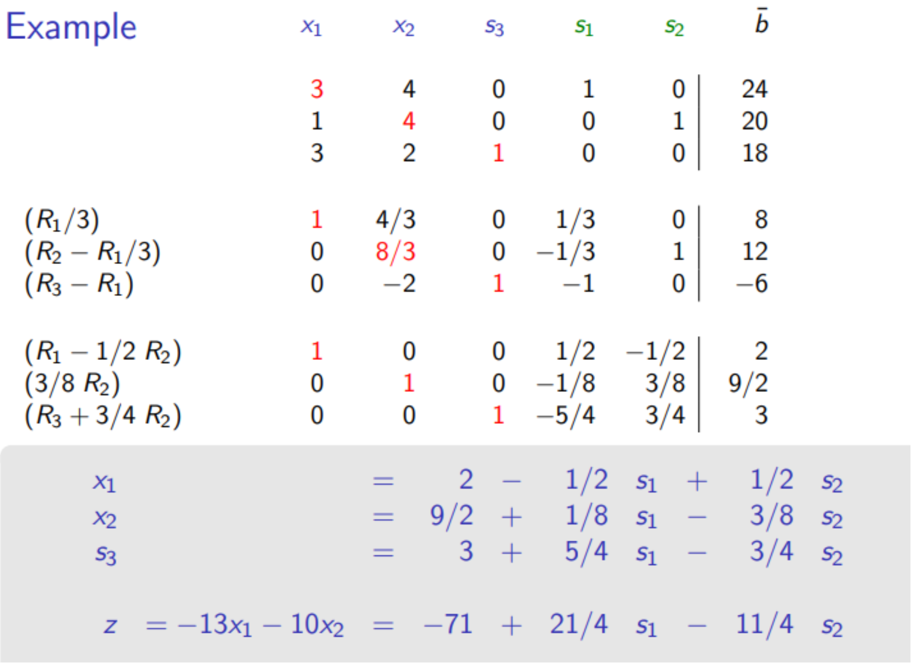
Write x_B and z as functions of only **non-basic** variables

For the sake of manual computation, use **Gauss-Jordan**:

$$Ax = b \quad \rightsquigarrow \quad [\,B\ N \mid b\,] \quad \rightsquigarrow \quad [\,B^{-1}B = I\ B^{-1}N = \bar{N} \mid B^{-1}b = \bar{b}\,]$$

$$x_B = \bar{b} - \bar{N}x_N \qquad z = \ldots$$

You see how Gauss-Jordan (row/column operations done algebraically) applies here:



Example

	x_1	x_2	s_3	s_1	s_2	$\bar{b}$
	3	4	0	1	0	24
	1	4	0	0	1	20
	3	2	1	0	0	18
$(R_1/3)$	1	$4/3$	0	$1/3$	0	8
$(R_2 - R_1/3)$	0	$8/3$	0	$-1/3$	1	12
$(R_3 - R_1)$	0	-2	1	-1	0	-6
$(R_1 - 1/2\ R_2)$	1	0	0	$1/2$	$-1/2$	2
$(3/8\ R_2)$	0	1	0	$-1/8$	$3/8$	$9/2$
$(R_3 + 3/4\ R_2)$	0	0	1	$-5/4$	$3/4$	3

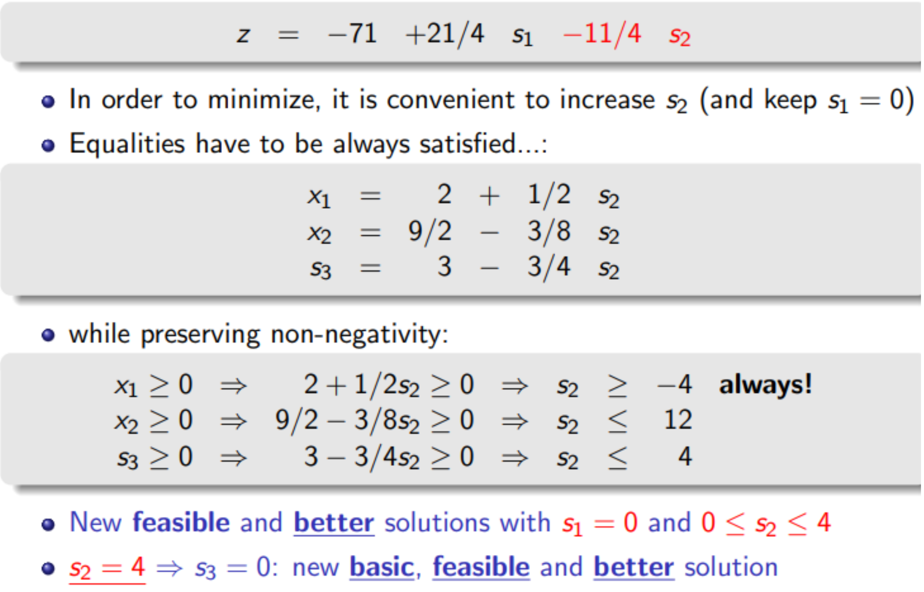
$$\begin{aligned} x_1 &= 2 - 1/2\ s_1 + 1/2\ s_2 \\ x_2 &= 9/2 + 1/8\ s_1 - 3/8\ s_2 \\ s_3 &= 3 + 5/4\ s_1 - 3/4\ s_2 \\ z = -13x_1 - 10x_2 &= -71 + 21/4\ s_1 - 11/4\ s_2 \end{aligned}$$

At each iteration, we want to modify a basic solution in a linear programming problem to achieve a better objective value, while maintaining feasibility and satisfying the problem constraints.

The key steps in this example are:

1. Identify the objective function and the equality constraints that must be satisfied
2. Recognize the opportunity to increase the value of s2 to improve the objective function, while maintaining feasibility.
3. Derive the novel solutions by expressing the basic variables (x1, x2, s2) in terms of the non-basic variable s2.
4. Determine the feasible range for s2 that preserves non-negativity.

5. Identify the new optimal basic solution when s2 = 4.



$$z = -71 + 21/4 \; s_1 - 11/4 \; s_2$$

- In order to minimize, it is convenient to increase s_2 (and keep $s_1 = 0$)
- Equalities have to be always satisfied...:

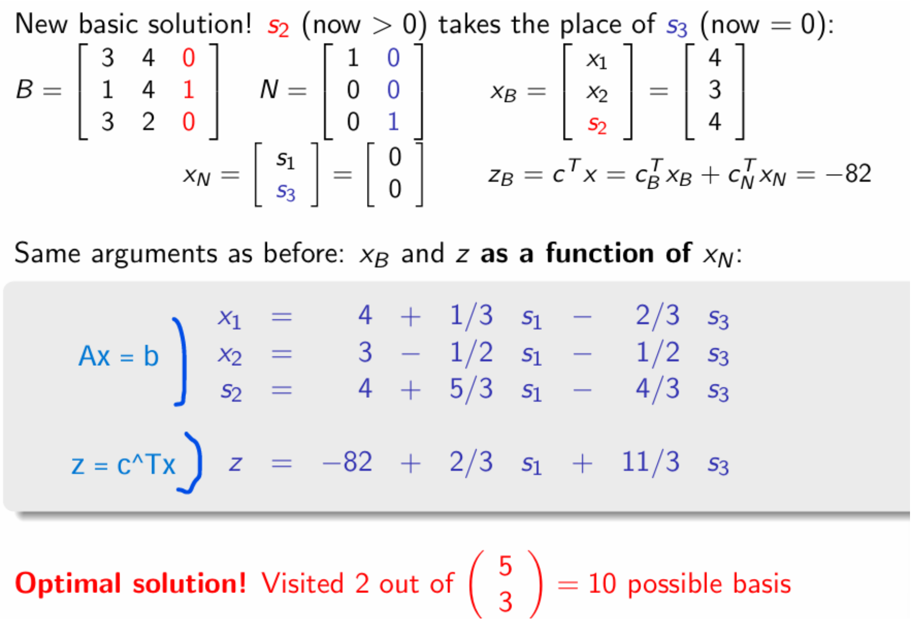
$$\begin{array}{rcrcrl} x_1 & = & 2 & + & 1/2 & s_2 \\ x_2 & = & 9/2 & - & 3/8 & s_2 \\ s_3 & = & 3 & - & 3/4 & s_2 \end{array}$$

- while preserving non-negativity:

$$\begin{array}{lcccccl} x_1 \geq 0 & \Rightarrow & 2 + 1/2 s_2 \geq 0 & \Rightarrow & s_2 & \geq & -4 \quad \textbf{always!} \\ x_2 \geq 0 & \Rightarrow & 9/2 - 3/8 s_2 \geq 0 & \Rightarrow & s_2 & \leq & 12 \\ s_3 \geq 0 & \Rightarrow & 3 - 3/4 s_2 \geq 0 & \Rightarrow & s_2 & \leq & 4 \end{array}$$

- New **feasible** and **better** solutions with $s_1 = 0$ and $0 \leq s_2 \leq 4$
- $s_2 = 4 \Rightarrow s_3 = 0$: new **basic**, **feasible** and **better** solution

Inside of the feasible region, it is impossible to obtain better value to the optimal solution value (z) inside of the base (basic solution), expressed in terms of the non-basic variables (so to understand up to which limit we enter the basis):



New basic solution! s_2 (now > 0) takes the place of s_3 (now $= 0$):

$$B = \begin{bmatrix} 3 & 4 & 0 \\ 1 & 4 & 1 \\ 3 & 2 & 0 \end{bmatrix} \quad N = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \quad x_B = \begin{bmatrix} x_1 \\ x_2 \\ s_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \\ 4 \end{bmatrix}$$

$$x_N = \begin{bmatrix} s_1 \\ s_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad z_B = c^T x = c_B^T x_B + c_N^T x_N = -82$$

Same arguments as before: x_B and z **as a function of** x_N:

Ax = b

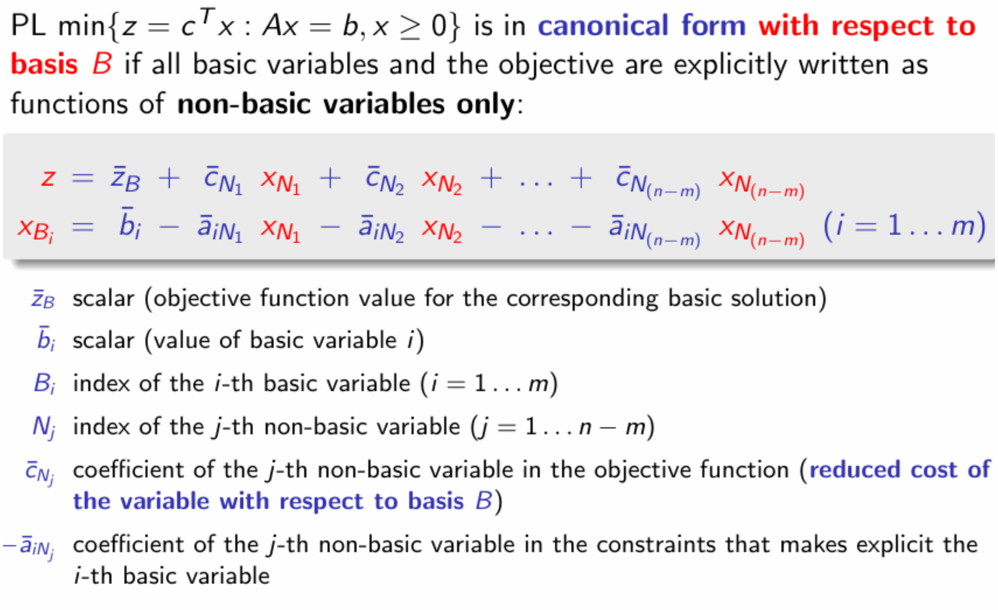
$$\begin{array}{rcrcrcrl} x_1 & = & 4 & + & 1/3 & s_1 & - & 2/3 \; s_3 \\ x_2 & = & 3 & - & 1/2 & s_1 & - & 1/2 \; s_3 \\ s_2 & = & 4 & + & 5/3 & s_1 & - & 4/3 \; s_3 \end{array}$$

z = c^Tx

$$z = -82 + 2/3 \; s_1 + 11/3 \; s_3$$

Optimal solution! Visited 2 out of $\binom{5}{3} = 10$ possible basis

The LP problem will be expressed in a canonical form with respect to a specific basis, combining linearly all non-basic variables with coefficients:



PL $\min\{z = c^T x : Ax = b, x \geq 0\}$ is in **canonical form with respect to basis** B if all basic variables and the objective are explicitly written as functions of **non-basic variables only**:

$$\begin{array}{rcl} z & = & \bar{z}_B + \bar{c}_{N_1} x_{N_1} + \bar{c}_{N_2} x_{N_2} + \ldots + \bar{c}_{N_{(n-m)}} x_{N_{(n-m)}} \\ x_{B_i} & = & \bar{b}_i - \bar{a}_{iN_1} x_{N_1} - \bar{a}_{iN_2} x_{N_2} - \ldots - \bar{a}_{iN_{(n-m)}} x_{N_{(n-m)}} \quad (i = 1 \ldots m) \end{array}$$

$\bar{z}_B$ scalar (objective function value for the corresponding basic solution)

$\bar{b}_i$ scalar (value of basic variable i)

B_i index of the i-th basic variable ($i = 1 \ldots m$)

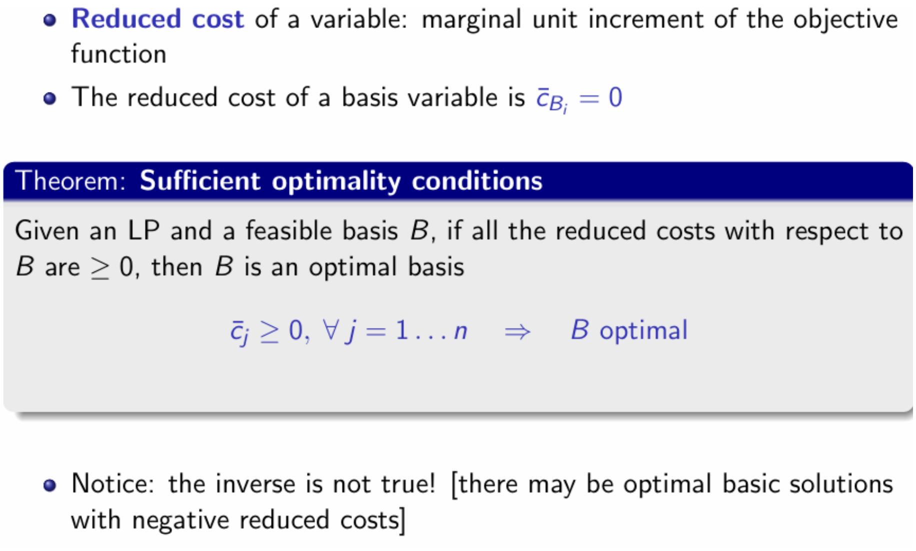
N_j index of the j-th non-basic variable ($j = 1 \ldots n - m$)

$\bar{c}_{N_j}$ coefficient of the j-th non-basic variable in the objective function (**reduced cost of the variable with respect to basis** B)

$-\bar{a}_{iN_j}$ coefficient of the j-th non-basic variable in the constraints that makes explicit the i-th basic variable

Each linear variable is written in the form of non-basic variables and vice versa, where each variable will increase or decrease according to the current value of the o.f.

We start from a feasible solution, and we put a system in a canonical form (system/function) with respect to a given basis. If all the reduced costs cannot be improved they are all positive and this is the optimality check; we stop when they are all positive:



- **Reduced cost** of a variable: marginal unit increment of the objective function
- The reduced cost of a basis variable is $\bar{c}_{B_i} = 0$

Theorem: **Sufficient optimality conditions**

Given an LP and a feasible basis B, if all the reduced costs with respect to B are ≥ 0, then B is an optimal basis

$$\bar{c}_j \geq 0, \ \forall\, j = 1 \ldots n \quad \Rightarrow \quad B \text{ optimal}$$

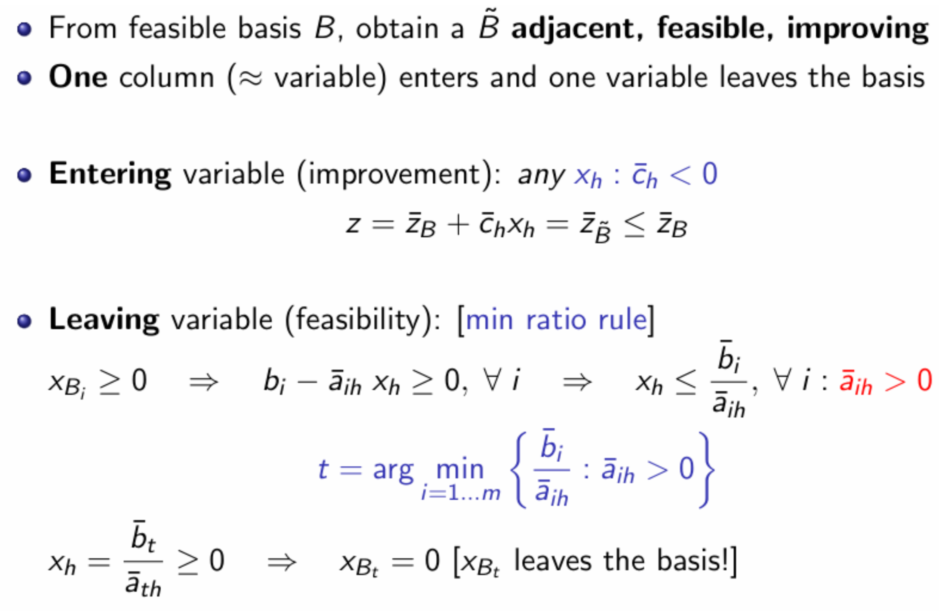
- Notice: the inverse is not true! [there may be optimal basic solutions with negative reduced costs]

The condition is sufficient though; for example, to see if we can improve/reduce the cost of the objective function we welcome variables with negative reduced costs.

The basis change is a fundamental operation in the simplex method that allows the algorithm to move from one basic feasible solution to another while potentially improving the objective function value.

This process involves two key steps:

1. The entering variable is chosen to potentially improve the objective function. Mathematically, this means selecting a variable x_h such that its reduced cost c_h is negative, keeping the problem feasible
2. The leaving variable is selected to maintain feasibility through the "minimum ratio rule". This ensures that no basic variable becomes negative during the transformation



- From feasible basis B, obtain a $\tilde{B}$ **adjacent, feasible, improving**
- **One** column ($\approx$ variable) enters and one variable leaves the basis

- **Entering** variable (improvement): *any* $x_h : \bar{c}_h < 0$

$$z = \bar{z}_B + \bar{c}_h x_h = \bar{z}_{\tilde{B}} \leq \bar{z}_B$$

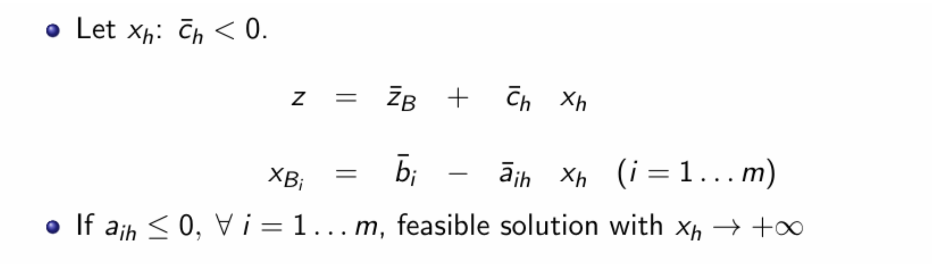
- **Leaving** variable (feasibility): [min ratio rule]

$$x_{B_i} \geq 0 \quad \Rightarrow \quad b_i - \bar{a}_{ih}\, x_h \geq 0,\ \forall\, i \quad \Rightarrow \quad x_h \leq \frac{\bar{b}_i}{\bar{a}_{ih}},\ \forall\, i : \bar{a}_{ih} > 0$$

$$t = \arg \min_{i=1\ldots m} \left\{ \frac{\bar{b}_i}{\bar{a}_{ih}} : \bar{a}_{ih} > 0 \right\}$$

$$x_h = \frac{\bar{b}_t}{\bar{a}_{th}} \geq 0 \quad \Rightarrow \quad x_{B_t} = 0 \ [x_{B_t} \text{ leaves the basis!}]$$

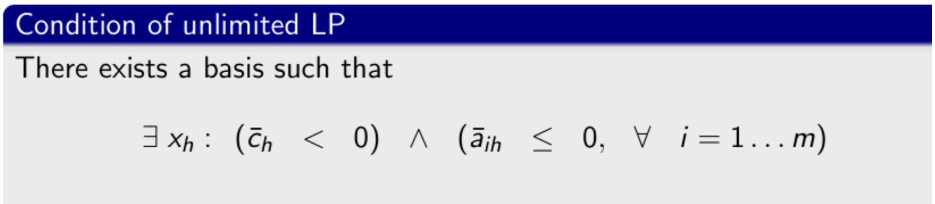
If there is strictly negative reduced costs and the coefficients related to the variables are non-positive, then the problem is considered unlimited:



- Let x_h: $\bar{c}_h < 0$.

$$\begin{aligned} z &= \bar{z}_B + \bar{c}_h \ x_h \\ x_{B_i} &= \bar{b}_i - \bar{a}_{ih} \ x_h \quad (i = 1 \ldots m) \end{aligned}$$

- If $a_{ih} \leq 0,\ \forall\ i = 1 \ldots m$, feasible solution with $x_h \to +\infty$



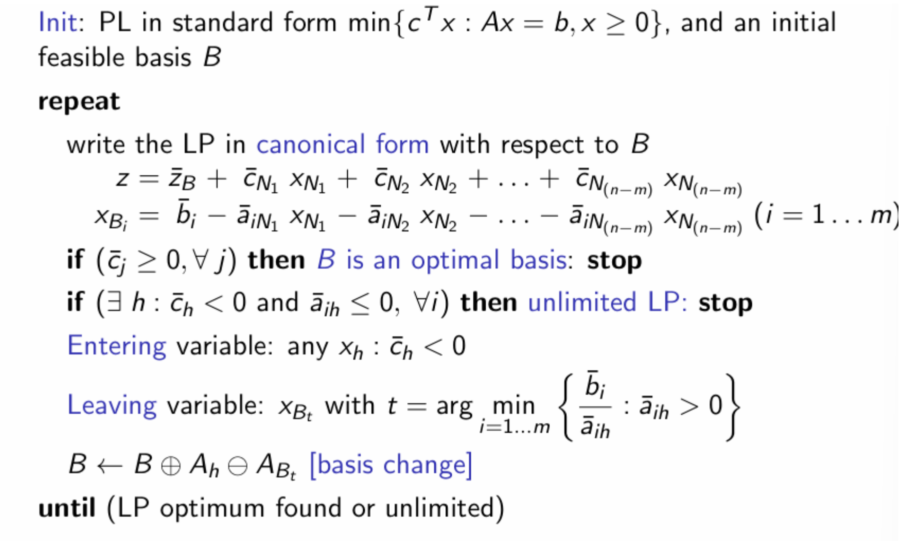
Condition of unlimited LP

There exists a basis such that

$$\exists\, x_h : \ (\bar{c}_h \ < \ 0) \ \wedge \ (\bar{a}_{ih} \ \leq \ 0, \ \forall \ \ i = 1 \ldots m)$$

The simplex method is a systematic geometric approach to solving linear programming problems by systematically exploring the vertices of a polyhedron defined by linear constraints. Its core strategy involves transforming the problem into a standardized form and strategically moving between basic feasible solutions to optimize the objective function.

The algorithm follows this schema seen below:

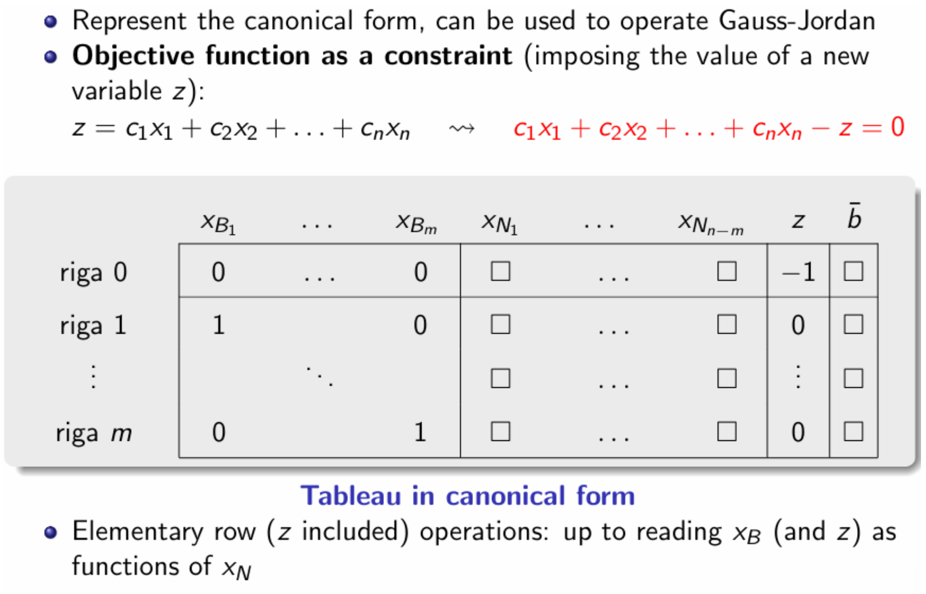


```
Init: PL in standard form $\min\{c^T x : Ax = b, x \geq 0\}$, and an initial feasible basis $B$
repeat
   write the LP in canonical form with respect to $B$
      $z = \bar{z}_B + \bar{c}_{N_1}\, x_{N_1} + \bar{c}_{N_2}\, x_{N_2} + \ldots + \bar{c}_{N_{(n-m)}}\, x_{N_{(n-m)}}$
      $x_{B_i} = \bar{b}_i - \bar{a}_{iN_1}\, x_{N_1} - \bar{a}_{iN_2}\, x_{N_2} - \ldots - \bar{a}_{iN_{(n-m)}}\, x_{N_{(n-m)}}$ $(i = 1 \ldots m)$
   if $(\bar{c}_j \geq 0, \forall\, j)$ then $B$ is an optimal basis: stop
   if $(\exists\, h : \bar{c}_h < 0$ and $\bar{a}_{ih} \leq 0,\ \forall i)$ then unlimited LP: stop
   Entering variable: any $x_h : \bar{c}_h < 0$
   Leaving variable: $x_{B_t}$ with $t = \arg\min_{i=1\ldots m} \left\{ \frac{\bar{b}_i}{\bar{a}_{ih}} : \bar{a}_{ih} > 0 \right\}$
   $B \leftarrow B \oplus A_h \ominus A_{B_t}$ [basis change]
until (LP optimum found or unlimited)
```

In summary:

1. Choose the variable to enter the base, so to find an adjacent base and a feasible solution
2. Choose the variable to exit the base: use the minimum ratio rule
3. Change the base to converge to optimum
4. When all reduced costs are non-negative, stop; but if all reduced costs are negative, problem is unlimited

Usually, there is a "human-readable" form so to represent the simplex operations, in the form of an augmented matrix, which is the simplex tableau:



- Represent the canonical form, can be used to operate Gauss-Jordan
- **Objective function as a constraint** (imposing the value of a new variable z):
  $z = c_1x_1 + c_2x_2 + \ldots + c_nx_n \quad \rightsquigarrow \quad c_1x_1 + c_2x_2 + \ldots + c_nx_n - z = 0$

	x_{B_1}	$\ldots$	x_{B_m}	x_{N_1}	$\ldots$	$x_{N_{n-m}}$	z	$\bar{b}$
riga 0	0	$\ldots$	0	□	$\ldots$	□	-1	□
riga 1	1		0	□	$\ldots$	□	0	□
$\vdots$		$\ddots$		□	$\ldots$	□	$\vdots$	□
riga m	0		1	□	$\ldots$	□	0	□

Tableau in canonical form

- Elementary row (z included) operations: up to reading x_B (and z) as functions of x_N

Recalling that the tableau is a schematized form of the canonical form for a linear programming problem, we note that:

- The last column of the table shows the solution of the problem compared to the current base: the value of the variables in base and, in the first row, the opposite of the value the objective function

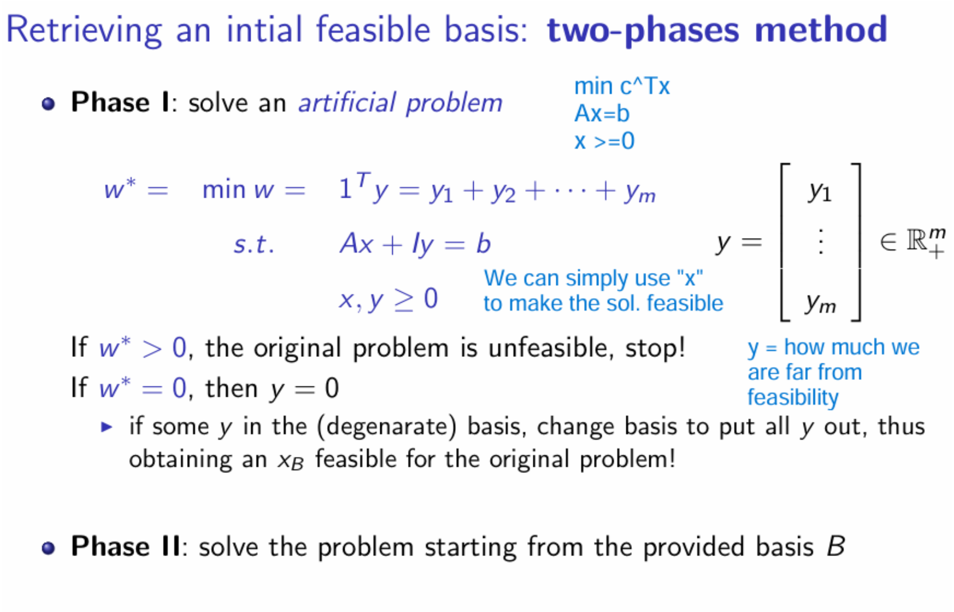
- The columns of the variables in base correspond (if properly ordered) to the identity matrix surmounted by a line of 0 (the reduced costs of the variables in base)
- The columns of the out-of-base variables correspond to the coefficients of the canonical form (where they are preceded by the minus sign) and, in the first row, show the reduced costs

The simplex method simply goes on and terminates when all reduced costs are non-negative. Problem is, if entering variable is not selected carefully, the method might loop encountering an already visited solution.

5.4 Two-Phase Method

The two-phase method is a systematic approach to finding an initial feasible basis or find out if the problem is not feasible when it is not immediately apparent. This technique ensures that we start with a valid starting point for the simplex method.

- Construct an *auxiliary optimization problem* designed to find a feasible initial basis for the original linear programming problem, while keeping it feasible (sometimes harder than the actual problem), using y as base (called vector of artificial variables)
- Solve the original LP problem using the simplex method, then use the initial feasible basis as the starting point, in order to make the problem tractable



Retrieving an intial feasible basis: **two-phases method**

- **Phase I**: solve an *artificial problem*

min c^Tx
Ax=b
x >=0

$$w^* = \min w = 1^T y = y_1 + y_2 + \cdots + y_m \qquad s.t. \quad Ax + Iy = b, \quad x, y \geq 0 \qquad y = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \in \mathbb{R}^m_+$$

We can simply use "x" to make the sol. feasible

If $w^* > 0$, the original problem is unfeasible, stop!

If $w^* = 0$, then $y = 0$

- if some y in the (degenarate) basis, change basis to put all y out, thus obtaining an x_B feasible for the original problem!

y = how much we are far from feasibility

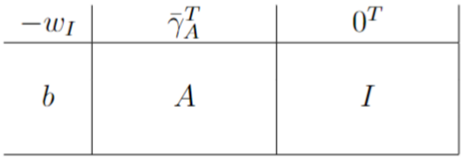
- **Phase II**: solve the problem starting from the provided basis B

Phase 1: Artificial problem solution

So, starting from the artificial problem, the simplex method can be used to remove all artificial variables and keep the problem solvable in some way. The following is the artificial problem:

$$\begin{aligned} w^* = \quad \min w = \quad & 1^T y = y_1 + y_2 + \ldots + y_m \\ s.t. \quad & Ax + Iy = b \qquad\qquad y \in \mathbb{R}^m_+ \qquad y = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \\ & x, y \geq 0 \end{aligned}$$

To change to the tableau (canonical) form, operations are needed on the first row, to turn the 1s into 0s and obtain, instead of 0s, the reduced costs of the variables out of base x with respect to base y.



$-w_I$	$\bar{\gamma}_A^T$	0^T
b	A	I

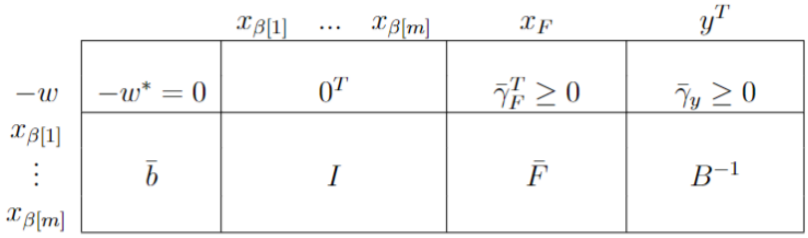
One can then start with the steps of the simplex described above until an optimal solution of the artificial problem is reached. It should be noted that the artificial problem is always feasible and cannot be unbounded.

In the end, the optimal value of the objective function of the artificial problem can be (having to exclude the case w∗ < 0):

- $w^* > 0$: it is concluded that the original problem is not feasible (and we obviously do not proceed with Phase II).
- $w^* = 0$: In this case, all artificial variables are necessarily null. They can therefore be eliminated from the system of constraints, and the same system will be satisfied with only the variables x. In other words, the problem is admissible. To identify the initial basis, two subcases are distinguished:
  - If all variables y are off-base, then the final tableau of Phase 1 directly locates the variables x in a feasible basis and the problem is feasible
  - If any variable y is in base, then it will be in base at the value 0. It is therefore always possible to perform pivot operations (one for each variable y in base) to exchange an in-base y for an out-of-base x. This yields an optimal alternative with only variables x in base, leading back to the first subcase

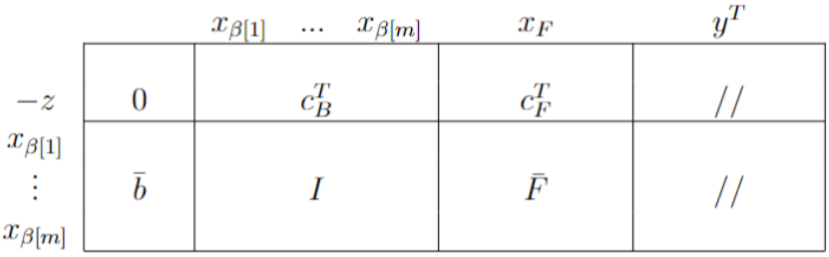
<u>Phase 2: Solution of the starting problem</u>

Any basis obtained at the end of Phase I can be used to initialize the simplex method. Using the simplex tableau, at the end of Phase I we will have:



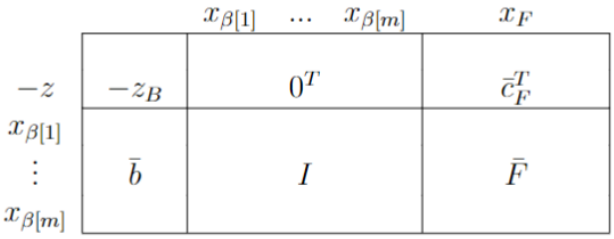
		$x_{\beta[1]} \;\dots\; x_{\beta[m]}$	x_F	y^T
$-w$	$-w^* = 0$	0^T	$\bar{\gamma}_F^T \geq 0$	$\bar{\gamma}_y \geq 0$
$x_{\beta[1]}$ $\vdots$ $x_{\beta[m]}$	$\bar{b}$	I	$\bar{F}$	B^{-1}

To restore the final tableau of Phase I in terms of the initial tableau of the original problem, the following steps are taken. The columns of artificial variables are removed and the costs of the original objective function and the value 0 for the objective function are returned to the first row:



		$x_{\beta[1]} \;\dots\; x_{\beta[m]}$	x_F	y^T
$-z$	0	c_B^T	c_F^T	//
$x_{\beta[1]}$ $\vdots$ $x_{\beta[m]}$	$\bar{b}$	I	$\bar{F}$	//

We then switch to the canonical tableau form with operations on the first row to return the reduced costs of the variables in the base to 0.

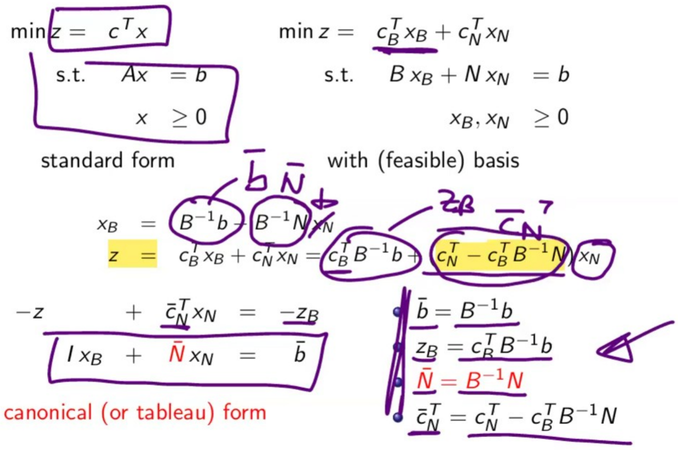


		$x_{\beta[1]} \; \dots \; x_{\beta[m]}$	x_F
$-z$	$-z_B$	0^T	$\bar{c}_F^T$
$x_{\beta[1]}$ $\vdots$ $x_{\beta[m]}$	$\bar{b}$	I	$\bar{F}$

At this point the tableau (and the system of equations it implies) is restored to its usual form for the application of Step 1 of the simplex.

5.5 SIMPLEX ALGORITHM IN MATRIX FORM AND REVISED ALGORITHM

The simplex algorithm can be *elegantly reformulated using matrix operations*, providing a systematic approach to solving LP problems in standard form, using linear algebraic techniques able to simplify complex optimization problems.



$$\begin{aligned} \min z = \;& c^T x \\ \text{s.t.} \quad Ax &= b \\ x &\geq 0 \end{aligned}$$

standard form

$$\begin{aligned} \min z = \;& c_B^T x_B + c_N^T x_N \\ \text{s.t.} \quad B\, x_B + N\, x_N &= b \\ x_B, x_N &\geq 0 \end{aligned}$$

with (feasible) basis

$$x_B = B^{-1}b - B^{-1}N x_N$$

$$z = c_B^T x_B + c_N^T x_N = c_B^T B^{-1} b + (c_N^T - c_B^T B^{-1} N) x_N$$

$$\begin{aligned} -z \quad + \bar{c}_N^T x_N &= -z_B \\ I\, x_B + \bar{N}\, x_N &= \bar{b} \end{aligned}$$

canonical (or tableau) form

- $\bar{b} = B^{-1} b$
- $z_B = c_B^T B^{-1} b$
- $\bar{N} = B^{-1} N$
- $\bar{c}_N^T = c_N^T - c_B^T B^{-1} N$

Basic and non-basic variable sets want to represent the numbers in such a way the problem becomes tractable, so to decompose the solution properly. So, the problem can be written as:

$$\begin{aligned} \min z = \;& c^T x \\ \text{s.t.} \quad Ax &= b \\ x &\geq 0 \end{aligned}$$

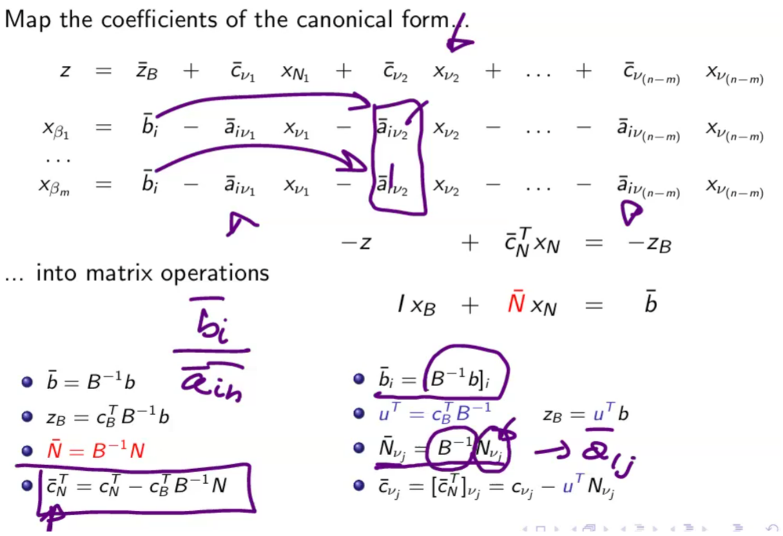
At any step, consider a basis B which allows us to write the same problem as:

$$\begin{aligned} \min z = \;& c_B^T x_B + c_F^T x_F \\ \text{s.t.} \quad Bx_B + Fx_F &= b \\ x_B, x_F &\geq 0 \end{aligned}$$

Or in the equivalent tableau form:

$$\begin{aligned} -z \qquad + \bar{c}_F^T x_F &= -z_B \\ Ix_B + \bar{F} x_F &= \bar{b} \end{aligned}$$

Solvers do not use the tableau but just compute what they need – it's like being in a maze (feasible solution space) and then taking a shortcut. This is done using only relevant parts.



Map the coefficients of the canonical form...

$$\begin{array}{lllllllllll} z & = & \bar{z}_B & + & \bar{c}_{\nu_1} & x_{N_1} & + & \bar{c}_{\nu_2} & x_{\nu_2} & + \dots + & \bar{c}_{\nu_{(n-m)}} \; x_{\nu_{(n-m)}} \\ x_{\beta_1} & = & \bar{b}_i & - & \bar{a}_{i\nu_1} & x_{\nu_1} & - & \bar{a}_{i\nu_2} & x_{\nu_2} & - \dots - & \bar{a}_{i\nu_{(n-m)}} \; x_{\nu_{(n-m)}} \\ \dots & & & & & & & & & & \\ x_{\beta_m} & = & \bar{b}_i & - & \bar{a}_{i\nu_1} & x_{\nu_1} & - & \bar{a}_{i\nu_2} & x_{\nu_2} & - \dots - & \bar{a}_{i\nu_{(n-m)}} \; x_{\nu_{(n-m)}} \end{array}$$

$$-z + \bar{c}_N^T x_N = -z_B$$

... into matrix operations

$$I x_B + \bar{N} x_N = \bar{b}$$

$\frac{\bar{b_i}}{\bar{a_{ih}}}$

- $\bar{b} = B^{-1}b$
- $z_B = c_B^T B^{-1} b$
- $\bar{N} = B^{-1}N$
- $\bar{c}_N^T = c_N^T - c_B^T B^{-1} N$

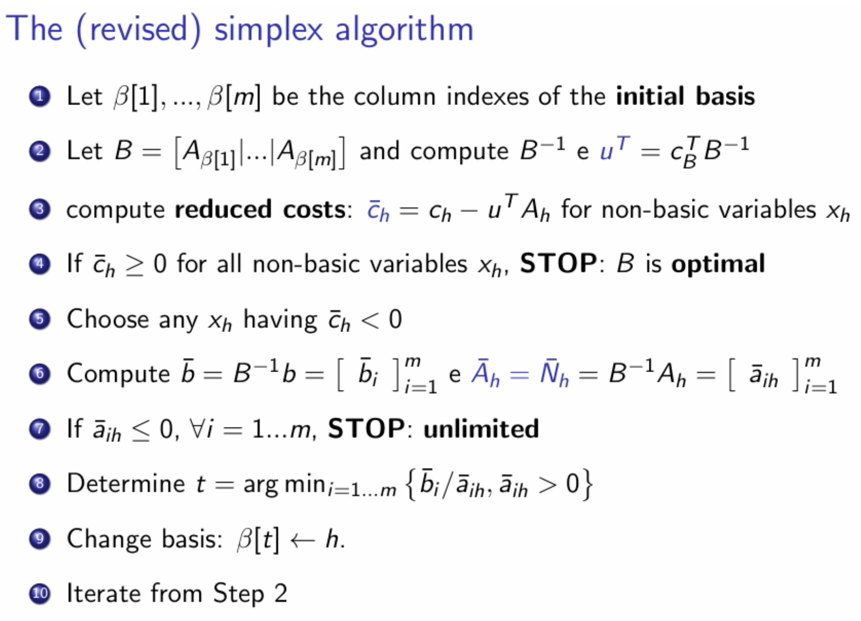
- $\bar{b}_i = [B^{-1}b]_i$
- $u^T = c_B^T B^{-1}$ $\quad z_B = u^T b$
- $\bar{N}_{\nu_j} = B^{-1} N_{\nu_j}$ → $\bar{a}_{ij}$
- $\bar{c}_{\nu_j} = [\bar{c}_N^T]_{\nu_j} = c_{\nu_j} - u^T N_{\nu_j}$

Here:

- $\overline{b} = B^{-1}b$ (value of basic variables in the current basic solution)
- $z_B = c_B^T\overline{b}$ (current value of the o.f.)
- $\overline{F} = B^{-1}F$ (columns of the non-basic variables expressed in terms of the current basis)
- $\overline{c}_F^T = c_F^T - c_B^T B^{-1} F$ (vector of reduced costs for non-basic variables)

At each step, then, simply invert the base matrix B and compute the elements listed above. In fact, the substitution steps seen above, as well as the pivot operations on the simplex tableau, correspond exactly to the algebraic steps on the matrices. Consider, for example, the second iteration of the simplex seen above, at the base x_B and non-basic variables x_F.

For this specific reason, we use the (revised) simplex algorithm:



The (revised) simplex algorithm

1. Let $\beta[1], ..., \beta[m]$ be the column indexes of the **initial basis**
2. Let $B = [A_{\beta[1]}|...|A_{\beta[m]}]$ and compute B^{-1} e $u^T = c_B^T B^{-1}$
3. compute **reduced costs**: $\bar{c}_h = c_h - u^T A_h$ for non-basic variables x_h
4. If $\bar{c}_h \geq 0$ for all non-basic variables x_h, **STOP**: B is **optimal**
5. Choose any x_h having $\bar{c}_h < 0$
6. Compute $\bar{b} = B^{-1}b = [\ \bar{b}_i\ ]_{i=1}^m$ e $\bar{A}_h = \bar{N}_h = B^{-1}A_h = [\ \bar{a}_{ih}\ ]_{i=1}^m$
7. If $\bar{a}_{ih} \leq 0$, $\forall i = 1...m$, **STOP**: **unlimited**
8. Determine $t = \arg\min_{i=1...m} \{\bar{b}_i/\bar{a}_{ih}, \bar{a}_{ih} > 0\}$
9. Change basis: $\beta[t] \leftarrow h$.
10. Iterate from Step 2

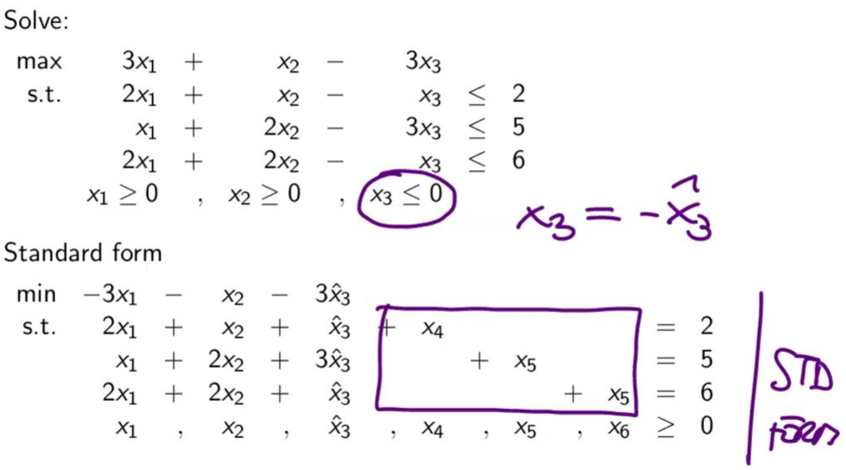
The original and revised simplex methods differ primarily in how they store and update information during the optimization process:

- The original simplex method maintains and updates the entire tableau at each iteration. This means it explicitly stores all the coefficients for both basic and non-basic variables, as well as the right-hand side values. When a pivot operation is performed, the entire tableau must be recalculated using elementary row operations.

- The revised simplex method, in contrast, only maintains the essential information needed for each iteration. Specifically, it stores:
  - The current basis matrix B and its inverse B^{-1}
  - The current basic solution values
  - The original problem data (A, b, and c)

When evaluating potential entering variables or performing updates, the revised method computes the necessary coefficients using matrix operations with this stored information.

Imagine you want to solve the following problem (here for complete resolution in Italian), which is then represented by the standard form:



Solve:

$$\begin{array}{lrcrcrcl} \max & 3x_1 & + & x_2 & - & 3x_3 & & \\ \text{s.t.} & 2x_1 & + & x_2 & - & x_3 & \le & 2 \\ & x_1 & + & 2x_2 & - & 3x_3 & \le & 5 \\ & 2x_1 & + & 2x_2 & - & x_3 & \le & 6 \\ & x_1 \ge 0 & , & x_2 \ge 0 & , & \boxed{x_3 \le 0} & & \end{array}$$

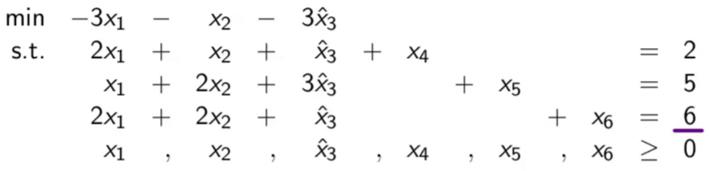
$x_3 = -\hat{x}_3$

Standard form

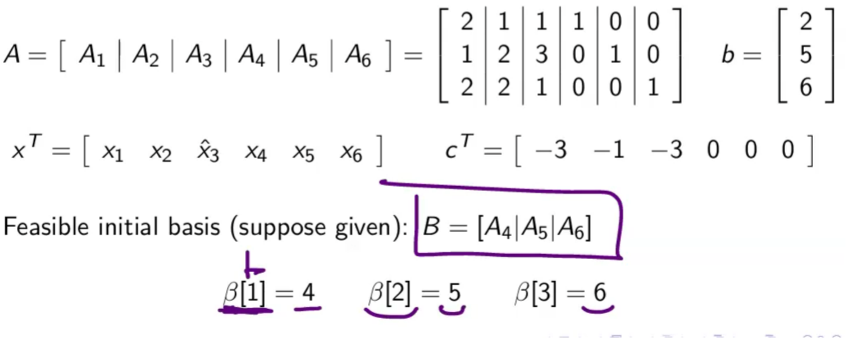
$$\begin{array}{lrcrcrcccccl} \min & -3x_1 & - & x_2 & - & 3\hat{x}_3 & & & & & & \\ \text{s.t.} & 2x_1 & + & x_2 & + & \hat{x}_3 & + x_4 & & & & = & 2 \\ & x_1 & + & 2x_2 & + & 3\hat{x}_3 & & + x_5 & & & = & 5 \\ & 2x_1 & + & 2x_2 & + & \hat{x}_3 & & & + x_5 & & = & 6 \\ & x_1 & , & x_2 & , & \hat{x}_3 & , x_4 & , x_5 & , x_6 & & \ge & 0 \end{array}$$

STD form

We now apply the simplex method, considering all the relevant parts needed – negative slack variable, expression in matrix terms and finding of a feasible basis in x_4, x_5, x_6:



$$\begin{array}{lrcrcrcccl} \min & -3x_1 & - & x_2 & - & 3\hat{x}_3 & & & & \\ \text{s.t.} & 2x_1 & + & x_2 & + & \hat{x}_3 & + x_4 & & & = 2 \\ & x_1 & + & 2x_2 & + & 3\hat{x}_3 & & + x_5 & & = 5 \\ & 2x_1 & + & 2x_2 & + & \hat{x}_3 & & & + x_6 & = \underline{6} \\ & x_1 & , & x_2 & , & \hat{x}_3 & , x_4 & , x_5 & , x_6 & \ge 0 \end{array}$$



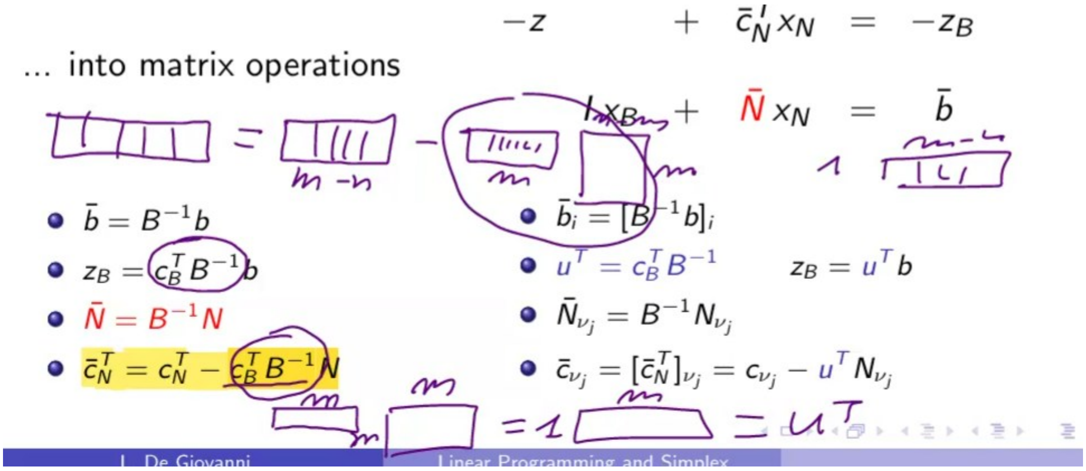
$$A = [\, A_1 \mid A_2 \mid A_3 \mid A_4 \mid A_5 \mid A_6 \,] = \left[\begin{array}{c|c|c|c|c|c} 2 & 1 & 1 & 1 & 0 & 0 \\ 1 & 2 & 3 & 0 & 1 & 0 \\ 2 & 2 & 1 & 0 & 0 & 1 \end{array}\right] \qquad b = \begin{bmatrix} 2 \\ 5 \\ 6 \end{bmatrix}$$

$$x^T = [\, x_1 \quad x_2 \quad \hat{x}_3 \quad x_4 \quad x_5 \quad x_6 \,] \qquad c^T = [\, -3 \quad -1 \quad -3 \quad 0 \quad 0 \quad 0 \,]$$

Feasible initial basis (suppose given): $\boxed{B = [A_4 | A_5 | A_6]}$

$$\underline{\beta[1]} = \underline{4} \qquad \underline{\beta[2]} = \underline{5} \qquad \beta[3] = \underline{6}$$

The formula is to always use vectors which are to be computed only once when needed, using what we had before, computing reduced costs one at a time:



$$-z \qquad + \quad \bar{c}_N' x_N = -z_B$$

... into matrix operations

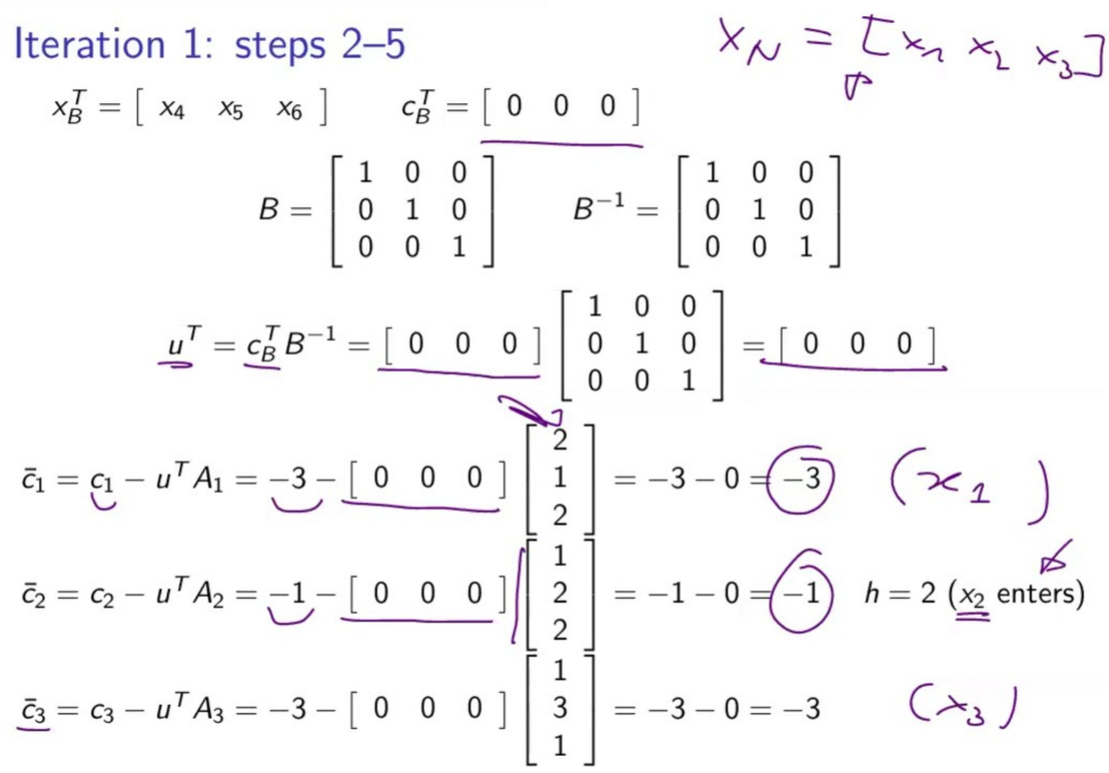
$$I x_B + \bar{N} x_N = \bar{b}$$

- $\bar{b} = B^{-1}b$
- $z_B = c_B^T B^{-1} b$
- $\bar{N} = B^{-1}N$
- $\bar{c}_N^T = c_N^T - c_B^T B^{-1} N$

- $\bar{b}_i = [B^{-1}b]_i$
- $u^T = c_B^T B^{-1}$ $\qquad z_B = u^T b$
- $\bar{N}_{\nu_j} = B^{-1} N_{\nu_j}$
- $\bar{c}_{\nu_j} = [\bar{c}_N^T]_{\nu_j} = c_{\nu_j} - u^T N_{\nu_j}$

Going on with the simplex application, keeping the base feasible.

- Iteration 1 – Step 2: Inverting the base and compute the u multipliers
- Iteration 1 – Step 3: Compute reduced costs
- Iteration 1 – Step 4: Optimality test
- Iteration 1 – Step 5: Choice of the entering variable for the basis exchange



Iteration 1: steps 2–5

$x_N = [x_1\ x_2\ x_3]$

$x_B^T = [\ x_4 \quad x_5 \quad x_6\ ] \qquad c_B^T = [\ 0 \quad 0 \quad 0\ ]$

$$B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad B^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

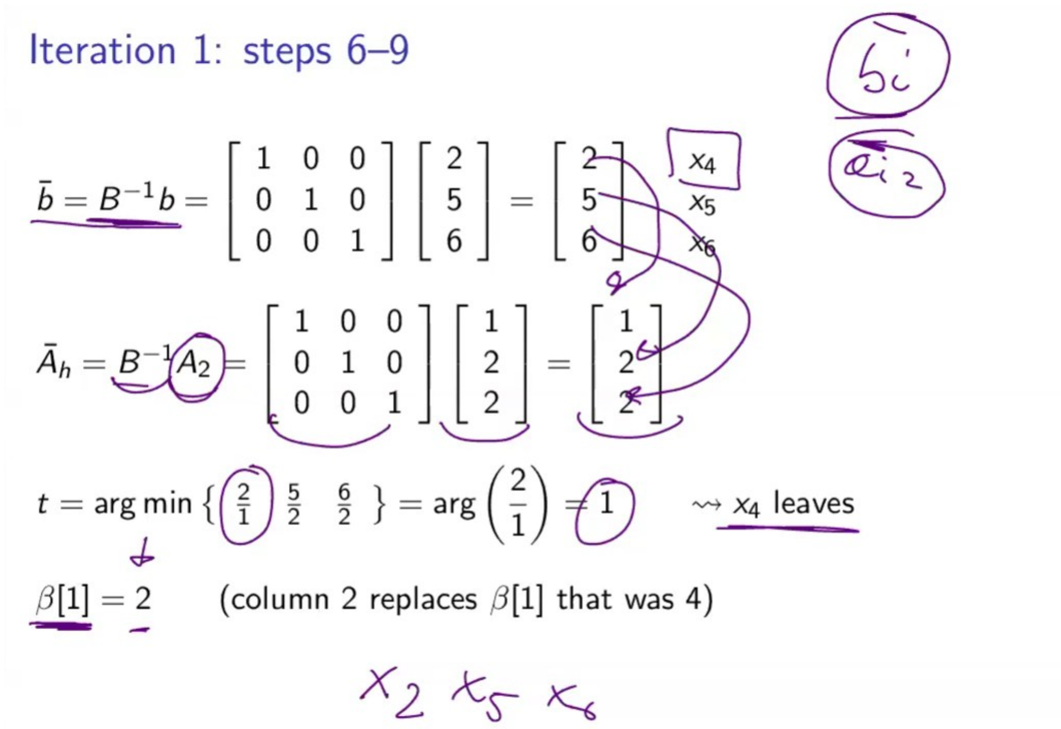
$$u^T = c_B^T B^{-1} = [\ 0 \quad 0 \quad 0\ ] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = [\ 0 \quad 0 \quad 0\ ]$$

$$\bar{c}_1 = c_1 - u^T A_1 = -3 - [\ 0 \quad 0 \quad 0\ ] \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix} = -3 - 0 = \boxed{-3} \qquad (x_1)$$

$$\bar{c}_2 = c_2 - u^T A_2 = -1 - [\ 0 \quad 0 \quad 0\ ] \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix} = -1 - 0 = \boxed{-1} \qquad h = 2\ (x_2 \text{ enters})$$

$$\bar{c}_3 = c_3 - u^T A_3 = -3 - [\ 0 \quad 0 \quad 0\ ] \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix} = -3 - 0 = -3 \qquad (x_3)$$

- Iteration 1 – Step 6: Updating columns (known terms and entering variable)
- Iteration 1 – Step 7: Unboundedness test
- Iteration 1 – Step 8: Determining the exiting variable for the basis exchange
- Iteration 1 – Step 9: Updating indices of the columns of the feasible base



Iteration 1: steps 6–9

$$\frac{\bar{b}_i}{\bar{a}_{i2}}$$

$$\bar{b} = B^{-1}b = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \\ 6 \end{bmatrix} \begin{matrix} \boxed{x_4} \\ x_5 \\ x_6 \end{matrix}$$

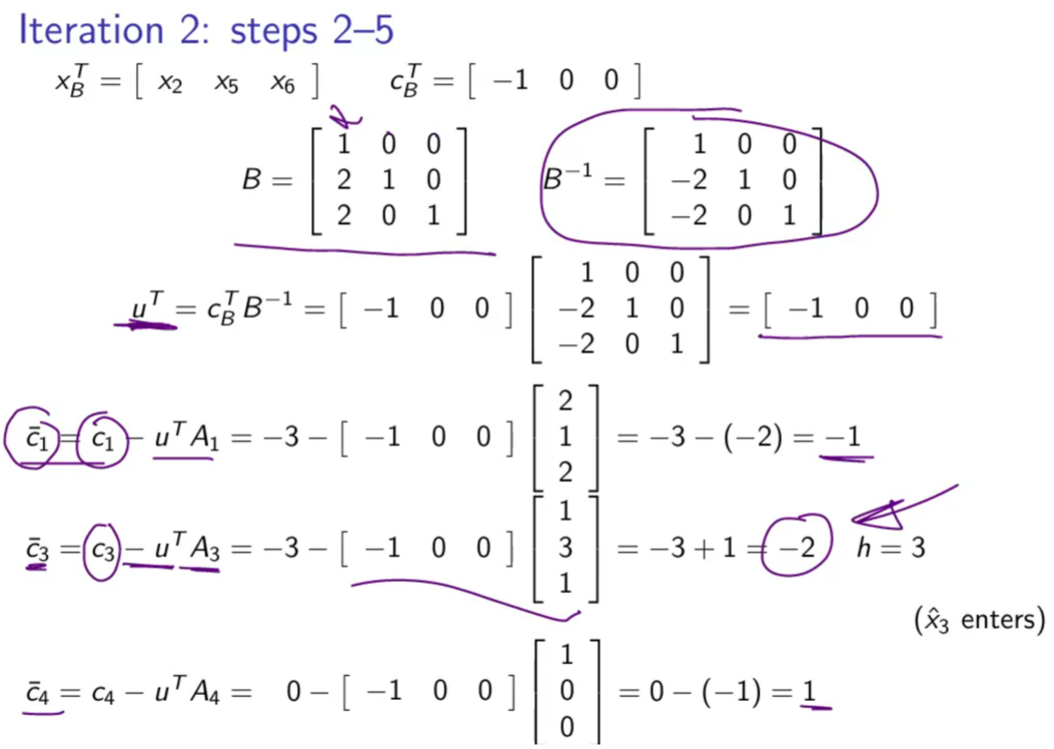
$$\bar{A}_h = B^{-1}A_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix}$$

$$t = \arg\min \left\{ \frac{2}{1} \quad \frac{5}{2} \quad \frac{6}{2} \right\} = \arg \left(\frac{2}{1} \right) = 1 \quad \rightsquigarrow x_4 \text{ leaves}$$

$\beta[1] = 2$ (column 2 replaces $\beta[1]$ that was 4)

$x_2 \; x_5 \; x_6$

- Iteration 2 – Step 2: Inverting the base and compute the u multipliers
- Iteration 2 – Step 3: Compute reduced costs
- Iteration 2 – Step 4: Optimality test
- Iteration 2 – Step 5: Choice of the entering variable for the basis exchange



Iteration 2: steps 2–5

$$x_B^T = [\; x_2 \quad x_5 \quad x_6 \;] \qquad c_B^T = [\; -1 \quad 0 \quad 0 \;]$$

$$B = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix} \qquad B^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix}$$

$$u^T = c_B^T B^{-1} = [\; -1 \quad 0 \quad 0 \;] \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -2 & 0 & 1 \end{bmatrix} = [\; -1 \quad 0 \quad 0 \;]$$

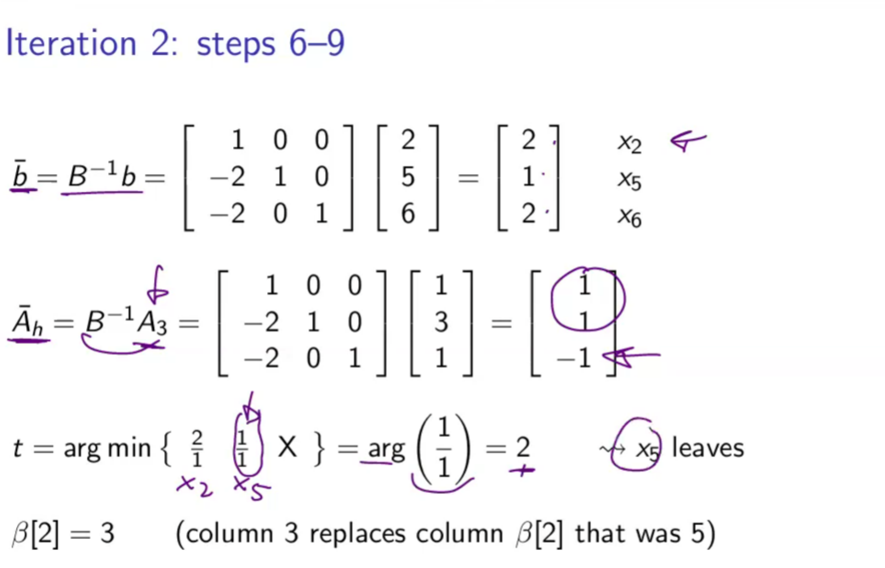
$$\bar{c}_1 = c_1 - u^T A_1 = -3 - [\; -1 \quad 0 \quad 0 \;] \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix} = -3 - (-2) = -1$$

$$\bar{c}_3 = c_3 - u^T A_3 = -3 - [\; -1 \quad 0 \quad 0 \;] \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix} = -3 + 1 = -2 \qquad h = 3$$

($\hat{x}_3$ enters)

$$\bar{c}_4 = c_4 - u^T A_4 = \quad 0 - [\; -1 \quad 0 \quad 0 \;] \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = 0 - (-1) = 1$$

- Iteration 2 – Step 6: Updating columns (known terms and entering variable)
- Iteration 2 – Step 7: Unboundedness test
- Iteration 2 – Step 8: Determining the exiting variable for the basis exchange
- Iteration 2 – Step 9: Updating indices of the columns of the feasible base



Iteration 2: steps 6–9

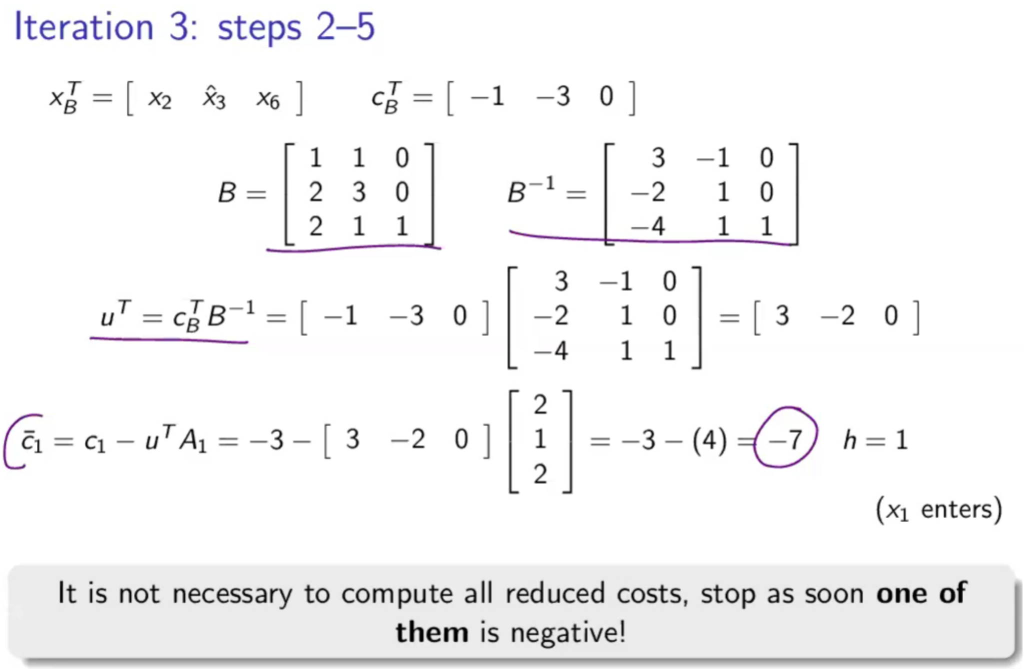
$$\underline{\bar{b}} = \underline{B^{-1}b} = \begin{bmatrix}1&0&0\\-2&1&0\\-2&0&1\end{bmatrix}\begin{bmatrix}2\\5\\6\end{bmatrix} = \begin{bmatrix}2\\1\\2\end{bmatrix} \quad \begin{matrix}x_2 \leftarrow\\x_5\\x_6\end{matrix}$$

$$\underline{\bar{A}_h} = B^{-1}A_3 = \begin{bmatrix}1&0&0\\-2&1&0\\-2&0&1\end{bmatrix}\begin{bmatrix}1\\3\\1\end{bmatrix} = \begin{bmatrix}1\\1\\-1\end{bmatrix} \leftarrow$$

$$t = \arg\min\left\{ \underset{x_2}{\tfrac{2}{1}} \quad \underset{x_5}{\tfrac{1}{1}} \quad \text{X} \right\} = \arg\left(\tfrac{1}{1}\right) = 2 \qquad \rightarrow x_5 \text{ leaves}$$

$\beta[2] = 3$ (column 3 replaces column $\beta[2]$ that was 5)

Contrary to the original simplex method, we *stop as soon as there is a single negative cost*:



Iteration 3: steps 2–5

$$x_B^T = \begin{bmatrix} x_2 & \hat{x}_3 & x_6 \end{bmatrix} \qquad c_B^T = \begin{bmatrix} -1 & -3 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix}1&1&0\\2&3&0\\2&1&1\end{bmatrix} \qquad B^{-1} = \begin{bmatrix}3&-1&0\\-2&1&0\\-4&1&1\end{bmatrix}$$

$$\underline{u^T = c_B^T B^{-1}} = \begin{bmatrix} -1 & -3 & 0 \end{bmatrix}\begin{bmatrix}3&-1&0\\-2&1&0\\-4&1&1\end{bmatrix} = \begin{bmatrix} 3 & -2 & 0 \end{bmatrix}$$

$$\bar{c}_1 = c_1 - u^T A_1 = -3 - \begin{bmatrix} 3 & -2 & 0 \end{bmatrix}\begin{bmatrix}2\\1\\2\end{bmatrix} = -3 - (4) = \boxed{-7} \qquad h = 1$$

(x_1 enters)

It is not necessary to compute all reduced costs, stop as soon **one of them** is negative!

Iteration 3: steps 6–9

$B \; [\, A_2 \mid A_3 \mid A_6 \,]$

$$\underline{\bar{b}} = \underline{B^{-1}b} = \begin{bmatrix} 3 & -1 & 0 \\ -2 & 1 & 0 \\ -4 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 3 \end{bmatrix} \begin{matrix} x_2 \\ \hat{x}_3 \\ x_6 \end{matrix}$$

$$\boxed{\bar{A}_h} = B^{-1}A_1 = \begin{bmatrix} 3 & -1 & 0 \\ -2 & 1 & 0 \\ -4 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 5 \\ -3 \\ -5 \end{bmatrix}$$

$$t = \arg\min \left\{ \tfrac{1}{5} \quad \text{X} \quad \text{X} \right\} = \arg\left(\frac{1}{5}\right) = 1 \qquad \rightsquigarrow \underline{x_2} \text{ leaves}$$

$\beta[1] = 1$ (column 1 replaces column $\beta[1]$ that was 2)

Iteration 4

$$x_B^T = [\; x_1 \quad \hat{x}_3 \quad x_6 \;] \qquad c_B^T = [\; -3 \quad -3 \quad 0 \;]$$

$$B = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 3 & 0 \\ 2 & 1 & 1 \end{bmatrix} \qquad B^{-1} = \begin{bmatrix} 3/5 & -1/5 & 0 \\ -1/5 & 2/5 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$

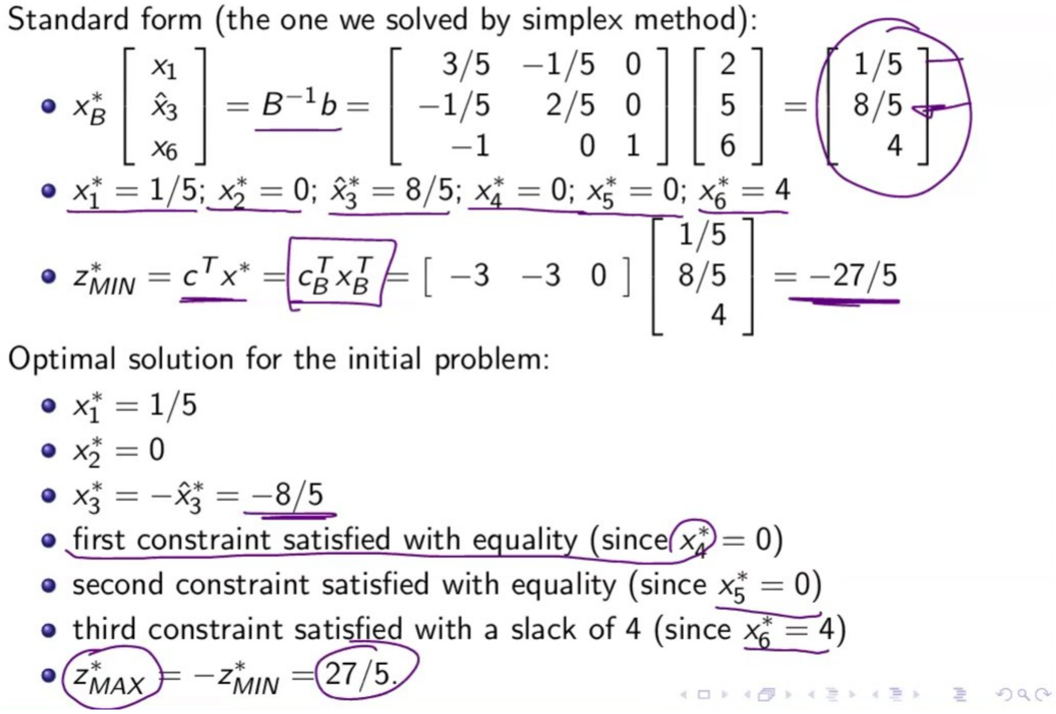
$$\underline{u^T = c_B^T B^{-1}} = [\; -3 \quad -3 \quad 0 \;] \begin{bmatrix} 3/5 & -1/5 & 0 \\ -1/5 & 2/5 & 0 \\ -1 & 0 & 1 \end{bmatrix} = [\; -6/5 \quad -3/5 \quad 0 \;]$$

$$\underline{\bar{c}_2} = c_2 - u^T A_2 = -1 - [\; -6/5 \quad -3/5 \quad 0 \;] \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix} = -1 - (12/5) = \underline{7/5}$$

$$\underline{\bar{c}_4} = c_4 - u^T A_4 = 0 - [\; -6/5 \quad -3/5 \quad 0 \;] \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = 0 - (6/5) = \underline{6/5}$$

$$\underline{\bar{c}_5} = c_5 - u^T A_5 = 0 - [\; -6/5 \quad -3/5 \quad 0 \;] \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = 0 - (3/5) = \underline{3/5}$$

We then arrive at the solution respecting all of the constraints, inverting at each iteration/step the matrix:



Standard form (the one we solved by simplex method):

- $x_B^* \begin{bmatrix} x_1 \\ \hat{x}_3 \\ x_6 \end{bmatrix} = \underline{B^{-1}b} = \begin{bmatrix} 3/5 & -1/5 & 0 \\ -1/5 & 2/5 & 0 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 1/5 \\ 8/5 \\ 4 \end{bmatrix}$
- $\underline{x_1^* = 1/5}; \underline{x_2^* = 0}; \underline{\hat{x}_3^* = 8/5}; \underline{x_4^* = 0; x_5^* = 0}; \underline{x_6^* = 4}$
- $z_{MIN}^* = \underline{c^T x^*} = \boxed{c_B^T x_B^T} = \begin{bmatrix} -3 & -3 & 0 \end{bmatrix} \begin{bmatrix} 1/5 \\ 8/5 \\ 4 \end{bmatrix} = \underline{-27/5}$

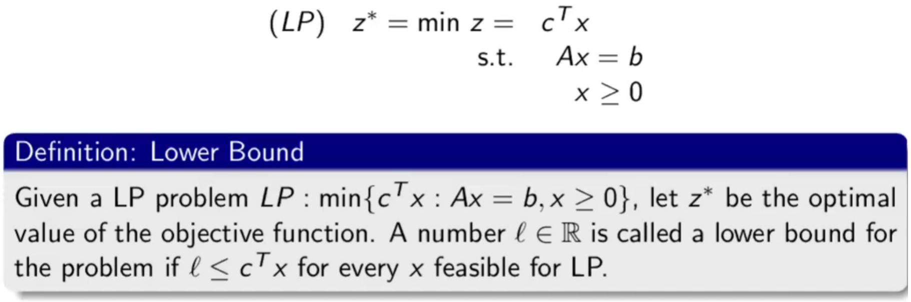
Optimal solution for the initial problem:

- $x_1^* = 1/5$
- $x_2^* = 0$
- $x_3^* = -\hat{x}_3^* = \underline{-8/5}$
- first constraint satisfied with equality (since $x_4^* = 0$)
- second constraint satisfied with equality (since $x_5^* = 0$)
- third constraint satisfied with a slack of 4 (since $x_6^* = 4$)
- $z_{MAX}^* = -z_{MIN}^* = 27/5.$

We will get to the "column generation methods", since at each step we take only the column that we need at a computation step. To get to that, we will see some basic concepts of *duality*, to be able to solve a particular problem even having an exponential number of variables.

6 REVIEW OF DUALITY IN LINEAR PROGRAMMING (5)

Given a linear programming problem in standard form, we want to provide a lower bound (LB) on the possible values that the objective function can take in the feasible region and we the problem is constrained to be not less than that variable (used to estimate the value of the o.f.).



$$(LP) \quad z^* = \min z = c^T x \quad \text{s.t.} \quad Ax = b, \quad x \geq 0$$

Definition: Lower Bound

Given a LP problem $LP : \min\{c^T x : Ax = b, x \geq 0\}$, let z^* be the optimal value of the objective function. A number $\ell \in \mathbb{R}$ is called a lower bound for the problem if $\ell \leq c^T x$ for every x feasible for LP.

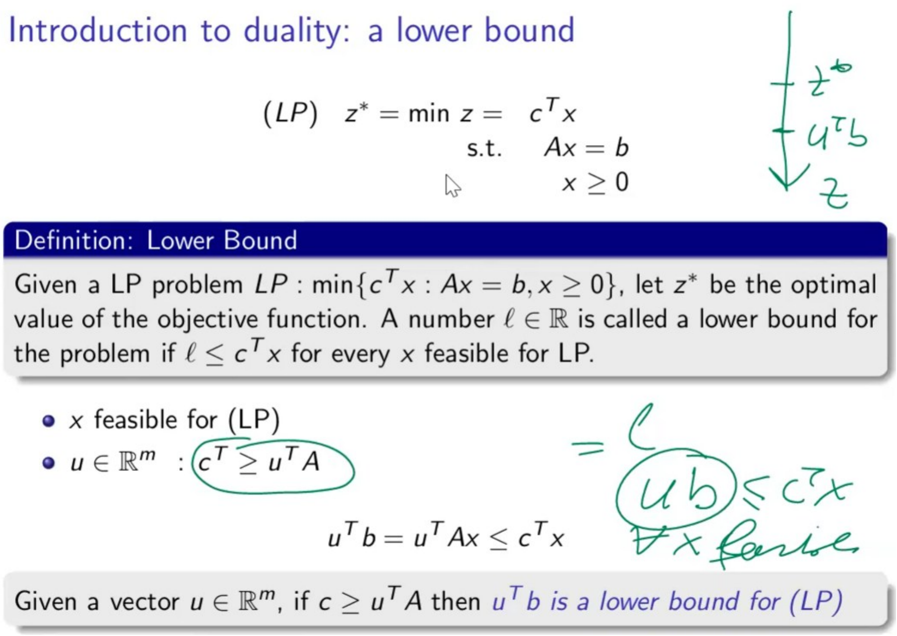
To obtain a lower bound, one can start from a vector $u \in R^m$ and impose the lower bound condition from the equation $Ax = b$.

$$u^T Ax = u^T b, \ \ \forall x \ feasible$$

For u to represent a lower bound it is necessary that $c^T \geq u^T A$, and this follows from the fact that the value of the objective function $c^T x$ must be greater than the lower bound:

$$c^T x \geq u^T Ax, \ \ \forall x \ feasible$$

Since there is a lower bound, this means that both inequalities are correctly satisfied and get a lower bound as close as possible to the solution:



Introduction to duality: a lower bound

$$(LP) \quad z^* = \min z = c^T x \quad \text{s.t.} \quad Ax = b, \quad x \geq 0$$

Definition: Lower Bound

Given a LP problem $LP : \min\{c^T x : Ax = b, x \geq 0\}$, let z^* be the optimal value of the objective function. A number $\ell \in \mathbb{R}$ is called a lower bound for the problem if $\ell \leq c^T x$ for every x feasible for LP.

- x feasible for (LP)
- $u \in \mathbb{R}^m \ : \boxed{c^T \geq u^T A}$

$$= \ell$$

$$u^T b = u^T Ax \leq c^T x \qquad \boxed{ub} \leq c^T x \ \ \forall x \text{ feasible}$$

Given a vector $u \in \mathbb{R}^m$, if $c \geq u^T A$ then $u^T b$ is a lower bound for (LP)

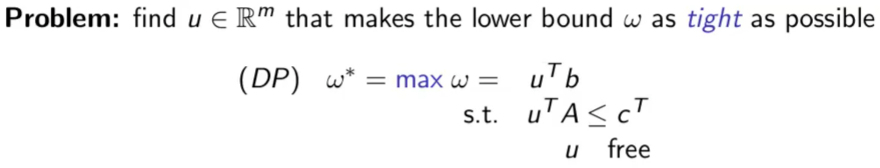
It therefore turns out to be important to have the highest possible LB, and this is done by appropriately choosing the vector u. The choice of this vector can be seen as a problem of maximization, in which the decision variables are contained in the vector u.

6.1 Dual Problem Definition and Duality Theorems

Duality theory in linear programming can be viewed as a tool for checking the optimality of a feasible solution. Given a linear programming problem in minimization form, the idea is to provide a lower bound on the possible values that the objective function can take over the feasible region.

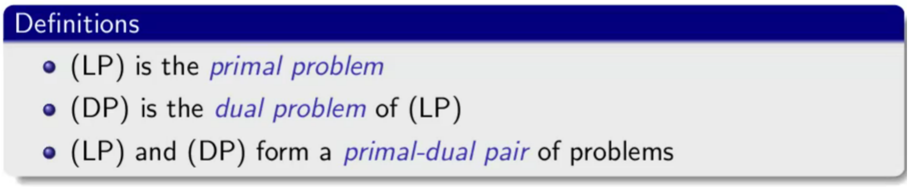
Problem (DP) is the dual problem of (LP). In this context, (LP) is called primal problem, and the pair of problems (LP) and (DP) is called primal-dual pair. Note that there is:

- A dual variable corresponding to each primal constraint
- A dual constraint corresponding to each primal variable



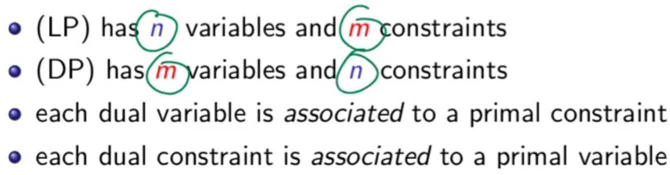
Problem: find $u \in \mathbb{R}^m$ that makes the lower bound ω as *tight* as possible

$$(DP) \quad \omega^* = \max \omega = \quad u^T b \\ \text{s.t.} \quad u^T A \leq c^T \\ u \quad \text{free}$$



Definitions

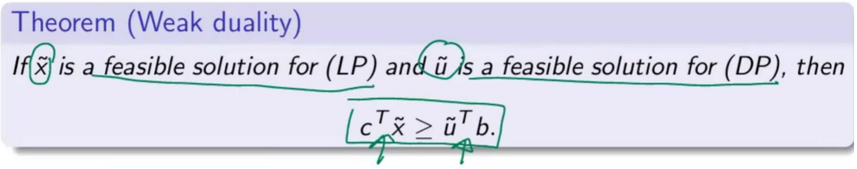
- (LP) is the *primal problem*
- (DP) is the *dual problem* of (LP)
- (LP) and (DP) form a *primal-dual pair* of problems



- (LP) has n variables and m constraints
- (DP) has m variables and n constraints
- each dual variable is *associated* to a primal constraint
- each dual constraint is *associated* to a primal variable

A solution to the dual problem is a lower bound to the primal problem and vice versa a solution to the primal problem is an upper bound to the dual problem.

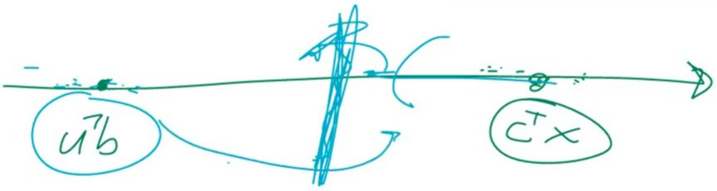
Weak duality provides an essential bounding mechanism. It tells us that the value of any feasible solution to the dual problem serves as a bound on the optimal value of the primal problem. This allows us to see that bounds are strict and not separated.



Theorem (Weak duality)

If $\tilde{x}$ is a feasible solution for (LP) and $\tilde{u}$ is a feasible solution for (DP), then

$$\boxed{c^T \tilde{x} \geq \tilde{u}^T b.}$$

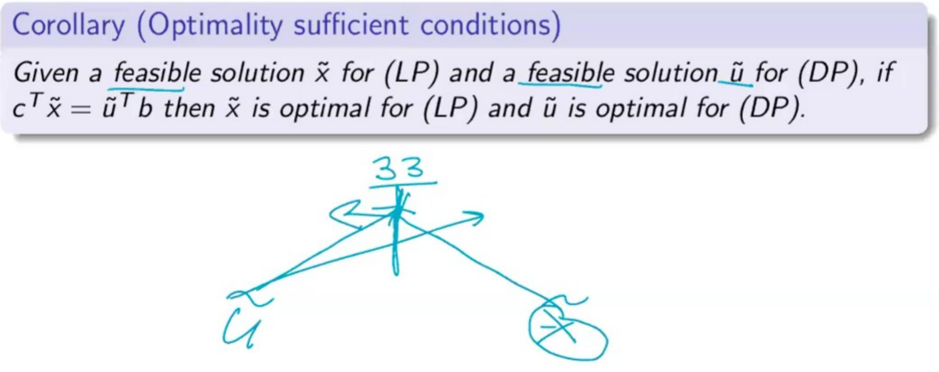


This has significant practical implications for optimization algorithms, as it allows us to:

- Establish quality guarantees on solutions
  - By comparing the objective values of feasible primal and dual solutions, we can determine how far a current solution is from optimality
  - This is particularly valuable when working with large-scale problems where finding the exact optimal solution may be computationally intensive

- Develop stopping criteria for algorithms
  - When the difference between primal and dual objective values becomes sufficiently small, we can be confident that we are close to the optimal solution, allowing algorithms to terminate efficiently

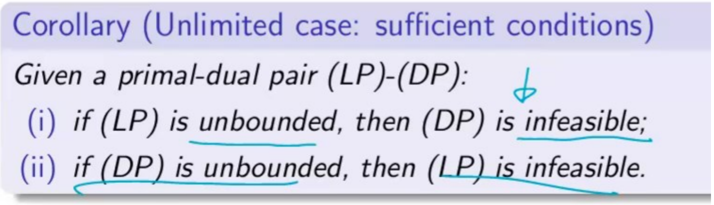
It's impossible to have another feasible solution not being related to the actual solution bounds (note: the conditions are sufficient):



Corollary (Optimality sufficient conditions)

Given a feasible solution $\tilde{x}$ for (LP) and a feasible solution $\tilde{u}$ for (DP), if $c^T\tilde{x} = \tilde{u}^T b$ then $\tilde{x}$ is optimal for (LP) and $\tilde{u}$ is optimal for (DP).

It also holds the following:



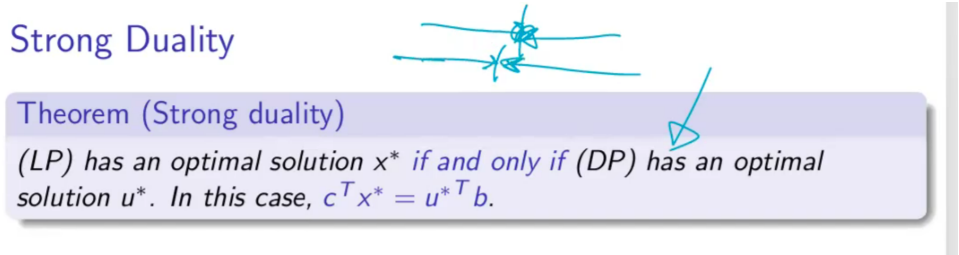
Corollary (Unlimited case: sufficient conditions)

Given a primal-dual pair (LP)-(DP):

(i) *if (LP) is unbounded, then (DP) is infeasible;*

(ii) *if (DP) is unbounded, then (LP) is infeasible.*

Strong duality, which states that the optimal values of primal and dual problems are equal under certain conditions, has even more profound implications (also for column generation methods). In every case the problem has an optimal solution, this value is optimal for sure. Since they are not unlimited and feasible, there is not a gap between the solutions, so the solutions are the same.



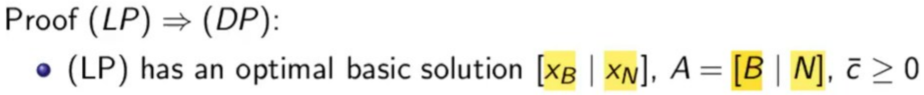
Strong Duality

Theorem (Strong duality)

(LP) has an optimal solution x^* if and only if (DP) has an optimal solution u^*. In this case, $c^T x^* = {u^*}^T b$.

To complete the proof, we will use the simplex theory.

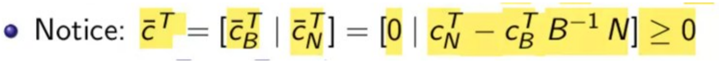
If PL has an admissible optimal solution, then it will have a basic one that can be derived by the (x^*) simplex method from which one can construct a vector $u \in R^m$ that is admissible and optimal solution for the dual problem.



Proof $(LP) \Rightarrow (DP)$:

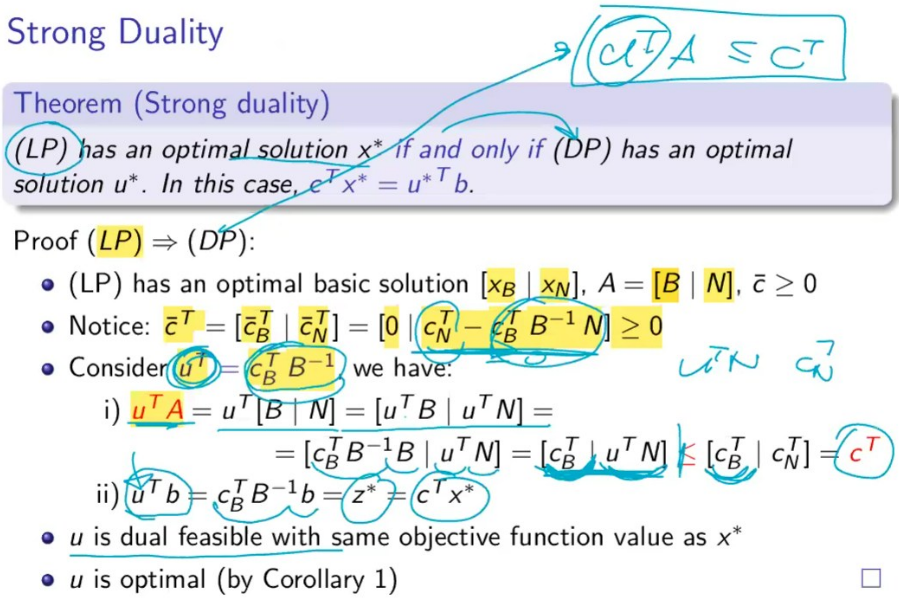
- (LP) has an optimal basic solution $[x_B \mid x_N]$, $A = [B \mid N]$, $\bar{c} \geq 0$

Since x^* is a solution found with the simplex, the reduced costs of in-base variables will be zero ($c_B = 0$) and those of out-of-base variables will be greater than or equal to 0.



- Notice: $\bar{c}^T = [\bar{c}_B^T \mid \bar{c}_N^T] = [0 \mid c_N^T - c_B^T B^{-1} N] \geq 0$

The simplex multipliers have got to be the feasible solution of the problem:



Strong Duality

$\boxed{u^T A \le c^T}$

Theorem (Strong duality)

(LP) has an optimal solution x^* if and only if (DP) has an optimal solution u^*. In this case, $c^T x^* = {u^*}^T b$.

Proof $(LP) \Rightarrow (DP)$:

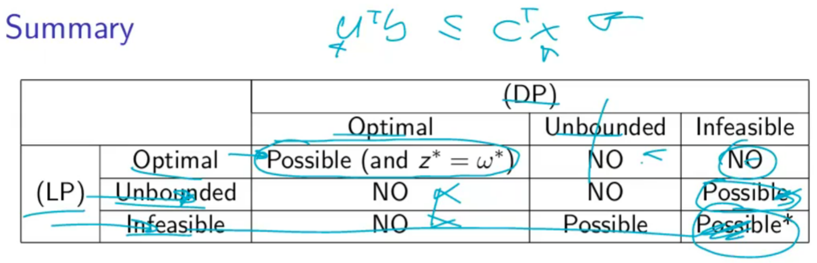
- (LP) has an optimal basic solution $[x_B \mid x_N]$, $A = [B \mid N]$, $\bar{c} \geq 0$
- Notice: $\bar{c}^T = [\bar{c}_B^T \mid \bar{c}_N^T] = [0 \mid c_N^T - c_B^T B^{-1} N] \geq 0$
- Consider $u^T = c_B^T B^{-1}$ we have: $u^T N \quad c_N^T$

  i) $u^T A = u^T [B \mid N] = [u^T B \mid u^T N] =$

  $= [c_B^T B^{-1} B \mid u^T N] = [c_B^T \mid u^T N] \leq [c_B^T \mid c_N^T] = c^T$

  ii) $u^T b = c_B^T B^{-1} b = z^* = c^T x^*$
- u is dual feasible with same objective function value as x^*
- u is optimal (by Corollary 1) □

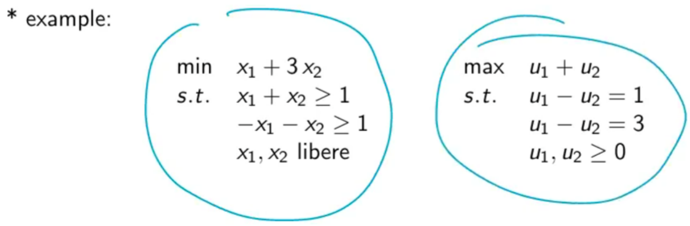
To make the transition from the primary problem to the dual problem, it is not necessary that the primary problem be in standard form, the only important thing is to respect the constraints given the relationships which logically appear between the actual constraints (below an example of primal/dual problems couple):



Summary

$u^T b \le c^T x$

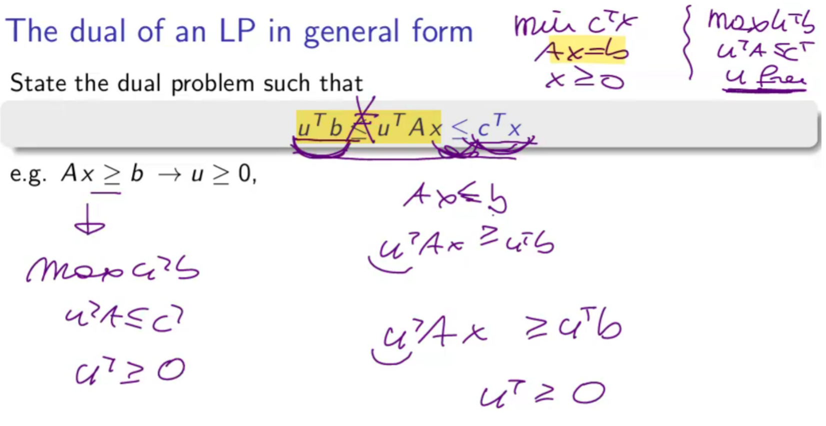
		(DP)		
		Optimal	Unbounded	Infeasible
(LP)	Optimal	Possible (and $z^* = \omega^*$)	NO	NO
	Unbounded	NO	NO	Possible
	Infeasible	NO	Possible	Possible*



* example:

$$\begin{array}{ll} \min & x_1 + 3x_2 \\ s.t. & x_1 + x_2 \geq 1 \\ & -x_1 - x_2 \geq 1 \\ & x_1, x_2 \text{ libere} \end{array} \qquad \begin{array}{ll} \max & u_1 + u_2 \\ s.t. & u_1 - u_2 = 1 \\ & u_1 - u_2 = 3 \\ & u_1, u_2 \geq 0 \end{array}$$

We just need to change the definition of primal/dual problem, to maintain upper/lower bound relationship between primal and dual problem.



The dual of an LP in general form

$\min c^T x$
$Ax = b$
$x \ge 0$

$\max u^T b$
$u^T A \le c^T$
u free

State the dual problem such that

$$\underbrace{u^T b}\ \cancel{\le}\ \underbrace{u^T A x \le c^T x}$$

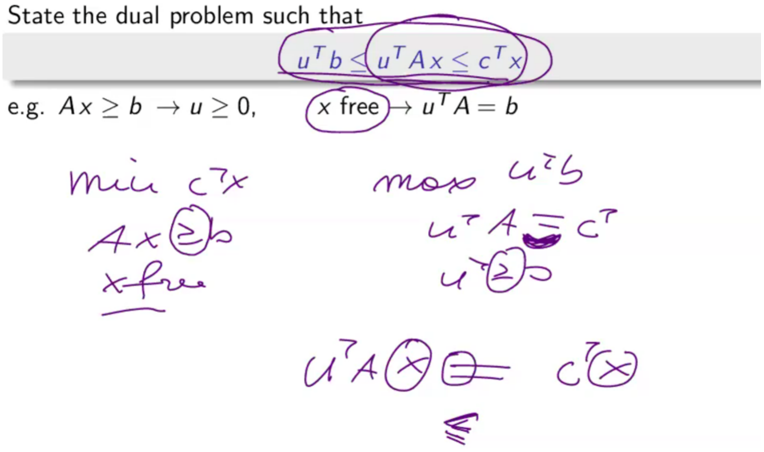
e.g. $Ax \ge b \to u \ge 0$,

$\max u^T b$
$u^T A \le c^T$
$u^T \ge 0$

$Ax \le b$
$u^T A x \ge u^T b$

$u^T A x \ge u^T b$
$u^T \ge 0$

Consider another case in which in the primal problem we have free variables. We always need the inequalities chain, so to make the same condition of before hold.



State the dual problem such that

$$u^T b \le u^T A x \le c^T x$$

e.g. $Ax \ge b \to u \ge 0$, x free $\to u^T A = b$

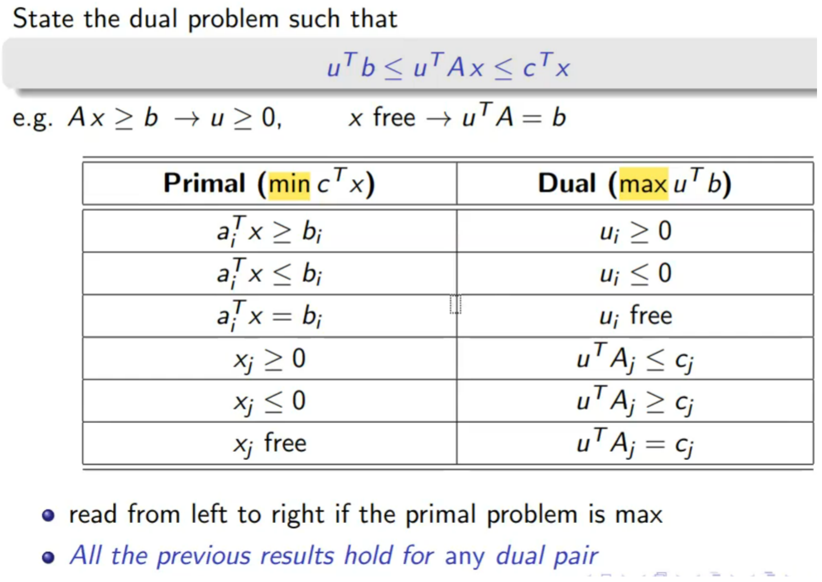
$\min c^T x$
$Ax \ge b$
x free

$\max u^T b$
$u^T A = c^T$
$u \ge 0$

$u^T A x = c^T x$

$\le$

If we have two solutions, this means they are optimal for both problems – the following is the general form for dual of an LP:



State the dual problem such that

$$u^T b \le u^T A x \le c^T x$$

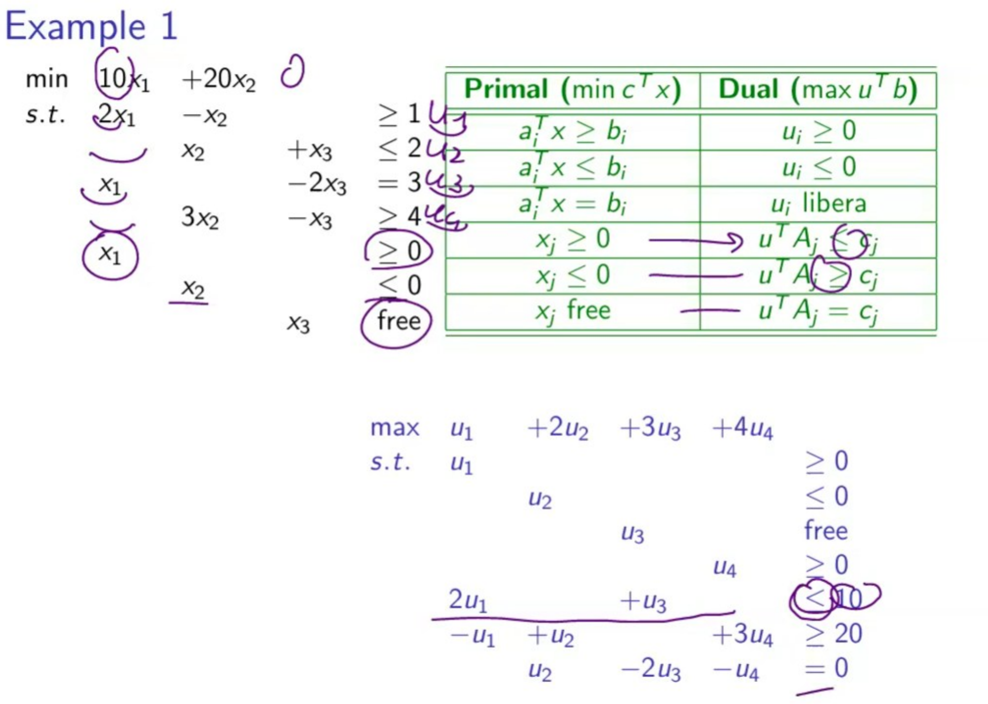
e.g. $Ax \ge b \to u \ge 0$, x free $\to u^T A = b$

Primal ($\min c^T x$)	**Dual** ($\max u^T b$)
$a_i^T x \ge b_i$	$u_i \ge 0$
$a_i^T x \le b_i$	$u_i \le 0$
$a_i^T x = b_i$	u_i free
$x_j \ge 0$	$u^T A_j \le c_j$
$x_j \le 0$	$u^T A_j \ge c_j$
x_j free	$u^T A_j = c_j$

- read from left to right if the primal problem is max
- *All the previous results hold for any dual pair*

We have to respect the chain of inequalities and use transformation according to the above table. Note that the table reads from left to right if you have a primal problem in minimum form and from right to left if you have a primal problem in maximum form.

A general example might be this one – now we apply the table of before:



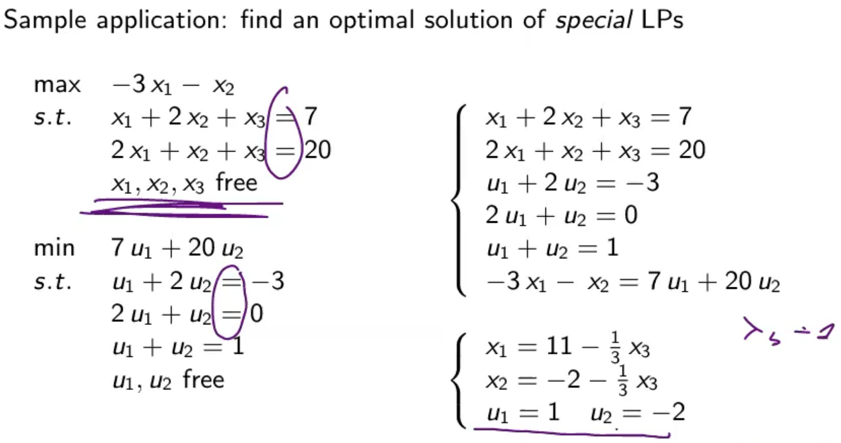
Example 1

$$
\begin{array}{llllll}
\min & 10x_1 & +20x_2 & & & \\
s.t. & 2x_1 & -x_2 & & \geq 1 & u_1\\
 & & x_2 & +x_3 & \leq 2 & u_2\\
 & x_1 & & -2x_3 & = 3 & u_3\\
 & & 3x_2 & -x_3 & \geq 4 & u_4\\
 & x_1 & & & \geq 0 & \\
 & & x_2 & & \leq 0 & \\
 & & & x_3 & \text{free} &
\end{array}
$$

Primal ($\min c^T x$)	Dual ($\max u^T b$)
$a_i^T x \geq b_i$	$u_i \geq 0$
$a_i^T x \leq b_i$	$u_i \leq 0$
$a_i^T x = b_i$	u_i libera
$x_j \geq 0$	$u^T A_j \leq c_j$
$x_j \leq 0$	$u^T A_j \geq c_j$
x_j free	$u^T A_j = c_j$

$$
\begin{array}{llllll}
\max & u_1 & +2u_2 & +3u_3 & +4u_4 & \\
s.t. & u_1 & & & & \geq 0\\
 & & u_2 & & & \leq 0\\
 & & & u_3 & & \text{free}\\
 & & & & u_4 & \geq 0\\
 & 2u_1 & & +u_3 & & \leq 10\\
 & -u_1 & +u_2 & & +3u_4 & \geq 20\\
 & & u_2 & -2u_3 & -u_4 & = 0
\end{array}
$$

There is a very special case using only free variables:



Sample application: find an optimal solution of *special* LPs

$$
\begin{array}{ll}
\max & -3x_1 - x_2\\
s.t. & x_1 + 2x_2 + x_3 = 7\\
 & 2x_1 + x_2 + x_3 = 20\\
 & x_1, x_2, x_3 \text{ free}
\end{array}
$$

$$
\begin{array}{ll}
\min & 7u_1 + 20u_2\\
s.t. & u_1 + 2u_2 = -3\\
 & 2u_1 + u_2 = 0\\
 & u_1 + u_2 = 1\\
 & u_1, u_2 \text{ free}
\end{array}
$$

$$
\begin{cases}
x_1 + 2x_2 + x_3 = 7\\
2x_1 + x_2 + x_3 = 20\\
u_1 + 2u_2 = -3\\
2u_1 + u_2 = 0\\
u_1 + u_2 = 1\\
-3x_1 - x_2 = 7u_1 + 20u_2
\end{cases}
$$

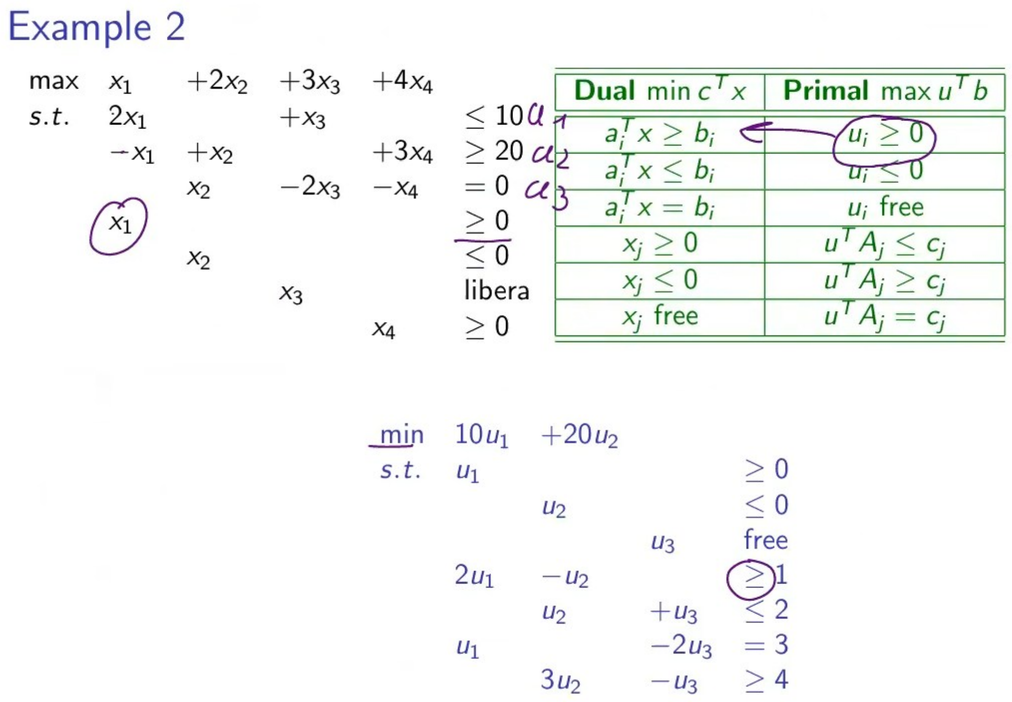
$x_3 = 1$

$$
\begin{cases}
x_1 = 11 - \frac{1}{3}x_3\\
x_2 = -2 - \frac{1}{3}x_3\\
u_1 = 1 \quad u_2 = -2
\end{cases}
$$

Hint: The presence of only equality constraints and only free variables suggests the direct application of the primal-dual optimality conditions by setting up a system of linear equations containing the constraints of the primal (equality) the constraints of the dual (free primary variables ⇒ constraints of the dual equality) and the equality constraint between the primal objective function and the primal objective function

As a result, we have infinite optimal solutions of type: $x_1 = 11 - \frac{1}{3}x_3, x_2 = -2 - \frac{1}{3}x_3$.

Now another example, where the main message is always the same: when we have a primal problem, get the dual problem and solve it to optimality. The value of the o.f. is a bound for the feasible solution of the other one.



Example 2

$$\begin{array}{llllll}
\max & x_1 & +2x_2 & +3x_3 & +4x_4 & \\
s.t. & 2x_1 & & +x_3 & & \leq 10 \quad u_1 \\
 & -x_1 & +x_2 & & +3x_4 & \geq 20 \quad u_2 \\
 & & x_2 & -2x_3 & -x_4 & = 0 \quad u_3 \\
 & x_1 & & & & \geq 0 \\
 & & x_2 & & & \leq 0 \\
 & & & x_3 & & \text{libera} \\
 & & & & x_4 & \geq 0
\end{array}$$

Dual $\min c^T x$	**Primal** $\max u^T b$
$a_i^T x \geq b_i$	$u_i \geq 0$
$a_i^T x \leq b_i$	$u_i \leq 0$
$a_i^T x = b_i$	u_i free
$x_j \geq 0$	$u^T A_j \leq c_j$
$x_j \leq 0$	$u^T A_j \geq c_j$
x_j free	$u^T A_j = c_j$

$$\begin{array}{lllll}
\min & 10u_1 & +20u_2 & & \\
s.t. & u_1 & & & \geq 0 \\
 & & u_2 & & \leq 0 \\
 & & & u_3 & \text{free} \\
 & 2u_1 & -u_2 & & \geq 1 \\
 & & u_2 & +u_3 & \leq 2 \\
 & u_1 & & -2u_3 & = 3 \\
 & & 3u_2 & -u_3 & \geq 4
\end{array}$$

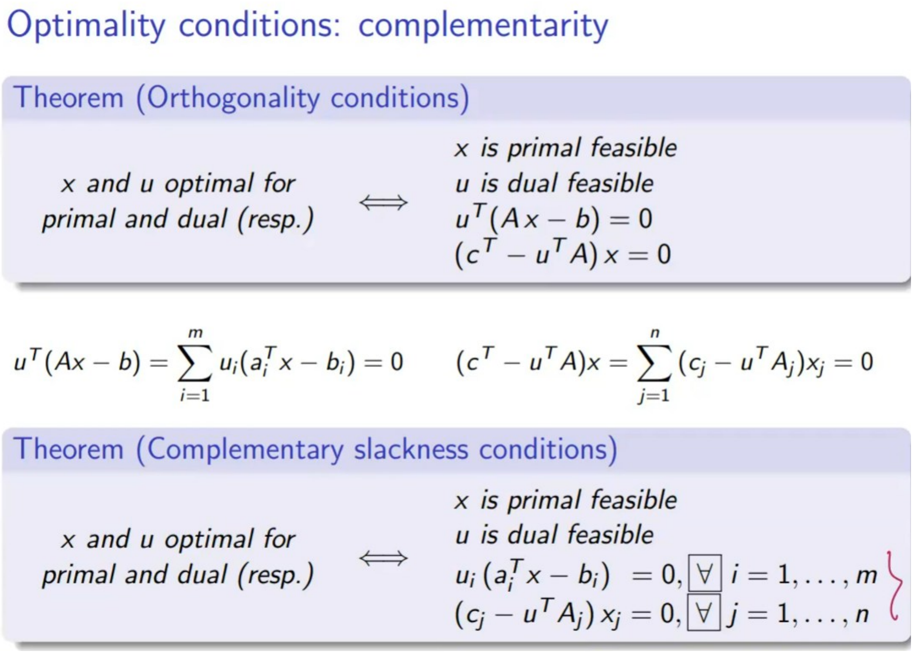
6.2 PRIMAL-DUAL OPTIMALITY CONDITIONS

The strong duality theorem provides optimality conditions: x^* and u^* are optimal solutions for the pair of problems if and only if (⇔):

- x^* is primal feasible, so $Ax^* \geq b \wedge x^* \geq 0$
- u^* is dual feasible, so $u^{*T}A \leq c^T \wedge u^* \geq 0$
- Strong duality holds, so $c^T x^* = u^{*T} b$

One can then think of directly applying the primal-dual optimality conditions by setting up a system of linear equations containing the constraints of the primal (equalities), the constraints of the dual (again the equalities) and adding as the last constraint the equality of the functions objective.

More formally, one can rewrite the optimality conditions as the following:



Optimality conditions: complementarity

Theorem (Orthogonality conditions)

$$x \text{ and } u \text{ optimal for primal and dual (resp.)} \iff \begin{cases} x \text{ is primal feasible} \\ u \text{ is dual feasible} \\ u^T(Ax-b)=0 \\ (c^T - u^T A)x = 0 \end{cases}$$

$$u^T(Ax-b) = \sum_{i=1}^{m} u_i(a_i^T x - b_i) = 0 \qquad (c^T - u^T A)x = \sum_{j=1}^{n} (c_j - u^T A_j)x_j = 0$$

Theorem (Complementary slackness conditions)

$$x \text{ and } u \text{ optimal for primal and dual (resp.)} \iff \begin{cases} x \text{ is primal feasible} \\ u \text{ is dual feasible} \\ u_i\,(a_i^T x - b_i) = 0, \boxed{\forall}\ i = 1,\dots,m \\ (c_j - u^T A_j)\,x_j = 0, \boxed{\forall}\ j = 1,\dots,n \end{cases}$$

Keeping in mind that for problem eligibility, all factors of the summations must be ≥ 0, we have that at the optimum it holds:

$$u_i(a_i^T x - b_i) = 0 \quad \forall i = 1 \dots m$$
$$(c_j - u^T A - J)x_j = 0 \quad \forall j = 1 \dots n$$

These conditions are met for each primary/dual constraint/variable. That is, two solutions x and u are optimal if and only if:

- a. Each positive primal variable $x_j > 0$ implies the saturated dual constraint $u^T A_j = c_j$ because (2) must be worth 0.
- b. Every loose dual constraint $u^T A_j < c_j$ implies the null primal variable $x_j = 0$ because 3.8 must be worth 0.
- c. Every positive dual variable $u_i > 0$ implies the saturated primal constraint $a_i^T x = b_i$ because (1) must be worth 0.
- d. Any loose primal constraint $a_i^T x > b_i$ implies the null dual variable $u_i = 0$ because 3.7 must be worth 0.

Because of these conditions, we can go and check whether a given solution is optimal or not by trying to construct a dual solution that is complementary to the given primal one.

$c^T x = u^T b$

$u^T(Ax - b) = 0 \quad u^T A x = u^T b$

$c^T x = u^T A x$

The orthogonality conditions state that for optimal solutions:

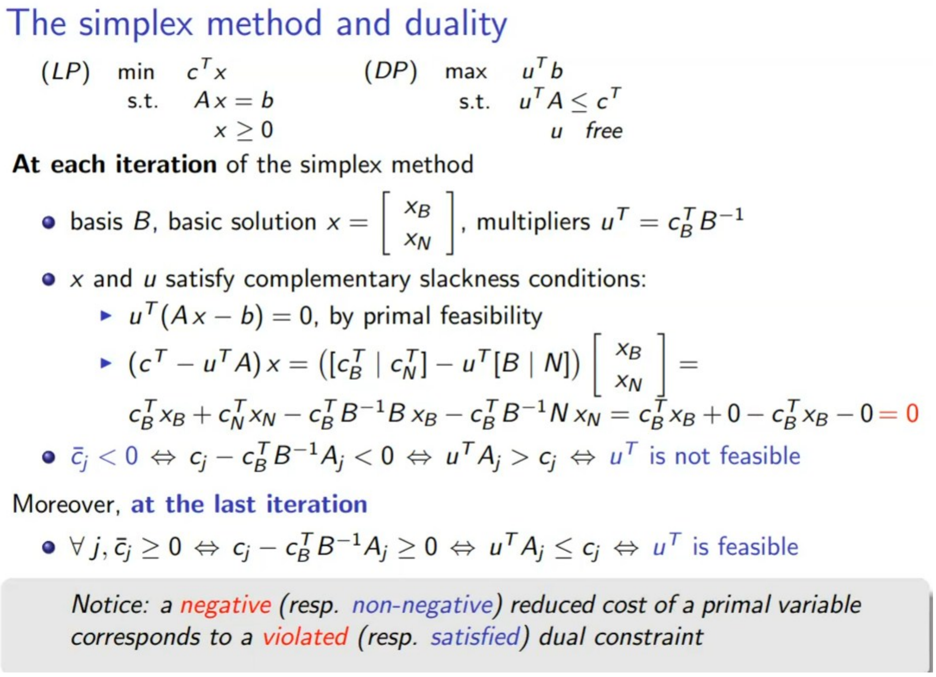
1. uT(Ax - b) = 0 This condition means that the dual variables (u) must be orthogonal to the slack in the primal constraints (Ax - b).
2. (cT - uTA)x = 0 This condition means that the primal variables (x) must be orthogonal to the slack in the dual constraints (cT - uTA).

Meanwhile, the complementary slackness conditions are powerful because they provide a way to verify optimality: if we have feasible primal and dual solutions that satisfy complementary slackness, those solutions must be optimal and one of them has to be zero when the other one has a value.

6.3 The Simplex Method and Duality

There is a connection between the simplex method and duality, because:

- During simplex iterations:
  - The multipliers give a dual solution (though not necessarily feasible)
  - Complementary slackness is always satisfied
  - Negative reduced costs indicate which dual constraints are violated
- At optimality:
  - All reduced costs are non-negative
  - The multipliers give a feasible dual solution
  - Both complementary slackness and feasibility are satisfied



The simplex method and duality

$$(LP)\quad \begin{array}{ll} \min & c^T x \\ \text{s.t.} & Ax = b \\ & x \geq 0 \end{array} \qquad (DP)\quad \begin{array}{ll} \max & u^T b \\ \text{s.t.} & u^T A \leq c^T \\ & u \ \textit{free} \end{array}$$

At each iteration of the simplex method

- basis B, basic solution $x = \begin{bmatrix} x_B \\ x_N \end{bmatrix}$, multipliers $u^T = c_B^T B^{-1}$
- x and u satisfy complementary slackness conditions:
  - $u^T(Ax - b) = 0$, by primal feasibility
  - $(c^T - u^T A)x = ([c_B^T \mid c_N^T] - u^T[B \mid N]) \begin{bmatrix} x_B \\ x_N \end{bmatrix} =$ $c_B^T x_B + c_N^T x_N - c_B^T B^{-1} B x_B - c_B^T B^{-1} N x_N = c_B^T x_B + 0 - c_B^T x_B - 0 = 0$
- $\bar{c}_j < 0 \Leftrightarrow c_j - c_B^T B^{-1} A_j < 0 \Leftrightarrow u^T A_j > c_j \Leftrightarrow u^T$ is not feasible

Moreover, **at the last iteration**

- $\forall j, \bar{c}_j \geq 0 \Leftrightarrow c_j - c_B^T B^{-1} A_j \geq 0 \Leftrightarrow u^T A_j \leq c_j \Leftrightarrow u^T$ is feasible

Notice: a negative (resp. non-negative) reduced cost of a primal variable corresponds to a violated (resp. satisfied) dual constraint

So, at each iteration:

- We have a basis and a basic solution derived by simplex multipliers
- Slackness holds because of primal feasibility, satisfying the linear system
- A negative reduced cost means dual constraint is violated

- At optimality (last iteration), all reduced costs are non-negative, all dual constraints are satisfied and multipliers from a feasible solution

Let us consider the (partial) proof seen for the strong duality theorem. We have seen that given an admissible solution of the basis and the corresponding multipliers of the simplex $u^T = C_B^T B^{-1}$ the condition “reduced cost of a variable with respect to the basis is nonnegative” is equivalent to saying “the corresponding dual constraint is satisfied by the dual solution obtained from the multipliers.” In fact, the definition of reduced cost traces the definition of the dual constraint:

$$\bar{c}_j = c_j - c_B^T B^{-1} A_j \geq 0 \Leftrightarrow c_j - u^T A_j \leq 0 \Leftrightarrow u^T A_j \leq c_j$$

Moreover, it can be seen that the multipliers themselves, viewed as solutions of the dual problem, are always, by construction, in complementary discards with the current admissible basis solution. In fact, considering that the problem in standard form has only equality constraints, the condition $u^T(Ax - b) = 0$ comes from the feasibility of the primal solution. For the condition $x_B = B^{-1}b$ and $x_F = 0$ in the basic solution, we have:

$$(c^T - u^T A)x = ([c_B^T | c_F^T] - u^T [B|F])[x_B | x_F] = c_B^T x_B + c_F^T x_F - c_B^T B^{-1} B x_B - c_B^T B^{-1} F x_F =$$
$$= c_B^T x_B + 0 - c_B^T x_B - 0 = 0.$$

In other words, x and u are a pair of primal-dual solutions in complementary slackness and this holds for *each simplex iteration*. We can therefore interpret the simplex in two ways:

- As a method that, at each step, determines an feasible primal solution and iteratively tries to make it optimal
- As a method that, at each step, determines a dual solution (the multipliers) in complementary scraps with a primal admissible solution and iteratively tries to make it dual admissible

In each case, *at the end* of the simplex we will have in hand a primal feasible solution and a dual feasible solution to each other in complementary scraps (and thus optimal primal and dual respectively).

- While running the simplex method, on the other hand, we will always have a pair of primal-dual solutions that are in complementary slackness, but with only the primal admissible, and thus the complementary slackness theorem does not apply except at the end of the simplex, when all reduced costs are non-negative (which is equivalent to saying that the multipliers are a dual admissible solution)

The reduced costs in simplex directly correspond to dual constraints:

- Negative reduced cost → violated dual constraint
- Non-negative reduced cost → satisfied dual constraint

6.4 Duality Example and Problem Modifications

Let’s go through a complete example so you can understand how it works. Consider the following problem:

$$\begin{aligned}&\min 2x_1 + 3x_2\\ &\text{s.t. } 3x_1 + x_2 \geq 11 \quad (u_1)\\ &\qquad x_2 \geq 2 \quad (u_2)\\ &\qquad x_1 \geq 1 \quad (u_3)\\ &\qquad x_1, x_2 \geq 0\end{aligned}$$

We want to check whether the solution x = (3,2) is optimal by applying the complementarity conditions.

- As a first step, it is necessary to test whether the solution is admissible by going to substitute within the constraints the values of the solution
- If all the constraints are met, the solution is admissible
- For the solution to also be optimal, it is then necessary to find a complementary solution for the dual problem:

$$\begin{aligned}&\max 11u_1 + 2u_2 + u_3\\ &\text{s.t. } 3u_1 + u_3 \leq 2\\ &\qquad u_1 + u_2 \leq 3\\ &\qquad u_1, u_2, u_3 \geq 0\end{aligned}$$

We then need to find equations to put into system to find the dual solution.

- The constraint 3x1 +x2 ≥ 11 is already at equality with the solution x1 = 3 and x2 = 2, so it gives no additional information
- Also x2 ≥ 2 is already equal with x2 = 2
- The constraint x1 ≥ 1 is not at equality and therefore, to make the complementarity condition satisfied, the dual variable associated with the constraint must be zero.
    - Therefore I derive u3 = 0
- Both primary variables are strictly greater than 0 and so I can impose the two equations associated with the variables at 0: 3u1 + u3 = 2 and u1 + u2 = 3

The final system of equations I obtain is:

$$\begin{cases} u_3 = 0 \\ 3u_1 + u_3 = 2 \\ u_1 + u_2 = 3 \end{cases} \quad \text{da cui ricavo} \quad \begin{cases} u_1 = 2/3 \\ u_2 = 7/3 \\ u_3 = 0 \end{cases}$$

We’re not done yet, because to make sure it is optimal, we have to check if it’s feasible. In this case it is, and therefore, that both primal/dual solutions are admissible and are complementary to each other, then they are also both optimal.

Suppose we have found the optimal solution of a primal-dual problem with the procedure just seen.

It may happen that the practical problem behind the model changes and it is necessary to find an optimal solution for the new problem. Obviously, one does not want to re-optimize the problem, but one wants to find out whether a solution for the new model is optimal. Suppose that the variable x3 is added to the previous problem and that it appears in the objective function with coefficient 2.

$$\begin{aligned}
&\min 2x_1 + 3x_2 + 2x_3 \\
&\text{s.t. } 3x_1 + x_2 + 2x_3 \geq 11 \quad (u_1) \\
&\qquad x_2 \geq 2 \quad (u_2) \\
&\qquad x_1 + x_3 \geq 1 \quad (u_3) \\
&\qquad x_1, x_2, x_3 \geq 0
\end{aligned}$$

The admissible solution given for this variant is the same as the previous one with x3 = 0. Since a variable was added to the primary problem, the dual obtains a new constraint:

$$\begin{aligned}
&\max 11u_1 + 2u_2 + u_3 \\
&\text{s.t. } 3u_1 + u_3 \leq 2 \\
&\qquad u_1 + u_2 \leq 3 \\
&\qquad 2u_1 + u_3 \leq 2 \\
&\qquad u_1, u_2, u_3 \geq 0
\end{aligned}$$

The previous dual solution does not change and remains feasible because it also satisfies the new constraint.

In this case I also know that they are optimal, because the dual solution was constructed in a complementary way, and the only thing that remains to be verified is that the complementarity also holds for the newly introduced primary variable-dual constraint pair, which is satisfied because in the solution x3 = 0.

It may happen, however, that the dual solution becomes infeasible. For example, if we add another variable x5 to the problem:

$$\begin{aligned}
&\min 2x_1 + 3x_2 + 2x_3 + 5.5x_5 \\
&\text{s.t. } 3x_1 + x_2 + 2x_3 + 2x_5 \geq 11 \quad (u_1) \\
&\qquad x_2 + 2x_5 \geq 2 \quad (u_2) \\
&\qquad x_1 + x_3 + 2x_5 \geq 1 \quad (u_3) \\
&\qquad x_1, x_2, x_3 \geq 0
\end{aligned}$$

The dual problem becomes:

$$\begin{aligned}
&\max 11u_1 + 2u_2 + u_3 \\
&\text{s.t. } 3u_1 + u_3 \leq 2 \\
&\qquad u_1 + u_2 \leq 3 \\
&\qquad 2u_1 + u_3 \leq 2 \\
&\qquad 2u_1 + 2u_2 + 2_u3 \leq 5.5 \\
&\qquad u_1, u_2, u_3 \geq 0
\end{aligned}$$

If in the starting optimal primal solution we also add x5 = 0 we still get an optimal solution admissible in complementary rejections with the dual one. The problem is that because of the new constraint, the dual solution is no longer admissible. The only thing that can be done in this case is to perform a new optimization. So: if a new variable (column) gets added to primal problem $\overline{x}$, the solutions remains optimal iff the dual solution $\overline{u}$ satisfies the dual constraint.

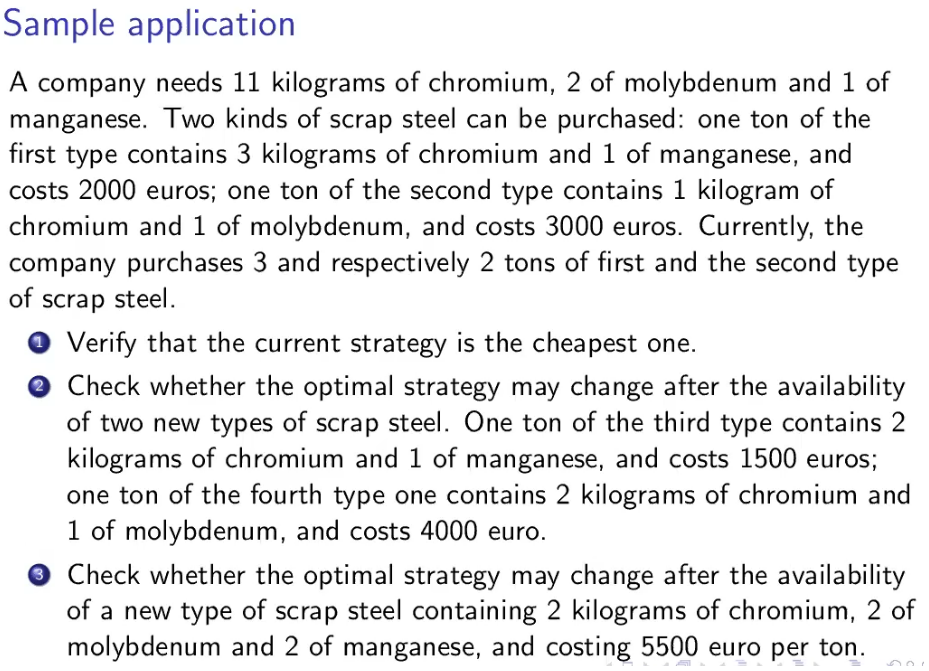
The fundamental insight is:

- Adding a column = Adding a dual constraint
- If dual solution violates new constraint = Column has negative reduced cost
- This means the column could improve solution

This leads naturally to *column generation* (which is the next theory module) because, instead of having all possible columns/variables:

- We work with a subset
- Use current dual values (u) to price potential new columns
- Only generate columns with negative reduced costs
- These are columns that could improve the solution

There is a specific example at the end of the slides (not treated in lessons – written by me below):



Sample application

A company needs 11 kilograms of chromium, 2 of molybdenum and 1 of manganese. Two kinds of scrap steel can be purchased: one ton of the first type contains 3 kilograms of chromium and 1 of manganese, and costs 2000 euros; one ton of the second type contains 1 kilogram of chromium and 1 of molybdenum, and costs 3000 euros. Currently, the company purchases 3 and respectively 2 tons of first and the second type of scrap steel.

1. Verify that the current strategy is the cheapest one.
2. Check whether the optimal strategy may change after the availability of two new types of scrap steel. One ton of the third type contains 2 kilograms of chromium and 1 of manganese, and costs 1500 euros; one ton of the fourth type one contains 2 kilograms of chromium and 1 of molybdenum, and costs 4000 euro.
3. Check whether the optimal strategy may change after the availability of a new type of scrap steel containing 2 kilograms of chromium, 2 of molybdenum and 2 of manganese, and costing 5500 euro per ton.

Let's define our variables: x1 = tons of type 1 scrap steel x2 = tons of type 2 scrap steel

We can write the constraints based on the requirements:

- Chromium: 3x1 + x2 ≥ 11 (need at least 11kg)
- Molybdenum: x2 ≥ 2 (need at least 2kg)
- Manganese: x1 ≥ 1 (need at least 1kg)
- Non-negativity: x1, x2 ≥ 0

The objective function (total cost) to minimize is: min z = 2000x1 + 3000x2

This is a linear programming problem. To solve part 1, we need to first verify if the current solution (x1=3, x2=2) is optimal.

Let’s rewrite this as a linear program and solve it using duality theory:

min z = 2000x1 + 3000x2 s.t. 3x1 + x2 ≥ 11 (u1) x2 ≥ 2 (u2) x1 ≥ 1 (u3) x1, x2 ≥ 0

The dual problem is: max 11u1 + 2u2 + u3 s.t. 3u1 + u3 ≤ 2000 u1 + u2 ≤ 3000 u1, u2, u3 ≥ 0

For x1=3, x2=2 to be optimal, we need to find dual variables that satisfy:

1. Dual feasibility
2. Complementary slackness conditions

The current solution x1=3, x2=2 satisfies all primal constraints:

- Chromium: 3(3) + 2 = 11 ≥ 11
- Molybdenum: 2 ≥ 2
- Manganese: 3 ≥ 1

Let's try dual values: u1=500, u2=2000, u3=500 These satisfy: 3(500) + 500 = 2000 500 + 2000 = 2500 ≤ 3000

The objective values match: Primal: 2000(3) + 3000(2) = 12000 Dual: 11(500) + 2(2000) + 1(500) = 12000

Since we found feasible dual variables that make the objectives equal, the current solution is indeed optimal. This verifies that the current strategy is the cheapest one.

For parts 2 and 3, we use similar analysis with the new variables/constraints:

Part 2: Adding two new variables makes new constraints but doesn't change optimality of current solution because:

- For type 3 scrap (2kg Cr, 1kg Mn for 1500€), this gives better cost per unit but adds new capacity. But optimal dual values show current solution remains optimal
- For type 4 scrap (2kg Cr, 1kg Mo for 4000€), cost is higher than type 2 scrap so won't improve solution

Part 3: The new type 5 scrap (2kg each of Cr, Mo, Mn for 5500€) does change optimal strategy because it can satisfy requirements with fewer tons needed, leading to lower total cost.

Therefore:

- Current strategy is optimal
- Strategy doesn't change with types 3 and 4 available
- Strategy does change with type 5 available

Side note: *the problem was presented a bit differently inside of Italian notes (here page 8 and onwards).*

To meet the demand for special steels, a manufacturing company needs 11 quintals of chromium, 2 quintals of molybdenum and 1 quintal of manganese. The market offers packages of two types. the first contains 3 kilograms of chromium and 1 of manganese and costs 200 euros; the second contains 1 kilogram of chromium and 1 of molybdenum and costs 300 euros.

Currently, the company purchases 300 packs of type 1 and 200 packs of type 2. It is desired to:

1. verify that the company implements an optimal procurement policy;

2. assess whether the policy should be changed due to the availability of a third and a fourth type of packages on the market. The third contains 2 kilograms of chromium and 1 of manganese and costs 200 euros. The fourth contains 2 kilograms of chromium and 1 of molybdenum and costs 400 euros.

3. consider whether the policy should be changed as a result of the availability on the market of a fifth type of packaging, containing 2 kilograms of chromium 2 of molybdenum and 2 of manganese, at a cost of 550 euros.

Note: the company is interested in knowing how the optimal supply policy is composed only approximately and expressed in hundreds of packages. For this reason, the model can be expressed by continuous variables and duality theory in linear programming can be applied.

Solution track: we first write the model of the problem. The variables are x_i: number of hundreds of packages of type $i = 1..2$ to be purchased. Since we are interested in an approximate solution, we can consider these variables continuous, rather than integer, as their nature would suggest.

Solving step 1 simply means checking the optimality of the solution $x_1 = 3, x_2 = 2$. The result is positive: the policy is optimal.

To solve point 2, consider that the new opportunities result, from the primary point of view, in two new variables. From the dual point of view, we have two new constraints ($2u_1 + u_3 \leq 2, 2u_1 + u_2 \leq 4$). It should be noted that, given the addition of more constraints to the dual problem, the optimal solution of the dual problem itself cannot improve but remain the same (if it does not violate the new constraints) or get worse (if the old optimal solution violates the constraints). Thus, with the addition of the two new dual constraints, two cases can occur:

- (a) The constraints are verified by the optimal dual variables obtained in Step 1
  - Then the optimal dual solution does not change and, due to strong duality, neither does the optimal solution of the primal, i.e., the addition of two new alternatives does not affect the optimality of the policy currently adopted by the firm
    - We achieve the optimal value of the objective function even if values of new variables remain at 0
    - It can be shown in this case that, if the dual constraints are satisfied, we are in the presence of a primary admissible solution and a dual admissible solution in complementary scraps, thus optimal solutions. You can find this in the Italian notes above quoted

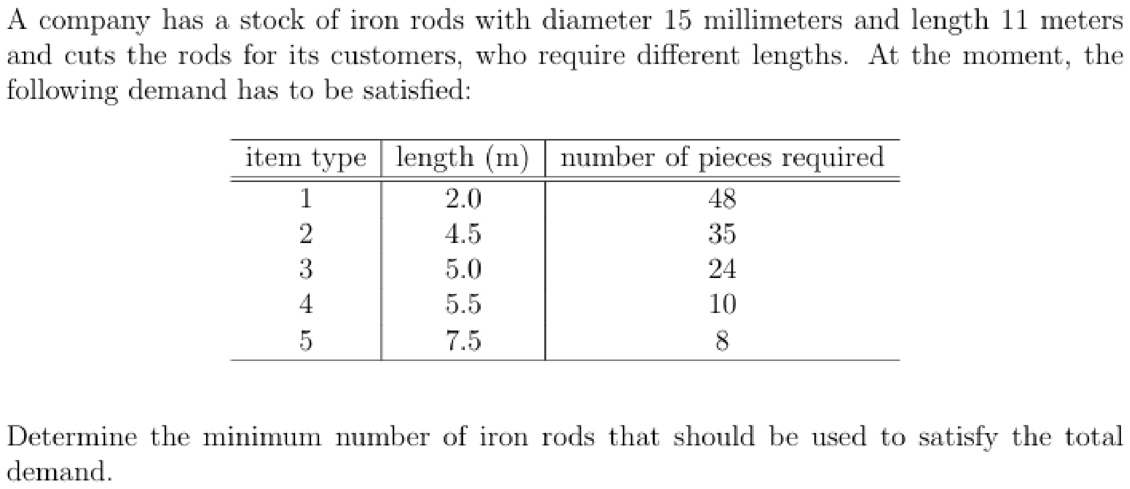
- (b) The constraints are not verified by the optimal dual variables obtained in step 1. So the optimal dual solution changes and, in particular, having added additional constraints, it gets worse, i.e., it decreases (dual objective function of max)
    - Again due to strong duality, the value of the objective function of the corresponding primal (the one with two new variables) will be equal to the new optimal value of the dual, thus lower than before.
    - As a result, the current policy could be improved by taking advantage of the new packages offered by the market
        - As alternative proof we would be in the presence of feasible primal solution in complementary slackness with an unfeasible dual solution, i.e., the two solutions are not optimal for their respective problems

Result: case (a).

The solution of point 3 is similar to point 2, with the outcome of having the policy adopted be not optimal and should be changed.

7 COLUMN GENERATION METHODS (6)

Let's start by considering the following problem (tondini di ferro – iron rods):



A company has a stock of iron rods with diameter 15 millimeters and length 11 meters and cuts the rods for its customers, who require different lengths. At the moment, the following demand has to be satisfied:

item type	length (m)	number of pieces required
1	2.0	48
2	4.5	35
3	5.0	24
4	5.5	10
5	7.5	8

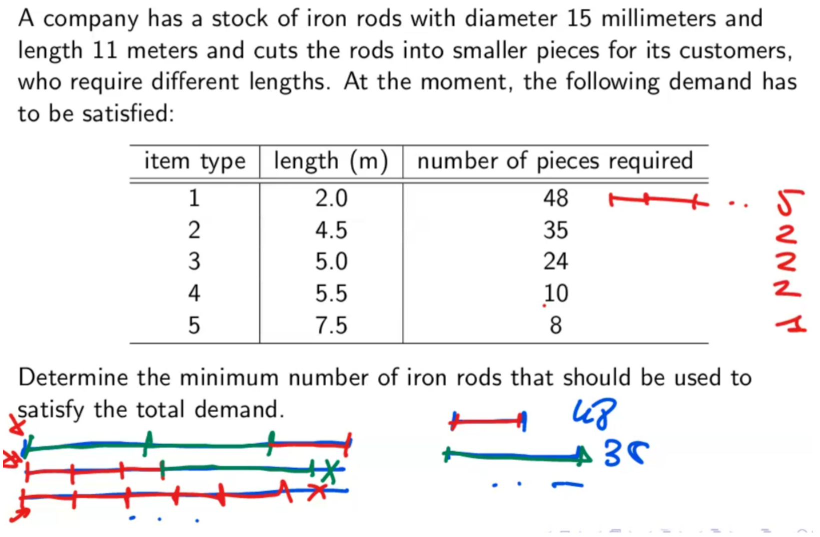
Determine the minimum number of iron rods that should be used to satisfy the total demand.

7.1 AN INTERESTING PROBLEM: CUTTING RODS – MODEL AND SOLUTION

We have several ways of cutting this; we want to decide "how" to cut all of these pieces and how many rods we want to cut with such technique. There are as many ways as the types of pieces to be cut here, but also industrially we may have a limited number of cuts to be executed.

- The problem structure presents a fundamental challenge typical of *column generation* applications: the number of possible cutting patterns (ways to cut the rods) is *extremely large and impractical to enumerate explicitly*
- For example, even with just these five different lengths, there are numerous possible combinations of cuts that could be made from an 11-meter rod while satisfying various piece requirements.

The real value of this example lies in how clearly it demonstrates the core principle of column generation: rather than dealing with an enormous number of variables upfront, we can work with a manageable subset and generate additional variables (columns) only when they have the potential to improve our solution.

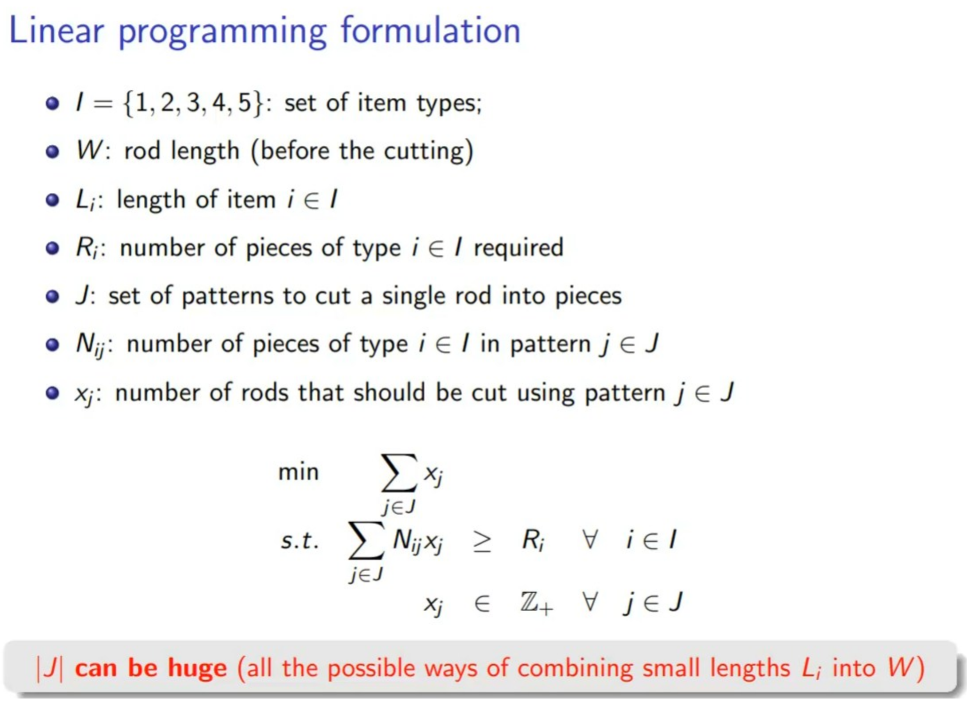


A company has a stock of iron rods with diameter 15 millimeters and length 11 meters and cuts the rods into smaller pieces for its customers, who require different lengths. At the moment, the following demand has to be satisfied:

item type	length (m)	number of pieces required
1	2.0	48
2	4.5	35
3	5.0	24
4	5.5	10
5	7.5	8

Determine the minimum number of iron rods that should be used to satisfy the total demand.

A possible model for the problem, proposed by Gilmore and Gomory in 1960 (see here) is the following – consider there are so many ways to solve this problem, exponential even! You do not have enough memory to create all of that for sure.



Linear programming formulation

- $I = \{1,2,3,4,5\}$: set of item types;
- W: rod length (before the cutting)
- L_i: length of item $i \in I$
- R_i: number of pieces of type $i \in I$ required
- J: set of patterns to cut a single rod into pieces
- N_{ij}: number of pieces of type $i \in I$ in pattern $j \in J$
- x_j: number of rods that should be cut using pattern $j \in J$

$$\begin{array}{lrcll} \min & \sum_{j \in J} x_j & & & \\ s.t. & \sum_{j \in J} N_{ij} x_j & \geq & R_i & \forall\ i \in I \\ & x_j & \in & \mathbb{Z}_+ & \forall\ j \in J \end{array}$$

$|J|$ **can be huge** (all the possible ways of combining small lengths L_i into W)

The model is very elegant since we do not worry about the feasibility constraints of the cuts, since the matrix of possible combinations only contains the valid ones. There are still other things to note:

- It assumes the availability of the set J and the parameters N_{ij}
- In order to generate this data, one needs to enumerate all possible cutting patterns
  - It is easy to realize that the number of possible cutting patterns is huge, and therefore direct implementation of the above model is unpractical for real-world instances
- So, two problems – integer variables and matrix of the cuts which can be too big!

We remark that it makes sense to solve the *continuous relaxation* of the above model.

- This is because, in practical situations, the demands are so high that the number of rods cut is also very large, and therefore a good heuristic solution can be determined by rounding up to the next integer each variable x_j found by solving the continuous relaxation
- Moreover, the solution of continuous relaxation may constitute the starting point for the application of an exact solution method (for instance, Branch-and Bound – next module)

We therefore analyze how to solve the continuous relaxation of the model ($x_j \in \mathbb{R}^+$). Such a solution can be constructed as follows:

- Consider single-item cutting patterns, i.e., $|I|$ configurations, each containing $N_{ii} = \lfloor \frac{W}{Li} \rfloor$ pieces of type i
  - In words: given a rod, produce only a type of piece and get the max possible
- Set $x_i = \frac{R_i}{N_{ii}}$ for pattern i (where pattern i is the pattern containing only pieces of type i)
  - In words, the number of times a pattern is applied is given rounding by excess the ratio between piece request and number of pieces produced by the schema

So: start from a relaxed version of the problem using a subset of the cutting patterns, making a good choice so to make sure there exists a feasible solution and cut off integrality constraints, since in reality we might have production waste.

The same solution can be obtained by *applying the simplex method to the model* (simple, since it's without integrality constraints), where only the decision variables corresponding to the above single-item patterns are considered (restrict to a subset of J). Here the relaxed version solved using simplex:

$$\begin{array}{llllllllllll}
\min & x_1 & + & x_2 & + & x_3 & + & x_4 & + & x_5 & & \\
s.t. & 5x_1 & & & & & & & & & \geq & 48 \\
 & & & 2x_2 & & & & & & & \geq & 35 \\
 & & & & & 2x_3 & & & & & \geq & 24 \\
 & & & & & & & 2x_4 & & & \geq & 10 \\
 & & & & & & & & & x_5 & \geq & 8 \\
 & x_1 & & x_2 & & x_3 & & x_4 & & x_5 & \geq & 0
\end{array}$$

$$u = (c_B^T B^{-1})^T = \begin{bmatrix} 0.2 \\ 0.5 \\ 0.5 \\ 0.5 \\ 1.0 \end{bmatrix}$$

$$\text{In fact, } x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 9.6 \\ 17.5 \\ 12.0 \\ 5.0 \\ 8.0 \end{bmatrix} \text{corresponding to the basis } B = \begin{bmatrix} 5 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Consider now a new possible pattern (number 6), containing one piece of type 1 and one piece of type 5. We ask ourselves: does the previous solution remain optimal if this new pattern is allowed? As we saw, we can answer a question like this by using *duality* or *simplex theory* (previous solution holds real values – hence not allowed – let's try to add then a pattern to try taking a better solution!)

- Recall that at every iteration the simplex method yields a feasible basic solution (corresponding to some basis B) for the primal problem and a dual solution (the multipliers) that satisfy the complementary slackness conditions
    - The dual solution will be feasible only at the last iteration
- The new pattern number 6 corresponds to including a new variable in the primal problem, with objective cost 1 (as each time pattern 6 is chosen, one rod is cut) and corresponding to the following column in the constraint matrix:

$$A_6 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

This variable creates a new dual constraint. We then have to check if this new constraint is violated by the current dual solution (u^T), i.e., if the reduced cost of the new variable with respect to basis B is negative. The new dual constraint and also the dual solution of relaxed problem are the following:

$$1u_1 + 0u_2 + 0u_3 + 0u_4 + 1u_5 \leq 1.$$

$$\text{The current dual solution associated to } B \text{ is } u^T = c_B^T B^{-1} = \begin{bmatrix} 0.2 & 0.5 & 0.5 & 0.5 & 1 \end{bmatrix}.$$

Considering the dual solution corresponding to the current optimal solution $u = c_B^T B^{-1}$, we get $0.2 + 1 = 1.2 > 1$, the new constraint is violated. This means that the current primal solution (in which the new variable is $x_6 = 0$) may not be optimal anymore (although it is still feasible).

We can verify that the fact that the dual constraint is violated corresponds to the fact that the *associated primal variable has negative reduced cost*:

$$\bar{c}_6 = c_6 - u^T A_6 = 1 - \left[\begin{array}{ccccc} 0.2 & 0.5 & 0.5 & 0.5 & 1 \end{array}\right] \left[\begin{array}{c} 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{array}\right] = -0.2.$$

It is then convenient to let x_6 enter the basis, since it means there is room for improvement for the primal relaxed solution. To do so, we modify the problem by inserting the new variable:

$$\begin{array}{lcccccccccccccc}
\min & x_1 & + & x_2 & + & x_3 & + & x_4 & + & x_5 & + & x_6 & & \\
s.t. & 5x_1 & & & & & & & & & + & x_6 & \geq & 48 \\
 & & & 2x_2 & & & & & & & & & \geq & 35 \\
 & & & & & 2x_3 & & & & & & & \geq & 24 \\
 & & & & & & & 2x_4 & & & & & \geq & 10 \\
 & & & & & & & & & x_5 & + & x_6 & \geq & 8 \\
 & x_1 & & x_2 & & x_3 & & x_4 & & x_5 & & x_6 & \geq & 0
\end{array}$$

If this problem is solved with the simplex method, the optimal solution is found but restricted only to patterns 1,…,6.

- If a new pattern is available, one can decide whether this new pattern should be used or not by proceeding as above (so: continue until an optimal relaxed solution not improvable is found)
- However, the problem is how to find a pattern (i.e., a variable; i.e., a column of the matrix) whose reduced cost is negative (i.e., it is convenient to include it in the formulation)
- Problem is: which columns to choose, since the set is very big and those are not defined explicitly (= not immediate to find a variable respecting this logic)
- Not only is the number of possible patterns exponentially large, but the patterns are not even known explicitly! The question then is:

> *Given a basic optimal solution for the problem in which only some variables are included, how can we find (if any exists) a variable with negative reduced cost (i.e., a constraint violated by the current dual solution)?*

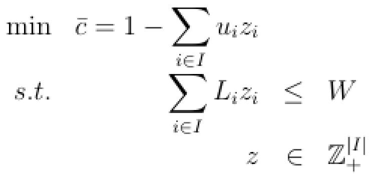
This question can be transformed into an optimization problem: in order to see whether a variable with negative reduced cost exists, we can look for the minimum of the reduced costs of all possible variables and check whether this minimum is negative:

$$\begin{array}{ll}
\min & \bar{c} = 1 - u^T z \\
s.t. & z \text{ is a possible column of the constraint matrix.}
\end{array}$$

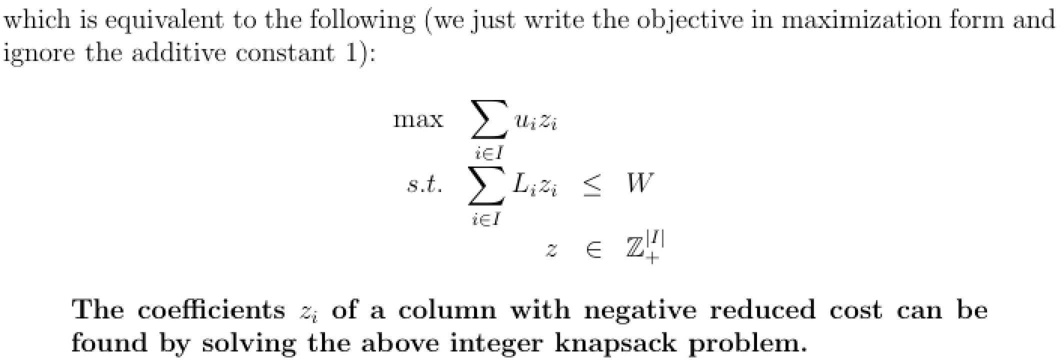
Recall that every column of the constraint matrix corresponds to a possible cutting pattern, and every entry of the column says how many pieces of a certain type are in that pattern. In order for z to be a possible column of the constraint matrix, the following condition must be satisfied:

$$z \in \mathbb{Z}_+^{|I|}$$
$$\sum_{i \in I} L_i z_i \leq W$$

Then the problem of finding a variable with negative reduced cost can be converted into the following ILP problem – basically, we transformed the solution into a *partial* problem, where we solve a different subproblem using the *knapsack* problem:



$$\begin{array}{lrcl}\min & \bar{c} = 1 - \sum_{i \in I} u_i z_i & & \\ s.t. & \sum_{i \in I} L_i z_i & \leq & W \\ & z & \in & \mathbb{Z}_+^{|I|}\end{array}$$

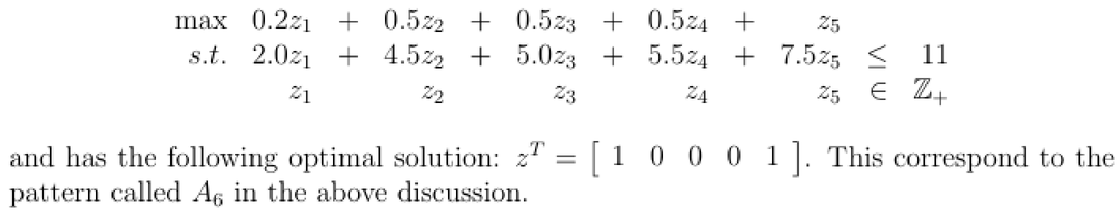


which is equivalent to the following (we just write the objective in maximization form and ignore the additive constant 1):

$$\begin{array}{lrcl}\max & \sum_{i \in I} u_i z_i & & \\ s.t. & \sum_{i \in I} L_i z_i & \leq & W \\ & z & \in & \mathbb{Z}_+^{|I|}\end{array}$$

The coefficients z_i of a column with negative reduced cost can be found by solving the above integer knapsack problem.

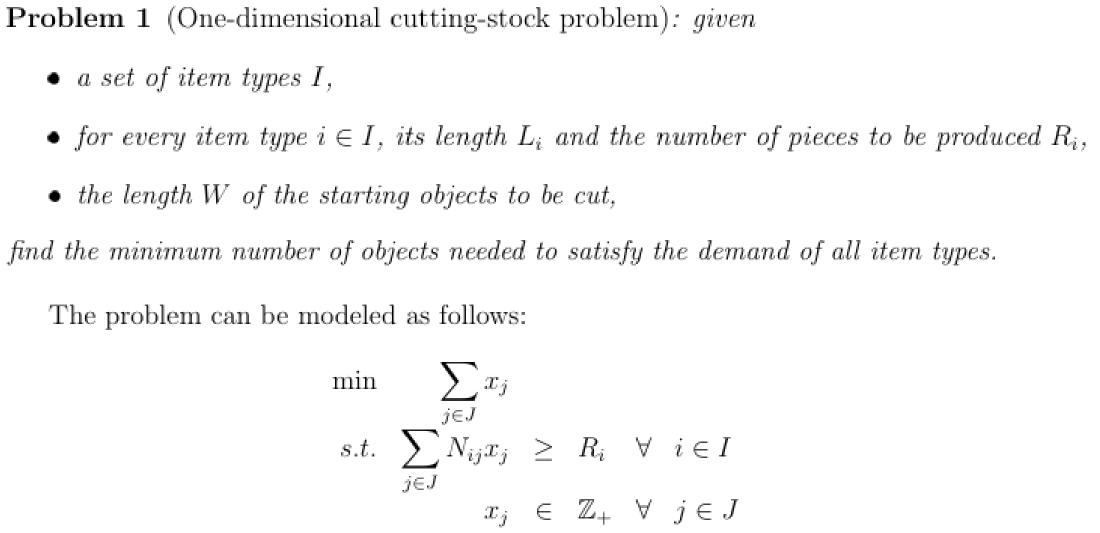
In our example, if we start from the problem restricted to the five single-item patterns, the above problem reads as:



$$\begin{array}{lccccccccccc}\max & 0.2z_1 & + & 0.5z_2 & + & 0.5z_3 & + & 0.5z_4 & + & z_5 & & \\ s.t. & 2.0z_1 & + & 4.5z_2 & + & 5.0z_3 & + & 5.5z_4 & + & 7.5z_5 & \leq & 11 \\ & z_1 & & z_2 & & z_3 & & z_4 & & z_5 & \in & \mathbb{Z}_+\end{array}$$

and has the following optimal solution: $z^T = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \end{bmatrix}$. This correspond to the pattern called A_6 in the above discussion.

The procedure described above can be generalized to an algorithm for one-dimensional cutting-stock problems, where the strategy is based upon continuous relaxation and application of a rounding heuristic.



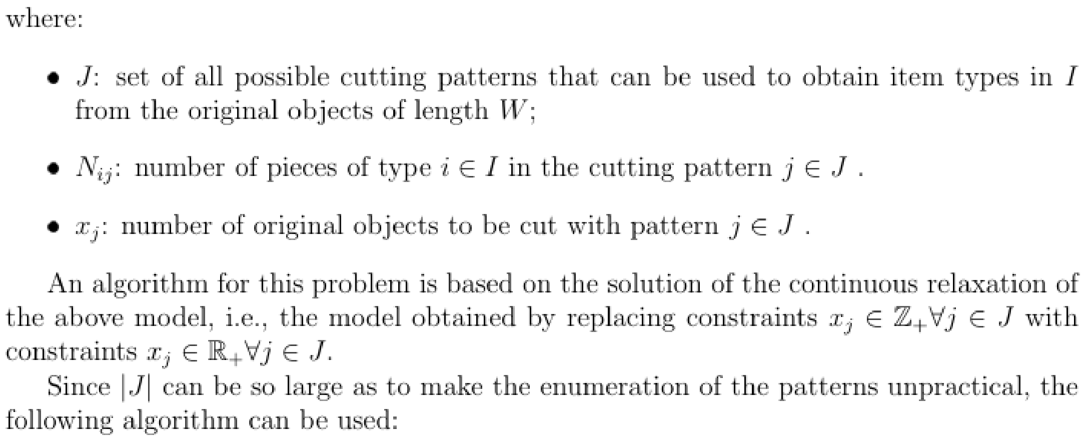
Problem 1 (One-dimensional cutting-stock problem)*: given*

- *a set of item types I,*
- *for every item type $i \in I$, its length L_i and the number of pieces to be produced R_i,*
- *the length W of the starting objects to be cut,*

find the minimum number of objects needed to satisfy the demand of all item types.

The problem can be modeled as follows:

$$\begin{array}{lrclcl}\min & \sum_{j \in J} x_j & & & & \\ s.t. & \sum_{j \in J} N_{ij} x_j & \geq & R_i & \forall & i \in I \\ & x_j & \in & \mathbb{Z}_+ & \forall & j \in J\end{array}$$



where:

- J: set of all possible cutting patterns that can be used to obtain item types in I from the original objects of length W;
- N_{ij}: number of pieces of type $i \in I$ in the cutting pattern $j \in J$.
- x_j: number of original objects to be cut with pattern $j \in J$.

An algorithm for this problem is based on the solution of the continuous relaxation of the above model, i.e., the model obtained by replacing constraints $x_j \in \mathbb{Z}_+ \forall j \in J$ with constraints $x_j \in \mathbb{R}_+ \forall j \in J$.

Since $|J|$ can be so large as to make the enumeration of the patterns unpractical, the following algorithm can be used:

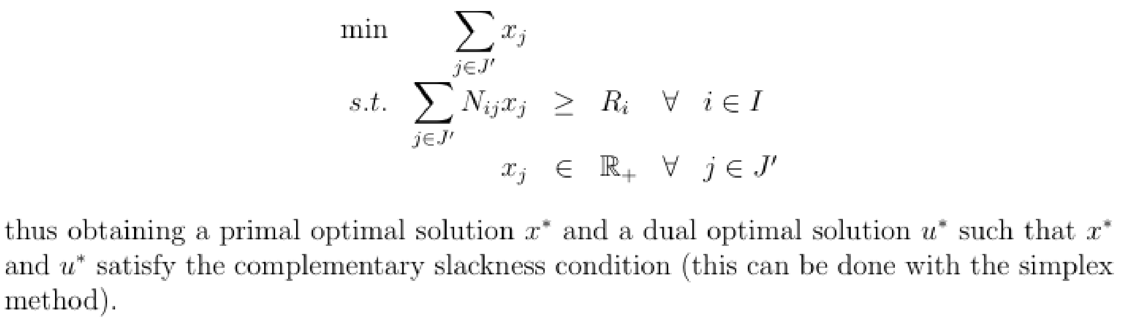
7.2 ALGORITHM FOR THE 1D-CSP

Step 0 – Initialization

Choose a subset J' of the cut patterns, such that the problem admits solution (feasible). An example of a subset is given by all the mono-cut schemes (e.g., $card(I)$ for single-item patterns).

Step 1 – Solving the master problem

Solve the *master* problem, considering only the previously defined subset of patterns. This results in an primal optimal solution x^* and a corresponding optimal dual u^* solution, which is in complementary slackness with x^*. This process can be done by simplex method.

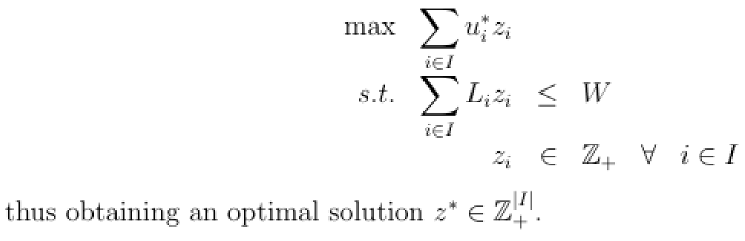


$$\begin{array}{lrcll} \min & \sum_{j \in J'} x_j & & & \\ s.t. & \sum_{j \in J'} N_{ij} x_j & \geq & R_i & \forall \ i \in I \\ & x_j & \in & \mathbb{R}_+ & \forall \ j \in J' \end{array}$$

thus obtaining a primal optimal solution x^* and a dual optimal solution u^* such that x^* and u^* satisfy the complementary slackness condition (this can be done with the simplex method).

Step 2 - Solving the sub-problem (slave problem)

Solve the *slave* problem (using the simplex method, to solve the primal problem) for determining the column to be introduced (i.e., the optimal solution z^* – a dual solution). The problem has variable $card(I)$ and only one constraint, thus obtaining the optimal solution.



$$\begin{array}{lrcll} \max & \sum_{i \in I} u_i^* z_i & & & \\ s.t. & \sum_{i \in I} L_i z_i & \leq & W & \\ & z_i & \in & \mathbb{Z}_+ & \forall \ i \in I \end{array}$$

thus obtaining an optimal solution $z^* \in \mathbb{Z}_+^{|I|}$.

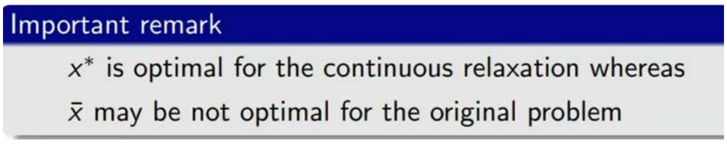
Step 3 - Optimality test (and heuristic solution to the starting problem)

If $\sum_{i\in I} u_i^* z_i^* \le 1$, then STOP: x^* is an *optimal solution* of the full continuous relaxation (including all patterns in J). Otherwise, *update the master problem* by including in J' the cutting pattern γ defined by $N_{i\gamma} = z_i^*$ (this means that column z^* has to be included in the master problems constraints matrix) and go to Step 1.

Finally, to go from the optimal solution of continuous relaxation x^* to a *heuristic* solution (i.e., not necessarily optimal but hopefully good) of the original problem (with integrality constraints), is possible, alternatively to:

- Round up by excess the entries of x^* (this is a good choice if these entries are large: 765.3 is not very different from 766…); note that rounding down is not allowed, as we would create an unfeasible integer solution
- Apply an ILP method (for instance Branch-and Bound) to the last master problem that was generated; this means solving the original problem (with integrality constraints) restricted to the only "good" patterns (those in J') generated by the solution of sub-problems

In either case you lose the guarantee of the optimality of the solution, but you still get a reasonably good solution.



Important remark

x^* is optimal for the continuous relaxation whereas

$\bar{x}$ may be not optimal for the original problem

7.3 Column generation methods for LP Problems

The idea developed above for the one-dimensional cutting-stock problem can be applied to more general LP problems (NOT integer, at least directly) whenever it is not possible or convenient to list explicitly all possible decision variables. This happens because of *simplex theory*.

Consider the following generic problem:

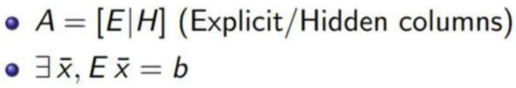
$$\begin{aligned}(P)\ \min\ & c^T x\\ \text{s.t.}\ \ & Ax = b\\ & x \ge 0\end{aligned}$$

such that the number of variables/columns (n) of A is very large or not known a priori, the algorithm becomes the following – let (D) be the dual problem of (P):

$$\begin{array}{llcl}(D) & \max & u^T b & \\ & \text{s.t.} & u^T A & \le\ c^T \\ & & u & \quad free\end{array}$$

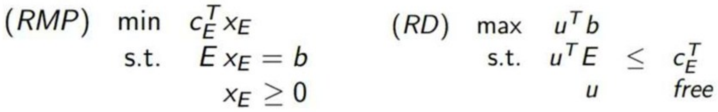
Step 0: Initialization

Find explicitly a (small) subset of columns of A such that, if only these columns are considered, the problem has a feasible solution. Let $E \in \mathbb{R}^{m\times q}\,(q \ll n)$ denote this submatrix of A s.t. it's composed only by the selected subset columns and let x_E, c_E be the corresponding vectors of variables and costs in the objective function. It's important the problem related to E is limited and feasible.

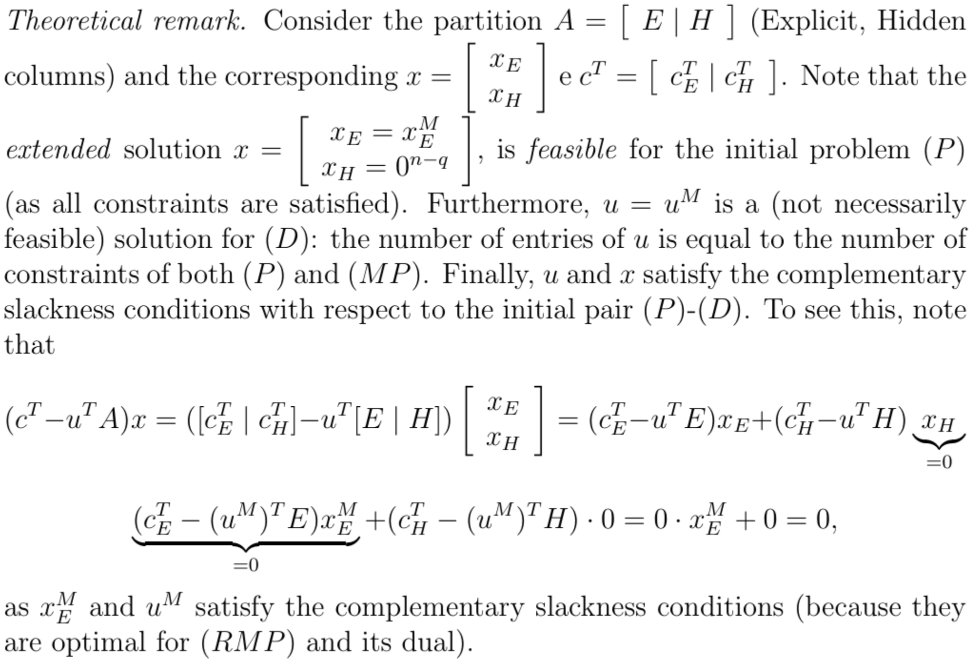


- $A = [E|H]$ (Explicit/Hidden columns)
- $\exists \bar{x}, E\bar{x} = b$

Step 1: Solve the Restricted Master Problem (RMP) obtaining x_E^M, u^M



$$(RMP)\quad \begin{array}{ll} \min & c_E^T x_E \\ \text{s.t.} & E\, x_E = b \\ & x_E \geq 0 \end{array} \qquad\qquad (RD)\quad \begin{array}{llll} \max & u^T b & & \\ \text{s.t.} & u^T E & \leq & c_E^T \\ & u & & \textit{free} \end{array}$$



Theoretical remark. Consider the partition $A = \left[\; E \mid H \;\right]$ (Explicit, Hidden columns) and the corresponding $x = \begin{bmatrix} x_E \\ x_H \end{bmatrix}$ e $c^T = \left[\; c_E^T \mid c_H^T \;\right]$. Note that the *extended* solution $x = \begin{bmatrix} x_E = x_E^M \\ x_H = 0^{n-q} \end{bmatrix}$, is *feasible* for the initial problem (P) (as all constraints are satisfied). Furthermore, $u = u^M$ is a (not necessarily feasible) solution for (D): the number of entries of u is equal to the number of constraints of both (P) and (MP). Finally, u and x satisfy the complementary slackness conditions with respect to the initial pair (P)-(D). To see this, note that

$$(c^T - u^T A)x = ([c_E^T \mid c_H^T] - u^T[E \mid H]) \begin{bmatrix} x_E \\ x_H \end{bmatrix} = (c_E^T - u^T E)x_E + (c_H^T - u^T H)\underbrace{x_H}_{=0} =$$

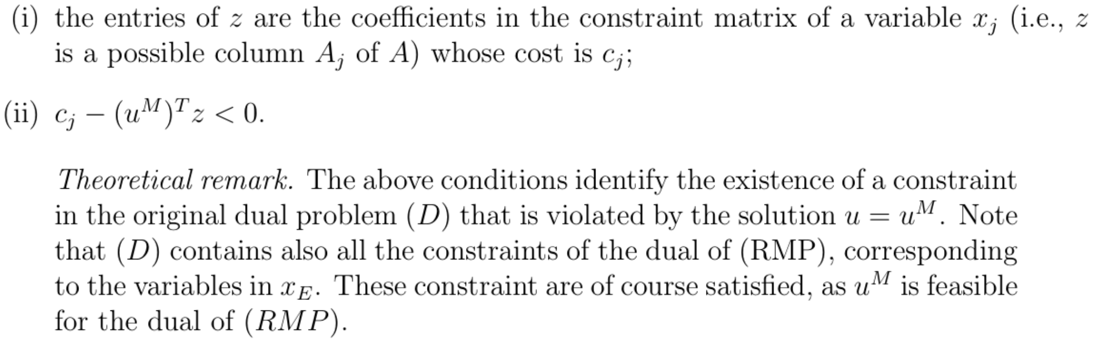
$$\underbrace{(c_E^T - (u^M)^T E)x_E^M}_{=0} + (c_H^T - (u^M)^T H) \cdot 0 = 0 \cdot x_E^M + 0 = 0,$$

as x_E^M and u^M satisfy the complementary slackness conditions (because they are optimal for (RMP) and its dual).

A pair of feasible solutions and optimal primal-dual for MP is obtained, for example, using the simplex method.

Step 2: Solution of the slave problem (sub-problem for the generation of a new column)

Find one or more vectors $z \in \mathbb{R}^m$ satisfying the following conditions:



(i) the entries of z are the coefficients in the constraint matrix of a variable x_j (i.e., z is a possible column A_j of A) whose cost is c_j;

(ii) $c_j - (u^M)^T z < 0$.

Theoretical remark. The above conditions identify the existence of a constraint in the original dual problem (D) that is violated by the solution $u = u^M$. Note that (D) contains also all the constraints of the dual of (RMP), corresponding to the variables in x_E. These constraint are of course satisfied, as u^M is feasible for the dual of (RMP).

To ensure the efficiency of the algorithm, this step needs to be performed quickly, and to limit the number of iterations, one may choose to generate more than one column at a time. In addition, the algorithm to be applied varies from problem to problem.

Step 3: Optimality test

If no vector z from the previous step exists, then STOP: $x = \begin{bmatrix} x_E^M \\ 0 \end{bmatrix}$ is an optimal solution of the initial problem (P). If no new columns are found, this means there are no dual constraints violated and also the dual solution is optimal.

Theoretical remark. As we saw, $x = \begin{bmatrix} x_E = x_E^M \\ x_H = 0^{n-q} \end{bmatrix}$ and $u = u^M$ are a primal-dual pair of solutions for (P)-(D) satisfying the complementary slackness conditions. The fact that the slave problem is infeasible means that no constraint of (D) is violated, i.e., $u = u^M$ is feasible for (D). We then have pair of *feasible* solutions for (P)-(D) satisfying the complementary slackness conditions. By the strong duality theorem, x and u are optimal for (P) and (D).

Step 4: Iteration

Update the master problem by including in matrix E one (or more) columns generated at Step 2; also update the corresponding costs in x_E and c_E. Go to Step 1 (so if new columns, add them back to the original matrix). Note that as the algorithm execution continues, the problem (MP) may become too complex, so you may choose to maintain a pool of active columns.

Theoretical remark. As we saw, violated dual constraints correspond to variables with negative reduced cost; thus these variables are worth being included in the problem to improve the objective value.

7.4 IMPLEMENTATION ISSUES – CONVERGENCE

The critical part of the method is Step 2, i.e., generating the new columns (solving slave problem). It is not reasonable to compute the reduced costs of all variables x_j for $j = 1, \dots, n$, otherwise this procedure would reduce to the simplex method. In fact, n can be very large (as in the cutting-stock problem) or, for some reason, it may not be possible or convenient to enumerate all decision variables.

- It is then necessary to construct a specific column generation algorithm for each problem; only if such an algorithm exists (and is efficient), can the method be fully developed

- In the one-dimensional cutting stock problem we transformed the column generation subproblem into a reasonable ILP (Branch and Bound or dynamic programming). In other cases, the computational effort required to solve the subproblem may be so high as to make the full procedure unpractical (in general NP-Hard or just inefficient overall)

A column generation algorithm considers, at each iteration, a primal-dual pair of feasible solutions. In order for Step 1 to be able to find such a pair, the *master problem needs to be always feasible and bounded*.

- At the first iteration feasibility can be achieved by taking any feasible solution for (P) and including in E only the columns corresponding to variables that take a strictly positive value in this solution
- At the next iterations, if the method adds new variables, the new master problems will be feasible because the initial variables will still be included in the model. Moreover, to ensure

boundedness, one can impose *box constraints*, i.e., constraints of the type $x_j \leq M, \forall j \in E$ (where M is a sufficiently large constant)

- In many cases such a value of M can be easily determined (for instance, in the rod cutting problem it is easy to find a safe upper bound M on the number of rods needed) and introducing them step by step to Step 4, boundedness is guaranteed

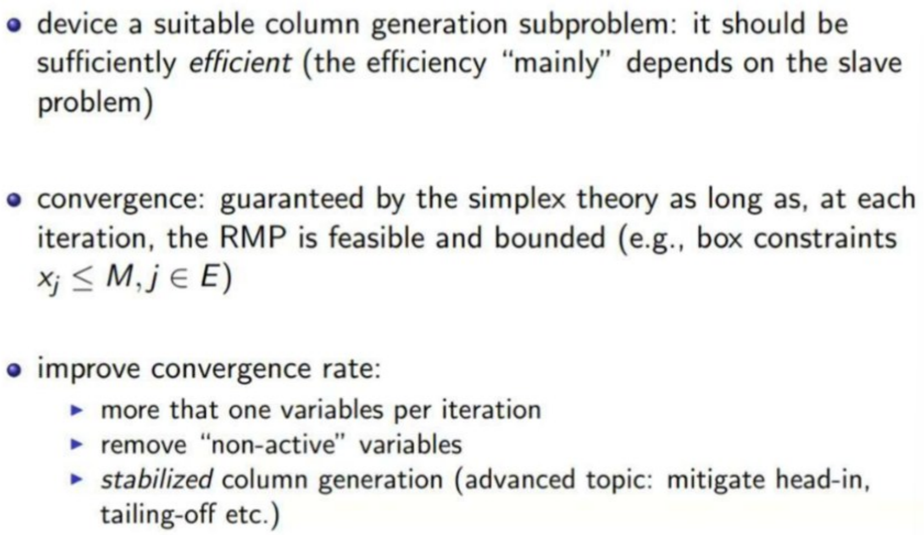
The convergence rate of column generation methods is guaranteed by the theory of the simplex method, provided that the column generation subproblem can be solved by an existing exact algorithm. However, from the practical point of view, convergence might be slow for several reasons (we only mention some of them below).

- One issue is the following: if, at Step 4, a single variable is introduced, many iterations may be needed before including all variables needed in an optimal solution of the original problem. To overcome this problem, when possible include and find more than one new variable at every iteration (so, find more columns at Step 2 to insert them into MP into Step 4)
- Another issue is the fact that, after some iterations, problem (RMP) will contain a large number of variables, and therefore solving (RMP) may become very hard. One way of overcoming this is the creation of a pool of non-active variables among all the variables introduced so far

In other words, the variables whose value has been zero for several iterations can be eliminated from the model but kept in a pool. However, when doing this, one has to ensure that the elimination of some variables does not make the problem infeasible.

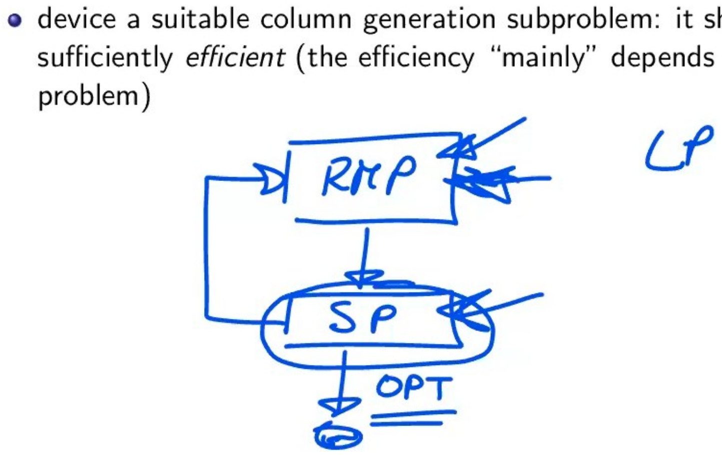
- If this approach is adopted, at every iteration one can check if one of the columns already generated but currently removed has negative reduced cost; only if this is false, a new variable will be generated
- Some other problems, not covered here, are known as *instability, tailing-off, head-in* etc.: dealing with this aspects is fundamental for the implementation of efficient column generation methods (*stabilized column generation*)

The basic notion coming from the theory is the fact if we have integer variables, this does not work. Even if we solve it to optimality with integer variables, we only solve the subproblem. To summarize:



- device a suitable column generation subproblem: it should be sufficiently *efficient* (the efficiency "mainly" depends on the slave problem)
- convergence: guaranteed by the simplex theory as long as, at each iteration, the RMP is feasible and bounded (e.g., box constraints $x_j \leq M, j \in E$)
- improve convergence rate:
  - more that one variables per iteration
  - remove "non-active" variables
  - *stabilized* column generation (advanced topic: mitigate head-in, tailing-off etc.)

We solve the RMP by decomposition, since we solve the problem by linear relaxation to optimality (1CSD). The RMP is a LP problem, but the SP (slave problem) has to be also fast enough to be solved. If SP is a NP-Hard problem, perhaps the column-generation approach might not work.



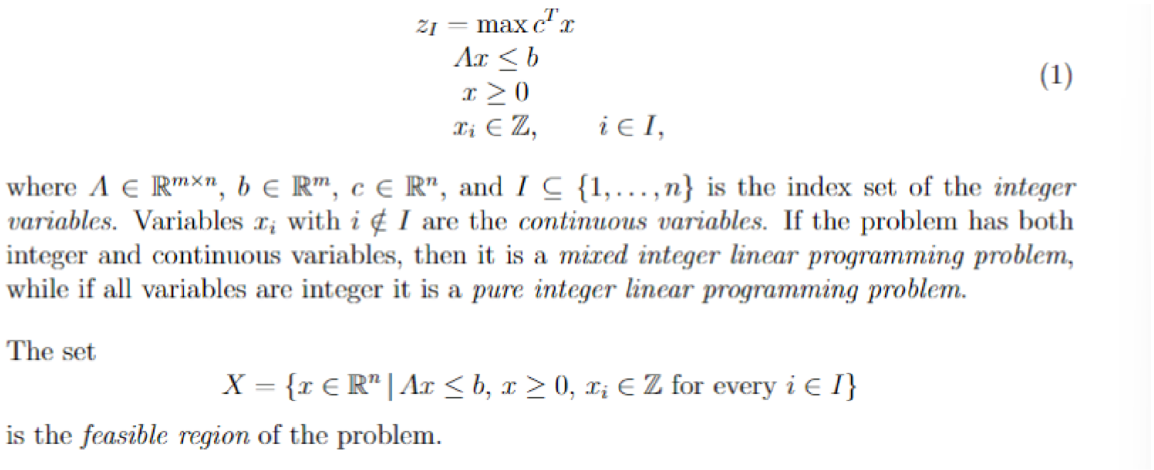
- device a suitable column generation subproblem: it should be sufficiently *efficient* (the efficiency "mainly" depends on the slave problem)

Since we need only some variables (a subset), we would need to handle multiple variables and remove inactive variables, generating poorer quality dual values (head-in) or having small improvements over the o.f. value.

8 SOLUTION METHODS FOR ILP – BRANCH AND BOUND AND ALTERNATIVE FORMULATIONS (7)

8.1 BRANCH AND BOUND – DEFINITION OF THE PROBLEM

A generic ILP problem is presented this way.



$$\begin{aligned} z_I = \max c^T x & \\ Ax \le b & \\ x \ge 0 & \\ x_i \in \mathbb{Z}, & \qquad i \in I, \end{aligned} \tag{1}$$

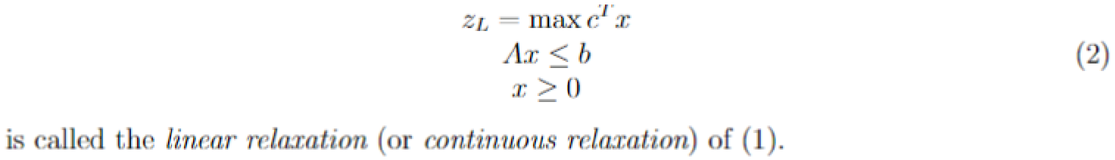
where $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, and $I \subseteq \{1,\dots,n\}$ is the index set of the *integer variables*. Variables x_i with $i \notin I$ are the *continuous variables*. If the problem has both integer and continuous variables, then it is a *mixed integer linear programming problem*, while if all variables are integer it is a *pure integer linear programming problem*.

The set

$$X = \{x \in \mathbb{R}^n \mid Ax \le b,\ x \ge 0,\ x_i \in \mathbb{Z} \text{ for every } i \in I\}$$

is the *feasible region* of the problem.

We analyze in the following image the linear relaxation of the problem:

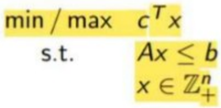


$$\begin{aligned} z_L = \max c^T x & \\ Ax \le b & \\ x \ge 0 & \end{aligned} \tag{2}$$

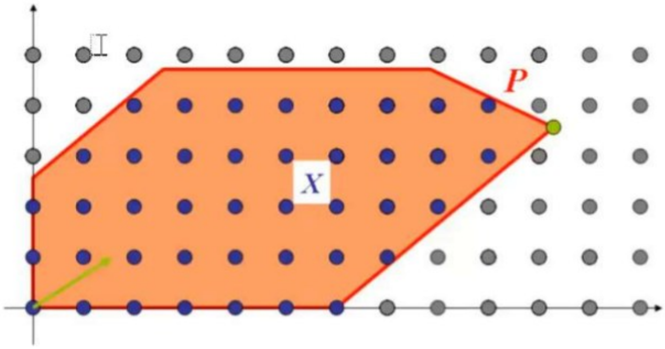
is called the *linear relaxation* (or *continuous relaxation*) of (1).

Note easily that: $z_i \le z_L$. Infact, if x^I is the optimal solution of (1) and x^L is the optimal solution of (2), then x^I satisfies (2) constraints, while $z_I = c^T x^I \le c^T x^L = z_L$. In the image:

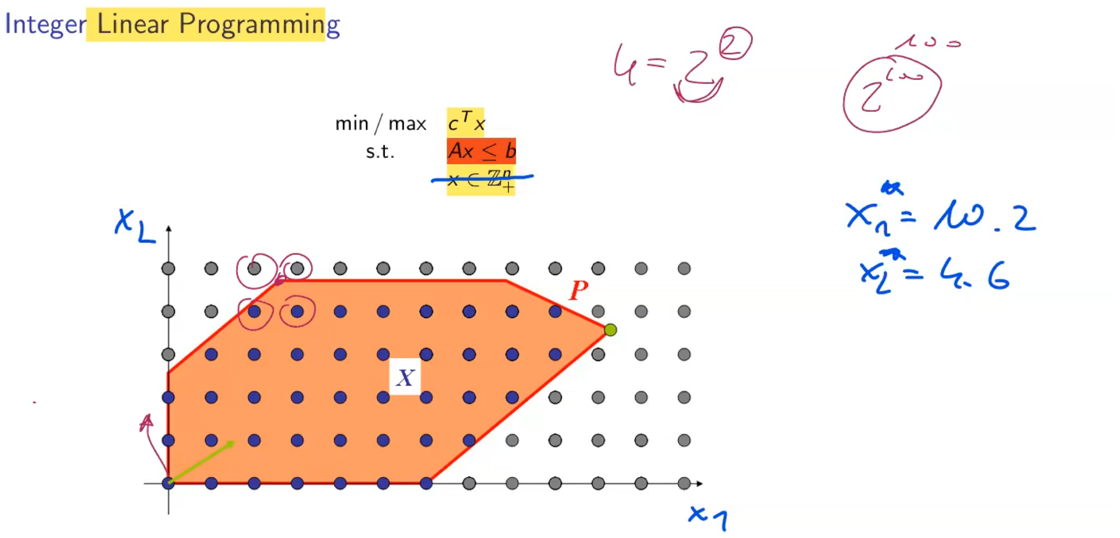
- The orange shaded region represents the feasible region defined by the linear constraints
- The blue dots represent the integer feasible solutions
- The grey dots are integer points that lie outside the feasible region
- The axes form a grid, representing the integer coordinates



$$\begin{aligned} &\text{min / max} && c^T x \\ &\text{s.t.} && Ax \le b \\ &&& x \in \mathbb{Z}^n_+ \end{aligned}$$



Solving linear relaxation can be useful but not so easy; rounding needs other procedures in order to work properly.



Integer Linear Programming

$$\begin{array}{ll} \min / \max & c^T x \\ \text{s.t.} & Ax \le b \\ & \cancel{x \in \mathbb{Z}^n_+} \end{array}$$

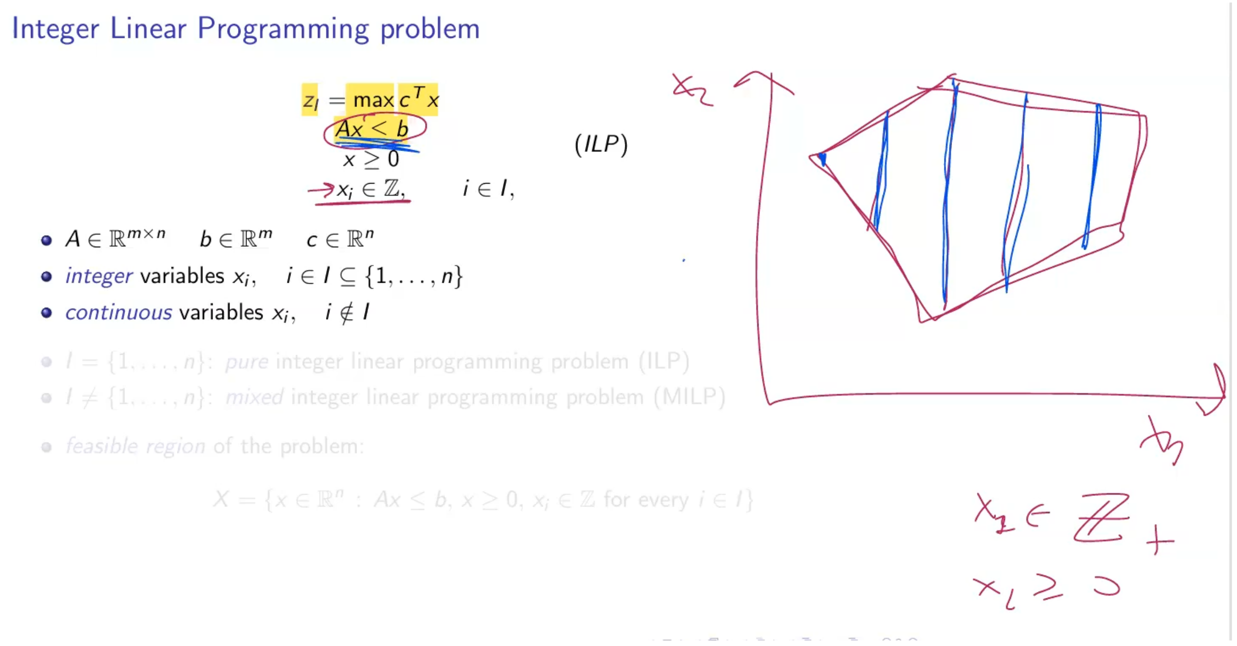
$4 = 2^2$

2^{100}

$x_1^* = 10.2$

$x_2^* = 4.6$

Solving the linear relaxation might be interesting, but not so easy. Let's go better into the details of the actual problem, with some visualizations coming from the actual lesson:



Integer Linear Programming problem

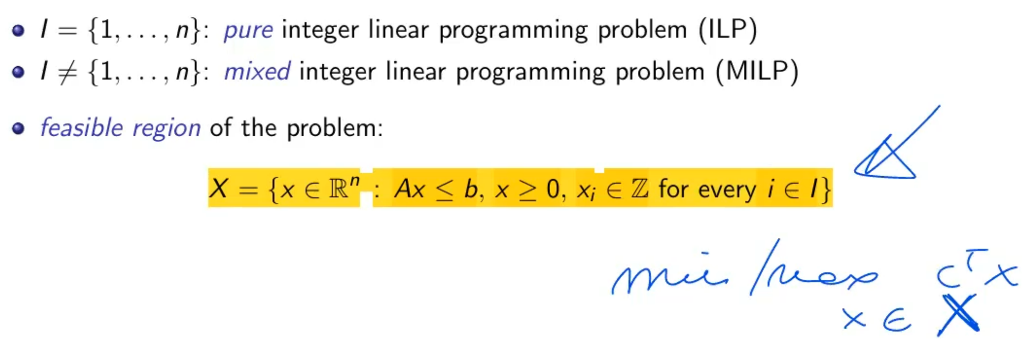
$$\begin{array}{ll} z_I = \max c^T x & \\ Ax \le b & \\ x \ge 0 & \\ x_i \in \mathbb{Z}, & i \in I, \end{array} \qquad (ILP)$$

- $A \in \mathbb{R}^{m \times n}$ $\quad b \in \mathbb{R}^m$ $\quad c \in \mathbb{R}^n$
- *integer* variables x_i, $\quad i \in I \subseteq \{1, \dots, n\}$
- *continuous* variables x_i, $\quad i \notin I$
- $I = \{1, \dots, n\}$: *pure* integer linear programming problem (ILP)
- $I \neq \{1, \dots, n\}$: *mixed* integer linear programming problem (MILP)
- *feasible region* of the problem:

$$X = \{x \in \mathbb{R}^n : Ax \le b, x \ge 0, x_i \in \mathbb{Z} \text{ for every } i \in I\}$$

$x_1 \in \mathbb{Z}_+$

$x_2 \ge 0$



- $I = \{1, \dots, n\}$: *pure* integer linear programming problem (ILP)
- $I \neq \{1, \dots, n\}$: *mixed* integer linear programming problem (MILP)
- *feasible region* of the problem:

$$X = \{x \in \mathbb{R}^n : Ax \le b, x \ge 0, x_i \in \mathbb{Z} \text{ for every } i \in I\}$$

$$\min / \max_{x \in X} c^T x$$

Pure ILP are variables constrained to be integer, into the mixed variant we have also continuous variable. The feasible region is made up by sets of real points satisfying the inequality (polyhedron) having points for some variables.

One positive feature of set X is that it is discrete, so it allows usage of two fundamental concepts used in solving Integer Linear Programming problems:

1. Divide et Impera (Divide and Conquer):

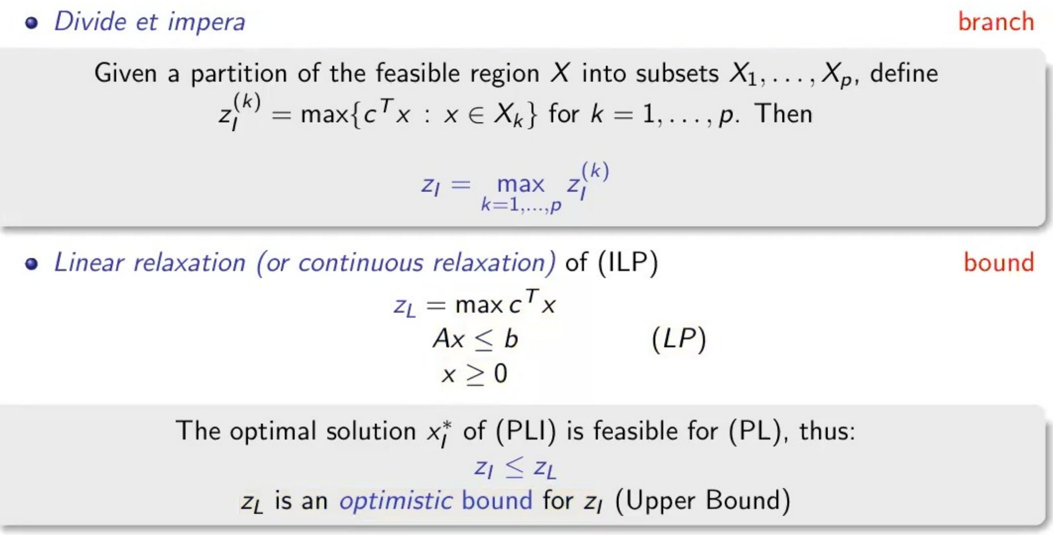
- The feasible region X is divided into p smaller subsets (X_1, X_2, ..., X_p)
- For each subset X_k, you solve a smaller problem to find $z_i^{(k)}$ = max{c^T x : x ∈ X_k}
- The optimal solution z_i is then the maximum among all these subsolutions
- This is useful because smaller problems are often easier to solve than the full problem

2. Linear Relaxation:

- You remove the integer constraint (x ∈ Z^+) and allow continuous values (x ≥ 0)
- This creates an easier-to-solve linear program (LP)
- The solution z_l to this relaxed problem gives an upper bound on z_i ($z_i \leq z_l$)
- This is very practical because:
  - LP problems are much easier to solve than ILP
  - The bound helps in branch-and-bound algorithms
  - If you're lucky, the LP solution might be integer anyway

These techniques are typically used together in branch-and-bound algorithms:

1. Start with the linear relaxation to get an upper bound
2. Use divide-and-conquer to partition the problem when the relaxation gives non-integer solutions
3. Continue this process recursively, using the bounds to prune branches that can't contain the optimal solution



- *Divide et impera* branch

Given a partition of the feasible region X into subsets $X_1, \dots, X_p$, define $z_I^{(k)} = \max\{c^T x \,:\, x \in X_k\}$ for $k = 1, \dots, p$. Then

$$z_I = \max_{k=1,\dots,p} z_I^{(k)}$$

- *Linear relaxation (or continuous relaxation)* of (ILP) bound

$$\begin{aligned} z_L &= \max c^T x \\ Ax &\leq b \qquad\qquad (LP) \\ x &\geq 0 \end{aligned}$$

The optimal solution x_I^* of (PLI) is feasible for (PL), thus:

$$z_I \leq z_L$$

z_L is an *optimistic* bound for z_I (Upper Bound)

The method exploits the following observation:

Given a partition of the feasible region X into subsets $X_1, \dots, X_p$, define $z_I^{(k)} = \max\{c^T x \mid x \in X_k\}$ for $k = 1, \dots, p$. Then

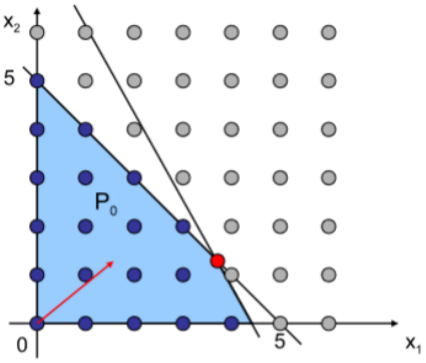
$$z_I = \max_{k=1,\dots,p} z_I^{(k)}.$$

The Branch-and-Bound method proceeds by partitioning X into smaller subsets and solving the problem $\max c^T x$ on every subset.

- This is done recursively, by further dividing the feasible regions of the subproblems in subsets. If this recursion was to be carried out completely, in the end we would enumerate all integer solutions of the problem
- In this case, at least two issues would arise: first, if the problem has infinitely many feasible solutions, so the complete enumeration is not possible; second, even assuming that the feasible region contains a finite number of points, this number might be extremely large and thus the enumeration would require an unpractical amount of time
- The Branch-and-Bound algorithm aims at exploring only the "promising" areas of the feasible region, by storing upper and lower bounds for the optimal value within a certain area and using these bounds to decide that certain subproblems do not need to be solved

8.1.1 Complete Branch and Bound example

Consider the problem P_0, where its feasible region (blue points) and the feasible region of its linear relaxation (light blue quadrilateral) are represented here (arrow is optimization direction).



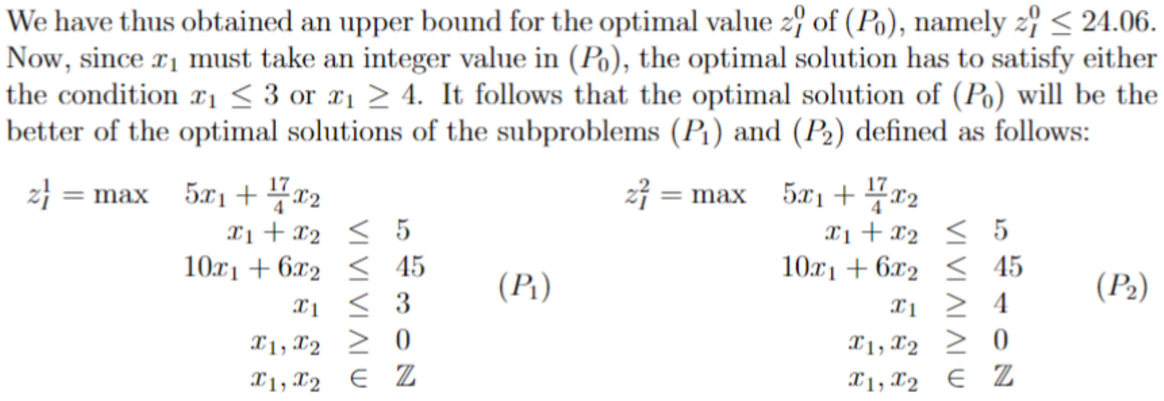
$$\begin{array}{rrcl} z_I^0 = \max & 5x_1 + \frac{17}{4}x_2 & & \\ & x_1 + x_2 & \leq & 5 \\ & 10x_1 + 6x_2 & \leq & 45 \\ & x_1, x_2 & \geq & 0 \\ & x_1, x_2 & \in & \mathbb{Z} \end{array} \qquad (P_0)$$

After solving the linear relaxation of (P_0), we obtain the optimal solution $x_1 = 3.75$, $x_2 = 1.25$ (red point), whose objective value is $z_L^0 = 24.06$.

We are using a divide-et-impera approach: choose one of the fractional variables then divide the problem into two subproblems. Based on the non-integer solution (3.75, 1.75), we can branch:

- For x_1: either $x_1 \leq 3$ or $x_1 \geq 4$
- For x_2: either $x_2 \leq 1$ or $x_2 \geq 2$

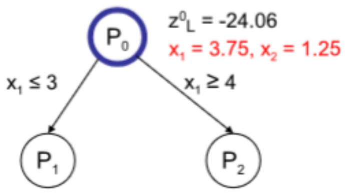
This example shows how linear relaxation gives a fractional solution, necessitating branching to find the optimal integer solution. The bounds help narrow down where the optimal integer solution must lie.

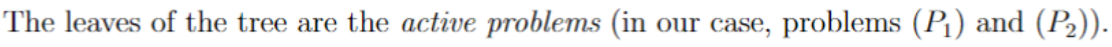


We have thus obtained an upper bound for the optimal value z_I^0 of (P_0), namely $z_I^0 \leq 24.06$. Now, since x_1 must take an integer value in (P_0), the optimal solution has to satisfy either the condition $x_1 \leq 3$ or $x_1 \geq 4$. It follows that the optimal solution of (P_0) will be the better of the optimal solutions of the subproblems (P_1) and (P_2) defined as follows:

$$
\begin{array}{llrcl}
z_I^1 = \max & 5x_1 + \frac{17}{4}x_2 & & & \\
 & & x_1 + x_2 & \leq & 5 \\
 & & 10x_1 + 6x_2 & \leq & 45 \\
 & & x_1 & \leq & 3 \\
 & & x_1, x_2 & \geq & 0 \\
 & & x_1, x_2 & \in & \mathbb{Z}
\end{array}
\quad (P_1)
\qquad
\begin{array}{llrcl}
z_I^2 = \max & 5x_1 + \frac{17}{4}x_2 & & & \\
 & & x_1 + x_2 & \leq & 5 \\
 & & 10x_1 + 6x_2 & \leq & 45 \\
 & & x_1 & \geq & 4 \\
 & & x_1, x_2 & \geq & 0 \\
 & & x_1, x_2 & \in & \mathbb{Z}
\end{array}
\quad (P_2)
$$

The operation used here is *branching* so to take a solution; we did *branching on variable* x_1. We can represent the subproblems and the corresponding bounds by means of a tree, called the Branch-and-Bound tree.



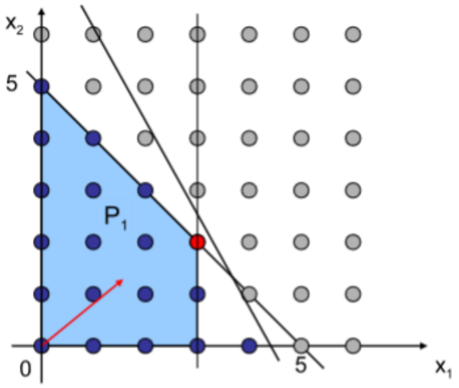


The leaves of the tree are the *active problems* (in our case, problems (P_1) and (P_2)).

If we take the union of the points represented, we take back the original divided blue points. The optimal integer solution of the problem is considered between the other optimal solutions. In particular:

- $x^0{}_1 \notin LP(P_1) \cup LP(P_2)$
  - The fractional solution (3.75, 1.25) is not feasible for either subproblem
- $X(P_1) \cup X(P_2) = X(P_0)$
  - The union of feasible regions of subproblems equals original problem's feasible region
  - Therefore, $z^0{}_i = \max\{z^1{}_i, z^2{}_i\}$

Consider problem (P_1), which is represented below.

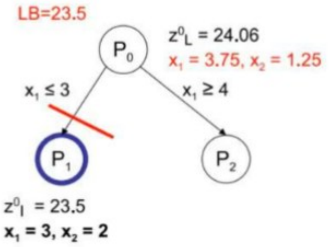


The optimal solution of the linear relaxation of (P_1) is $x_1 = 3$, $x_2 = 2$, with objective value $z_L^1 = 23.5$. Since this solution is integer, $(3, 2)$ is also the optimal integer solution of (P_1). For this reason, there is no need to branch on node (P_1), which can be pruned. We say that (P_1) is *pruned by optimality*. Also note that the optimal solution of (P_0) will necessarily have objective value $z_I^0 \geq z_I^1 = 23.5$. Therefore $LB = 23.5$ is a lower bound for the optimal value, and $(3, 2)$ is called the *incumbent solution*, i.e., the best integer solution found so far.

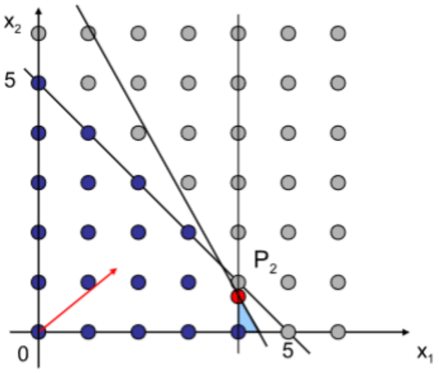
So, to summarize:

$LP(P_1)$: $x_1 = 3,\ x_2 = 2 \quad z_L^1 = 23.5$

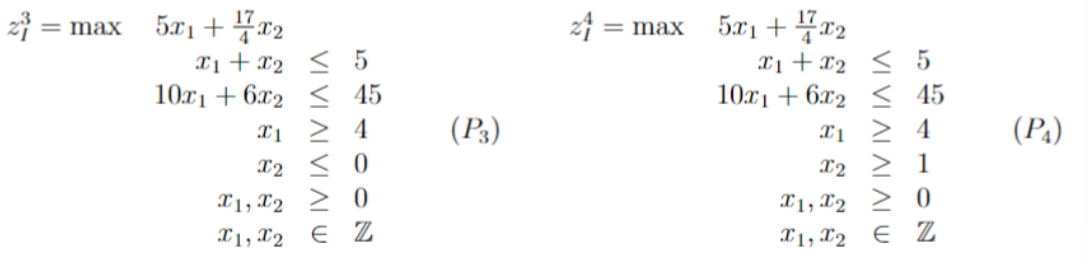
- $z_I^1 = z_L^1$: (P_1) is *pruned by optimality*
- $z_I^0 \geq z_I^1 = 23.5$: *lower bound* for (P_0)
- $(3\ , 2)$ is the *incumbent solution*



The only non-pruned leaf is (P_2), which therefore is the only active problem, and is represented below:



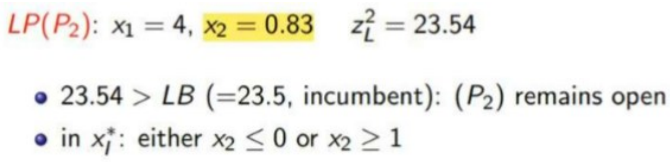
The optimal solution of the linear relaxation of (P_2) is $x_1 = 4$, $x_2 = 0.83$, with objective value $z_L^2 = 23.54$. Then $z_I^2 \leq 23.54$ and therefore 23.54 is an upper-bound for the optimal value of (P_2). Note that $LB = 23.5 < 23.54$, thus (P_2) might have a better solution than the incumbent solution. Since the value of x_2 is 0.83, which is not an integer, we branch on x_2, obtaining the subproblems (P_3) and (P_4) shown below.



$$z_I^3 = \max \quad 5x_1 + \tfrac{17}{4}x_2 \qquad \begin{array}{rcl} x_1 + x_2 & \le & 5 \\ 10x_1 + 6x_2 & \le & 45 \\ x_1 & \ge & 4 \\ x_2 & \le & 0 \\ x_1, x_2 & \ge & 0 \\ x_1, x_2 & \in & \mathbb{Z} \end{array} \qquad (P_3)$$

$$z_I^4 = \max \quad 5x_1 + \tfrac{17}{4}x_2 \qquad \begin{array}{rcl} x_1 + x_2 & \le & 5 \\ 10x_1 + 6x_2 & \le & 45 \\ x_1 & \ge & 4 \\ x_2 & \ge & 1 \\ x_1, x_2 & \ge & 0 \\ x_1, x_2 & \in & \mathbb{Z} \end{array} \qquad (P_4)$$

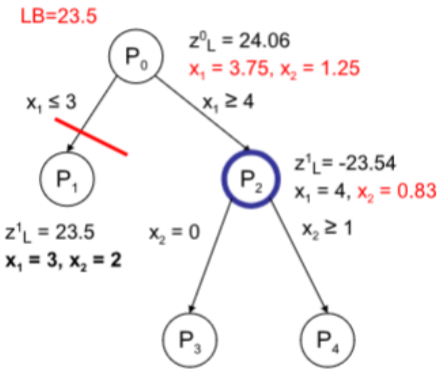
So, to summarize:



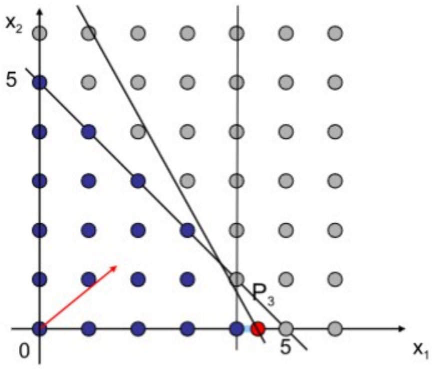
$LP(P_2)$: $x_1 = 4$, $x_2 = 0.83$ $\quad z_L^2 = 23.54$

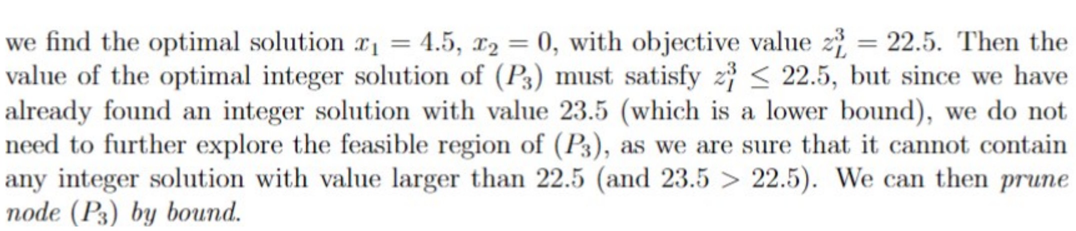
- $23.54 > LB$ (=23.5, incumbent): (P_2) remains open
- in x_I^*: either $x_2 \le 0$ or $x_2 \ge 1$

Now we have the B&B-tree as the following, with active nodes as P_3 and P_4.



Active nodes are P_3 and P_4. Now once again we solve the linear relaxation of P_3.



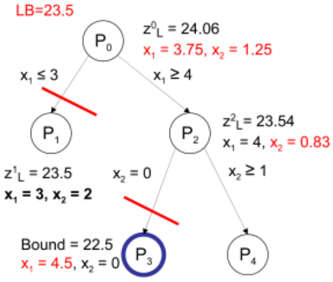


we find the optimal solution $x_1 = 4.5$, $x_2 = 0$, with objective value $z_L^3 = 22.5$. Then the value of the optimal integer solution of (P_3) must satisfy $z_I^3 \le 22.5$, but since we have already found an integer solution with value 23.5 (which is a lower bound), we do not need to further explore the feasible region of (P_3), as we are sure that it cannot contain any integer solution with value larger than 22.5 (and $23.5 > 22.5$). We can then *prune node* (P_3) *by bound*.

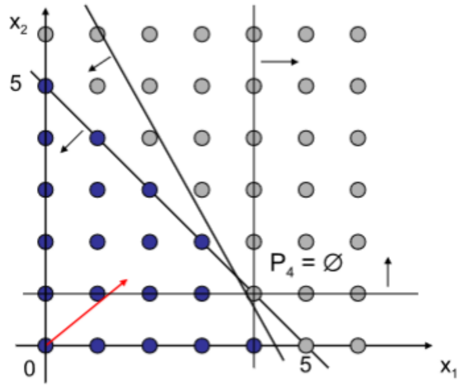
We get summarizing the following B&B tree, showing the single active problem below:

$LP(P_3)$: $x_1 = 4.5$, $x_2 = 0$ $\quad z_L^3 = 22.5$

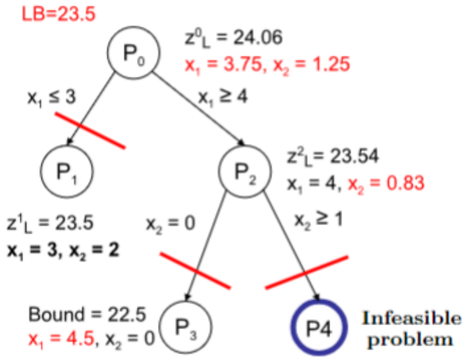
- $z_I^3 \leq 22.5 < LB(= 23.5)$
- (P3) is *pruned by bound*



Now we solve the linear relaxation of P_4, determining there is no feasible solution in the linear relaxation, therefore having P_4 having no integer solution.



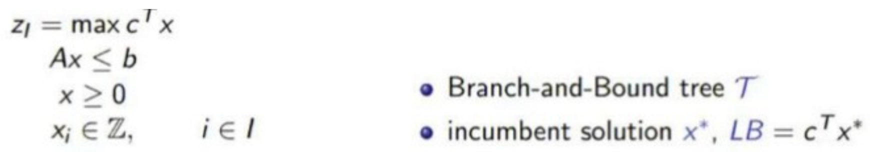
Node (P_4) can then be *pruned by infeasibility*. The Branch-and-Bound tree, shown below, does not have any active node and therefore the incumbent solution is the best integer solution of the problem; in other words, $(3, 2)$ is an optimal solution of (P_0).



8.1.2 Formal Description and Model

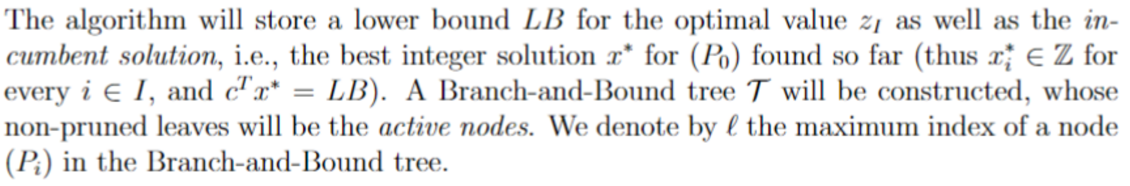
We give now a formal description. Consider the original problem and we want to find the current good solution (incumbent solution) so to construct a proper Branch-and-Bound tree, removing all of the non-active nodes.

Starting from the problem to be solved (P_0):



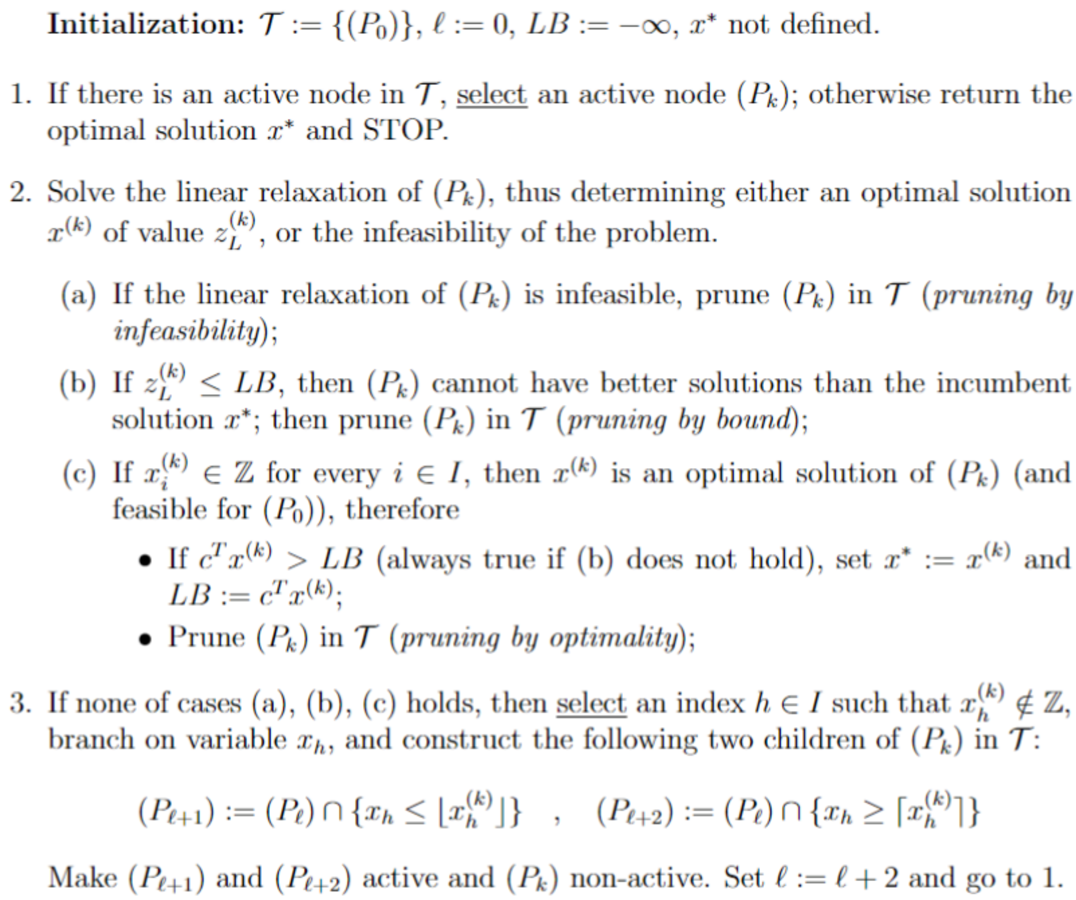
$$
\begin{aligned}
z_I = \max\ & c^T x \\
& Ax \leq b \\
& x \geq 0 \\
& x_i \in \mathbb{Z}, \quad i \in I
\end{aligned}
$$

- Branch-and-Bound tree $\mathcal{T}$
- incumbent solution x^*, $LB = c^T x^*$



The algorithm will store a lower bound LB for the optimal value z_I as well as the *incumbent solution*, i.e., the best integer solution x^* for (P_0) found so far (thus $x_i^* \in \mathbb{Z}$ for every $i \in I$, and $c^T x^* = LB$). A Branch-and-Bound tree $\mathcal{T}$ will be constructed, whose non-pruned leaves will be the *active nodes*. We denote by ℓ the maximum index of a node (P_i) in the Branch-and-Bound tree.

Now, consider the entire formulation of the Branch-and-Bound method:



Initialization: $\mathcal{T} := \{(P_0)\}$, $\ell := 0$, $LB := -\infty$, x^* not defined.

1. If there is an active node in $\mathcal{T}$, select an active node (P_k); otherwise return the optimal solution x^* and STOP.
2. Solve the linear relaxation of (P_k), thus determining either an optimal solution $x^{(k)}$ of value $z_L^{(k)}$, or the infeasibility of the problem.
   (a) If the linear relaxation of (P_k) is infeasible, prune (P_k) in $\mathcal{T}$ (*pruning by infeasibility*);
   (b) If $z_L^{(k)} \leq LB$, then (P_k) cannot have better solutions than the incumbent solution x^*; then prune (P_k) in $\mathcal{T}$ (*pruning by bound*);
   (c) If $x_i^{(k)} \in \mathbb{Z}$ for every $i \in I$, then $x^{(k)}$ is an optimal solution of (P_k) (and feasible for (P_0)), therefore
   - If $c^T x^{(k)} > LB$ (always true if (b) does not hold), set $x^* := x^{(k)}$ and $LB := c^T x^{(k)}$;
   - Prune (P_k) in $\mathcal{T}$ (*pruning by optimality*);
3. If none of cases (a), (b), (c) holds, then select an index $h \in I$ such that $x_h^{(k)} \notin \mathbb{Z}$, branch on variable x_h, and construct the following two children of (P_k) in $\mathcal{T}$:

$$(P_{\ell+1}) := (P_\ell) \cap \{x_h \leq \lfloor x_h^{(k)} \rfloor\} \quad , \quad (P_{\ell+2}) := (P_\ell) \cap \{x_h \geq \lceil x_h^{(k)} \rceil\}$$

Make $(P_{\ell+1})$ and $(P_{\ell+2})$ active and (P_k) non-active. Set $\ell := \ell + 2$ and go to 1.

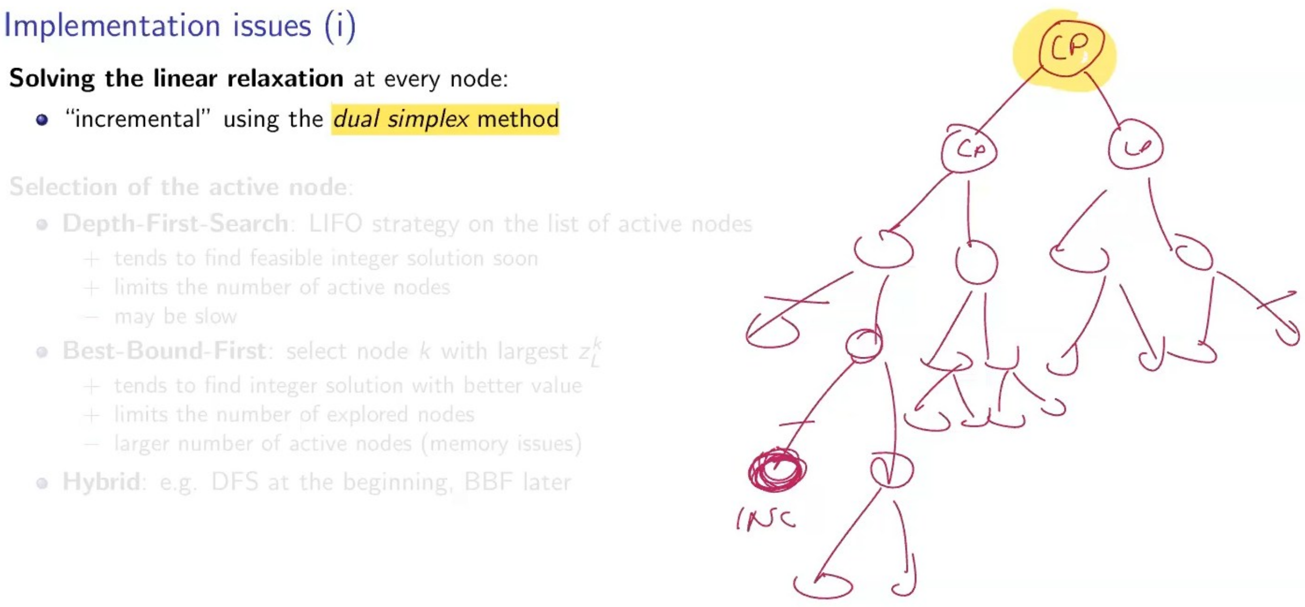
Where floor or ceiling notations are used, it means values are rounded-down or rounded-up.

8.1.3 Implementation Issues

There are many fundamental details to take care of in order to make a Branch-and-Bound method efficient. Here we examine the following implementation issues.

- *Solution of the linear relaxation of every node*
    - The linear relaxation of any node corresponds to the linear relaxation of the parent node plus a single constraint
    - If the relaxation of the parent node has been solved with the simplex method, we know an optimal basic solution of the relaxation of the parent node
    - By using a variant of the simplex method called "dual simplex method", one can efficiently obtain an optimal solution for the same problem with a new constraint added ("incremental") – this was present inside of older Italian notes here (online also)
    - This feature allows for a fast exploration of the nodes of the Branch-and-Bound tree in (M)ILP problems

We have a method able to exploit the linear relaxation of parent nodes but also for children nodes, which is the dual simplex method and given the simple nature of Branch and Bound using it is not really an issue.



Implementation issues (i)

Solving the linear relaxation at every node:

- "incremental" using the *dual simplex* method

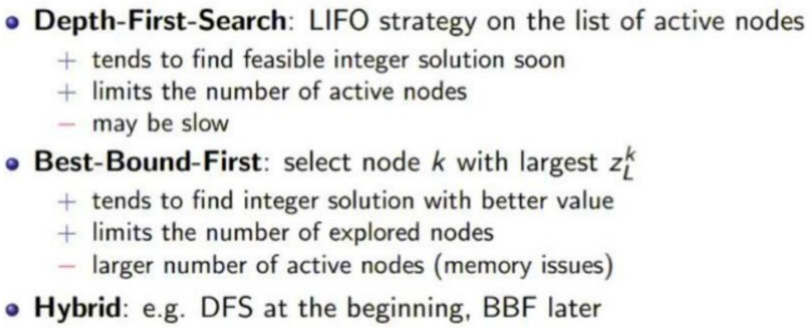
Selection of the active node:

- Depth-First-Search: LIFO strategy on the list of active nodes
    - + tends to find feasible integer solution soon
    - + limits the number of active nodes
    - − may be slow
- Best-Bound-First: select node k with largest z_L^k
    - + tends to find integer solution with better value
    - + limits the number of explored nodes
    - − larger number of active nodes (memory issues)
- Hybrid: e.g. DFS at the beginning, BBF later

- *Selection of an active node*
    - Step 1 of the algorithm requires to select a node from the list of active nodes
    - The number of nodes that will be opened overall depends on how this list is handled; in particular, this depends on the criteria used to select an active node
    - In fact, there are two conflicting targets to keep in mind when choosing an active node:
        - Finding a (good) feasible integer solution as soon as possible
            - This brings at least two advantages: an integer solution provides a lower bound for the optimal value of the problem, and having a good lower bound increases the chances of pruning some nodes by bound

            - Furthermore, in the event that one needs to stop the algorithm before its natural termination, we have at least found a (good) feasible solution for the problem, though maybe not the optimal one
        - Exploring a small number of nodes

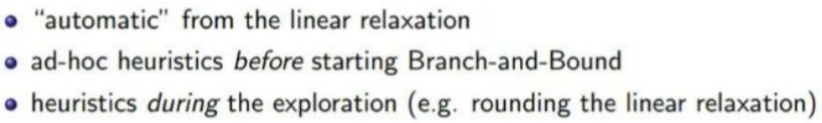
The selection of the active node revolves around these strategies:



- **Depth-First-Search**: LIFO strategy on the list of active nodes
  - + tends to find feasible integer solution soon
  - + limits the number of active nodes
  - − may be slow
- **Best-Bound-First**: select node k with largest z_L^k
  - + tends to find integer solution with better value
  - + limits the number of explored nodes
  - − larger number of active nodes (memory issues)
- **Hybrid**: e.g. DFS at the beginning, BBF later

- *Evaluation of feasible solutions*
    - In order to prune nodes by bound, good quality feasible solutions are needed
    - For this reason, when designing a Branch-and-Bound algorithm we have to decide how and when feasible solutions should be computed
    - There are several options, among which we mention the following:
        - Waiting for the enumeration to generate a leaf node whose linear relaxation has an integer optimal solution
        - Implementing a heuristic algorithm that finds a good integer solution before starting the exploration
        - Exploiting (several times during the algorithm, with frequency depending on the specific problem) the information obtained during the exploration of the tree to construct better and better feasible solutions
            - E.g., by rounding the solution of the linear relaxation in a suitable way, so that a feasible integer solution is obtained

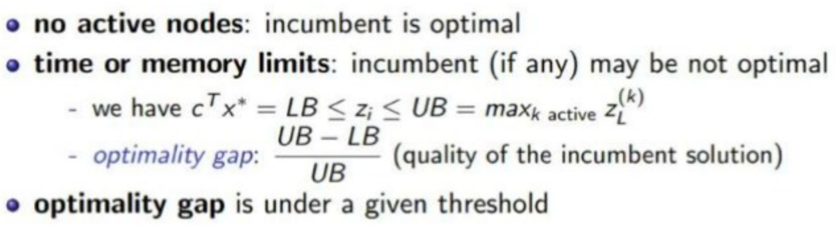
In any case, the trade-off between the quality of the incumbent solution and the computational effort needed to obtain it has to be considered.



- "automatic" from the linear relaxation
- ad-hoc heuristics *before* starting Branch-and-Bound
- heuristics *during* the exploration (e.g. rounding the linear relaxation)

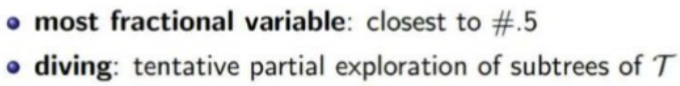
- *Stopping criteria*
    - The Branch-and-Bound method naturally stops when there are no active nodes left (all *closed/pruned*). In this case, the current incumbent solution is an optimal integer solution
    - However, one can stop the algorithm when a given time limit or memory limit has been reached, but in this case the incumbent solution (if any has been found) is not guaranteed to be optimal
    - Indeed, at any time during the construction of the Branch-and-Bound tree we know a lower bound LB (given by the value of the incumbent solution), but also an upper bound UB, given by the maximum of all values $z_L^{(k)}$ of the active nodes: this value is an optimistic estimation of the integer optimal value z_I (meaning that $z_I \leq UB$)

- If the algorithm is stopped before its natural termination, the difference between the value of the incumbent solution LB and the bound UB is an estimation of the quality of the incumbent solution available
- For this reason, a possible stopping criterion might be to terminate the algorithm when the difference between these two bounds is smaller than a given value (fixed in advance), when we keep this difference "sufficient" given the quality of the solution



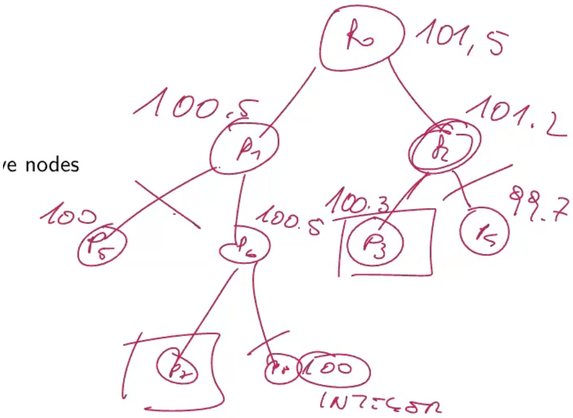
- **no active nodes**: incumbent is optimal
- **time or memory limits**: incumbent (if any) may be not optimal
  - we have $c^T x^* = LB \leq z_i \leq UB = max_{k \text{ active}} z_L^{(k)}$
  - *optimality gap*: $\dfrac{UB - LB}{UB}$ (quality of the incumbent solution)
- **optimality gap** is under a given threshold

- *Selection/choice of the branching variable*
  - There are several applicable options for the choice of the branching variable, but a common one is to select the variable with the *most fractional value*, i.e., the variable whose fractional part is the closest to 0.5
  - In other words, we define $f_i = x_i^{(k)} - \lfloor x_i^{(k)} \rfloor$, we choose $h \in I\ s.t.\ h = argmin_{i \in I}\{\min\{f_i, 1 - f_i\}\}$

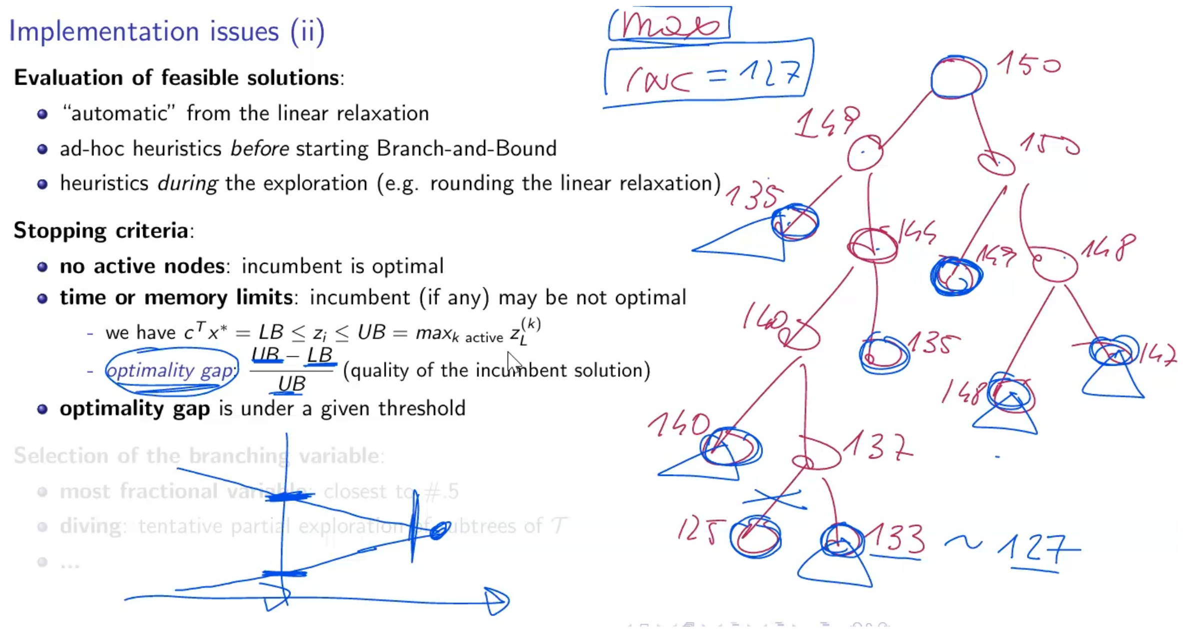


- **most fractional variable**: closest to #.5
- **diving**: tentative partial exploration of subtrees of $\mathcal{T}$

You can see at each iteration, more and more nodes are opening, which means there are more active nodes (there may be memory issues), but this means we will find more solutions and possibly stop before. The problem is already solved by CPLEX internally, but that's also the reason why it has to be licensed.



In this case, the Branch and Bound method eventually converges up to a given point. This means understanding which will be the most promising solution given the conditions of the problem.



Implementation issues (ii)

Evaluation of feasible solutions:

- "automatic" from the linear relaxation
- ad-hoc heuristics *before* starting Branch-and-Bound
- heuristics *during* the exploration (e.g. rounding the linear relaxation)

Stopping criteria:

- **no active nodes**: incumbent is optimal
- **time or memory limits**: incumbent (if any) may be not optimal
  - we have $c^T x^* = LB \leq z_I \leq UB = \max_{k \text{ active}} z_L^{(k)}$
  - *optimality gap*: $\frac{UB - LB}{UB}$ (quality of the incumbent solution)
- **optimality gap** is under a given threshold

Selection of the branching variable:

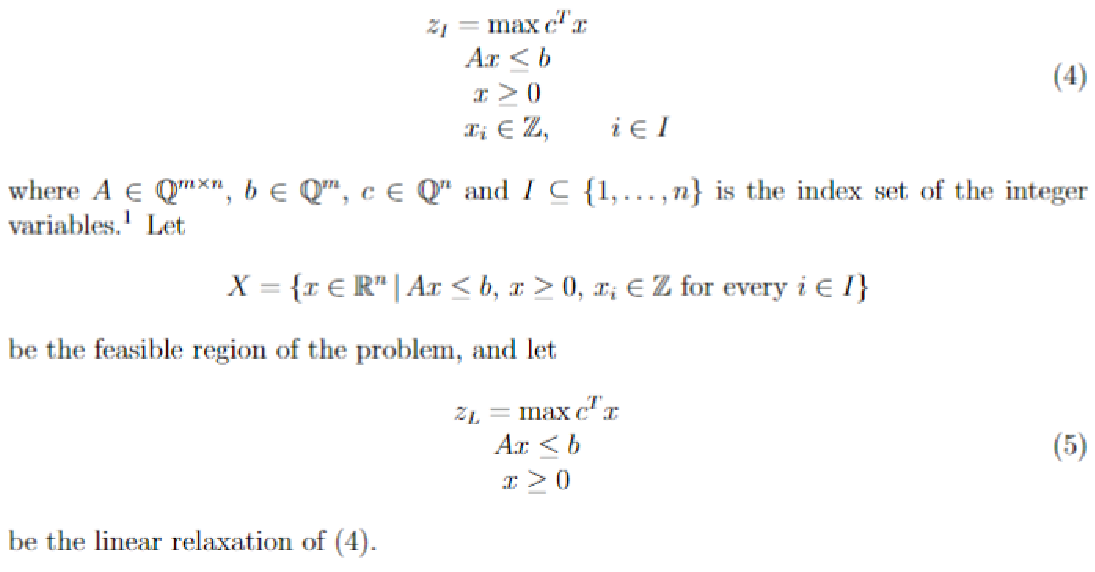
- most fractional variable: closest to #.5
- diving: tentative partial exploration of subtrees of $\mathcal{T}$
- ...

Important

There is a section from 11 to 15 dedicated for the general principles of B&B as a combinatorial optimization problem method which you can have a read to – but this is not part of the course unit (and of course not asked inside of the exam). This is section 2.1 inside of the file.

8.2 ALTERNATIVE FORMULATIONS – POLYHEDRAL APPROACH TO LP

We identify a set of linear inequalities s.t. within the polyhedron we have feasible solutions. As a benchmark we use the classical minimization problem in standard form, with a subset of variables x_i such that $x_i \in Z, \forall i \in I$. The problem to consider is the following:



$$\begin{aligned} z_I &= \max c^T x \\ Ax &\leq b \\ x &\geq 0 \\ x_i &\in \mathbb{Z}, \qquad i \in I \end{aligned} \tag{4}$$

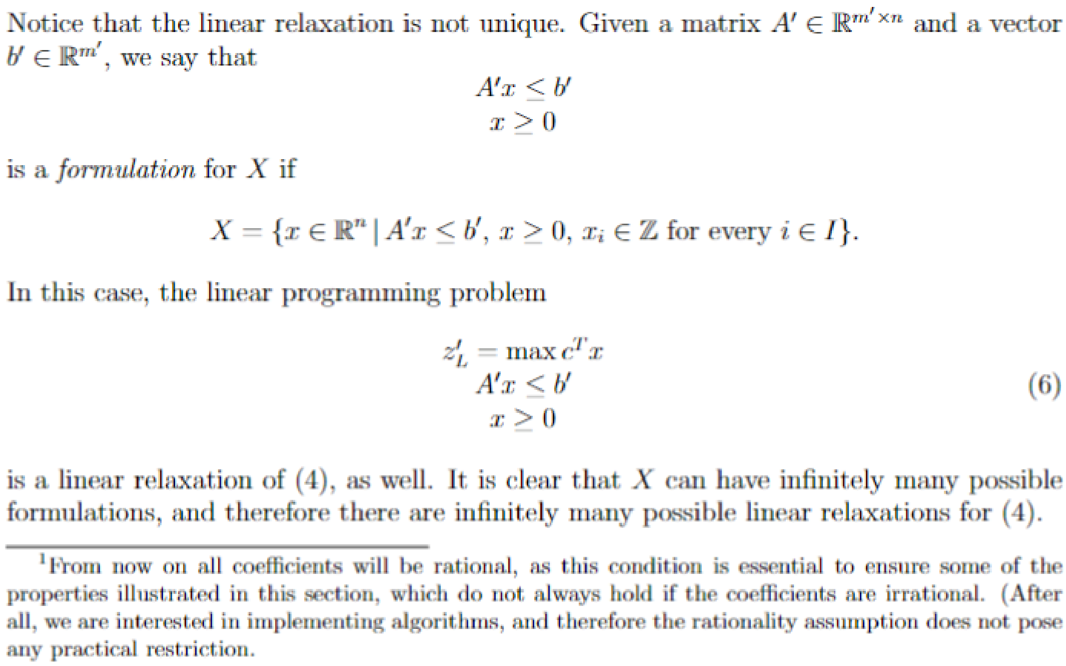
where $A \in \mathbb{Q}^{m \times n}$, $b \in \mathbb{Q}^m$, $c \in \mathbb{Q}^n$ and $I \subseteq \{1, \ldots, n\}$ is the index set of the integer variables.[1] Let

$$X = \{x \in \mathbb{R}^n \mid Ax \leq b,\ x \geq 0,\ x_i \in \mathbb{Z} \text{ for every } i \in I\}$$

be the feasible region of the problem, and let

$$\begin{aligned} z_L &= \max c^T x \\ Ax &\leq b \\ x &\geq 0 \end{aligned} \tag{5}$$

be the linear relaxation of (4).



Notice that the linear relaxation is not unique. Given a matrix $A' \in \mathbb{R}^{m' \times n}$ and a vector $b' \in \mathbb{R}^{m'}$, we say that

$$\begin{aligned} A'x &\leq b' \\ x &\geq 0 \end{aligned}$$

is a *formulation* for X if

$$X = \{x \in \mathbb{R}^n \mid A'x \leq b',\ x \geq 0,\ x_i \in \mathbb{Z} \text{ for every } i \in I\}.$$

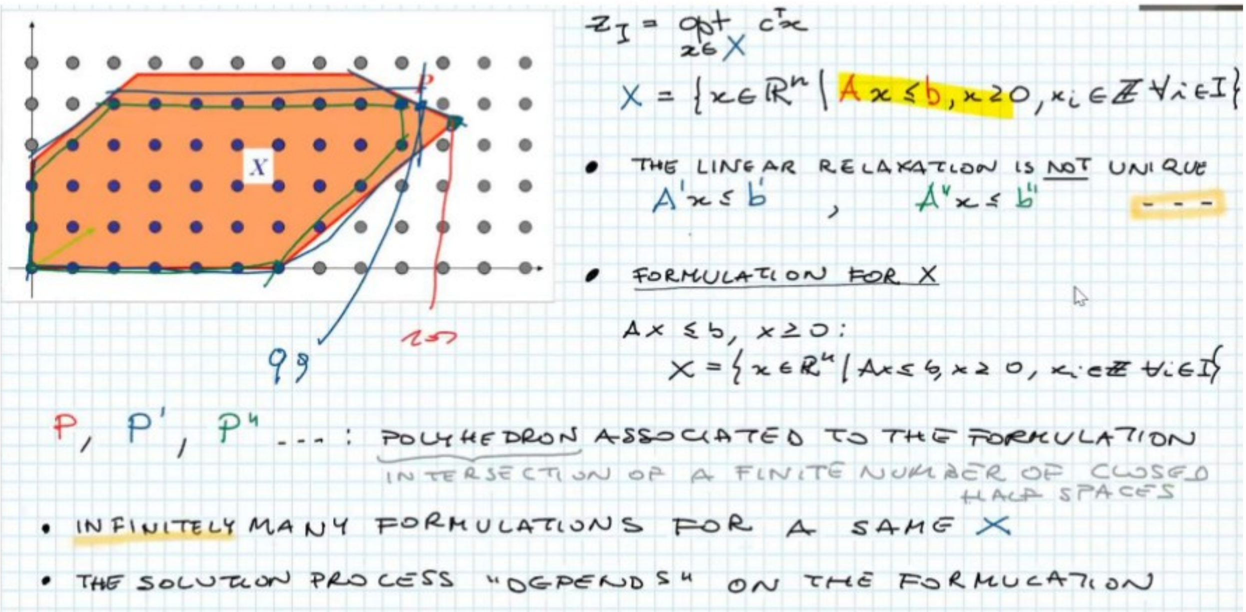
In this case, the linear programming problem

$$\begin{aligned} z'_L &= \max c^T x \\ A'x &\leq b' \\ x &\geq 0 \end{aligned} \tag{6}$$

is a linear relaxation of (4), as well. It is clear that X can have infinitely many possible formulations, and therefore there are infinitely many possible linear relaxations for (4).

[1] From now on all coefficients will be rational, as this condition is essential to ensure some of the properties illustrated in this section, which do not always hold if the coefficients are irrational. (After all, we are interested in implementing algorithms, and therefore the rationality assumption does not pose any practical restriction.

We may have infinite many other formulations given a specific problem. This is to be represented by the following representation:



$z_I = \underset{x \in X}{\text{opt}}\ c^T x$

$X = \{x \in \mathbb{R}^n \mid Ax \leq b, x \geq 0, x_i \in \mathbb{Z}\ \forall i \in I\}$

- THE LINEAR RELAXATION IS NOT UNIQUE
  $A'x \leq b'$, $A''x \leq b''$ - - -
- FORMULATION FOR X

  $Ax \leq b,\ x \geq 0$:
  $X = \{x \in \mathbb{R}^n \mid Ax \leq b, x \geq 0, x_i \in \mathbb{Z}\ \forall i \in I\}$

P, P', P'' - - - : $\underbrace{\text{POLYHEDRON}}$ ASSOCIATED TO THE FORMULATION
INTERSECTION OF A FINITE NUMBER OF CLOSED HALF SPACES

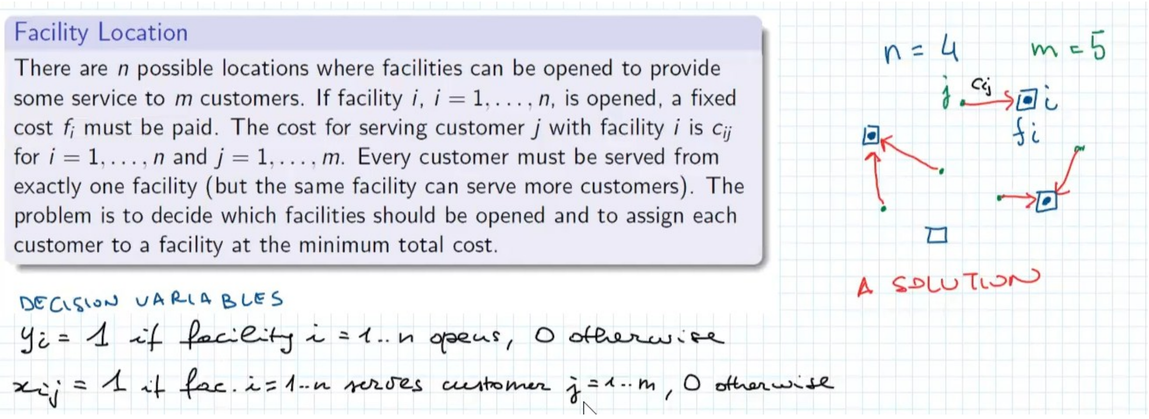
- INFINITELY MANY FORMULATIONS FOR A SAME X
- THE SOLUTION PROCESS "DEPENDS" ON THE FORMULATION

So here, every formulation is every polyhedron inside of the feasible region. In MILP every problem has a lot of formulations.

- Therefore it would be convenient to choose as the relaxation a polyhedron that is as close as possible to the optimal integer solution, because smaller polyhedra (formulations) give better bound
- There is to keep in mind that it is not always possible to determine whether one formulation is larger or smaller than the other, because it may happen that neither contains exactly the other

8.2.1 Example: Facility Location Problem and Better Formulations

Let's consider the *facility location problem*:



Facility Location

There are n possible locations where facilities can be opened to provide some service to m customers. If facility i, $i = 1, \dots, n$, is opened, a fixed cost f_i must be paid. The cost for serving customer j with facility i is c_{ij} for $i = 1, \dots, n$ and $j = 1, \dots, m$. Every customer must be served from exactly one facility (but the same facility can serve more customers). The problem is to decide which facilities should be opened and to assign each customer to a facility at the minimum total cost.

DECISION VARIABLES

$y_i = 1$ if facility $i = 1..n$ opens, 0 otherwise

$x_{ij} = 1$ if fac. $i = 1..n$ serves customer $j = 1..m$, 0 otherwise

The goal is to minimize the cost of reducing the total cost of opening of different facilities:

$$\sum_{i=1}^{n} f_i y_i + \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij}$$

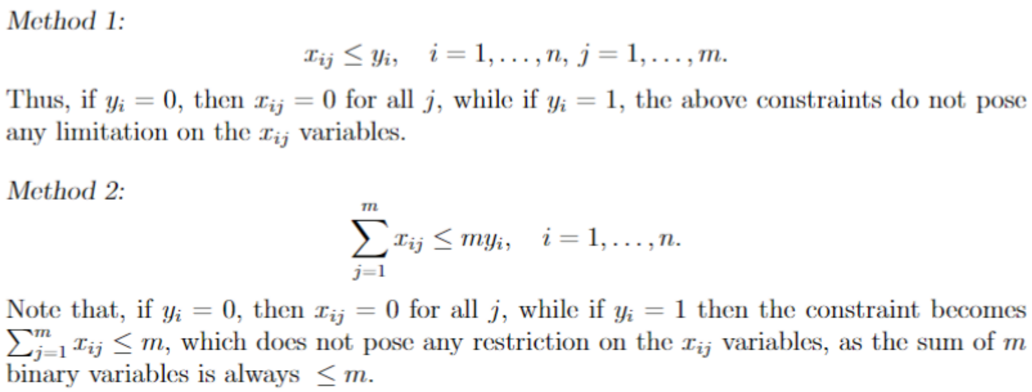
Each customers should be served by exactly one facility:

$$\sum_{i=1}^{n} x_{ij} = 1, \quad j = 1, \dots, m.$$

Customer *j* can be served by facility *i* if only if *i* will be opened:

$$y_i = 0 \Rightarrow x_{ij} = 0, \quad i = 1, \dots, n, \, j = 1, \dots, m.$$

This can be modeled by linear constraints in at least two ways:



Method 1:

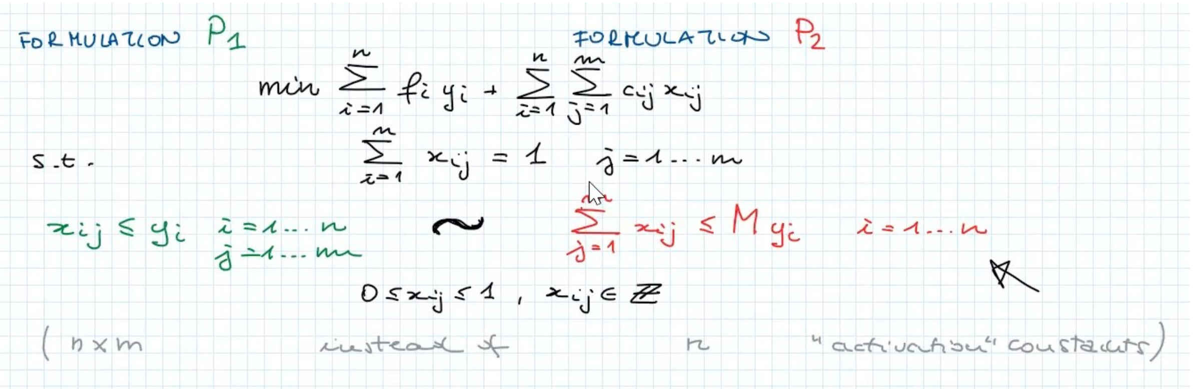
$$x_{ij} \le y_i, \quad i = 1, \dots, n, \, j = 1, \dots, m.$$

Thus, if $y_i = 0$, then $x_{ij} = 0$ for all j, while if $y_i = 1$, the above constraints do not pose any limitation on the x_{ij} variables.

Method 2:

$$\sum_{j=1}^{m} x_{ij} \le m y_i, \quad i = 1, \dots, n.$$

Note that, if $y_i = 0$, then $x_{ij} = 0$ for all j, while if $y_i = 1$ then the constraint becomes $\sum_{j=1}^{m} x_{ij} \le m$, which does not pose any restriction on the x_{ij} variables, as the sum of m binary variables is always $\le m$.



FORMULATION P_1 FORMULATION P_2

$$\min \sum_{i=1}^{n} f_i y_i + \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij}$$

s.t.

$$\sum_{i=1}^{m} x_{ij} = 1 \quad j = 1 \dots m$$

$x_{ij} \le y_i$ $\quad i = 1 \dots n$, $j = 1 \dots m$ $\quad \sim \quad$ $\sum_{j=1} x_{ij} \le M y_i$ $\quad i = 1 \dots n$

$$0 \le x_{ij} \le 1, \quad x_{ij} \in \mathbb{Z}$$

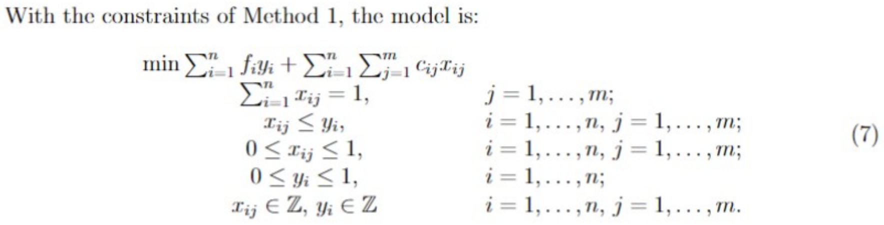
($n \times m$ instead of n "activation" constraints)

The two formulations are equivalent, but the first requires more constraints. As the value of M for the second formulation we can use the number of users m, because the maximum value of the summation is m and having a minimum M leads to good constraints.

But which of the two solutions is better to choose?

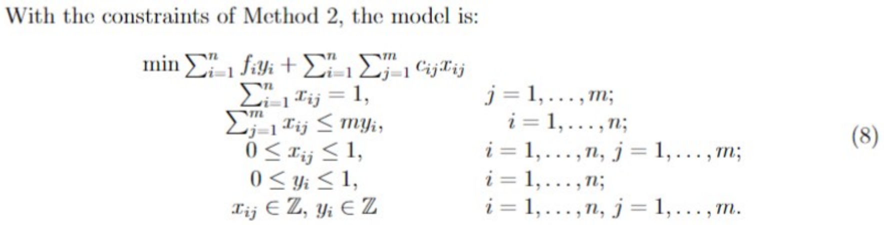
- In the end, we come always to the optimal solution anyway
- Typically the fewer constraints there are, the easier it is to solve the problem, but having so many constraints can lead to a smaller polyhedron
- It can be verified that the *first* formulation is better, in the sense that it defines a smaller polyhedron, because if (x, y) satisfies the first formulation, this also satisfies the second formulation

The constraints in Method 1 are called *non-aggregated* constraints (because they work on individual relationships), those in Method 2 are called *aggregated constraints* (combine multiple relationships into one constraint). Even if they enclose exactly the same points, the formulations differ in how they handle the relationship between variables:



With the constraints of Method 1, the model is:

$$\begin{array}{lll} \min \sum_{i=1}^n f_i y_i + \sum_{i=1}^n \sum_{j=1}^m c_{ij} x_{ij} & & \\ \sum_{i=1}^n x_{ij} = 1, & j = 1, \dots, m; & \\ x_{ij} \leq y_i, & i = 1, \dots, n,\ j = 1, \dots, m; & \\ 0 \leq x_{ij} \leq 1, & i = 1, \dots, n,\ j = 1, \dots, m; & (7) \\ 0 \leq y_i \leq 1, & i = 1, \dots, n; & \\ x_{ij} \in \mathbb{Z},\ y_i \in \mathbb{Z} & i = 1, \dots, n,\ j = 1, \dots, m. & \end{array}$$



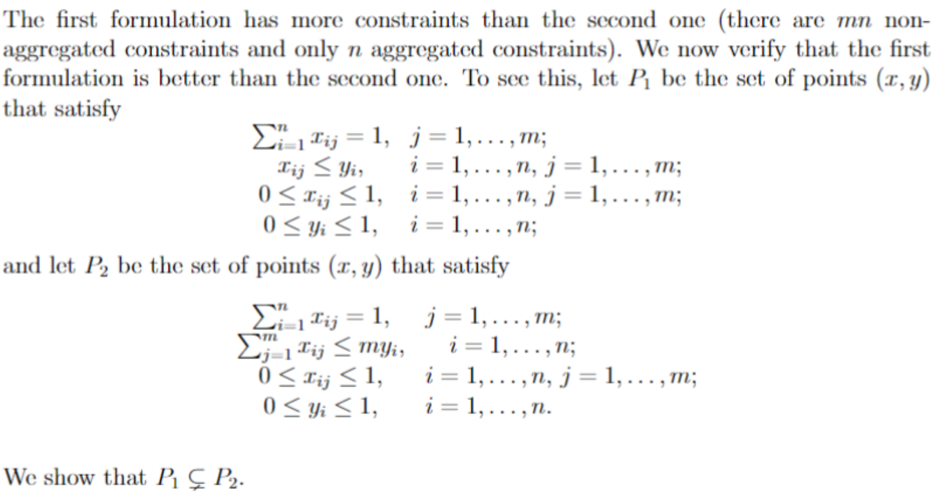
With the constraints of Method 2, the model is:

$$\begin{array}{lll} \min \sum_{i=1}^n f_i y_i + \sum_{i=1}^n \sum_{j=1}^m c_{ij} x_{ij} & & \\ \sum_{i=1}^n x_{ij} = 1, & j = 1, \dots, m; & \\ \sum_{j=1}^m x_{ij} \leq m y_i, & i = 1, \dots, n; & \\ 0 \leq x_{ij} \leq 1, & i = 1, \dots, n,\ j = 1, \dots, m; & (8) \\ 0 \leq y_i \leq 1, & i = 1, \dots, n; & \\ x_{ij} \in \mathbb{Z},\ y_i \in \mathbb{Z} & i = 1, \dots, n,\ j = 1, \dots, m. & \end{array}$$

Key differences:

1. P1 uses individual constraints $x_{ij} \leq y_i$ for each pair (i, j) to ensure a user can only be served by an open facility

2. P2 aggregates these into single constraints for each i, where M is a large enough constant (in this case $M = m$ works)



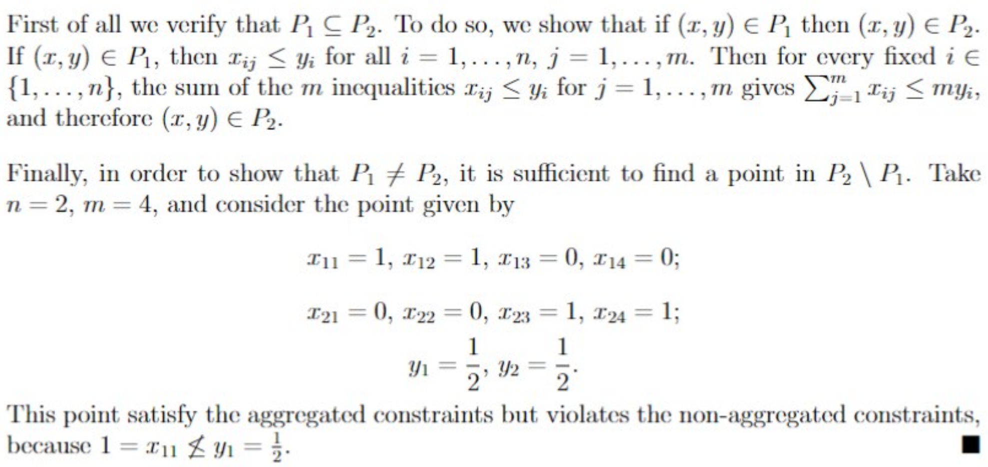
The first formulation has more constraints than the second one (there are mn non-aggregated constraints and only n aggregated constraints). We now verify that the first formulation is better than the second one. To see this, let P_1 be the set of points (x, y) that satisfy

$$\begin{array}{ll} \sum_{i=1}^n x_{ij} = 1, & j = 1, \dots, m; \\ x_{ij} \leq y_i, & i = 1, \dots, n,\ j = 1, \dots, m; \\ 0 \leq x_{ij} \leq 1, & i = 1, \dots, n,\ j = 1, \dots, m; \\ 0 \leq y_i \leq 1, & i = 1, \dots, n; \end{array}$$

and let P_2 be the set of points (x, y) that satisfy

$$\begin{array}{ll} \sum_{i=1}^n x_{ij} = 1, & j = 1, \dots, m; \\ \sum_{j=1}^m x_{ij} \leq m y_i, & i = 1, \dots, n; \\ 0 \leq x_{ij} \leq 1, & i = 1, \dots, n,\ j = 1, \dots, m; \\ 0 \leq y_i \leq 1, & i = 1, \dots, n. \end{array}$$

We show that $P_1 \subsetneq P_2$.



First of all we verify that $P_1 \subseteq P_2$. To do so, we show that if $(x,y) \in P_1$ then $(x,y) \in P_2$. If $(x,y) \in P_1$, then $x_{ij} \leq y_i$ for all $i = 1, \ldots, n$, $j = 1, \ldots, m$. Then for every fixed $i \in \{1, \ldots, n\}$, the sum of the m inequalities $x_{ij} \leq y_i$ for $j = 1, \ldots, m$ gives $\sum_{j=1}^{m} x_{ij} \leq m y_i$, and therefore $(x,y) \in P_2$.

Finally, in order to show that $P_1 \neq P_2$, it is sufficient to find a point in $P_2 \setminus P_1$. Take $n = 2$, $m = 4$, and consider the point given by

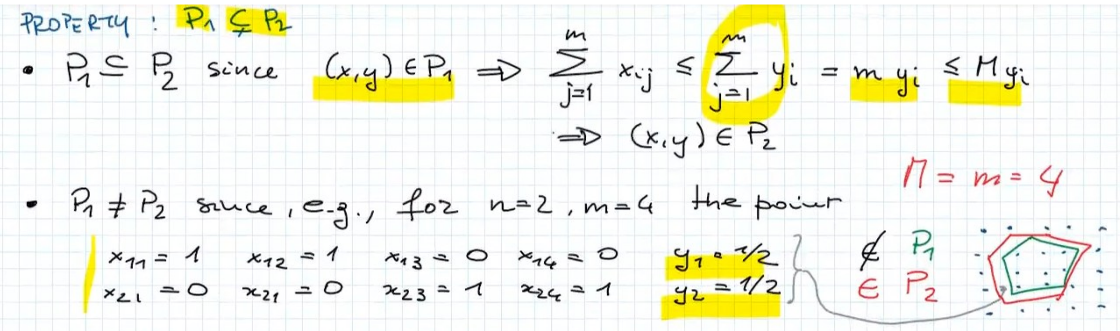
$$x_{11} = 1,\ x_{12} = 1,\ x_{13} = 0,\ x_{14} = 0;$$

$$x_{21} = 0,\ x_{22} = 0,\ x_{23} = 1,\ x_{24} = 1;$$

$$y_1 = \frac{1}{2},\ y_2 = \frac{1}{2}.$$

This point satisfy the aggregated constraints but violates the non-aggregated constraints, because $1 = x_{11} \not\leq y_1 = \frac{1}{2}$. ■

This proves that P_1 is strictly contained in P_2, making P_1 a tighter formulation despite having more constraints. A tighter formulation often provides better bounds for branch-and-bound, though it may be computationally more expensive to solve due to the larger number of constraints.



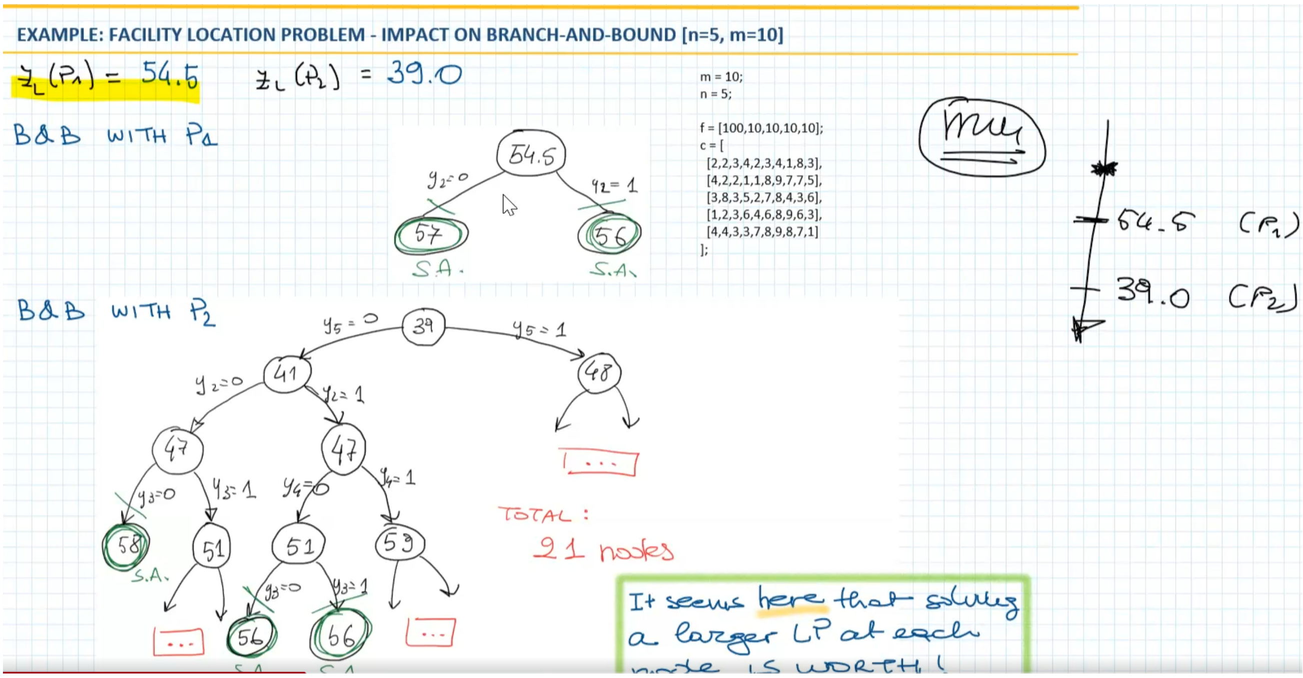
PROPERTY : $P_1 \subsetneq P_2$

- $P_1 \subseteq P_2$ since $(x,y) \in P_1 \Rightarrow \sum_{j=1}^{m} x_{ij} \leq \sum_{j=1}^{m} y_i = m y_i \leq M y_i$
  $\Rightarrow (x,y) \in P_2$

$M = m = 4$

- $P_1 \neq P_2$ since, e.g., for $n=2$, $m=4$ the point
  $x_{11} = 1$ $x_{12} = 1$ $x_{13} = 0$ $x_{14} = 0$ $y_1 = 1/2$
  $x_{21} = 0$ $x_{21} = 0$ $x_{23} = 1$ $x_{24} = 1$ $y_2 = 1/2$
  $\notin P_1$
  $\in P_2$

Imagine now using the Branch and Bound method, solving the linear relaxation of the problem. This is a special instance of the problem with 5 customers and 10 facilities.



EXAMPLE: FACILITY LOCATION PROBLEM - IMPACT ON BRANCH-AND-BOUND [n=5, m=10]

$z_L(P_1) = 54.5$ $z_L(P_2) = 39.0$

B&B WITH P_1

```
m = 10;
n = 5;

f = [100,10,10,10,10];
c = [
  [2,2,3,4,2,3,4,1,8,3],
  [4,2,2,1,1,8,9,7,7,5],
  [3,8,3,5,2,7,8,4,3,6],
  [1,2,3,6,4,6,8,9,6,3],
  [4,4,3,3,7,8,9,8,7,1]
];
```

B&B WITH P_2

TOTAL : 21 nodes

It seems here that solving a larger LP at each node IS WORTH !

Note: we do not consider the multi-period production in these examples (for time reasons, as it seems).

Written by Gabriel R.

P1 Tree (First Formulation):

- Root node: $z_1(P_1) = 54.6$
- Branching on y_2 creates 2 children:
    - Left ($y_2 \leq 0$): Value 57 (relatively close to root)
    - Right ($y_2 \geq 1$): Value 55.6 (also close to root)
- Only 3 nodes total
- Small gap between parent and child bounds

P2 Tree (Second Formulation):

- Root node: $z_1(P_1) = 39.0$
- Much weaker initial bound
- Required branching on multiple y variables:
    - First on y_2
    - Then y_3
    - Then y_4
    - And so on...
- Total of 21 nodes
- Large gaps between parent and child bounds

The key insight is that the bounds in P_1's tree are much closer to each other. When the bound at any node is closer to the optimal solution (the incumbent), it allows for:

- Better decisions about which nodes to prune
- Fewer branches needed to reach integer solutions
- Earlier pruning of suboptimal branches
- Faster convergence to optimality

This explains why P_1 only needed 3 nodes while P_2 required 21 nodes - the stronger bounds in P_1 allowed for more effective pruning and exploration of the search tree. The closer the bounds are to the incumbent solution, the more powerful the branch and bound algorithm becomes at eliminating suboptimal regions of the search space

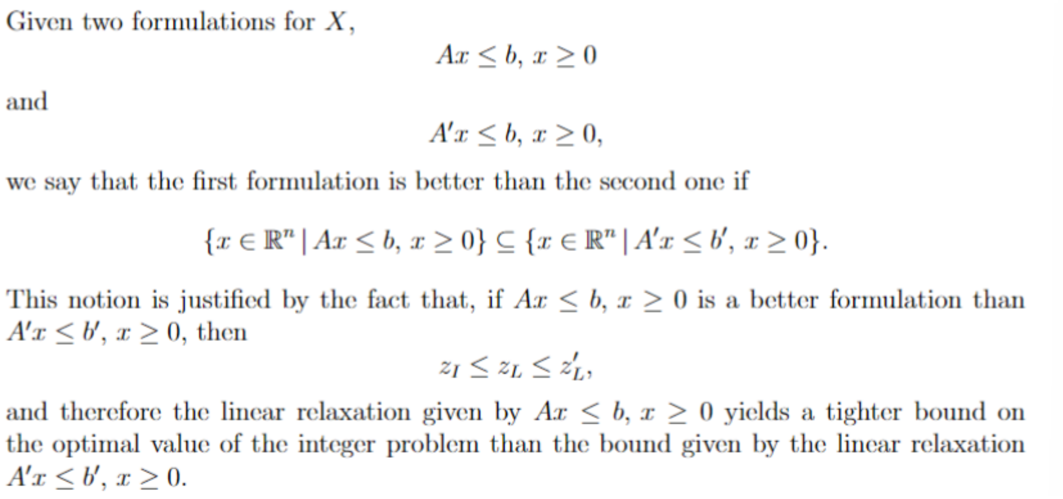
This example clearly illustrates why having a tighter formulation (P_1) can be beneficial despite having more constraints - it leads to stronger bounds and a more efficient B&B process, even though each individual LP relaxation might be more expensive to solve. Given the representation is larger, it is not so close to the integer points. The closer the bound is to the optimal solution, the easier that bound is closer to an incumbent.

A formulation is considered <u>better</u> compared to another one when:

$$\underbrace{\{Ax = b, x \geq 0\}}_{polyhedron} \subseteq \{A'x = b', x \geq 0\}$$

so, when the points of a polyhedron are all contained inside of another one, which can be determined algebraically.

This is to be represented by the following:



Given two formulations for X,

$$Ax \le b,\ x \ge 0$$

and

$$A'x \le b,\ x \ge 0,$$

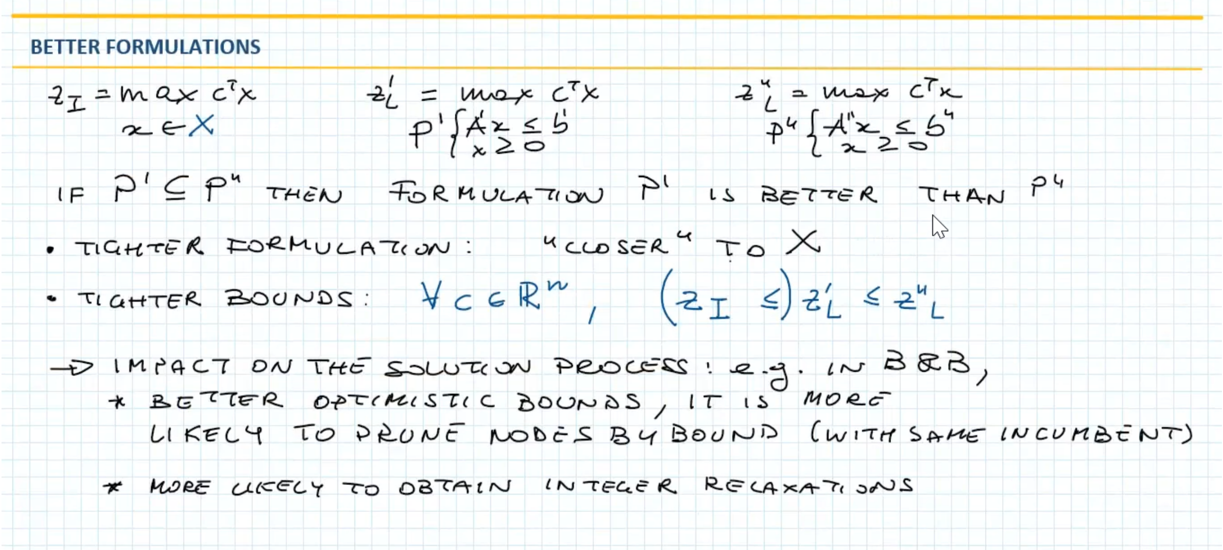
we say that the first formulation is better than the second one if

$$\{x \in \mathbb{R}^n \,|\, Ax \le b,\ x \ge 0\} \subseteq \{x \in \mathbb{R}^n \,|\, A'x \le b',\ x \ge 0\}.$$

This notion is justified by the fact that, if $Ax \le b$, $x \ge 0$ is a better formulation than $A'x \le b'$, $x \ge 0$, then

$$z_I \le z_L \le z'_L,$$

and therefore the linear relaxation given by $Ax \le b$, $x \ge 0$ yields a tighter bound on the optimal value of the integer problem than the bound given by the linear relaxation $A'x \le b'$, $x \ge 0$.



BETTER FORMULATIONS

$z_I = \max c^T x$, $x \in X$ $\quad$ $z'_L = \max c^T x$, $P'\{A'x \le b', x \ge 0\}$ $\quad$ $z''_L = \max c^T x$, $P''\{A''x \le b'', x \ge 0\}$

IF $P' \subseteq P''$ THEN FORMULATION P' IS BETTER THAN P''

- TIGHTER FORMULATION: "CLOSER" TO X
- TIGHTER BOUNDS: $\forall c \in \mathbb{R}^n$, $(z_I \le) z'_L \le z''_L$

→ IMPACT ON THE SOLUTION PROCESS: e.g. IN B&B,

* BETTER OPTIMISTIC BOUNDS, IT IS MORE LIKELY TO PRUNE NODES BY BOUND (WITH SAME INCUMBENT)

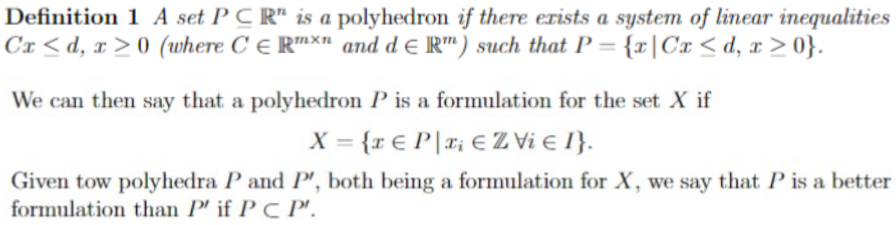
* MORE LIKELY TO OBTAIN INTEGER RELAXATIONS

This means that even the problem formulation may be interesting from a computational point of view.

8.2.2 Convex Hull and Ideal Formulation

At the geometric level, however, it is possible to define the ideal formulation, that is, the one that is geometrically best (alternatively – the formulation for X whose continuous relaxation is as small as possible with respect to set inclusion).

- Thus, the ideal solution is the convex hull of the feasible region, that is, the minimal convex set containing the feasible-region (in Italian – "inviluppo convesso") – see here
- A convex set C is a set of points, such that $\forall x, y \in C$, the segment joining them is completely contained in C – smallest convex set that encloses all the points, forming a convex polygon
- All polyhedra associated with formulation are always convex sets

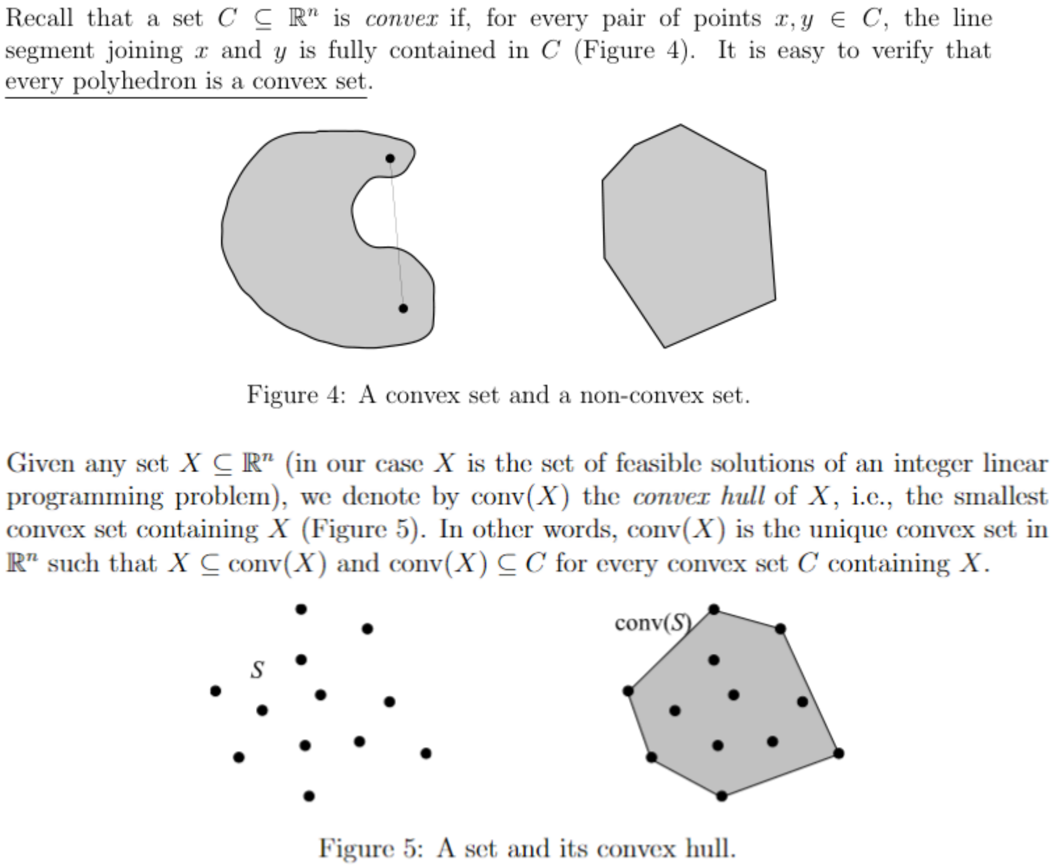


Definition 1 *A set $P \subseteq \mathbb{R}^n$ is a* polyhedron *if there exists a system of linear inequalities $Cx \le d$, $x \ge 0$ (where $C \in \mathbb{R}^{m\times n}$ and $d \in \mathbb{R}^m$) such that $P = \{x \,|\, Cx \le d, x \ge 0\}$.*

We can then say that a polyhedron P is a formulation for the set X if

$$X = \{x \in P \,|\, x_i \in \mathbb{Z}\ \forall i \in I\}.$$

Given tow polyhedra P and P', both being a formulation for X, we say that P is a better formulation than P' if $P \subset P'$.



Recall that a set $C \subseteq \mathbb{R}^n$ is *convex* if, for every pair of points $x, y \in C$, the line segment joining x and y is fully contained in C (Figure 4). It is easy to verify that every polyhedron is a convex set.

Figure 4: A convex set and a non-convex set.

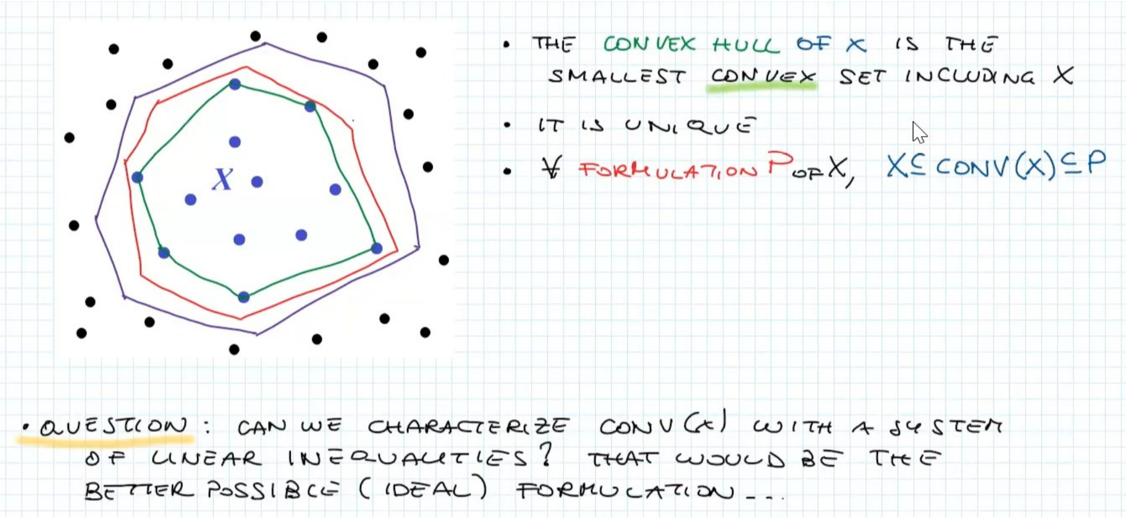
Given any set $X \subseteq \mathbb{R}^n$ (in our case X is the set of feasible solutions of an integer linear programming problem), we denote by $\mathrm{conv}(X)$ the *convex hull* of X, i.e., the smallest convex set containing X (Figure 5). In other words, $\mathrm{conv}(X)$ is the unique convex set in $\mathbb{R}^n$ such that $X \subseteq \mathrm{conv}(X)$ and $\mathrm{conv}(X) \subseteq C$ for every convex set C containing X.

Figure 5: A set and its convex hull.

Given a formulation $P = \{x \mid Cx \leq d, x \geq 0\}$ for X, since P is a convex set containing X, we have that

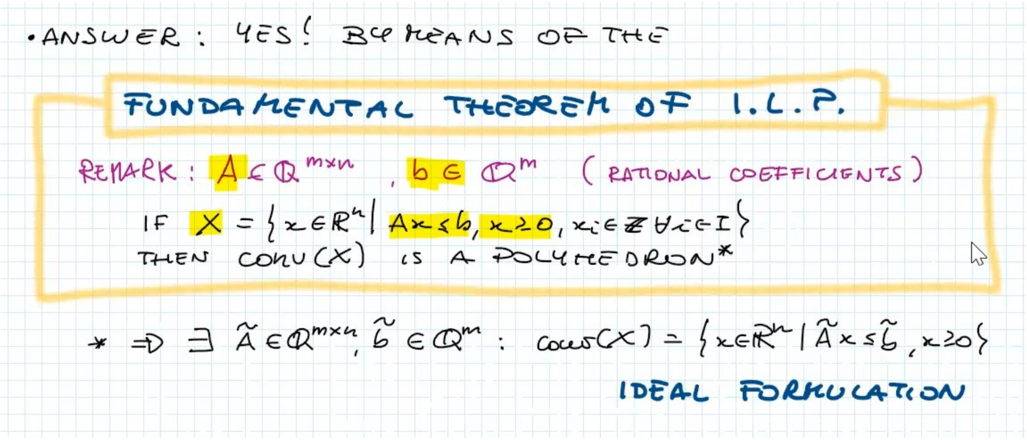
$$X \subseteq conv(X) \subseteq P$$

Then $conv(X)$ is contained in the feasible region of the continuous relaxation of every formulation of X. The following is a fundamental result in integer linear programming. It shows that there exists a formulation for whose continuous relaxation is precisely $conv(X)$.



- THE CONVEX HULL OF X IS THE SMALLEST CONVEX SET INCLUDING X
- IT IS UNIQUE
- ∀ FORMULATION P OF X, $X \subseteq CONV(X) \subseteq P$

- QUESTION: CAN WE CHARACTERIZE CONV(X) WITH A SYSTEM OF LINEAR INEQUALITIES? THAT WOULD BE THE BETTER POSSIBLE (IDEAL) FORMULATION...



- ANSWER: YES! BY MEANS OF THE

FUNDAMENTAL THEOREM OF I.L.P.

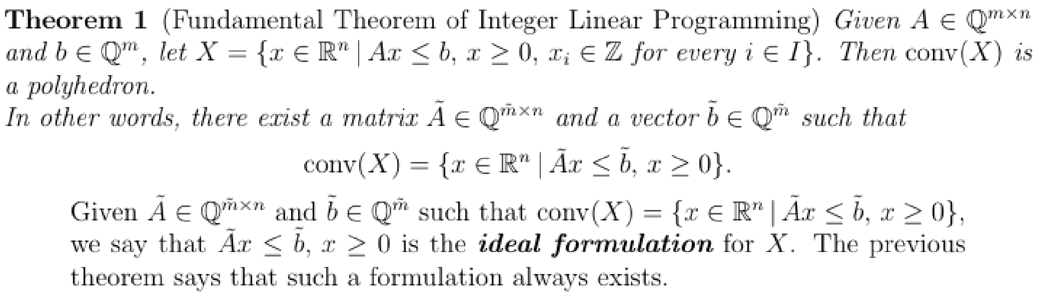
REMARK: $A \in \mathbb{Q}^{m \times n}$, $b \in \mathbb{Q}^m$ (RATIONAL COEFFICIENTS)

IF $X = \{x \in \mathbb{R}^n \mid Ax \leq b, x \geq 0, x_i \in \mathbb{Z} \; \forall i \in I\}$

THEN $conv(X)$ IS A POLYHEDRON*

$* \Rightarrow \exists \; \tilde{A} \in \mathbb{Q}^{m \times n}, \tilde{b} \in \mathbb{Q}^m : conv(X) = \{x \in \mathbb{R}^n \mid \tilde{A}x \leq \tilde{b}, x \geq 0\}$

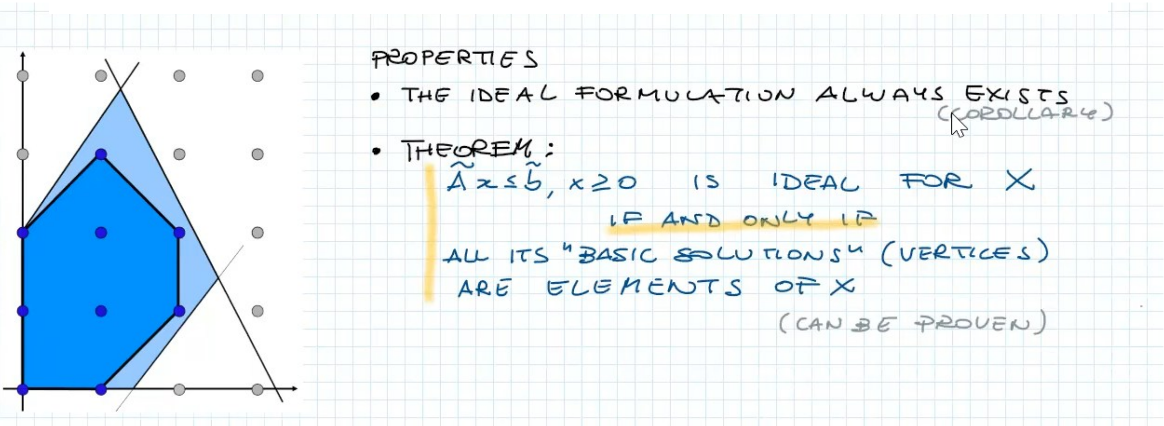
IDEAL FORMULATION



Theorem 1 (Fundamental Theorem of Integer Linear Programming) *Given $A \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{Q}^m$, let $X = \{x \in \mathbb{R}^n \mid Ax \leq b, \, x \geq 0, \, x_i \in \mathbb{Z}$ for every $i \in I\}$. Then* conv(X) *is a polyhedron.*
In other words, there exist a matrix $\tilde{A} \in \mathbb{Q}^{\tilde{m} \times n}$ and a vector $\tilde{b} \in \mathbb{Q}^{\tilde{m}}$ such that

$$\text{conv}(X) = \{x \in \mathbb{R}^n \mid \tilde{A}x \leq \tilde{b}, \, x \geq 0\}.$$

Given $\tilde{A} \in \mathbb{Q}^{\tilde{m} \times n}$ and $\tilde{b} \in \mathbb{Q}^{\tilde{m}}$ such that conv$(X) = \{x \in \mathbb{R}^n \mid \tilde{A}x \leq \tilde{b}, \, x \geq 0\}$, we say that $\tilde{A}x \leq \tilde{b}, \, x \geq 0$ is the ***ideal formulation*** for X. The previous theorem says that such a formulation always exists.

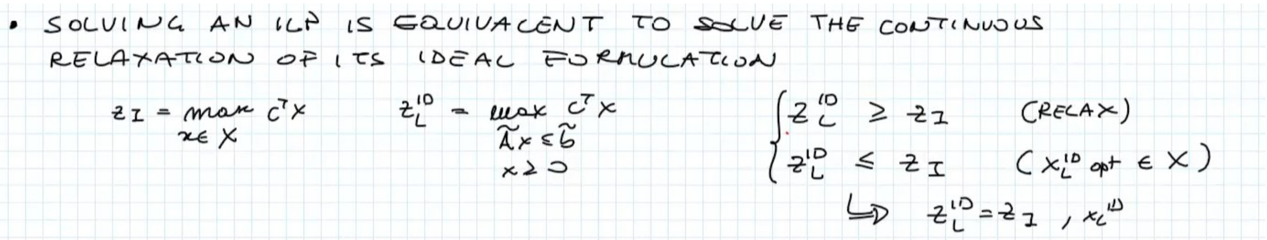


PROPERTIES

- THE IDEAL FORMULATION ALWAYS EXISTS (COROLLARY)
- THEOREM:
  $\tilde{A}x \leq \tilde{b}, x \geq 0$ IS IDEAL FOR X
  IF AND ONLY IF
  ALL ITS "BASIC SOLUTIONS" (VERTICES) ARE ELEMENTS OF X
  (CAN BE PROVEN)

The Fundamental theorem of Integer Linear Programming says that given an (M)ILP (in the form of a maximum), such that the matrix A contains only rational values and the convex hull of the feasible region is a polyhedron, that is, it can be expressed as $\{Cx \leq d, x \geq 0\}$. We assume this theorem.

The impact on solving ILP with ideal formulations is given by the fact that solving an ILP (min/max) is equivalent to solving the continuous relaxation of its ideal formulation $(conv(X))$ with the simplex method.



- SOLVING AN ILP IS EQUIVALENT TO SOLVE THE CONTINUOUS RELAXATION OF ITS IDEAL FORMULATION

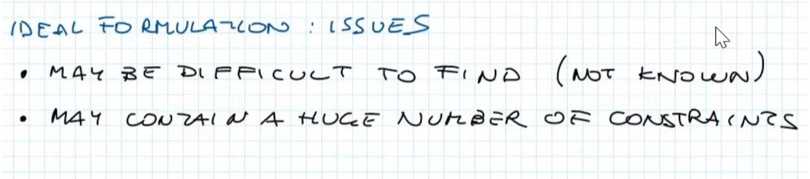
$z_I = \max_{x \in X} c^T x$ $\quad$ $z_L^{ID} = \max c^T x$, $\tilde{A}x \leq \tilde{b}$, $x \geq 0$

$$\begin{cases} z_L^{ID} \geq z_I & \text{(RELAX)} \\ z_L^{ID} \leq z_I & (x_L^{ID} \text{ opt} \in X) \end{cases}$$

$\hookrightarrow z_L^{ID} = z_I \;, x_L^{ID}$

Therefore, in principle, solving an ILP equivalent to solving a LP problem (with no integer variables) in which the constraints define the ideal formulation. However, there are two major issues which make integer linear programming harder than linear programming:

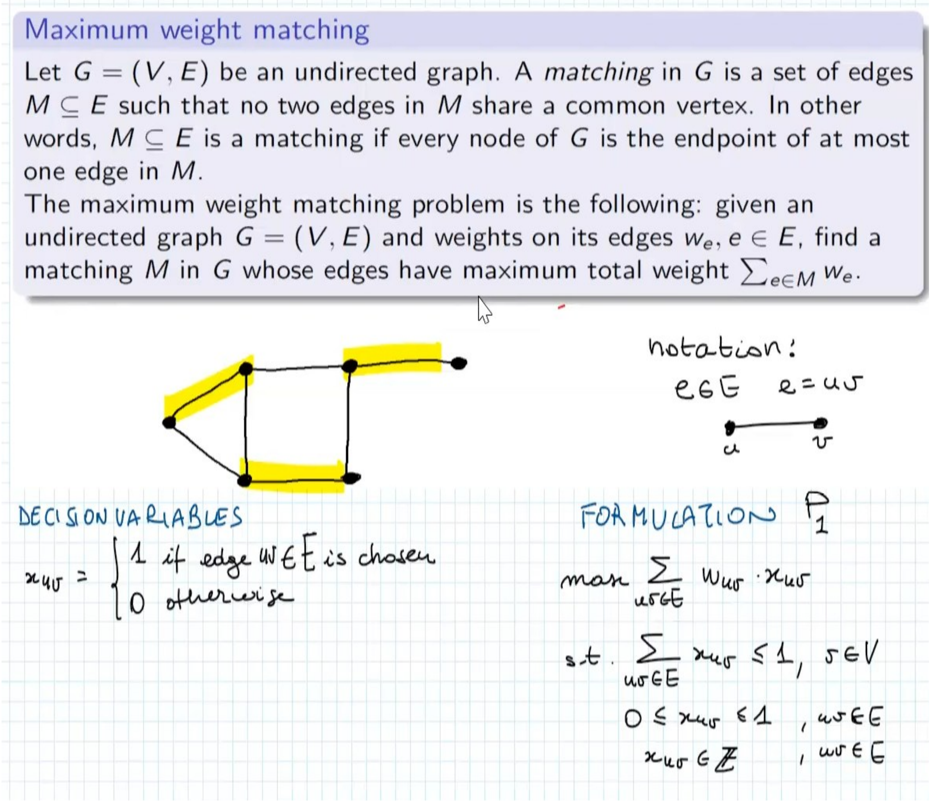
- The ideal formulation, in general, is not known, and it can be very difficult to find it
- Even in cases in which the ideal formulation is known, it is often described by a huge number of constraints, and therefore problem (10) cannot be solved directly with standard LP algorithms (such as the simplex method) – since $conv(x)$ is NP-complete



IDEAL FORMULATION : ISSUES

- MAY BE DIFFICULT TO FIND (NOT KNOWN)
- MAY CONTAIN A HUGE NUMBER OF CONSTRAINTS

Let's consider another example: *maximum weight matching*.



Maximum weight matching

Let $G = (V, E)$ be an undirected graph. A *matching* in G is a set of edges $M \subseteq E$ such that no two edges in M share a common vertex. In other words, $M \subseteq E$ is a matching if every node of G is the endpoint of at most one edge in M.
The maximum weight matching problem is the following: given an undirected graph $G = (V, E)$ and weights on its edges $w_e, e \in E$, find a matching M in G whose edges have maximum total weight $\sum_{e \in M} w_e$.

notation:
$e \in E \quad e = uv$

DECISION VARIABLES

$$x_{uv} = \begin{cases} 1 & \text{if edge } uv \in E \text{ is chosen} \\ 0 & \text{otherwise} \end{cases}$$

FORMULATION P_1

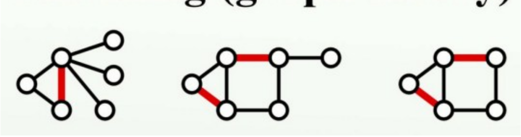
$$\max \sum_{uv \in E} w_{uv} \cdot x_{uv}$$

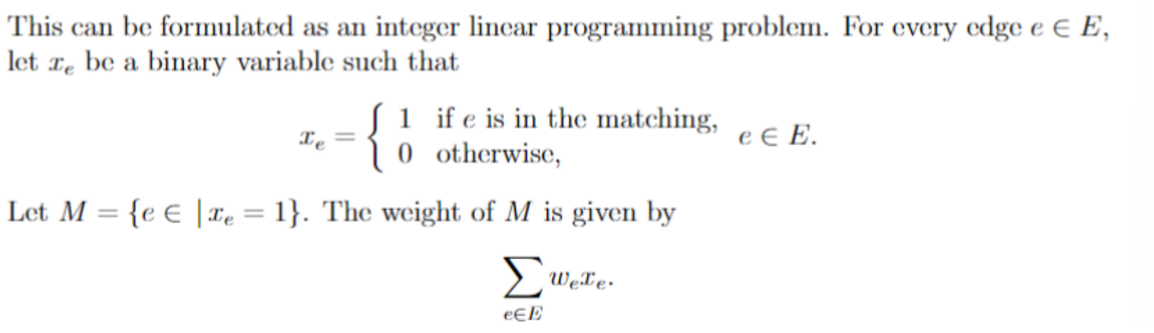
$$\text{s.t.} \sum_{uv \in E} x_{uv} \leq 1, \quad v \in V$$

$$0 \leq x_{uv} \leq 1, \quad uv \in E$$

$$x_{uv} \in \mathbb{Z}, \quad uv \in E$$

Matching (graph theory)



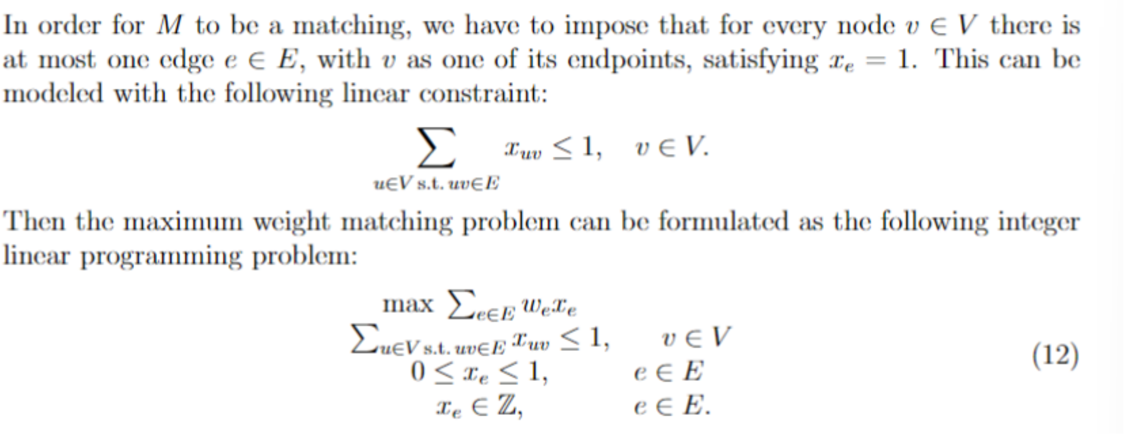


This can be formulated as an integer linear programming problem. For every edge $e \in E$, let x_e be a binary variable such that

$$x_e = \begin{cases} 1 & \text{if } e \text{ is in the matching,} \\ 0 & \text{otherwise,} \end{cases} \quad e \in E.$$

Let $M = \{e \in | x_e = 1\}$. The weight of M is given by

$$\sum_{e \in E} w_e x_e.$$



In order for M to be a matching, we have to impose that for every node $v \in V$ there is at most one edge $e \in E$, with v as one of its endpoints, satisfying $x_e = 1$. This can be modeled with the following linear constraint:

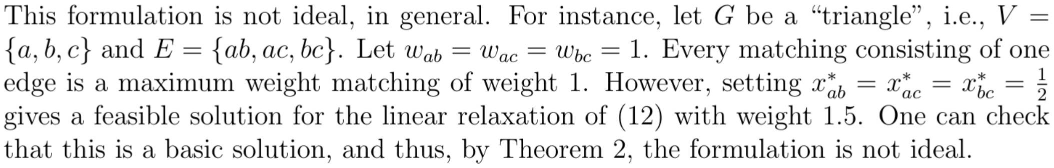
$$\sum_{u \in V \text{ s.t. } uv \in E} x_{uv} \leq 1, \quad v \in V.$$

Then the maximum weight matching problem can be formulated as the following integer linear programming problem:

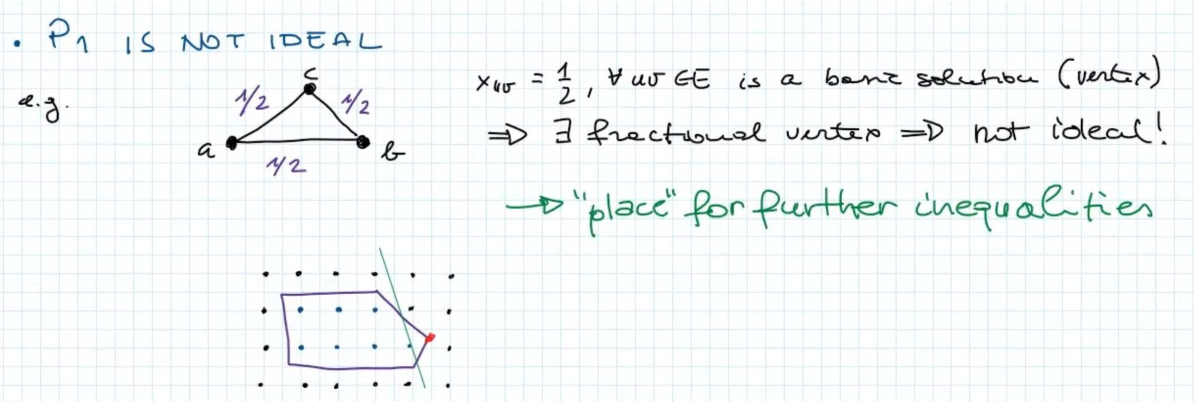
$$\begin{array}{rl} \max \sum_{e \in E} w_e x_e & \\ \sum_{u \in V \text{ s.t. } uv \in E} x_{uv} \leq 1, & v \in V \\ 0 \leq x_e \leq 1, & e \in E \\ x_e \in \mathbb{Z}, & e \in E. \end{array} \tag{12}$$

This formulation is not ideal, in general.

- If the graph is a triangle, every matching consisting of one edge is a maximum weight matching of weight 1
- All solutions give $\frac{1}{2}$ giving a feasible solution which is a basic solution, but not an ideal formulation



This formulation is not ideal, in general. For instance, let G be a "triangle", i.e., $V = \{a, b, c\}$ and $E = \{ab, ac, bc\}$. Let $w_{ab} = w_{ac} = w_{bc} = 1$. Every matching consisting of one edge is a maximum weight matching of weight 1. However, setting $x^*_{ab} = x^*_{ac} = x^*_{bc} = \frac{1}{2}$ gives a feasible solution for the linear relaxation of (12) with weight 1.5. One can check that this is a basic solution, and thus, by Theorem 2, the formulation is not ideal.



• P_1 IS NOT IDEAL

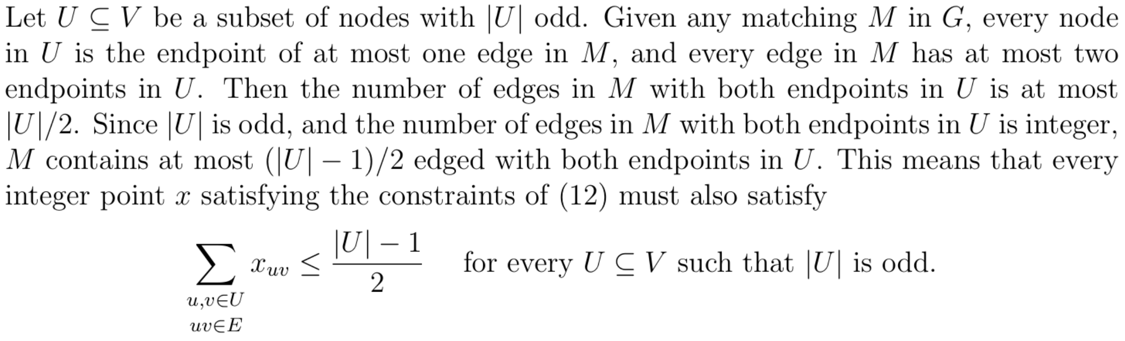
e.g.

$x_{uv} = \frac{1}{2}, \forall uv \in E$ is a basic solution (vertex)

$\Rightarrow \exists$ fractional vertex $\Rightarrow$ not ideal!

→ "place" for further inequalities

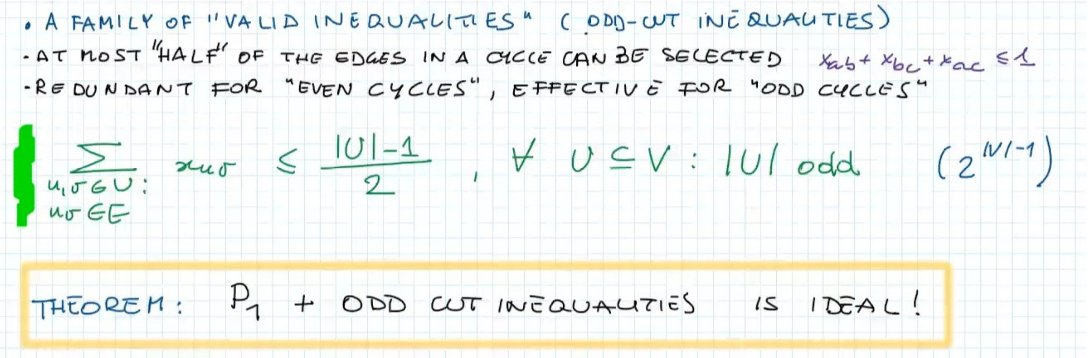
We can find a better formulation by adding "ad-hoc" inequalities, which will be obtained by exploiting the structure of the problem.



Let $U \subseteq V$ be a subset of nodes with $|U|$ odd. Given any matching M in G, every node in U is the endpoint of at most one edge in M, and every edge in M has at most two endpoints in U. Then the number of edges in M with both endpoints in U is at most $|U|/2$. Since $|U|$ is odd, and the number of edges in M with both endpoints in U is integer, M contains at most $(|U|-1)/2$ edged with both endpoints in U. This means that every integer point x satisfying the constraints of (12) must also satisfy

$$\sum_{\substack{u,v \in U \\ uv \in E}} x_{uv} \leq \frac{|U|-1}{2} \quad \text{for every } U \subseteq V \text{ such that } |U| \text{ is odd.}$$

These inequalities are called odd-cut inequalities (below called “o.c.i”), since at least one edge in the cycle has to be selected and this means such cut off fractional solutions that would otherwise be feasible in the LP relaxation (valid for any matching), making the formulation tighter to the extreme points in general.



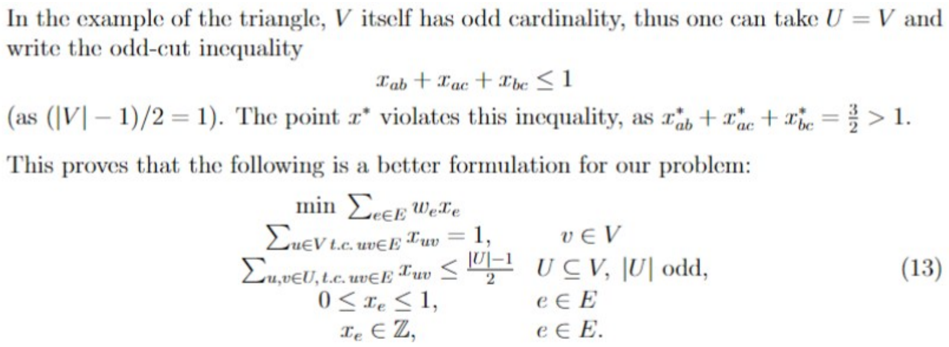
• A FAMILY OF "VALID INEQUALITIES" (ODD-CUT INEQUALITIES)

- AT MOST "HALF" OF THE EDGES IN A CYCLE CAN BE SELECTED $x_{ab}+x_{bc}+x_{ac} \le 1$

- REDUNDANT FOR "EVEN CYCLES", EFFECTIVE FOR "ODD CYCLES"

$$\sum_{\substack{u,v\in U:\\ uv\in E}} x_{uv} \le \frac{|U|-1}{2}, \quad \forall\ U \subseteq V : |U| \text{ odd} \qquad (2^{|V|-1})$$

THEOREM: P_1 + ODD CUT INEQUALITIES IS IDEAL!



In the example of the triangle, V itself has odd cardinality, thus one can take $U = V$ and write the odd-cut inequality

$$x_{ab} + x_{ac} + x_{bc} \le 1$$

(as $(|V| - 1)/2 = 1$). The point x^* violates this inequality, as $x^*_{ab} + x^*_{ac} + x^*_{bc} = \frac{3}{2} > 1$.

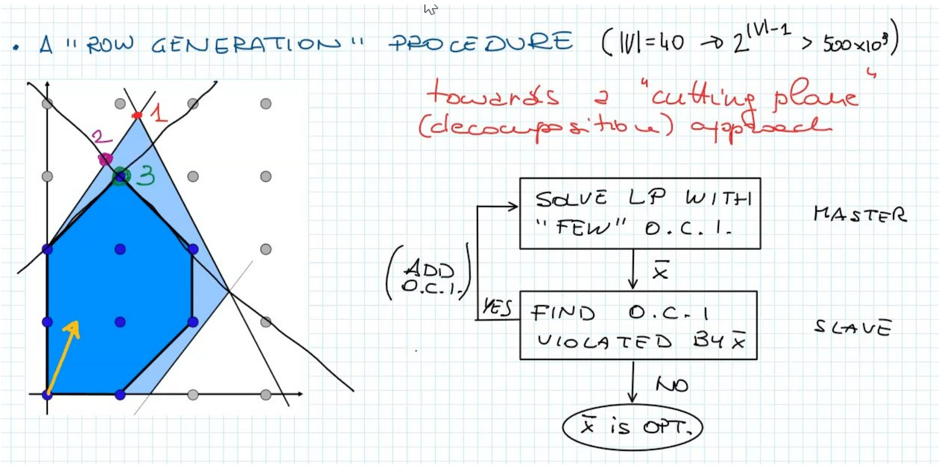
This proves that the following is a better formulation for our problem:

$$\begin{array}{lll} \min \sum_{e\in E} w_e x_e & & \\ \sum_{u\in V\,\text{t.c.}\,uv\in E} x_{uv} = 1, & v \in V & \\ \sum_{u,v\in U,\,\text{t.c.}\,uv\in E} x_{uv} \le \frac{|U|-1}{2} & U \subseteq V,\ |U| \text{ odd}, & (13) \\ 0 \le x_e \le 1, & e \in E & \\ x_e \in \mathbb{Z}, & e \in E. & \end{array}$$

Theorem: P_1 + odd cut inequalities is ideal

(therefore it would be sufficient to solve its continuous relaxation to find the optimum of the integer problem)

- However, the number of constraints is exponential (there are $2^{|V|-1}$ subsets of V with odd cardinality) and it is therefore practically impossible to solve the relaxation
  - Even for a graph with just 40 nodes, there are more than 500 billion ($500 * 10^9$) odd-cut inequalities
- A better strategy is to solve a sequence of linear relaxations, starting from problem (12) and at every iteration adding one or more odd-cut inequalities that exclude the current optimal solution, until an optimal solution of the integer problem is found
- This idea is discussed in the next section (with a more general framework)



• A "ROW GENERATION" PROCEDURE ($|V|=40 \rightarrow 2^{|V|-1} > 500\times10^9$)

towards a "cutting plane" (decomposition) approach

Written by Gabriel R.

When we have an Integer Linear Programming (ILP) problem, we often face a situation where:

- There are too many potential constraints (like said above)
- Including all these constraints upfront is computationally impossible

Row generation is an approach where:

- We start with a basic formulation (shown as the yellow region in the image), which is a subset of constraints
- We identify violated inequalities (rows/constraints)
- We add these constraints one at a time solving the relaxed problem
- Check if the solution violates any of the omitted constraints and if violations are found, add ("generate") those constraints and repeat
- This gradually tightens the formulation towards the convex hull (blue region)

Outcomes by this:

- We're trying to approach the ideal formulation (convex hull of integer solutions)
- The convex hull would give us the tightest possible LP relaxation
- But describing it completely is usually impractical
- Row generation gives us a practical way to get closer to it

The golden arrow in the image suggests the direction of improvement – we're trying to shrink the relaxation (yellow) toward the convex hull (blue) by adding strategic cuts.

This leads to the Cutting Plane Approach:

- Solve the LP relaxation
- If the solution x^* is not integer:
  - Find a valid inequality that is violated by x^*
  - Add this inequality as a "cut"
  - Re-solve the LP
- Repeat until we get an integer solution or can't find more cuts

The key advantages are:

- We avoid dealing with exponentially many constraints upfront
- We only generate constraints that are actually needed
- Each iteration makes the formulation tighter
- We can often solve large problems that would be impossible to handle if we included all constraints initially

This approach is particularly powerful because:

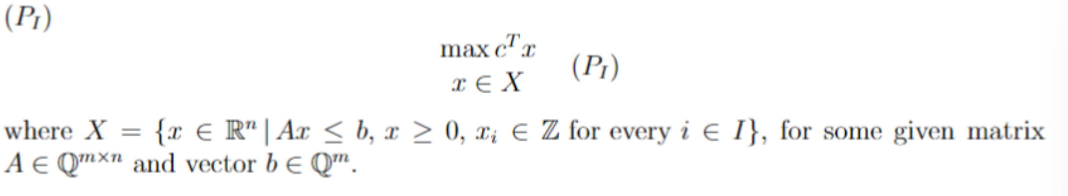
- It's more efficient than enumerating all constraints
- It can be combined with branch-and-bound (leading to branch-and-cut)
- Many problems have efficient separation procedures to find violated inequalities

Row generation is used do deal with *known* families of valid inequalities, while cutting plane method find new inequalities, adding cuts valid for integer solutions.

8.3 CUTTING PLANE METHODS

The idea behind the cutting plane methods (“metodi dei piani di taglio”) is to solve a sequence of linear relaxations that approximate better and better the convex hull of the feasible region around the optimal solution. Cutting planes *strictly* improve formulations of X!

Consider more formally we want to solve the following ILP:

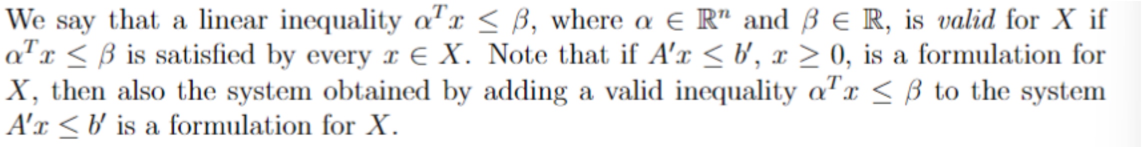


(P_I)

$$\max c^T x \quad x \in X \qquad (P_I)$$

where $X = \{x \in \mathbb{R}^n \mid Ax \le b,\ x \ge 0,\ x_i \in \mathbb{Z} \text{ for every } i \in I\}$, for some given matrix $A \in \mathbb{Q}^{m\times n}$ and vector $b \in \mathbb{Q}^m$.

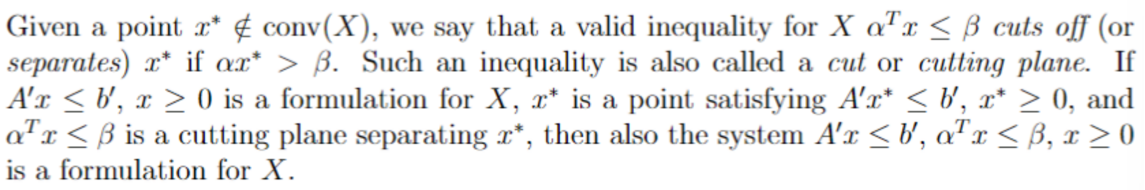
A valid inequality does not cut off valid points:



We say that a linear inequality $\alpha^T x \le \beta$, where $\alpha \in \mathbb{R}^n$ and $\beta \in \mathbb{R}$, is *valid* for X if $\alpha^T x \le \beta$ is satisfied by every $x \in X$. Note that if $A'x \le b'$, $x \ge 0$, is a formulation for X, then also the system obtained by adding a valid inequality $\alpha^T x \le \beta$ to the system $A'x \le b'$ is a formulation for X.

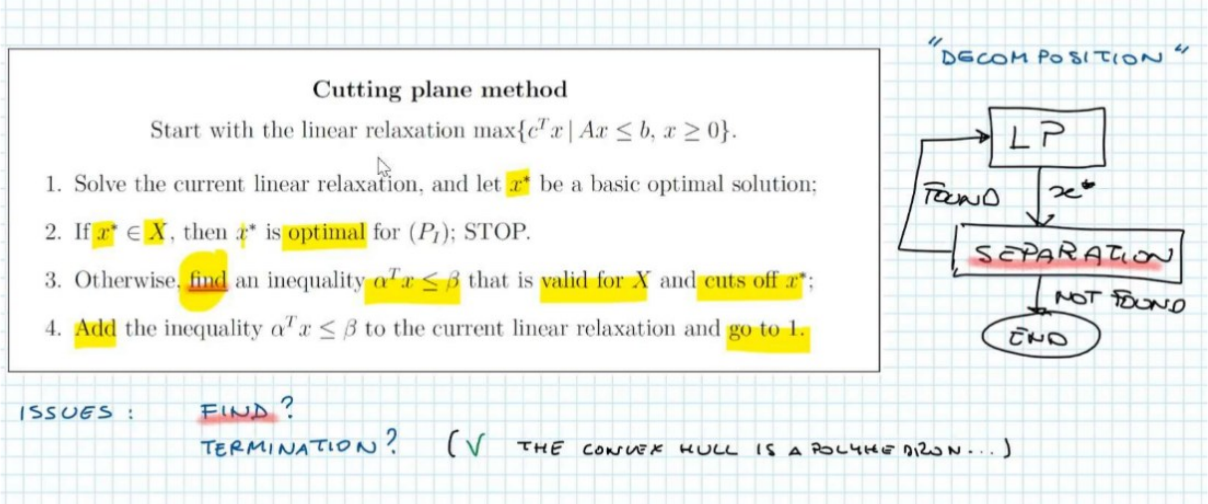
Basically, it is a constraint that is satisfied by ALL feasible integer solutions of your original problem, even if it's not part of your initial formulation. Think of it as a "hidden rule" that you discover.

A cut separates the convex hull from X (so to cut away non-feasible solutions):



Given a point $x^* \notin \text{conv}(X)$, we say that a valid inequality for X $\alpha^T x \le \beta$ *cuts off* (or *separates*) x^* if $\alpha x^* > \beta$. Such an inequality is also called a *cut* or *cutting plane*. If $A'x \le b'$, $x \ge 0$ is a formulation for X, x^* is a point satisfying $A'x^* \le b'$, $x^* \ge 0$, and $\alpha^T x \le \beta$ is a cutting plane separating x^*, then also the system $A'x \le b'$, $\alpha^T x \le \beta$, $x \ge 0$ is a formulation for X.

The general method given below is a general framework for tackling integer linear programming problems, but in order to implement it one needs an automatic technique to find valid inequalities that cut off the current solution.



Cutting plane method

Start with the linear relaxation $\max\{c^T x \mid Ax \le b, x \ge 0\}$.

1. Solve the current linear relaxation, and let x^* be a basic optimal solution;
2. If $x^* \in X$, then x^* is optimal for (P_I); STOP.
3. Otherwise, find an inequality $\alpha^T x \le \beta$ that is valid for X and cuts off x^*;
4. Add the inequality $\alpha^T x \le \beta$ to the current linear relaxation and go to 1.

ISSUES : FIND ?
TERMINATION ? (✓ THE CONVEX HULL IS A POLYHEDRON...)

As said above – this method works like this:

- You solve the LP relaxation (ignore integer constraints)
- If you get a fractional solution x^*, you try to find a valid inequality that:
    - Is satisfied by all feasible integer solutions
    - It is violated by your current fractional solution x^*
- This inequality "cuts off" the fractional solution while keeping all integer solutions.
- Add this cut to your problem and resolve
- Repeat until you get an integer solution

The term "cutting plane" comes from the geometric interpretation:

- The new inequality is like drawing a line (plane in higher dimensions)
- This line "cuts off" part of your feasible region
- Specifically, it cuts off the current fractional solution
- But it doesn't cut off any integer solutions you care about

The decomposition idea appears because we're essentially breaking the problem into two parts:

- MASTER PROBLEM:
    - The LP relaxation we keep solving
    - Gets progressively tighter with each cut
- SUBPROBLEM:
    - The separation problem of finding violated inequalities
    - Functions as a "cut generator"

The decomposition is natural because:

- It's often easier to solve these two parts separately
- The separation problem (finding cuts) might have a special structure we can exploit
- We can develop specialized algorithms for each part

This method is powerful because it dynamically generates only the necessary constraints, rather than starting with all possible constraints upfront.

There are *issues* however:

- The major challenge is how to systematically *find* valid inequalities that cut off fractional solutions. This isn't trivial because:
    - There could be many possible valid inequalities
    - We need an efficient way to identify which ones are helpful
    - We need to ensure they're actually valid for all integer solutions
- We need to know if and when the algorithm *will stop*. This relates to:
    - Whether we can keep finding useful cuts
    - Whether we'll reach an integer solution in finite time
    - The theoretical guarantee that the process converges (convex hull is a polyhedron)

It is clear that the above method is a general framework for tackling integer linear programming problems, but in order to implement it one needs an automatic technique to find valid inequalities that cut off the current solution. Below, we give a possible technique to do this.

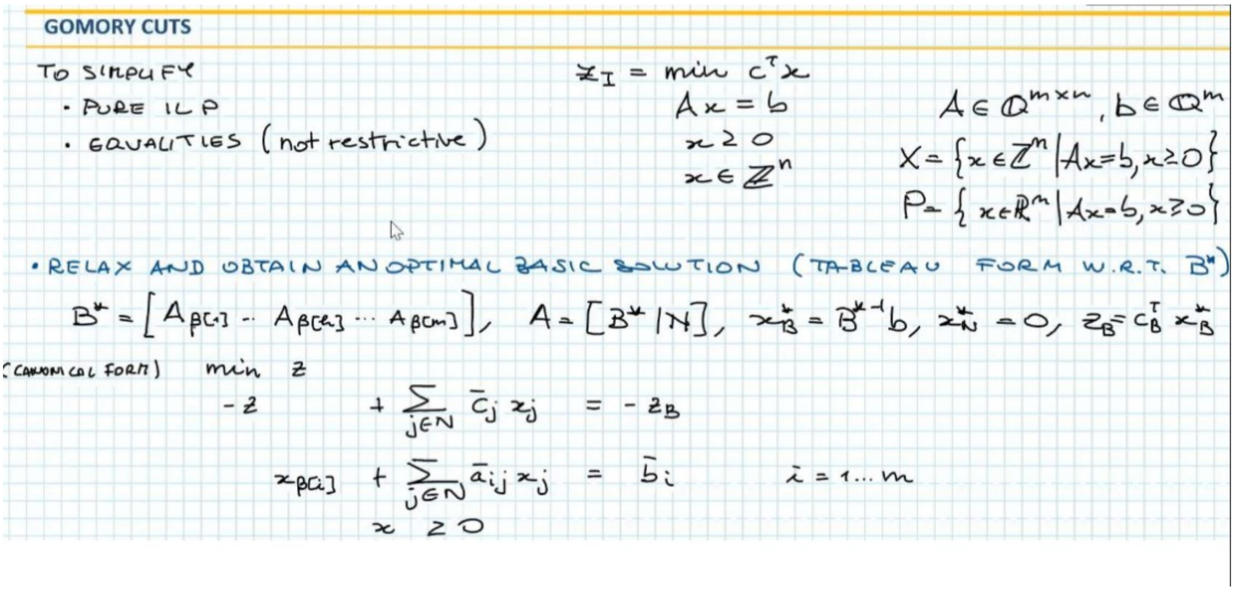
8.3.1 Gomory Cuts

Gomory cutting plane method can be applied only to *pure* ILP problems – all variables integer/all constraints linear – (although there are extensions to the mixed integer case) and done to simplify pure ILP or inequalities (not restrictive). Thus we consider the problem:

$$\begin{aligned} z_I &= \min c^t x \\ Ax &= b \\ x &\geq 0 \\ x &\in \mathbb{Z}^n \end{aligned} \qquad (14)$$

where $A \in \mathbb{Q}^{m \times n}$ and $b \in \mathbb{Q}^m$. Define $X = \{x \mid Ax = b,\ x \geq 0,\ x \in Z^n\}$.

Solve the continuous relaxation with the simplex method, thus obtaining the problem in tableau form with respect to an optimal basis B (where below, N is the set of indices of the non-basic variables):



GOMORY CUTS

TO SIMPLIFY
- PURE ILP
- EQUALITIES (not restrictive)

$$z_I = \min c^T x \quad Ax = b \quad x \geq 0 \quad x \in \mathbb{Z}^n$$

$A \in \mathbb{Q}^{m \times n}$, $b \in \mathbb{Q}^m$

$X = \{x \in \mathbb{Z}^n \mid Ax = b, x \geq 0\}$

$P = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$

- RELAX AND OBTAIN AN OPTIMAL BASIC SOLUTION (TABLEAU FORM W.R.T. B^*)

$B^* = [A_{\beta[1]} \cdots A_{\beta[2]} \cdots A_{\beta[m]}]$, $A = [B^* \mid N]$, $x^*_B = B^{*-1} b$, $x^*_N = 0$, $z_B = c_B^T x^*_B$

(CANONICAL FORM) $\min z$

$$\begin{aligned} -z \quad & + \sum_{j \in N} \bar{c}_j x_j = -z_B \\ x_{\beta[i]} & + \sum_{j \in N} \bar{a}_{ij} x_j = \bar{b}_i \qquad i = 1 \dots m \\ x & \geq 0 \end{aligned}$$

If the right hand side is fractional (not integer), we can create a cut satisfying any integer solution and cutting off the current fractional solution.

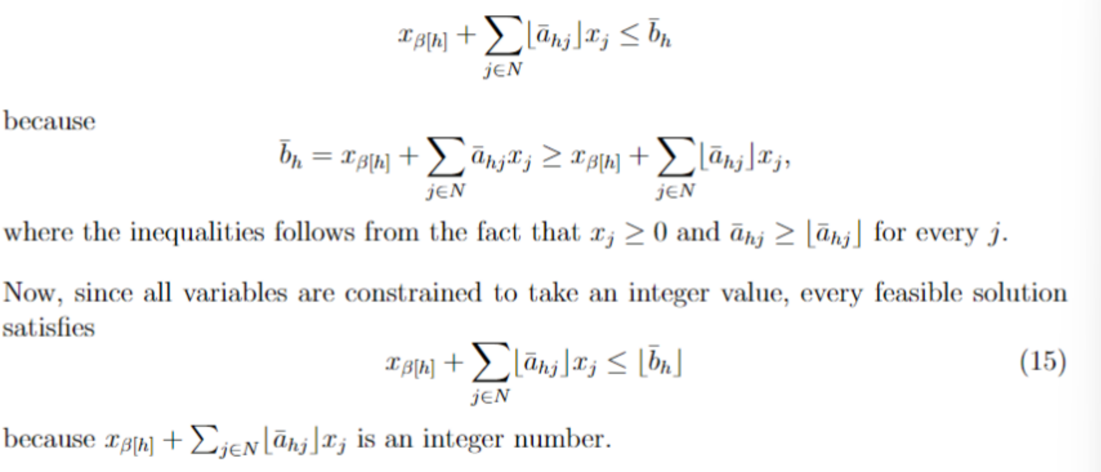
Since B is an optimal basis, the reduced costs are non-negative: $\bar{c}_j \geq 0$ for every $j \in N$. The optimal basic solution x^* is given by

$$\begin{aligned} x^*_{\beta[i]} &= \bar{b}_i, \quad i = 1, \dots, m; \\ x^*_j &= 0, \quad j \in N; \end{aligned}$$

therefore $x^* \in \mathbb{Z}^n$ if and only if $\bar{b}_i \in \mathbb{Z}$ for all $i = 1, \dots, m$.

If this is not the case, let $h \in \{1, \dots, m\}$ be an index such that $\bar{b}_h \notin \mathbb{Z}$.

Note that every vector x satisfying the linear equation system $Ax = b, x \geq 0$ also satisfies:



$$x_{\beta[h]} + \sum_{j \in N} \lfloor \bar{a}_{hj} \rfloor x_j \leq \bar{b}_h$$

because

$$\bar{b}_h = x_{\beta[h]} + \sum_{j \in N} \bar{a}_{hj} x_j \geq x_{\beta[h]} + \sum_{j \in N} \lfloor \bar{a}_{hj} \rfloor x_j,$$

where the inequalities follows from the fact that $x_j \geq 0$ and $\bar{a}_{hj} \geq \lfloor \bar{a}_{hj} \rfloor$ for every j.

Now, since all variables are constrained to take an integer value, every feasible solution satisfies

$$x_{\beta[h]} + \sum_{j \in N} \lfloor \bar{a}_{hj} \rfloor x_j \leq \lfloor \bar{b}_h \rfloor \tag{15}$$

because $x_{\beta[h]} + \sum_{j \in N} \lfloor \bar{a}_{hj} \rfloor x_j$ is an integer number.

The inequality (15) is a Gomory cut, since it is a valid inequality for X.

- It examines a fractional solution from the simplex method and creates a new constraint that removes this fractional solution while keeping all valid integer solutions
- It does this by exploiting the fact that if all variables must be integer, then certain combinations of these variables must also result in integer values
    - When we find a fractional value in our solution, we can create a constraint that essentially says "this combination must be integer" which cuts off our current fractional solution but preserves all true integer solutions

In simpler terms:

- Start with a linear relaxation of an integer program
- Solve it using simplex method
- If we get a fractional solution, we want to cut it off

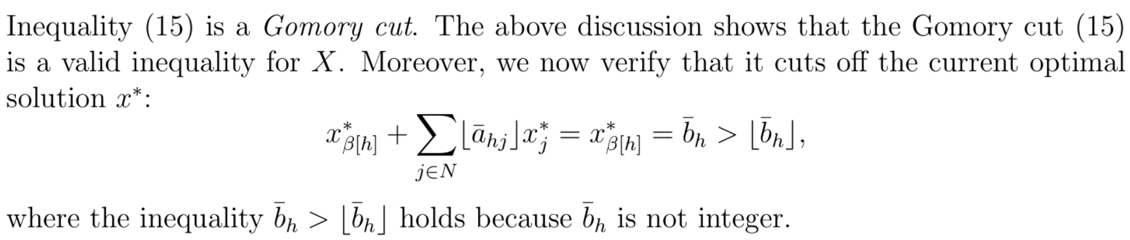
This is how it works:

- When we have a simplex tableau with fractional basic variables:
    - Pick a row where the right-hand side is fractional
    - Round down all the coefficients to have all left side integer and floor to be the largest
    - From this row, create a new inequality that:
        - Must be satisfied by all integer solutions
        - Is violated by the current fractional solution
- If you have a constraint like:

$$x_1 + 2.7x_2 = 3.4$$

- The Gomory cut would say: "Since x_1 and x_2 must be integer, the left side must be integer too"
- So we can create a cut like:

$$x_1 + 2x_2 \leq 3$$



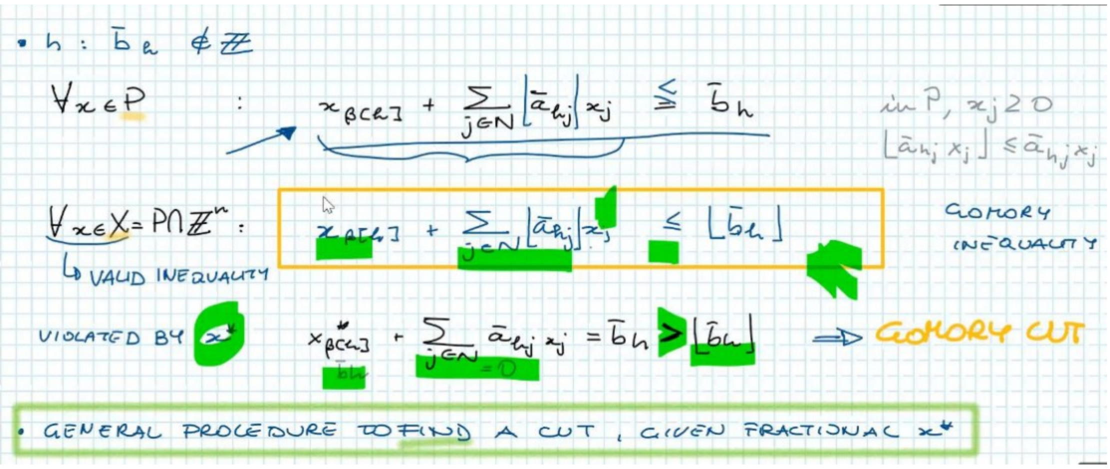
Inequality (15) is a *Gomory cut.* The above discussion shows that the Gomory cut (15) is a valid inequality for X. Moreover, we now verify that it cuts off the current optimal solution x^*:

$$x^*_{\beta[h]} + \sum_{j\in N} \lfloor \bar{a}_{hj} \rfloor x^*_j = x^*_{\beta[h]} = \bar{b}_h > \lfloor \bar{b}_h \rfloor,$$

where the inequality $\bar{b}_h > \lfloor \bar{b}_h \rfloor$ holds because $\bar{b}_h$ is not integer.

This cuts off the fractional solution but keeps all integer solutions.

- It's called a "cut" because it cuts off part of the feasible region of the LP relaxation while preserving all integer solutions – inequality is “valid”, so satisfied by any integer solution
- Think of it as adding constraints that enforce "integrality" by using the fractional parts of the current solution to generate new constraints, in a systematic way (generated by tableau)



• $h : \bar{b}_h \notin \mathbb{Z}$

$\forall x \in P$: $\underbrace{x_{\beta[h]} + \sum_{j\in N} \lfloor \bar{a}_{hj} \rfloor x_j}$ $\leq$ $\bar{b}_h$

in P, $x_j \geq 0$
$\lfloor \bar{a}_{hj} x_j \rfloor \leq \bar{a}_{hj} x_j$

$\forall x \in X = P \cap \mathbb{Z}^n$: $\boxed{x_{\beta[h]} + \sum_{j\in N} \lfloor \bar{a}_{hj} \rfloor x_j \leq \lfloor \bar{b}_h \rfloor}$

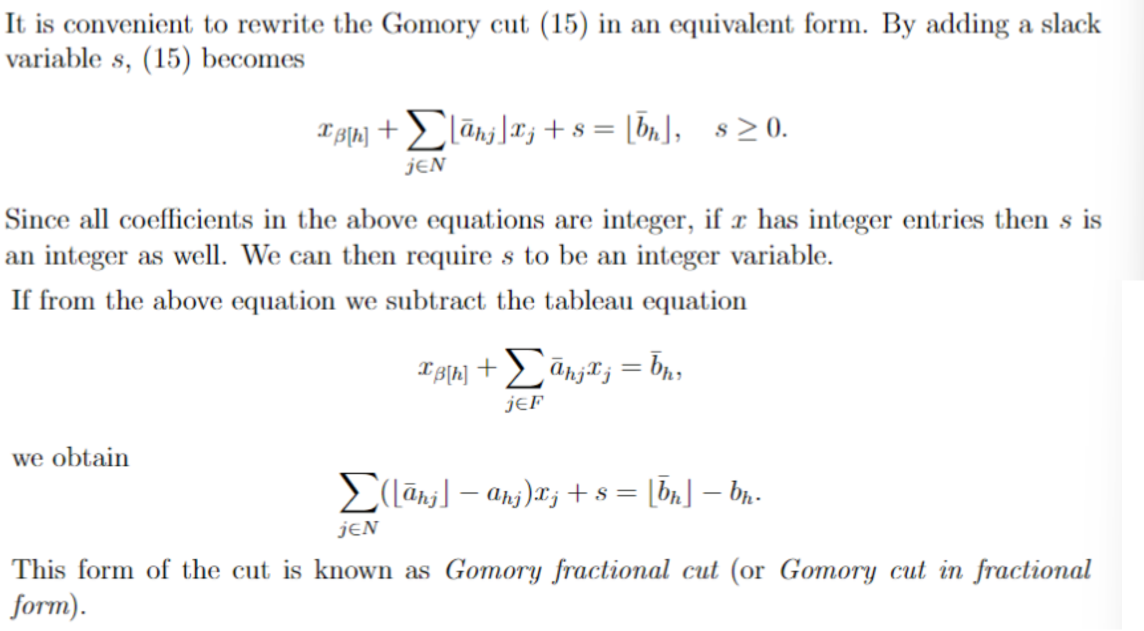
↳ VALID INEQUALITY

GOMORY INEQUALITY

VIOLATED BY x^* $\underset{\bar{b}_h}{x^*_{\beta[h]}} + \underbrace{\sum_{j\in N} \bar{a}_{hj} x_j}_{=0} = \bar{b}_h > \lfloor \bar{b}_h \rfloor$ ⟹ GOMORY CUT

• GENERAL PROCEDURE TO FIND A CUT, GIVEN FRACTIONAL x^*

There is an implementation issue to be solved to integrate Gomory cuts into the cutting plane procedure – transform this problem into an equivalent form adding a slack variable, which must be integer.



It is convenient to rewrite the Gomory cut (15) in an equivalent form. By adding a slack variable s, (15) becomes

$$x_{\beta[h]} + \sum_{j\in N} \lfloor \bar{a}_{hj} \rfloor x_j + s = \lfloor \bar{b}_h \rfloor, \quad s \geq 0.$$

Since all coefficients in the above equations are integer, if x has integer entries then s is an integer as well. We can then require s to be an integer variable.

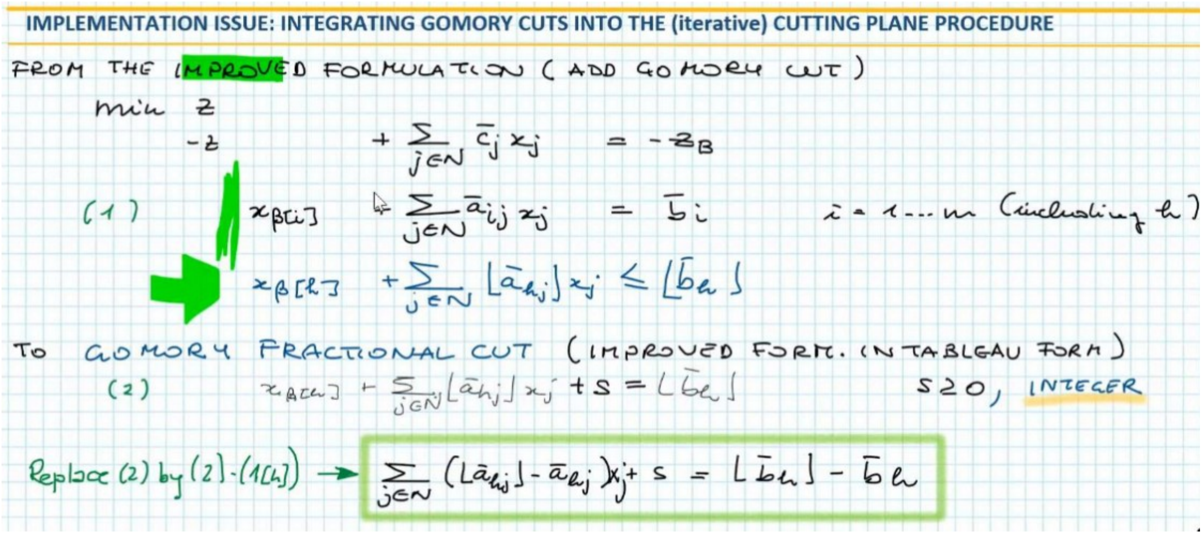
If from the above equation we subtract the tableau equation

$$x_{\beta[h]} + \sum_{j\in F} \bar{a}_{hj} x_j = \bar{b}_h,$$

we obtain

$$\sum_{j\in N} (\lfloor \bar{a}_{hj} \rfloor - a_{hj}) x_j + s = \lfloor \bar{b}_h \rfloor - b_h.$$

This form of the cut is known as *Gomory fractional cut* (or *Gomory cut in fractional form*).



IMPLEMENTATION ISSUE: INTEGRATING GOMORY CUTS INTO THE (iterative) CUTTING PLANE PROCEDURE

FROM THE IMPROVED FORMULATION (ADD GOMORY CUT)

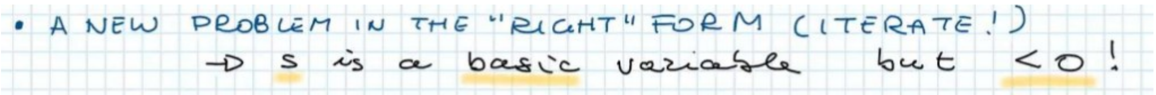
$$\begin{array}{llllll}
\min & z & & & & \\
 & -z & + \sum_{j \in N} \bar{c}_j x_j & = & -z_B & \\
(1) & x_{\beta[i]} & + \sum_{j \in N} \bar{a}_{ij} x_j & = & \bar{b}_i & i = 1 \dots m \text{ (including } h\text{)} \\
 & x_{\beta[h]} & + \sum_{j \in N} \lfloor \bar{a}_{hj} \rfloor x_j & \leq & \lfloor \bar{b}_h \rfloor &
\end{array}$$

TO GOMORY FRACTIONAL CUT (IMPROVED FORM. IN TABLEAU FORM)

$$(2) \quad x_{\beta[h]} + \sum_{j \in N} \lfloor \bar{a}_{hj} \rfloor x_j + s = \lfloor \bar{b}_h \rfloor \qquad s \geq 0, \text{ INTEGER}$$

Replace (2) by (2)-(1[h]) $\rightarrow \boxed{\sum_{j \in N} (\lfloor \bar{a}_{hj} \rfloor - \bar{a}_{hj}) x_j + s = \lfloor \bar{b}_h \rfloor - \bar{b}_h}$

We thus obtain a new problem in the right form where the variable is basic but negative, so given simplex rules, we would need to continue iterating.

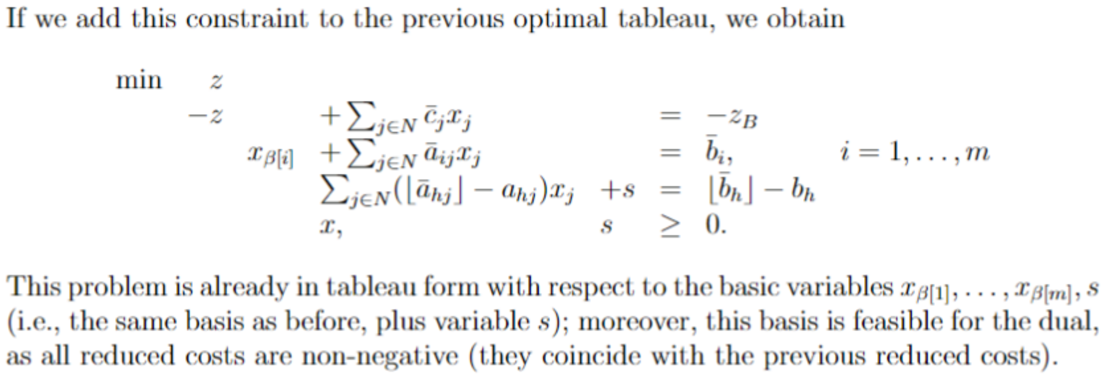


- A NEW PROBLEM IN THE "RIGHT" FORM (ITERATE!)

  → s is a basic variable but < 0 !

So: the Gomory fractional cut comes in when we try to make the original Gomory cut more computationally efficient – add a slack variable/subtract the original tableau equation and get the fractional cut.

- This is good since it's already in the right form to be added inside of the tableau and also the coefficients are small (just the fractional parts), with the slack variable leaving the basis first for the dual simplex
- This makes the cut more numerically stable and easier to work with while maintaining its cutting properties



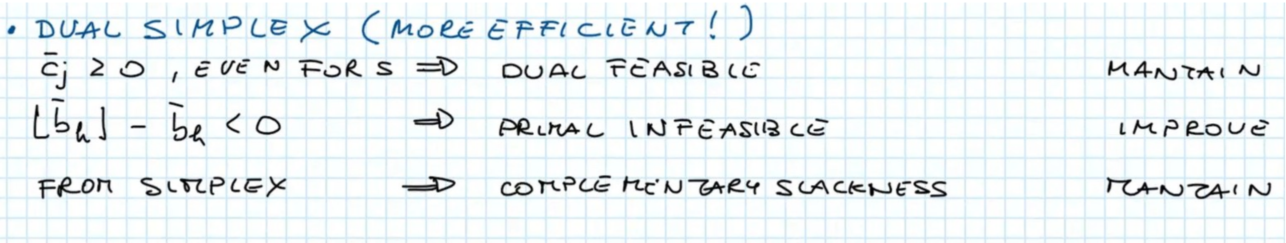
If we add this constraint to the previous optimal tableau, we obtain

$$\begin{array}{lllllll}
\min & z & & & & & \\
 & -z & + \sum_{j \in N} \bar{c}_j x_j & & = & -z_B & \\
 & x_{\beta[i]} & + \sum_{j \in N} \bar{a}_{ij} x_j & & = & \bar{b}_i, & i = 1, \dots, m \\
 & & \sum_{j \in N} (\lfloor \bar{a}_{hj} \rfloor - a_{hj}) x_j & +s & = & \lfloor \bar{b}_h \rfloor - b_h & \\
 & & x, & s & \geq & 0. &
\end{array}$$

This problem is already in tableau form with respect to the basic variables $x_{\beta[1]}, \dots, x_{\beta[m]}, s$ (i.e., the same basis as before, plus variable s); moreover, this basis is feasible for the dual, as all reduced costs are non-negative (they coincide with the previous reduced costs).

In summary – given any problem:

- We can solve it with simplex method: if feasible good, otherwise, we apply a Gomory cut (take the floor of everything and is a cut since violated by current solution)
- Iterate this method thus obtaining an integer solution



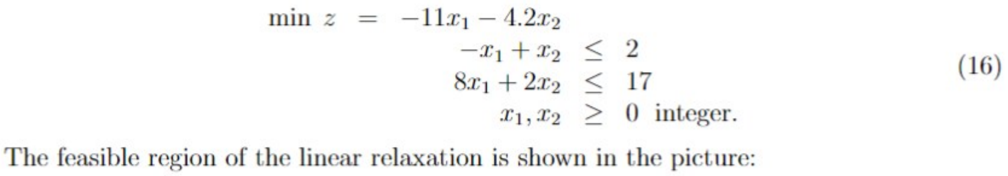
• DUAL SIMPLEX (MORE EFFICIENT!)

$\bar{c}_j \geq 0$, EVEN FOR S	⇒	DUAL FEASIBLE	MANTAIN
$\lfloor \bar{b}_h \rfloor - \bar{b}_h < 0$	⇒	PRIMAL INFEASIBLE	IMPROVE
FROM SIMPLEX	⇒	COMPLEMENTARY SLACKNESS	MANTAIN

Gomory cuts provide a systematic way to solve pure integer linear programs by iteratively solving LP relaxations with the simplex method, and whenever we get a fractional solution, we generate a valid inequality (cut) from the tableau that is guaranteed to remove this fractional solution while preserving all integer solutions; we then resolve using dual simplex (which is efficient here due to the structure of the cuts) until we eventually obtain an integer solution.

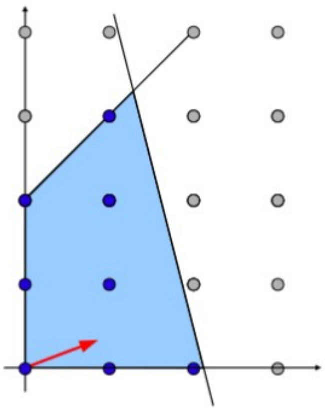
8.3.2 Complete Example

Consider the following problem/example (before you attack me since this is a "collection of images" as other ungrateful people have done overtime (a few, but present) – this was done in 5 minutes in lesson, so the best thing is to combine prof. notes in all forms – be thankful given the file completeness, instead reach me in case of feedback):

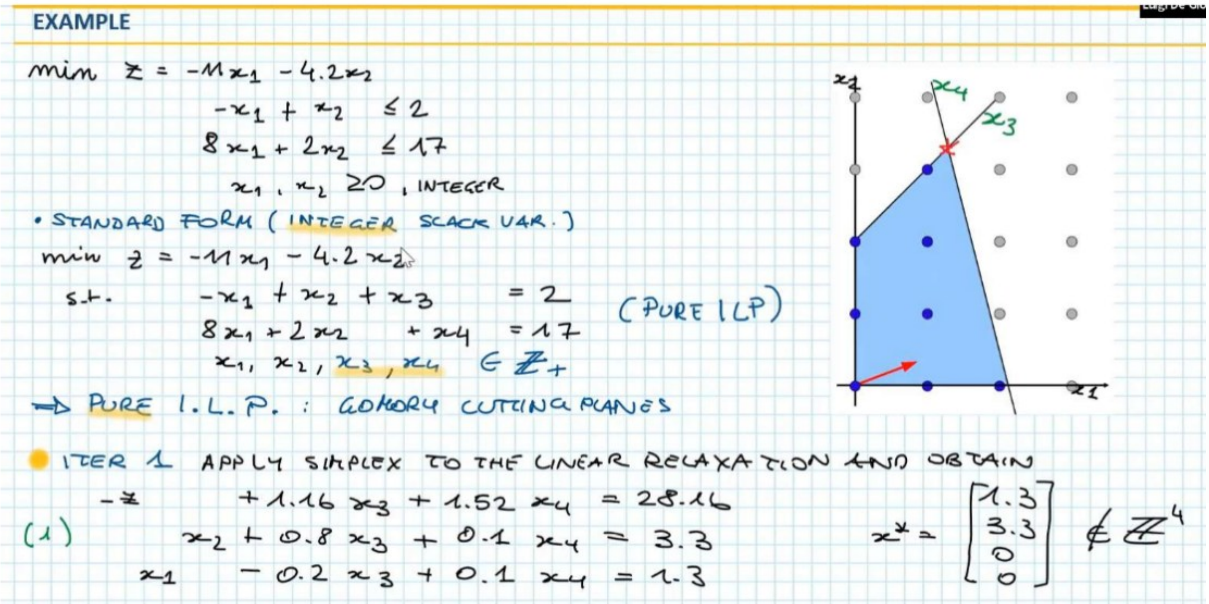


$$\begin{aligned} \min\ z &= -11x_1 - 4.2x_2 \\ -x_1 + x_2 &\leq 2 \\ 8x_1 + 2x_2 &\leq 17 \\ x_1, x_2 &\geq 0 \text{ integer.} \end{aligned} \tag{16}$$

The feasible region of the linear relaxation is shown in the picture:



Transform the problem into standard form adding slack variables and have it into pure ILP form:



EXAMPLE

$$\min \ z = -11x_1 - 4.2x_2$$
$$-x_1 + x_2 \leq 2$$
$$8x_1 + 2x_2 \leq 17$$
$$x_1, x_2 \geq 0, \text{ INTEGER}$$

• STANDARD FORM (INTEGER SLACK VAR.)

$$\min \ z = -11x_1 - 4.2x_2$$
$$\text{s.t.} \quad -x_1 + x_2 + x_3 = 2$$
$$8x_1 + 2x_2 + x_4 = 17$$
$$x_1, x_2, x_3, x_4 \in \mathbb{Z}_+$$

(PURE ILP)

⇒ PURE I.L.P. : GOMORY CUTTING PLANES

ITER 1 APPLY SIMPLEX TO THE LINEAR RELAXATION AND OBTAIN

$$\begin{array}{lllll} -z & & +1.16x_3 & +1.52x_4 & = 28.16 \\ (1) & x_2 & +0.8x_3 & +0.1x_4 & = 3.3 \\ x_1 & & -0.2x_3 & +0.1x_4 & = 1.3 \end{array}$$

$$x^* = \begin{bmatrix} 1.3 \\ 3.3 \\ 0 \\ 0 \end{bmatrix} \notin \mathbb{Z}^4$$

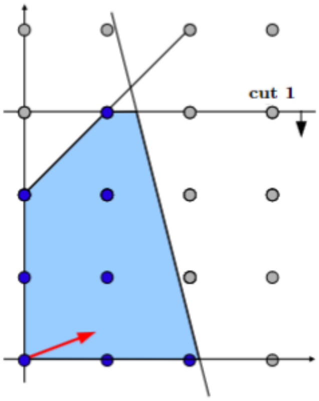
The corresponding basic solution is $x_3 = x_4 = 0$, $x_1 = 1.3$, $x_2 = 3.3$ (the upper vertex of the quadrilateral in the above picture), with objective function value $z = -28.16$. Since the values of x_1 and x_2 are not integer, this is not a feasible solution of (16). We can derive, e.g., a Gomory cut from the equation $x_2 + 0.8x_3 + 0.1x_4 = 3.3$, thus obtaining

$$x_2 \leq 3.$$

If this constraint is added to the original linear relaxation, we obtain a better formulation:

$$\begin{array}{rcl} \min z = 11x_1 + 4.2x_2 & & \\ -x_1 + x_2 & \leq & 2 \\ 8x_1 + 2x_2 & \leq & 17 \\ x_2 & \leq & 3 \\ x_1, x_2 & \geq & 0 \end{array}$$

whose feasible region is the following:

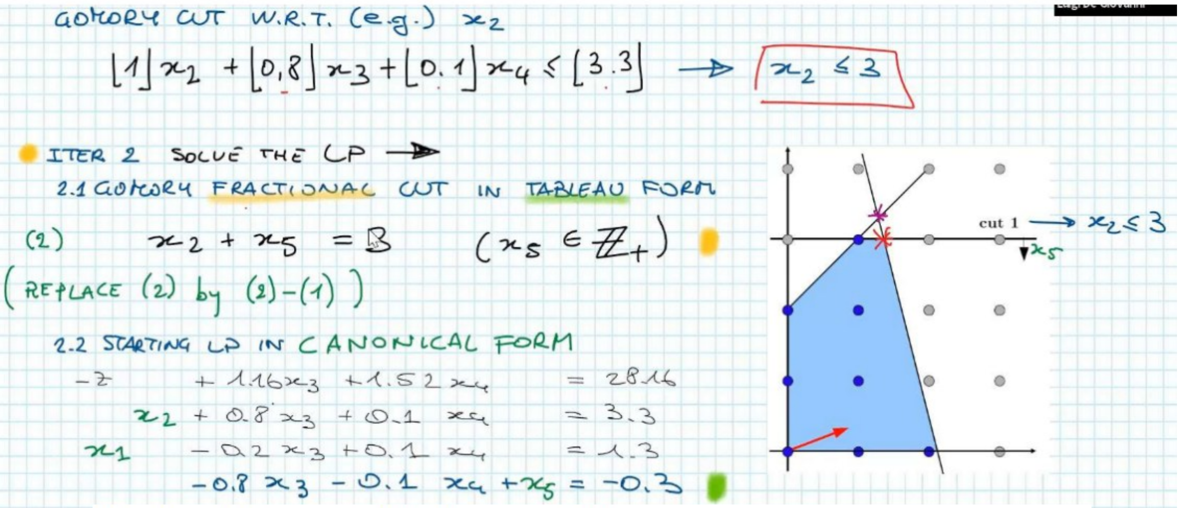


In order to solve this new problem, we first write the cut in fractional form with a slack variable x_5:

$$-0.8x_3 - 0.1x_4 + x_5 = -0.3.$$

We then add the constraint to the previous tableau:

$$\begin{array}{ccccccc} -z & & +1.16x_3 & +1.52x_4 & & = & 28.16 \\ & x_2 & +0.8x_3 & +0.1x_4 & & = & 3.3 \\ x_1 & & -0.2x_3 & +0.1x_4 & & = & 1.3 \\ & & -0.8x_3 & -0.1x_4 & +x_5 & = & -0.3 \end{array}$$



GOMORY CUT W.R.T. (e.g.) x_2

$\lfloor 1 \rfloor x_2 + \lfloor 0.8 \rfloor x_3 + \lfloor 0.1 \rfloor x_4 \leq \lfloor 3.3 \rfloor \rightarrow \boxed{x_2 \leq 3}$

ITER 2 SOLVE THE LP →

2.1 GOMORY FRACTIONAL CUT IN TABLEAU FORM

(2) $x_2 + x_5 = 3 \quad (x_5 \in \mathbb{Z}_+)$

(REPLACE (2) by (2)−(1))

2.2 STARTING LP IN CANONICAL FORM

$$\begin{array}{lllll} -z & & +1.16x_3 & +1.52x_4 & = 28.16 \\ & x_2 & +0.8x_3 & +0.1x_4 & = 3.3 \\ x_1 & & -0.2x_3 & +0.1x_4 & = 1.3 \\ & & -0.8x_3 & -0.1x_4 \quad +x_5 & = -0.3 \end{array}$$

We then add the constraint to the previous tableau:

$$\begin{array}{rrrrrcl} -z & & & +1.16x_3 & +1.52x_4 & & = & 28.16 \\ & x_2 & & +0.8x_3 & +0.1x_4 & & = & 3.3 \\ x_1 & & & -0.2x_3 & +0.1x_4 & & = & 1.3 \\ & & & -0.8x_3 & -0.1x_4 & +x_5 & = & -0.3 \end{array}$$

If the dual simplex method is applied, x_5 leaves the basis and x_3 enters, as $\min\left\{\frac{1.16}{0.8}, \frac{1.52}{0.1}\right\} = \frac{1.16}{0.8}$. After this single iteration, we obtain the following optimal tableau:

$$\begin{array}{rrrrrcl} -z & & & +1.375x_4 & +1.45x_5 & = & 27.725 \\ & x_2 & & & +x_5 & = & 3 \\ x_1 & & & +0.125x_4 & -0.25x_5 & = & 1.375 \\ & & x_3 & +0.125x_4 & -1.25x_5 & = & 0.375 \end{array}$$

The corresponding basic solution is $x_1 = 1.375$, $x_2 = 3$ $x_3 = 0.375$ (upper-right vertex in the previous picture), with objective value $z = 27.725$.

From the equation $x_3 + 0.125x_4 - 1.25x_5 = 0.375$ of the tableau, we obtain the Gomory cut

$$x_3 - 2x_5 \leq 0.$$

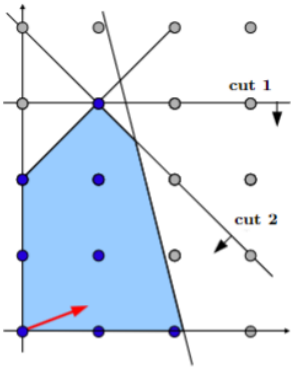
Since $x_3 = 2 + x_1 - x_2$ and $x_5 = 3 - x_2$, in the original space of variables (x_1, x_2) the above inequality can be rewritten as

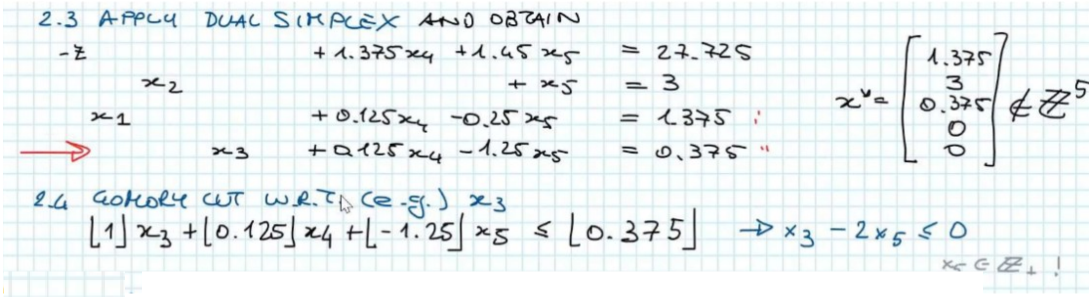
$$x_1 + x_2 \leq 4.$$

If this constraint is added to the original problem, we obtain the new linear relaxation

$$\begin{array}{rcl} \min z = 11x_1 + 4.2x_2 & & \\ -x_1 + x_2 & \leq & 2 \\ 8x_1 + 2x_2 & \leq & 17 \\ x_2 & \leq & 3 \\ x_1 + x_2 & \leq 4 & \\ x_1, x_2 & \geq & 0 \end{array}$$

shown in the picture.





2.3 APPLY DUAL SIMPLEX AND OBTAIN

$$\begin{array}{lllllll} -z & & & +1.375x_4 & +1.45x_5 & = & 27.725 \\ & x_2 & & & +x_5 & = & 3 \\ x_1 & & & +0.125x_4 & -0.25x_5 & = & 1.375 \\ & & x_3 & +0.125x_4 & -1.25x_5 & = & 0.375 \end{array} \qquad x^* = \begin{bmatrix} 1.375 \\ 3 \\ 0.375 \\ 0 \\ 0 \end{bmatrix} \notin \mathbb{Z}^5$$

2.4 GOMORY CUT W.R.T. (e.g.) x_3

$$\lfloor 1 \rfloor x_3 + \lfloor 0.125 \rfloor x_4 + \lfloor -1.25 \rfloor x_5 \leq \lfloor 0.375 \rfloor \rightarrow x_3 - 2x_5 \leq 0$$

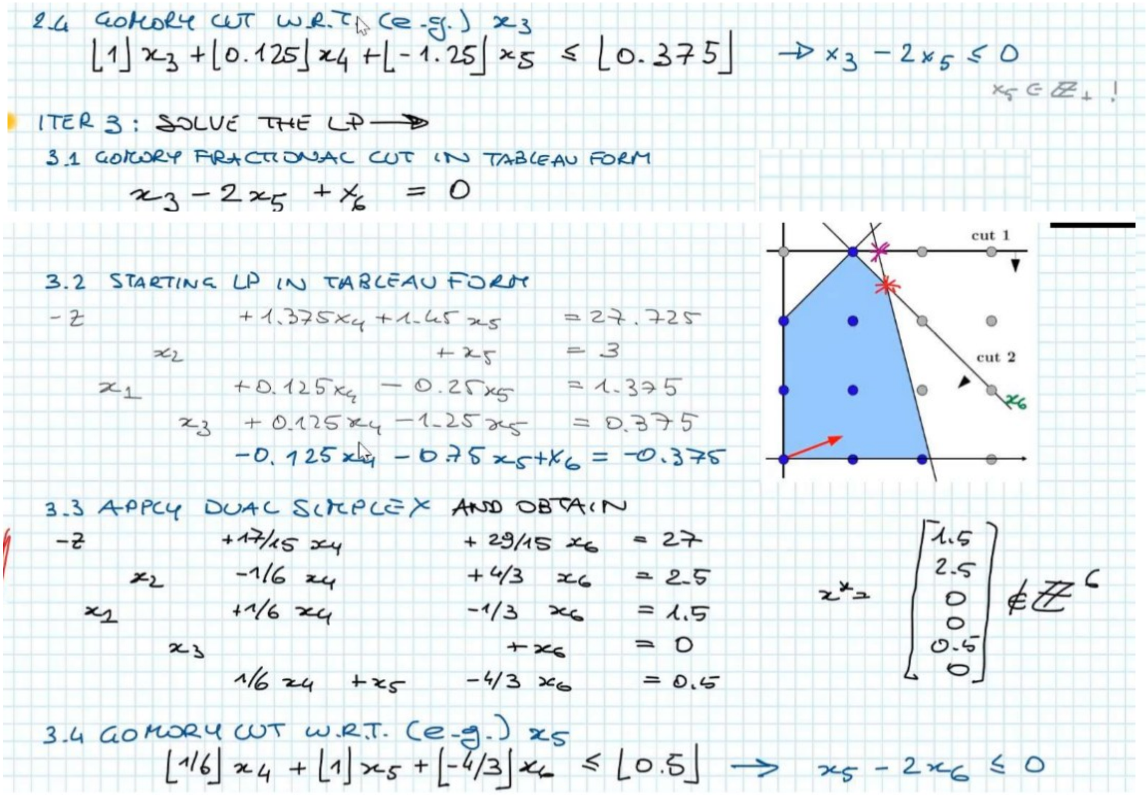
$x_5 \in \mathbb{Z}_+$!

Written in fractional form, the cut is

$$-0.125x_4 - 0.75x_5 + x_6 = -0.375.$$

If we add this constraint to the tableau, we obtain

$$\begin{array}{cccccccc} -z & & & & +1.375x_4 & +1.45x_5 & & = & 27.725 \\ & & x_2 & & & +x_5 & & = & 3 \\ & x_1 & & & +0.125x_4 & -0.25x_5 & & = & 1.375 \\ & & & x_3 & +0.125x_4 & -1.25x_5 & & = & 0.375 \\ & & & & -0.125x_4 & -0.75x_5 & +x_6 & = & -0.375 \end{array}$$



2.4 GOMORY CUT W.R.T. (e.g.) x_3

$$\lfloor 1 \rfloor x_3 + \lfloor 0.125 \rfloor x_4 + \lfloor -1.25 \rfloor x_5 \leq \lfloor 0.375 \rfloor \rightarrow x_3 - 2x_5 \leq 0$$

$x_5 \in \mathbb{Z}_+$!

ITER 3: SOLVE THE LP →

3.1 GOMORY FRACTIONAL CUT IN TABLEAU FORM

$$x_3 - 2x_5 + x_6 = 0$$

3.2 STARTING LP IN TABLEAU FORM

$$\begin{array}{llllllll} -z & & & +1.375x_4 & +1.45x_5 & & = & 27.725 \\ & x_2 & & & +x_5 & & = & 3 \\ x_1 & & & +0.125x_4 & -0.25x_5 & & = & 1.375 \\ & & x_3 & +0.125x_4 & -1.25x_5 & & = & 0.375 \\ & & & -0.125x_4 & -0.75x_5 & +x_6 & = & -0.375 \end{array}$$

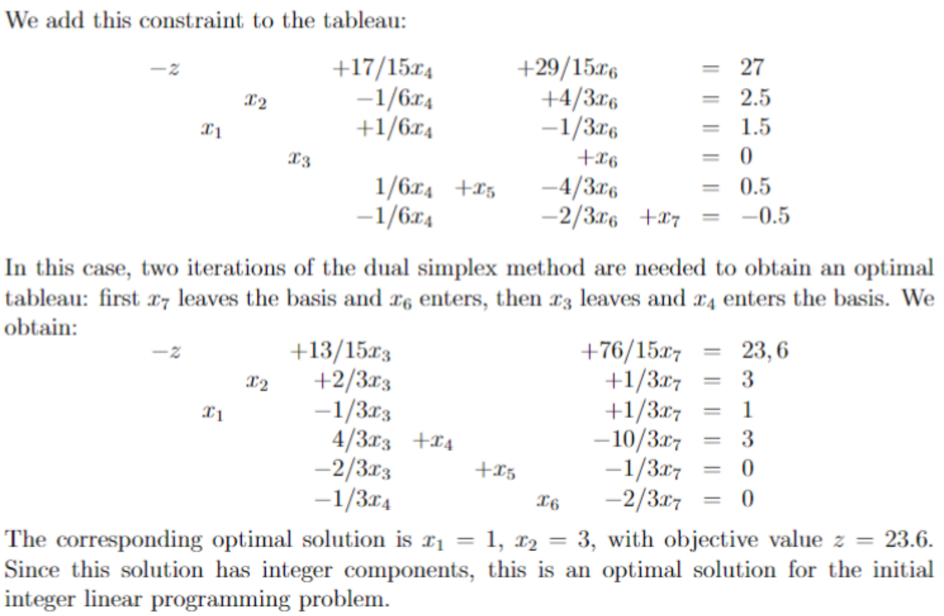
cut 1
cut 2
x_6

3.3 APPLY DUAL SIMPLEX AND OBTAIN

$$\begin{array}{llllllll} -z & & & +17/15\ x_4 & & +29/15\ x_6 & = & 27 \\ & x_2 & & -1/6\ x_4 & & +4/3\ x_6 & = & 2.5 \\ x_1 & & & +1/6\ x_4 & & -1/3\ x_6 & = & 1.5 \\ & & x_3 & & & +x_6 & = & 0 \\ & & & 1/6\ x_4 & +x_5 & -4/3\ x_6 & = & 0.5 \end{array} \qquad x^* = \begin{bmatrix} 1.5 \\ 2.5 \\ 0 \\ 0 \\ 0.5 \\ 0 \end{bmatrix} \notin \mathbb{Z}^6$$

3.4 GOMORY CUT W.R.T. (e.g.) x_5

$$\lfloor 1/6 \rfloor x_4 + \lfloor 1 \rfloor x_5 + \lfloor -4/3 \rfloor x_6 \leq \lfloor 0.5 \rfloor \rightarrow x_5 - 2x_6 \leq 0$$



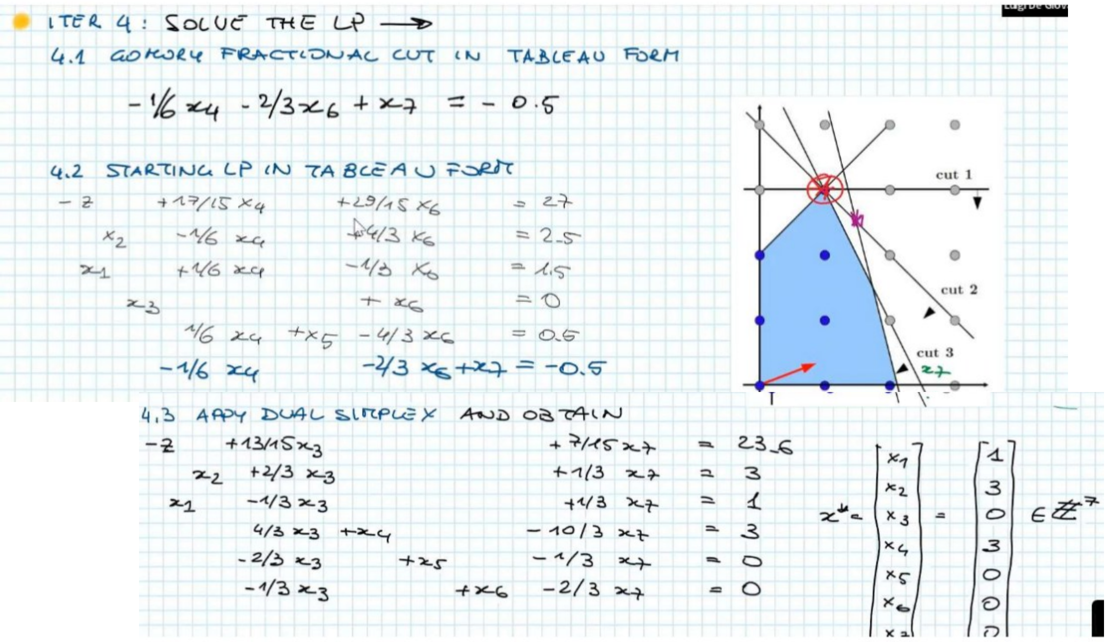
We add this constraint to the tableau:

$$\begin{array}{rrrrrrrcl}
-z & & & +17/15x_4 & & +29/15x_6 & & = & 27 \\
& x_2 & & -1/6x_4 & & +4/3x_6 & & = & 2.5 \\
x_1 & & & +1/6x_4 & & -1/3x_6 & & = & 1.5 \\
& & x_3 & & & +x_6 & & = & 0 \\
& & & 1/6x_4 & +x_5 & -4/3x_6 & & = & 0.5 \\
& & & -1/6x_4 & & -2/3x_6 & +x_7 & = & -0.5
\end{array}$$

In this case, two iterations of the dual simplex method are needed to obtain an optimal tableau: first x_7 leaves the basis and x_6 enters, then x_3 leaves and x_4 enters the basis. We obtain:

$$\begin{array}{rrrrrrrcl}
-z & & +13/15x_3 & & & & +76/15x_7 & = & 23,6 \\
& x_2 & +2/3x_3 & & & & +1/3x_7 & = & 3 \\
x_1 & & -1/3x_3 & & & & +1/3x_7 & = & 1 \\
& & 4/3x_3 & +x_4 & & & -10/3x_7 & = & 3 \\
& & -2/3x_3 & & +x_5 & & -1/3x_7 & = & 0 \\
& & -1/3x_4 & & & x_6 & -2/3x_7 & = & 0
\end{array}$$

The corresponding optimal solution is $x_1 = 1$, $x_2 = 3$, with objective value $z = 23.6$. Since this solution has integer components, this is an optimal solution for the initial integer linear programming problem.



ITER 4 : SOLVE THE LP ⟶

4.1 GOMORY FRACTIONAL CUT IN TABLEAU FORM

$$-1/6\,x_4 - 2/3\,x_6 + x_7 = -0.5$$

4.2 STARTING LP IN TABLEAU FORM

$$\begin{array}{rrrrrrrcl}
-z & & & +17/15\,x_4 & & +29/15\,x_6 & & = & 27 \\
& x_2 & & -1/6\,x_4 & & +4/3\,x_6 & & = & 2.5 \\
x_1 & & & +1/6\,x_4 & & -1/3\,x_6 & & = & 1.5 \\
& & x_3 & & & +x_6 & & = & 0 \\
& & & 1/6\,x_4 & +x_5 & -4/3\,x_6 & & = & 0.5 \\
& & & -1/6\,x_4 & & -2/3\,x_6 & +x_7 & = & -0.5
\end{array}$$

4.3 APPLY DUAL SIMPLEX AND OBTAIN

$$\begin{array}{rrrrrrrcl}
-z & & +13/15x_3 & & & & +7/15\,x_7 & = & 23.6 \\
& x_2 & +2/3\,x_3 & & & & +1/3\,x_7 & = & 3 \\
x_1 & & -1/3\,x_3 & & & & +1/3\,x_7 & = & 1 \\
& & 4/3\,x_3 & +x_4 & & & -10/3\,x_7 & = & 3 \\
& & -2/3\,x_3 & & +x_5 & & -1/3\,x_7 & = & 0 \\
& & -1/3\,x_3 & & & +x_6 & -2/3\,x_7 & = & 0
\end{array}$$

$$x^* = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 0 \\ 3 \\ 0 \\ 0 \\ 0 \end{bmatrix} \in \mathbb{Z}^7$$

Let me break down this example step by step:

1. Initial Problem:

min z = $-11x_1 - 4.2x_2$

$-x_1 + x_2 \leq 2$

$8x_1 + 2x_2 \leq 17$

$x_1, x_2 \geq 0$ integer

2. First Step - Standard Form:

- Add slack variables x_3 and x_4
- Convert to equalities
- Get initial tableau through simplex method
- Find optimal solution: $x_1 = 1.3$, $x_2 = 3.3$ (NON-INTEGER!)

3. First Gomory Cut:

- From equation $x_2 + 0.8x_3 + 0.1x_4 = 3.3$
- Generate cut $x_2 \leq 3$
- Feasible region gets smaller (see first cut in diagram)

4. Add Cut in Fractional Form:

- $0.8x_3 - 0.1x_4 + x_5 = -0.3$
- Add to tableau
- Solve with dual simplex (x_5 leaves, x_3 enters)
- Get new solution: $x_1 = 1.375$, $x_2 = 3$, $x_3 = 0.375$ (still fractional!)

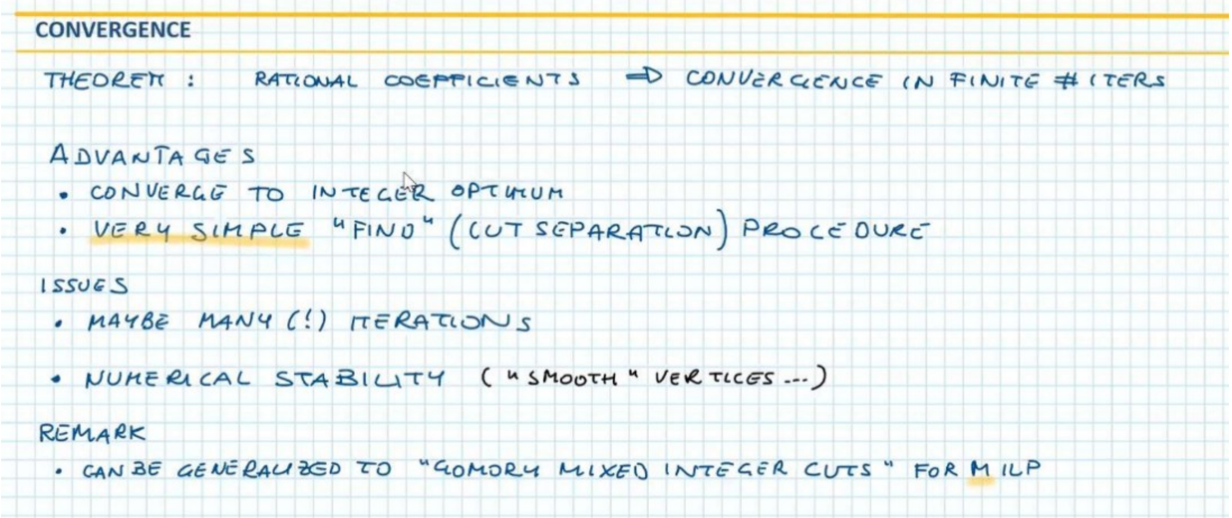
5. Second Gomory Cut:

- From $x_3 + 0.125x_4 - 1.25x_5 = 0.375$
- Get $x_3 - 2x_5 \leq 0$
- Translates to $x_1 + x_2 \leq 4$ in original variables
- Add to tableau in fractional form

6. Final Steps:

- Continue process
- Eventually reach integer solution $x_1 = 1$, $x_2 = 3$
- This is optimal for original problem with $z = 23.6$

The example shows how each Gomory cut progressively restricts the feasible region until we reach an integer solution, while the dual simplex method efficiently handles the new cuts at each iteration.



CONVERGENCE

THEOREM : RATIONAL COEFFICIENTS ⇒ CONVERGENCE IN FINITE # ITERS

ADVANTAGES

- CONVERGE TO INTEGER OPTIMUM
- VERY SIMPLE "FIND" (CUT SEPARATION) PROCEDURE

ISSUES

- MAYBE MANY (!) ITERATIONS
- NUMERICAL STABILITY ("SMOOTH" VERTICES ...)

REMARK

- CAN BE GENERALIZED TO "GOMORY MIXED INTEGER CUTS" FOR MILP

In the end: using rational coefficients leads to convergence in finite iterations. This is demonstrated in the example where each cut (x2 ≤ 3, x1 + x2 ≤ 4, 2x1 + x2 ≤ 5) progressively tightens the feasible region until reaching the integer optimum (x1 = 1, x2 = 3).

Advantages:

1. Converge to integer optimum - The example clearly shows this, converging to integer values after 3 cuts
2. Quite simple 'find' (cut separation) procedure - Each cut is mechanically derived from fractional values in the tableau

Issues:

1. Maybe many (!!) iterations – While this example took only 3 cuts, larger problems could require many more
2. Numerical stability ('smooth' vertices) – The example shows increasingly complex fractions in the tableaus (like 17/15, 29/15), highlighting potential numerical issues

Remark:

- Can be generalized to “Gomory mixed integer cuts” for MILP

Side note: the other notes for this course present the *branch-and-cut*, which essentially merges branch and bound and cutting planes where:

- You start with continuous relaxation and add cutting planes
- You apply branch-and-bound
- You can add more cuts at each node of the branch-and-bound tree

The idea is that you can combine both approaches in various ways:

- Adding cuts only at the beginning
- Adding cuts before branching
- Adding cuts throughout the branch-and-bound process

9 COVER INEQUALITIES (8)

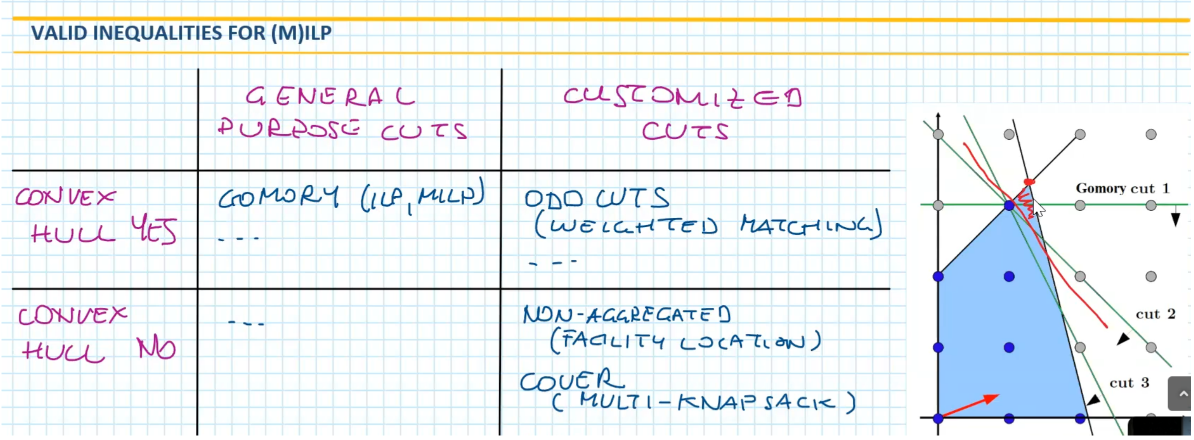
Understanding how cutting planes strengthen formulations is central to modern optimization. When we work with integer programming problems, we're often dealing with their linear relaxations – continuous approximations of discrete feasible regions. The quality of these relaxations directly impacts solution efficiency. This is important since:

- The convex hull of integer solutions represents our ideal: it's the tightest possible continuous relaxation we could hope for. Any valid integer solution lies within this hull, and its boundaries precisely define the space of feasible solutions. However, explicitly describing the convex hull is usually impractical due to the potentially enormous number of constraints needed

- Cuts that belong to the convex hull, so to help describe facets of the convex hull of integer solutions. When added to the linear relaxation, these cuts help make the relaxation tighter and closer to the integer polytope.

- Cuts not belonging to the convex hull may still be valid and useful but don't necessarily correspond to facets of the convex hull. These include specialized cuts depending on the problem nature, called customized cuts:
  - Non-aggregated cuts
  - Cover inequalities (for knapsack problems)
  - Facility location specific cuts

So basically: what happens when we decide to stop after a certain number of iterations or X cuts? Maybe we will find better formulation, from where to start the better formulation/resolution. This is what the solvers do. So: problems might have different constraints.



VALID INEQUALITIES FOR (M)ILP

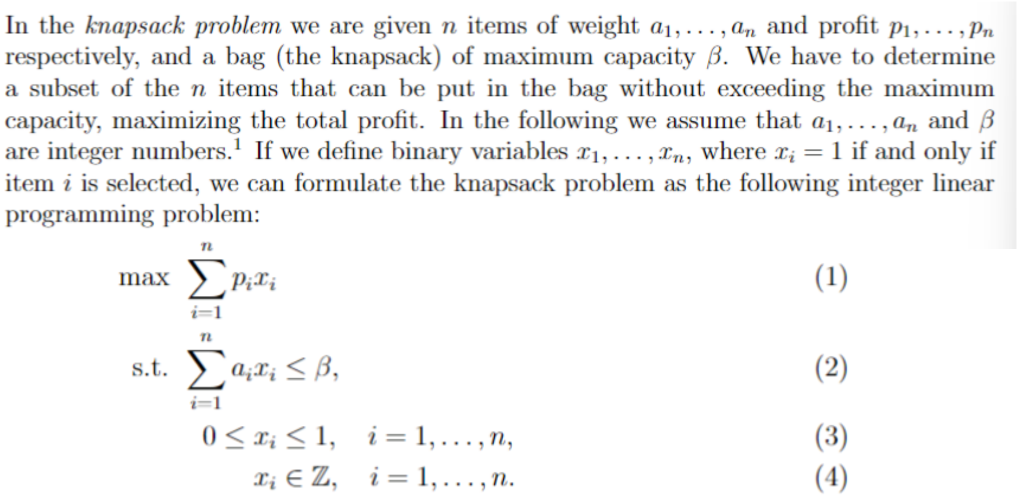
	GENERAL PURPOSE CUTS	CUSTOMIZED CUTS
CONVEX HULL YES	GOMORY (ILP, MILP) ...	ODD CUTS (WEIGHTED MATCHING) ...
CONVEX HULL NO	...	NON-AGGREGATED (FACILITY LOCATION) COVER (MULTI-KNAPSACK)

Can we put together these two things in a convenient way? Yes. Modern solvers don't attempt to add all possible cuts – doing so would be computationally prohibitive. Instead, they employ sophisticated strategies to:

- Identify the most violated inequalities
- Select cuts that provide the greatest improvement in bound quality
- Balance the strengthening effect against the computational overhead
- Recognize when additional cuts yield diminishing returns

9.1 COVER INEQUALITIES FOR THE KNAPSACK PROBLEM

Recall the knapsack problem:



In the *knapsack problem* we are given n items of weight $a_1, \dots, a_n$ and profit $p_1, \dots, p_n$ respectively, and a bag (the knapsack) of maximum capacity β. We have to determine a subset of the n items that can be put in the bag without exceeding the maximum capacity, maximizing the total profit. In the following we assume that $a_1, \dots, a_n$ and β are integer numbers.[1] If we define binary variables $x_1, \dots, x_n$, where $x_i = 1$ if and only if item i is selected, we can formulate the knapsack problem as the following integer linear programming problem:

$$\max \sum_{i=1}^{n} p_i x_i \tag{1}$$

$$\text{s.t. } \sum_{i=1}^{n} a_i x_i \le \beta, \tag{2}$$

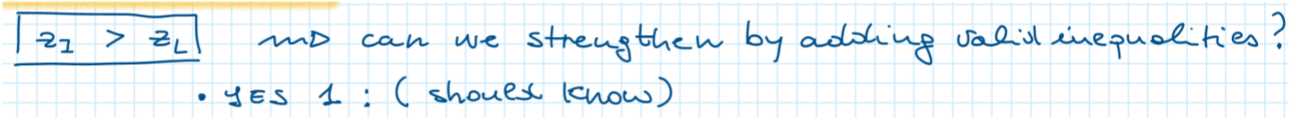
$$0 \le x_i \le 1, \quad i = 1, \dots, n, \tag{3}$$

$$x_i \in \mathbb{Z}, \quad i = 1, \dots, n. \tag{4}$$

The feasible region of the continuous relaxation, given by constraints (2)–(3), is usually much larger than the ideal formulation. It is then reasonable to introduce inequalities to strengthen the formulation: these inequalities must be satisfied by all the integer solutions but should hopefully yield a continuous relaxation closer to the ideal formulation.

The formulation is not ideal, however, since we get non-integer solutions out of this. We may decide to run, for example, the Branch-and-Bound method directly or improve the formulation if possible.

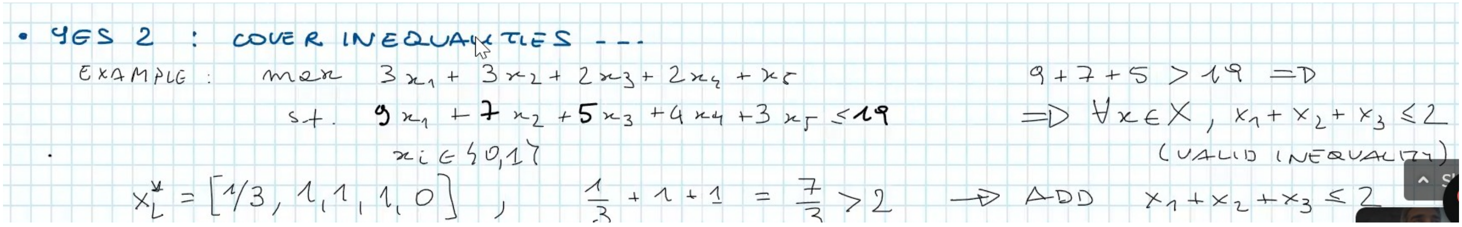
Assume we can apply cutting plane methods (aka Gomory cuts application); can we find a deeper cut (based on deeper reasoning of the problem).



$\boxed{z_I > z_L}$ $\rightsquigarrow$ can we strengthen by adding valid inequalities?

- YES 1 : (should know)

Any possible integer solution in the knapsack problem satisfies the example inequality; to cut it away, we must apply constraints over the sum, using better formulation principles (= cut away fractional value). Of course, value (=meaning we cut away more) depends on the problem reasoning. Here we show the formulation is not integer.



- YES 2 : COVER INEQUALITIES - - -

EXAMPLE : max $3x_1 + 3x_2 + 2x_3 + 2x_4 + x_5$

s.t. $9x_1 + 7x_2 + 5x_3 + 4x_4 + 3x_5 \le 19$

$x_i \in \{0,1\}$

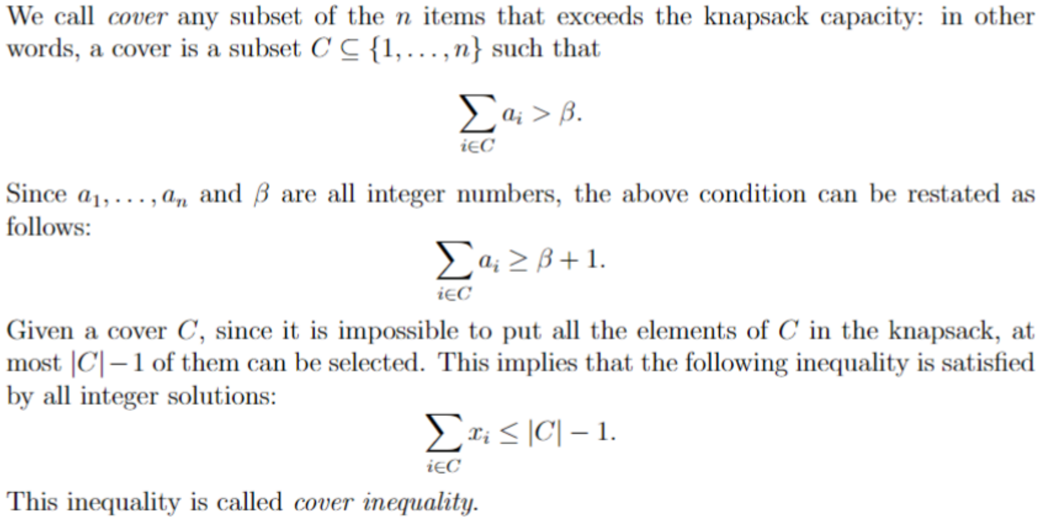
$9 + 7 + 5 > 19 \Rightarrow$

$\Rightarrow \forall x \in X,\ x_1 + x_2 + x_3 \le 2$

(VALID INEQUALITY)

$x_L^* = [1/3, 1, 1, 1, 0]$, $\frac{1}{3} + 1 + 1 = \frac{7}{3} > 2 \rightarrow$ ADD $x_1 + x_2 + x_3 \le 2$

So, this ideas can be applied this way, defining the concepts of cover and cover inequality:



We call *cover* any subset of the n items that exceeds the knapsack capacity: in other words, a cover is a subset $C \subseteq \{1, \dots, n\}$ such that

$$\sum_{i \in C} a_i > \beta.$$

Since $a_1, \dots, a_n$ and β are all integer numbers, the above condition can be restated as follows:

$$\sum_{i \in C} a_i \geq \beta + 1.$$

Given a cover C, since it is impossible to put all the elements of C in the knapsack, at most $|C| - 1$ of them can be selected. This implies that the following inequality is satisfied by all integer solutions:

$$\sum_{i \in C} x_i \leq |C| - 1.$$

This inequality is called *cover inequality*.

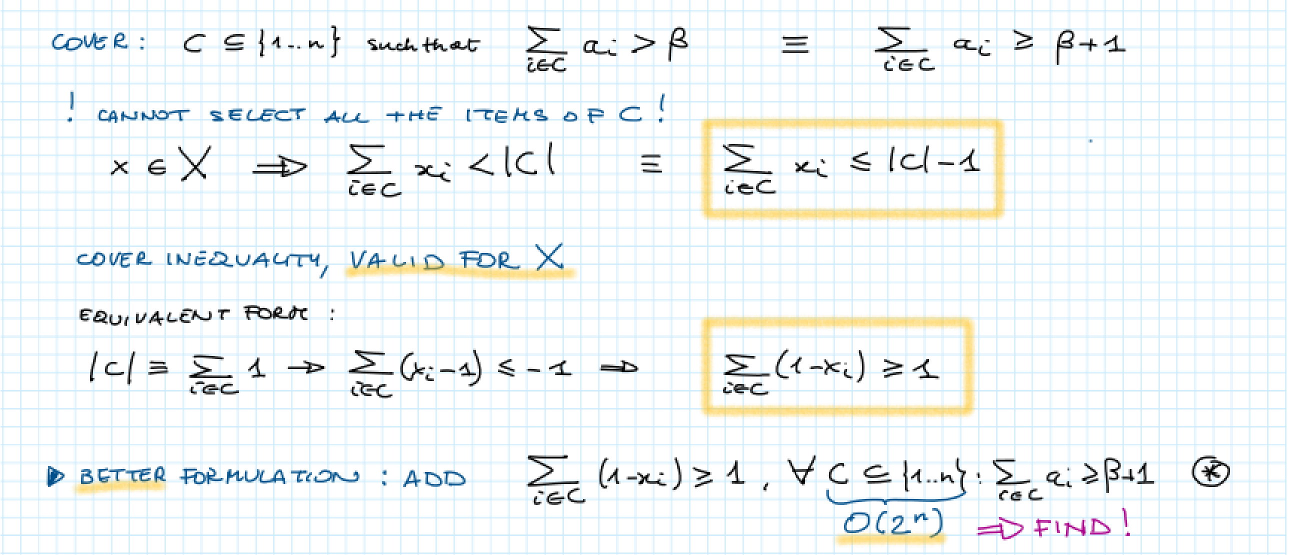
Cover inequalities are a subset of cutting planes tailored specifically for binary ILP problems.

The cover inequality is valid for X. Equivalently, given the cardinality of the variables ($|C|$), the inequality is equivalent to:

$$\sum_{i \in C} (1 - x_i) \geq 1.$$

If all possible cover inequalities are added to the original formulation of the knapsack problem, we obtain better formulation.

- However, it can be shown that this is not the ideal formulation yet. This is a better formulation for sure, but definitely we might not get the convex hull



COVER : $C \subseteq \{1..n\}$ such that $\sum_{i \in C} a_i > \beta \equiv \sum_{i \in C} a_i \geq \beta + 1$

! CANNOT SELECT ALL THE ITEMS OF C !

$$x \in X \Rightarrow \sum_{i \in C} x_i < |C| \equiv \boxed{\sum_{i \in C} x_i \leq |C| - 1}$$

COVER INEQUALITY, VALID FOR X

EQUIVALENT FORM :

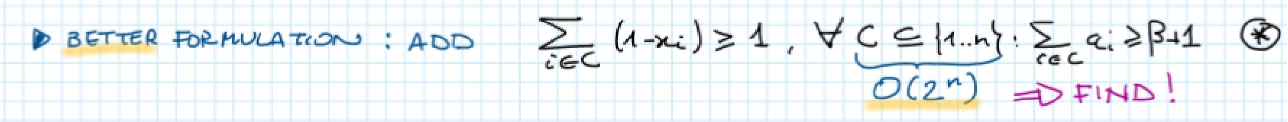
$$|C| = \sum_{i \in C} 1 \rightarrow \sum_{i \in C} (x_i - 1) \leq -1 \Rightarrow \boxed{\sum_{i \in C} (1 - x_i) \geq 1}$$

▶ BETTER FORMULATION : ADD $\sum_{i \in C} (1 - x_i) \geq 1, \ \forall \underbrace{C \subseteq \{1..n\}}_{O(2^n)} : \sum_{i \in C} a_i \geq \beta + 1$ ⊛

⇒ FIND !

Unfortunately, the number of cover inequalities is exponentially large, as in the worst case there is one cover inequality for every non-empty subset C of $\{1,\dots,n\}$

- Even though in most practical cases only some of the subsets of $\{1,\dots,n\}$ are actually covers, the number of covers is too large. The idea is then to add dynamically the cover inequalities only when they are really needed, adding to the original formulation



▶ BETTER FORMULATION : ADD $\sum_{i\in C}(1-x_i)\geq 1 \,,\ \forall \underbrace{C\subseteq\{1..n\}}_{O(2^n)} : \sum_{i\in C} a_i \geq \beta+1$ ⊛

⇒ FIND !

Question: Is it worth applying to cover inequalities?

- Yes, we solve the linear relaxation and find one of the cover inequalities
- Even adding all of the cover inequalities, we have no guarantee to be in the convex hull (with Gomory cuts we have this guarantee)
- With Gomory cuts, thanks to the rounding down, separation procedure is trivial
- Here, we have to formalize it, forming an optimization problem

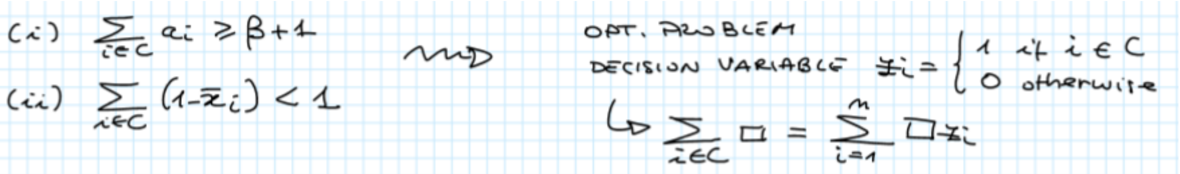
9.2 Separation Procedure

These inequalities are much faster than usual inequalities; if we were able to find them, we can add them instead of Gomory cuts. Violated inequalities would take much more time (aka – exponential). That's why we would need to add the separation procedure. Can we find this for odd cut inequalities and find the most violated inequality?

To do so:

- Suppose that we have solved the continuous relaxation (1)–(3) (for instance with the simplex method), thus obtaining a solution $\bar{x}$ that is not an integer vector
- We want to verify if there is a cover inequality violated by $\bar{x}$, with the purpose of adding this inequality to the formulation and solving the new linear programming problem
- The problem of finding a cover inequality violated by $\bar{x}$, or deciding that none exists, is the separation problem for the cover inequalities. Note that since the number of cover inequalities is exponentially large, we cannot simply enumerate them and look for a violated one

Deciding if there is a cover inequality violated by $\bar{x}$ means deciding if there exists a subset $C \subseteq \{1,\dots,n\}$ such that:



(i) $\sum_{i\in C} a_i \geq \beta+1$

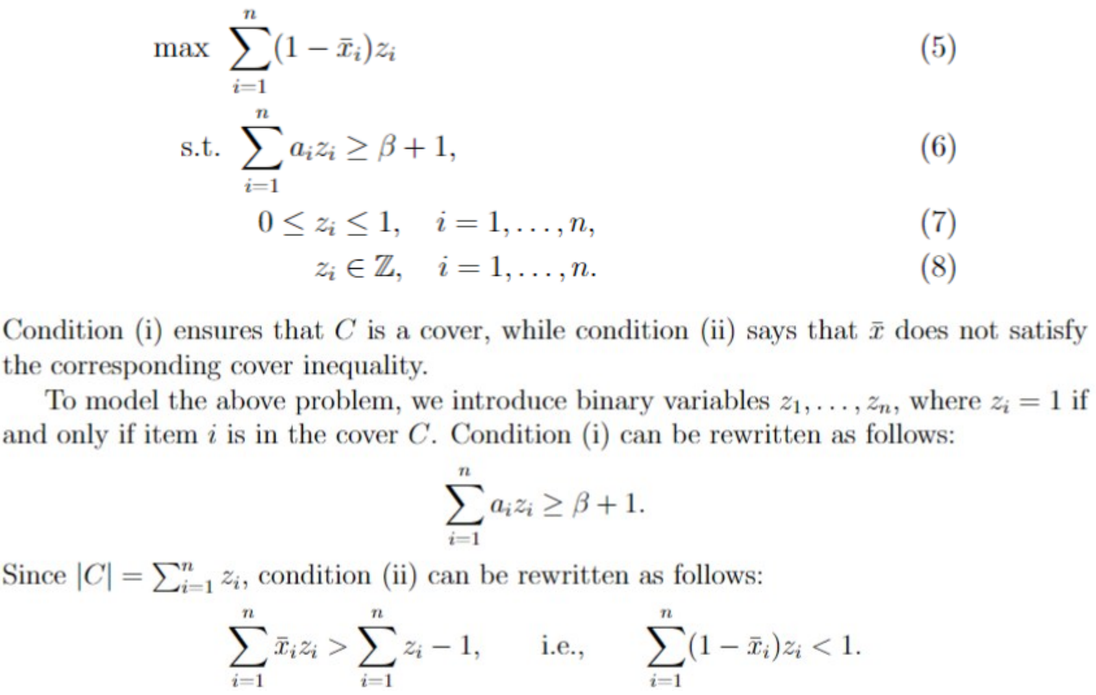
(ii) $\sum_{i\in C} (1-\bar{x}_i) < 1$

⇝ OPT. PROBLEM

DECISION VARIABLE $z_i = \begin{cases} 1 & \text{if } i\in C \\ 0 & \text{otherwise} \end{cases}$

↳ $\sum_{i\in C} \square = \sum_{i=1}^{n} \square z_i$

The decision variable is the subset, which is binary since the item is present in the knapsack we have 1, otherwise 0. The set of items to be selected must be a cover (capacity of knapsack plus 1):



$$\max \sum_{i=1}^{n} (1 - \bar{x}_i) z_i \tag{5}$$

$$\text{s.t.} \sum_{i=1}^{n} a_i z_i \geq \beta + 1, \tag{6}$$

$$0 \leq z_i \leq 1, \quad i = 1, \ldots, n, \tag{7}$$

$$z_i \in \mathbb{Z}, \quad i = 1, \ldots, n. \tag{8}$$

Condition (i) ensures that C is a cover, while condition (ii) says that $\bar{x}$ does not satisfy the corresponding cover inequality.

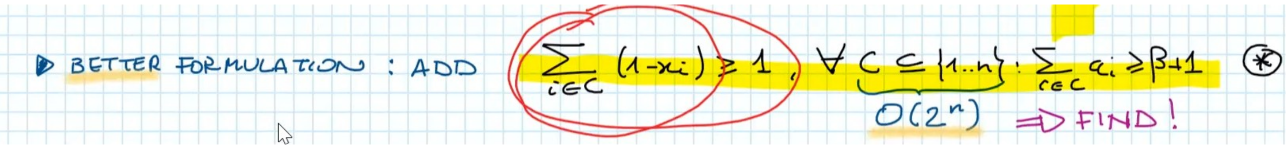
To model the above problem, we introduce binary variables $z_1, \ldots, z_n$, where $z_i = 1$ if and only if item i is in the cover C. Condition (i) can be rewritten as follows:

$$\sum_{i=1}^{n} a_i z_i \geq \beta + 1.$$

Since $|C| = \sum_{i=1}^{n} z_i$, condition (ii) can be rewritten as follows:

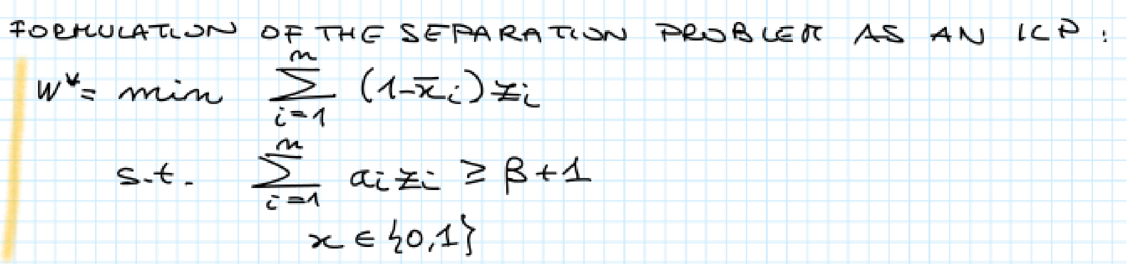
$$\sum_{i=1}^{n} \bar{x}_i z_i > \sum_{i=1}^{n} z_i - 1, \qquad \text{i.e.,} \qquad \sum_{i=1}^{n} (1 - \bar{x}_i) z_i < 1.$$

We have the subset to be a cover and more over the corresponding constraint has to be violated. This means the following sum has to be as small as possible, possibly less than 1:



▶ BETTER FORMULATION : ADD $\sum_{i \in C} (1-x_i) \geq 1$, $\forall \underbrace{C \subseteq \{1..n\}}_{O(2^n)} : \sum_{c \in C} a_i \geq \beta + 1$ ⊛ ⇒ FIND !

The constraints ensure that the z_i variables define a cover. Given the optimal integer solution $\overline{z}$ of the problem, is the optimal value is smaller than 1, then the cover inequality defined by the optimal solution is violated by $\overline{x}$. Otherwise, if the optimal value is ≥ 1, then there is no cover inequality violated by $\overline{x}$.



FORMULATION OF THE SEPARATION PROBLEM AS AN ICP :

$$W^* = \min \sum_{i=1}^{m} (1-\overline{x}_i) z_i$$

$$\text{s.t.} \sum_{i=1}^{m} a_i z_i \geq \beta + 1$$

$$x \in \{0,1\}$$

We can then decide if there is a cover inequality violated by $\overline{x}$ (and find it) by solving the integer linear programming problem (5)–(8). Note that this problem is very similar to the original knapsack problem (1)–(4), and therefore it would probably be better to solve directly the original problem rather than solving a similar problem just to find a new inequality to include in the formulation (and then iterate this procedure!). Nonetheless, we will see below that this approach is much more promising in more general situations.

If the equality is greater than 1, no violated covers are present. Find the inequality that violates the most the cover.

- IF $w^* < 1$, ADD COVER INEQUALITY FOR $C = \{i = 1..n \mid z_i^* = 1\}$
  OTHERWISE NO VIOLATED COVER INEQUALITY EXISTS

Let’s integrate such separation procedure by decomposition by integrating it into a cutting plane procedure:

MASTER : SOLVE THE LINEAR RELAXATION

SLAVE : SOLVE THE SEPARATION PROBLEM

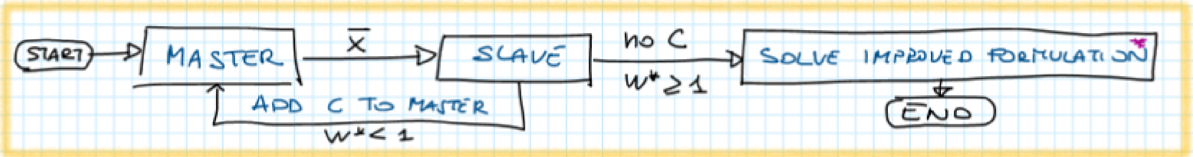
TRANSFORM INTO A KP-0/1 ! $z_i = 1 - y_i$

- $\min \sum_{i=1}^{m} (1-\bar{x}_i) z_i \sim \max - \sum_{i=1}^{m} (1-\bar{x}_i)(1-y_i) =$

$$= - \sum_{i=1}^{m} (1-\bar{x}_i) + (1-\bar{x}_i) y_i \sim \max \sum_{i=1}^{m} (1-\bar{x}_i) y_i$$

- $\sum_{i=1}^{m} a_i z_i = \sum_{i=1}^{m} a_i (1-y_i) = \sum_{i=1}^{m} a_i - \sum_{i=1}^{m} a_i y_i \geq \beta + 1 \Rightarrow$

$$\Rightarrow \sum_{i=1}^{m} a_i y_i \leq \sum_{i=1}^{m} a_i - \beta - 1 \qquad \left(\text{Notice } \sum_{i=1}^{m} a_i > \beta\right)$$



* USE, E.G., B&B OR CUTTING PLANE WITH GOMORY CUTS
(WE DONOT KNOW IF THE IMPROVED FORMULATION IS IDEAL!

It can be interesting to restrict the execution time then use Gomory cuts to make the procedure faster. Or even, use the same formulation then add Branch-and-Bound. Note that this problem is very similar to the original knapsack problem (1)–(4), and therefore it would probably be better to solve directly the original problem rather than solving a similar problem just to find a new inequality to include in the formulation (and then iterate this procedure! This adds the same complexity into the same problem.

9.3 Cover Inequalities for General Binary Problems and General Procedure

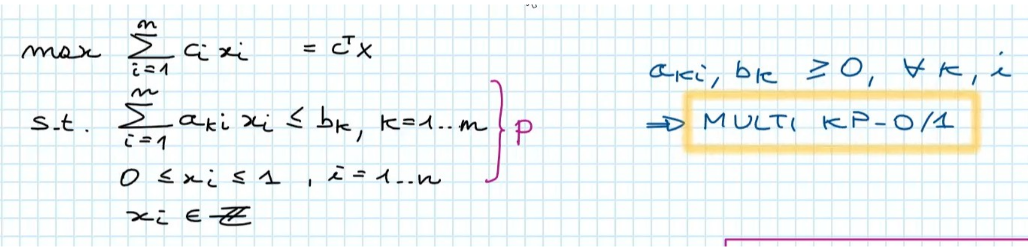
Consider a general integer linear programming problem in which all variables are binary:

$$\max \ c^T x \tag{9}$$
$$\text{s.t. } Ax \leq b, \tag{10}$$
$$0 \leq x_i \leq 1, \quad i = 1, \ldots, n, \tag{11}$$
$$x_i \in \mathbb{Z}, \quad i = 1, \ldots, n. \tag{12}$$

where $A \geq 0$.

- A crucial observation is that every single constraint of the system $Ax \leq b$ can be seen as a knapsack-type constraint of the form (2)
- In other words, if we replace the system $Ax \leq b$ with any of its constraints, and remove all the others, we obtain a relaxation of the problem that looks exactly like a knapsack problem
- It is then possible to add to the formulation the cover inequalities associated with each of the knapsack problems obtained this way (i.e., removing all constraints but one)

- The number of possible cover inequalities will be huge, but we can employ the approach described above to add the inequalities only when they are really needed



$$\max \sum_{i=1}^{m} c_i x_i = c^T x$$

$$\left.\begin{aligned} \text{s.t. } & \sum_{i=1}^{m} a_{ki} x_i \le b_k, \quad k=1..m \\ & 0 \le x_i \le 1, \quad i=1..n \end{aligned}\right\} P$$

$$x_i \in \mathbb{Z}$$

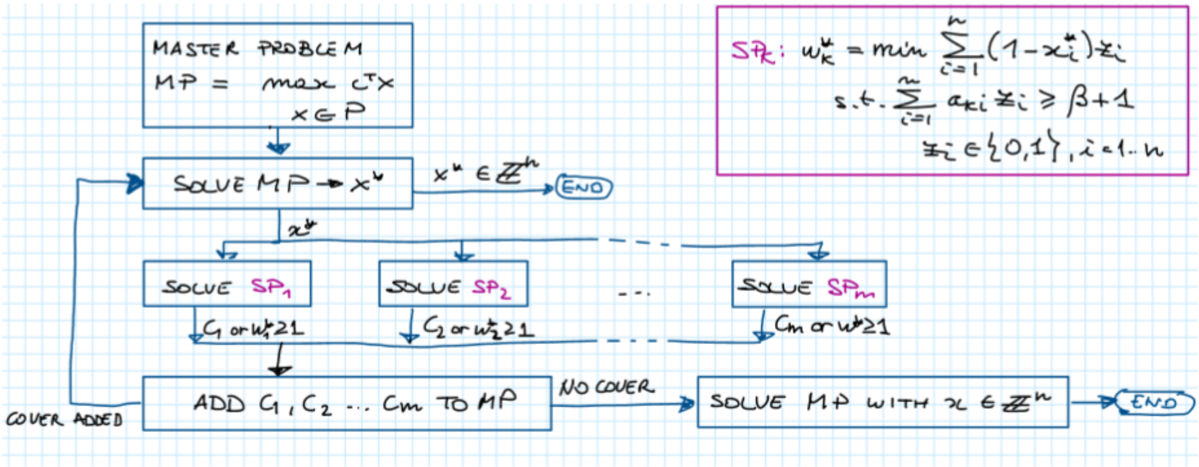
$a_{ki}, b_k \ge 0, \forall k, i$

$\Rightarrow$ MULTI KP-0/1

Suppose that we have solved the continuous relaxation (9) – (11), thus obtaining a non-integer solution $\bar{x}$.

- For each knapsack problem obtained by removing all the constraints of the system $Ax \le b$ except one, we ask ourselves whether there is a cover inequality violated by $\bar{x}$
- To this purpose, it is sufficient to solve a problem of the form (5) – (8)
- If the optimal value of this problem is smaller than 1, then we have found a cover inequality violated by $\bar{x}$ that can be added to the formulation; otherwise there is no cover inequality violated by $\bar{x}$
- We can then solve the new continuous relaxation (including the cover inequalities that have been added) and iterate this procedure until the current solution satisfies all cover inequalities

Basically, each inequality is a one-dimensional knapsack problem (NP-hard in its weak formulation) – this is the following algorithm, represented in both ways:


MASTER PROBLEM
MP = max $c^T x$, $x \in P$
SOLVE MP → x^*
$x^* \in \mathbb{Z}^n$
END
x^*
SOLVE SP_1
SOLVE SP_2
SOLVE SP_m
C_1 or $w_1^* \ge 1$
C_2 or $w_2^* \ge 1$
C_m or $w_m^* \ge 1$
ADD $C_1, C_2 \dots C_m$ TO MP
COVER ADDED
NO COVER
SOLVE MP WITH $x \in \mathbb{Z}^n$
END
$SP_k: w_k^* = \min \sum_{i=1}^{n} (1-x_i^*) z_i$
s.t. $\sum_{i=1}^{m} a_{ki} z_i \ge \beta+1$
$z_i \in \{0,1\}, i=1..n$

The diagram shows a clear iterative process starting with a Master Problem (MP) defined as $max\ c^T x$ over a feasible region P. When we solve this MP, we get a solution x^* in E^n space.

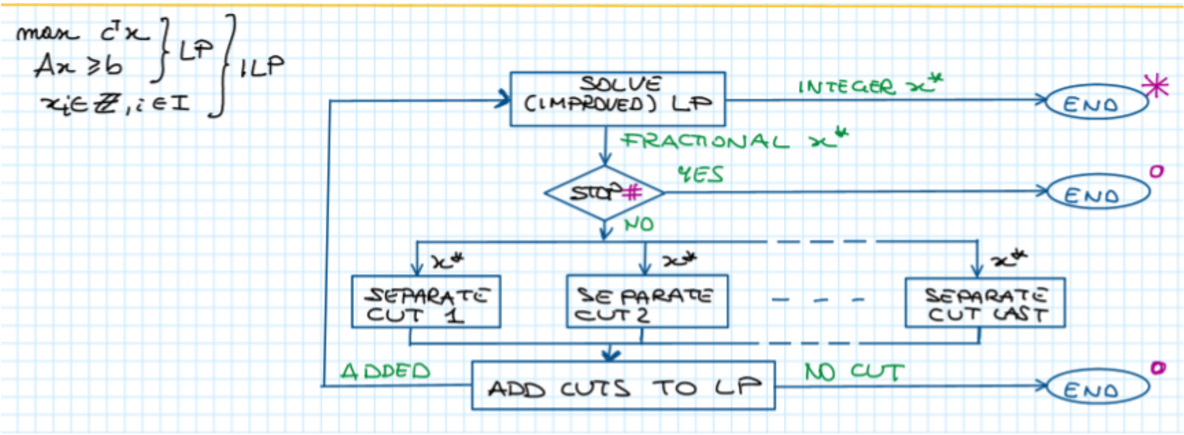
This solution then feeds into multiple subproblems. The diagram specifically shows their mathematical formulation.

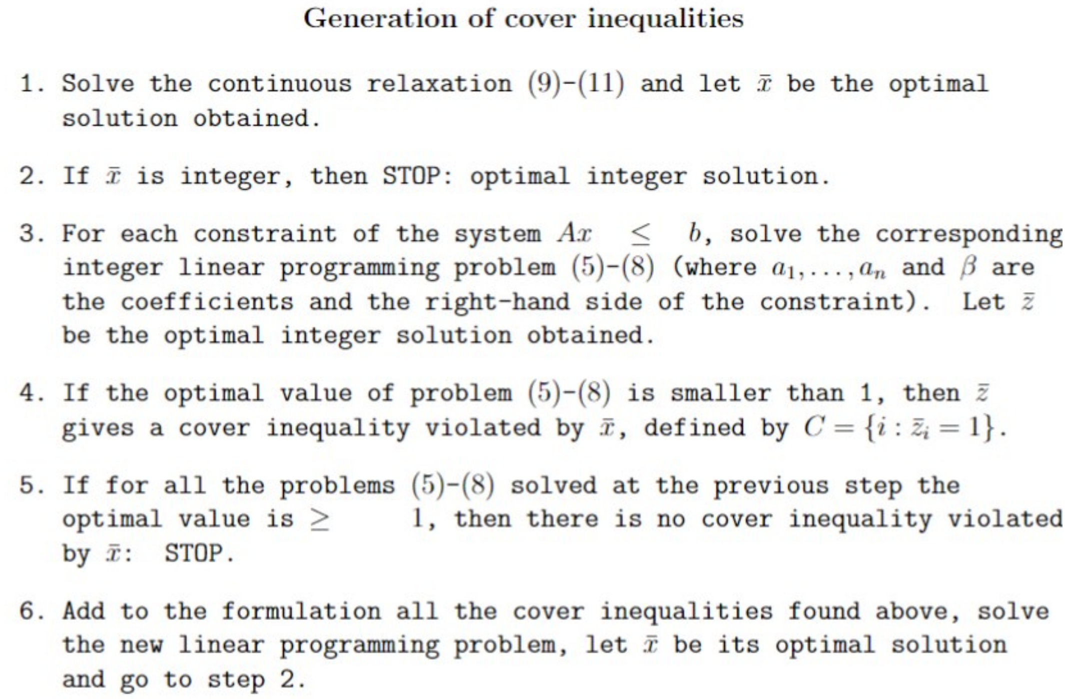
Once these subproblems are solved, they generate columns which are added back to the Master Problem. The process then follows one of two paths:

- If new columns were found, we loop back to solve the enhanced Master Problem
- If no columns were found ("NO COLS"), we solve a final version of MP with and terminate

Add all possible cover inequalities, then get an improved formulation after some time.

Now, we give the general cutting plane procedure:





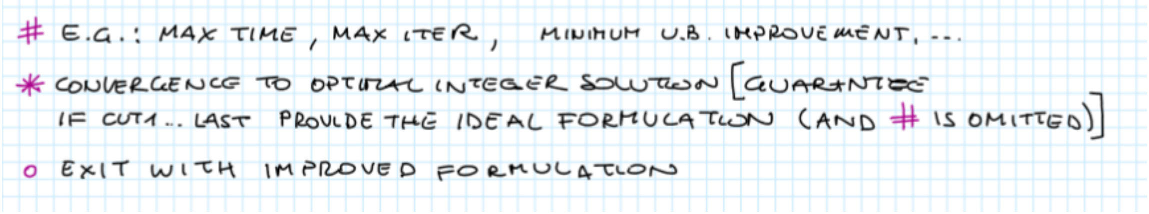
Generation of cover inequalities

1. Solve the continuous relaxation (9)-(11) and let $\bar{x}$ be the optimal solution obtained.
2. If $\bar{x}$ is integer, then STOP: optimal integer solution.
3. For each constraint of the system $Ax \leq b$, solve the corresponding integer linear programming problem (5)-(8) (where $a_1, \ldots, a_n$ and β are the coefficients and the right-hand side of the constraint). Let $\bar{z}$ be the optimal integer solution obtained.
4. If the optimal value of problem (5)-(8) is smaller than 1, then $\bar{z}$ gives a cover inequality violated by $\bar{x}$, defined by $C = \{i : \bar{z}_i = 1\}$.
5. If for all the problems (5)-(8) solved at the previous step the optimal value is ≥ 1, then there is no cover inequality violated by $\bar{x}$: STOP.
6. Add to the formulation all the cover inequalities found above, solve the new linear programming problem, let $\bar{x}$ be its optimal solution and go to step 2.

You can always separate such knapsack problem once it has been found. With the above algorithm we have to solve many problems of the form (5)–(8), which are integer linear programming problems and therefore hard in general.

- However, problem (5)–(8) is one of the simplest ILP problems and therefore, although in principle an exponential time might be needed to solve it, in practice an optimal solution can be found in a reasonable amount of time
- As seen above, the algorithm terminates when the current solution does not violate any cover inequality
    - Note however that when this happens, $\overline{x}$ might still be non-integer, because the cover inequalities are not sufficient to describe the ideal formulation of the problem (which is not known
- Of course it is possible to stop the algorithm before its natural termination if we think that the inequalities that we have added are sufficient to give a good formulation of the problem. In both cases, if $\overline{x}$ is not integer, we can apply the Branch-and-Bound method

- The fact that some cover inequalities have been added allows us to start from a better formulation and usually makes the Branch-and-Bound method terminate in a shorter time



E.G.: MAX TIME, MAX ITER, MINIMUM U.B. IMPROVEMENT, ...

* CONVERGENCE TO OPTIMAL INTEGER SOLUTION [GUARANTEE IF CUT1 .. LAST PROVIDE THE IDEAL FORMULATION (AND # IS OMITTED)]

o EXIT WITH IMPROVED FORMULATION

The procedure shown in the flowchart illustrates an iterative optimization process that starts with solving a relaxed linear program (LP), then systematically strengthens it by adding cuts when fractional solutions are found.

- The method continues in a cycle – solving the improved LP, checking if the solution is integer, and either adding more cuts through separation procedures or terminating based on specific criteria (like time limits or iteration counts)
- The process is guaranteed to converge to an optimal integer solution if the separation procedures can provide an ideal formulation and no arbitrary stopping criteria are enforced
- This creates a powerful framework that bridges the gap between continuous and integer solutions by progressively shaping the feasible region through strategically chosen cutting planes – how to use them depends on the problem

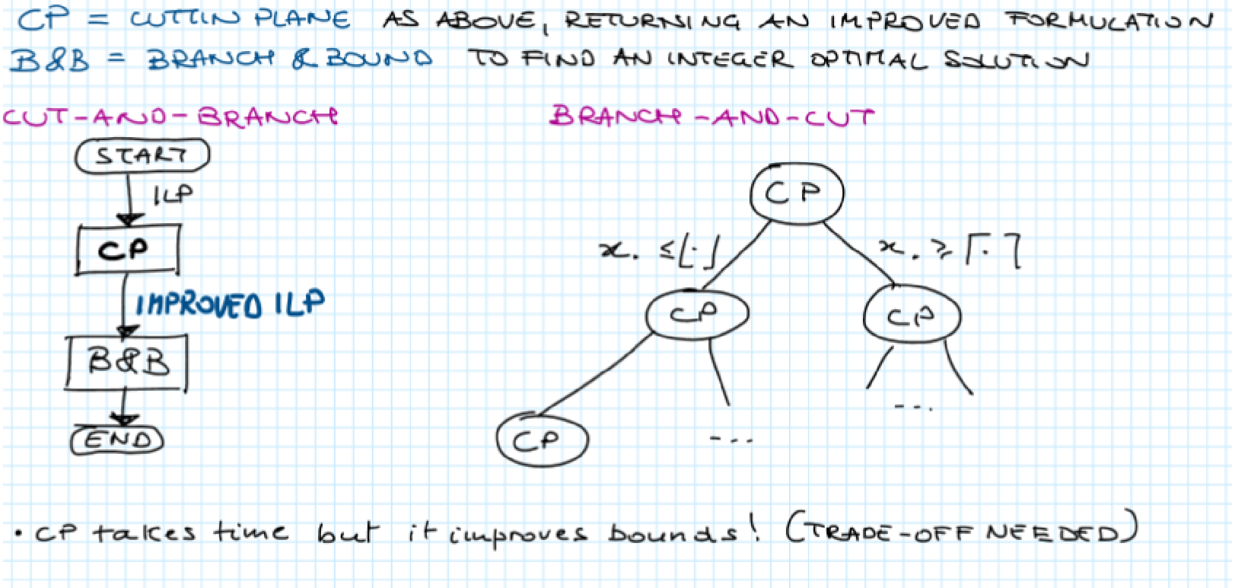
9.4 Hybrid Methods, Exercises and CPLEX Output

The image presents two different but complementary approaches to solving integer linear programming problems: Cut-and-Branch versus Branch-and-Cut. Let's understand what each does:

- Cut-and-Branch
    - This is a sequential approach
    - We start with our ILP, apply cutting plane methods (CP) to strengthen the formulation, and then use this improved ILP as input for a branch-and-bound (B&B) procedure
    - Think of it as "clean up first, then solve" - we strengthen our formulation upfront before trying to find the integer solution

- Branch-and-Cut (right side)
    - This shows a more integrated approach. Instead of separating the cutting and branching phases, we apply cutting planes throughout the branch-and-bound tree. At each node we can generate cuts based on the current solution
    - The branching decisions create new subproblems where we can again apply cutting planes
    - Here the formulation is strengthened with valid inequalities and the B&B is applied

The key insight noted at the bottom is crucial: While cutting planes take time to generate, they improve our bounds, potentially reducing the total nodes we need to explore in the branch-and-bound tree.


CP = CUTTING PLANE AS ABOVE, RETURNING AN IMPROVED FORMULATION
B&B = BRANCH & BOUND TO FIND AN INTEGER OPTIMAL SOLUTION
CUT-AND-BRANCH
BRANCH-AND-CUT
START
ILP
CP
IMPROVED ILP
B&B
END
CP
$x_\cdot \le \lfloor \cdot \rfloor$
$x_\cdot \ge \lceil \cdot \rceil$
CP
CP
CP
...
...
• CP takes time but it improves bounds! (TRADE-OFF NEEDED)

Cutting plane approaches let us work with a single formulation instead of creating a potentially massive branching tree.

- However, success hinges entirely on having separation procedures that can efficiently identify violated inequalities
- The challenge lies in finding cuts quickly enough to justify avoiding branching, while ensuring these cuts are strong enough to meaningfully improve the formulation
- This fundamental balance between speed of cut generation and strength of formulation improvement determines whether cutting planes will outperform traditional branching methods

We add here two things which might help on preparation – solving first the “exercises” (since this year OPL was not done, I’m gonna answer only the two questions present).

1. In which cases a cutting plane method is sufficient to solve an ILP?

The cutting plane procedure alone can provide the ideal formulation of the problem, meaning that all generated cuts lead to a polyhedron whose vertices are all integer solutions. This occurs when the separation procedures can identify all necessary valid inequalities to describe the convex hull of integer solutions. Additionally, no artificial stopping criteria (like maximum time or iteration limits) should be enforced, allowing the method to generate all required cuts.

However, it's important to note that this is relatively rare in practice. Even for well-studied problems like the knapsack problem, cover inequalities alone do not provide the ideal formulation, as mentioned in the provided text.

2. How can we improve the performance of the branch-and-bound method for ILP using cutting planes?

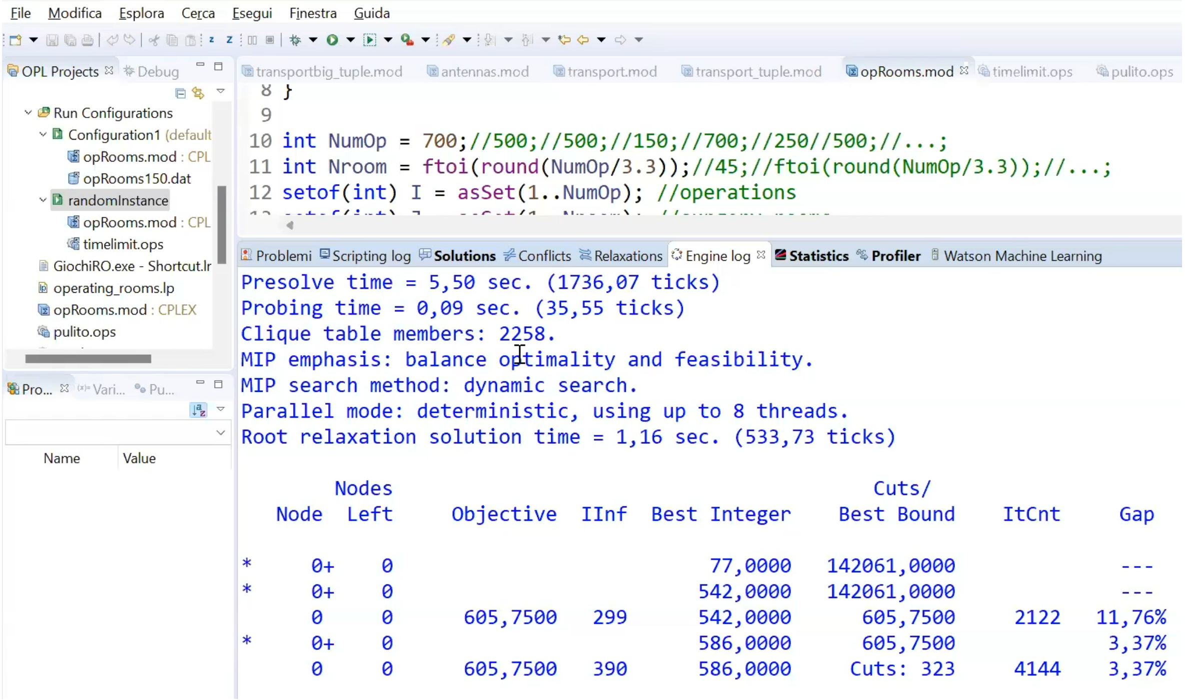
The performance of branch-and-bound for ILPs can be improved using cutting planes through two main approaches:

- The first approach is branch-and-cut, where we strengthen the initial formulation by adding cutting planes before starting the branch-and-bound procedure
    - This provides tighter bounds at the root node, potentially reducing the total number of nodes that need to be explored in the branch-and-bound tree

- The second, more sophisticated approach is branch-and-cut, where cutting planes are generated and added at each node of the branch-and-bound tree
    - This continuously strengthens the formulation throughout the solution process, providing tighter bounds at each node
    - While this requires more computational effort to generate cuts at multiple nodes, it can significantly reduce the total number of nodes that need to be explored, often leading to faster overall solution times

In both cases, the key advantage comes from the improved bounds provided by the cutting planes, which allow for more effective pruning of the branch-and-bound tree and consequently faster convergence to the optimal integer solution.

We will now (lastly) discuss the OPL example of surgery rooms (seen here if you don't remember), which is important by the point of view of this course unit, since it allows us to understand the output of a complex solution using a commercial solver like CPLEX.

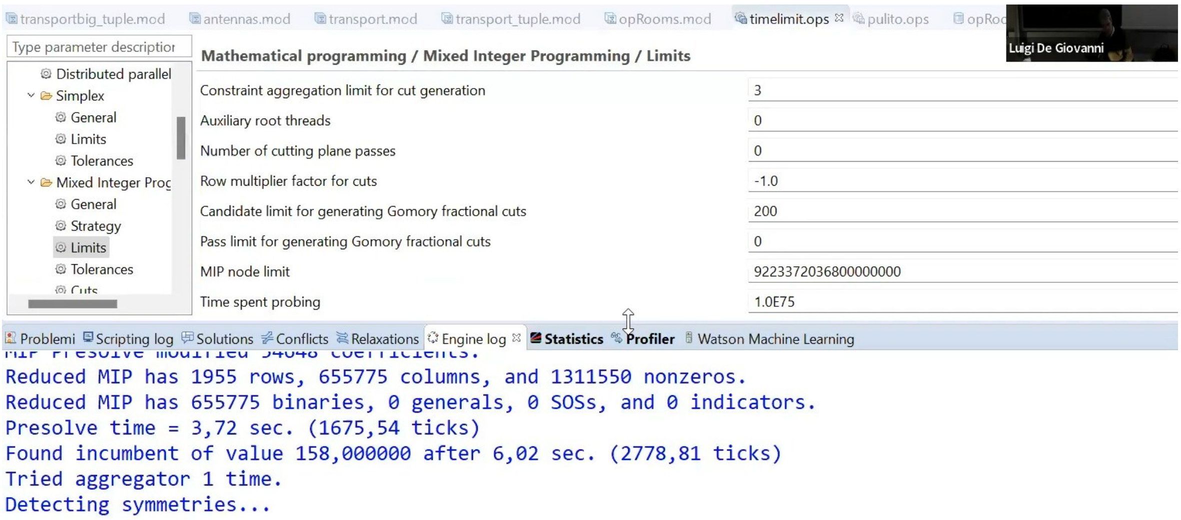


```
}

int NumOp = 700;//500;//500;//150;//700;//250//500;//...;
int Nroom = ftoi(round(NumOp/3.3));//45;//ftoi(round(NumOp/3.3));//...;
setof(int) I = asSet(1..NumOp); //operations
```

```
Presolve time = 5,50 sec. (1736,07 ticks)
Probing time = 0,09 sec. (35,55 ticks)
Clique table members: 2258.
MIP emphasis: balance optimality and feasibility.
MIP search method: dynamic search.
Parallel mode: deterministic, using up to 8 threads.
Root relaxation solution time = 1,16 sec. (533,73 ticks)

        Nodes                                         Cuts/
   Node  Left     Objective  IInf  Best Integer    Best Bound    ItCnt     Gap

*     0+    0                         77,0000   142061,0000             ---
*     0+    0                        542,0000   142061,0000             ---
      0     0      605,7500   299    542,0000      605,7500     2122   11,76%
*     0+    0                        586,0000      605,7500              3,37%
      0     0      605,7500   390    586,0000    Cuts: 323      4144    3,37%
```

Different things can be noticed here:

- Customized cuts are applied for the sake of separation procedure – a lot of separators are present internally
- For example they can be customized as you can see here

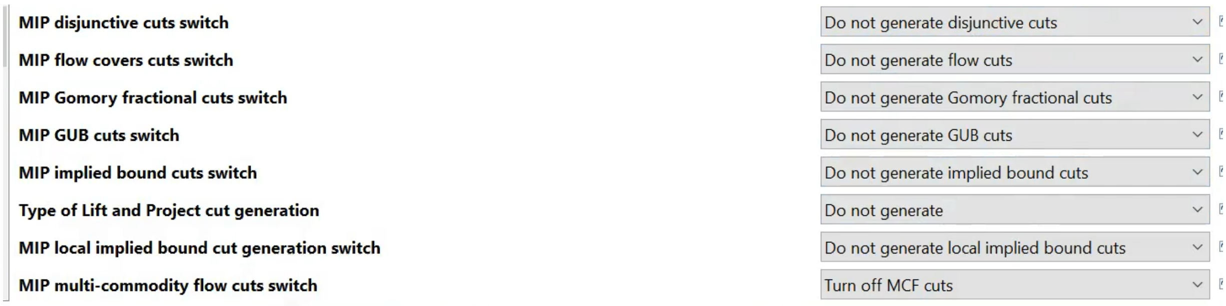


Mathematical programming / Mixed Integer Programming / Limits

Constraint aggregation limit for cut generation	3
Auxiliary root threads	0
Number of cutting plane passes	0
Row multiplier factor for cuts	-1.0
Candidate limit for generating Gomory fractional cuts	200
Pass limit for generating Gomory fractional cuts	0
MIP node limit	9223372036800000000
Time spent probing	1.0E75

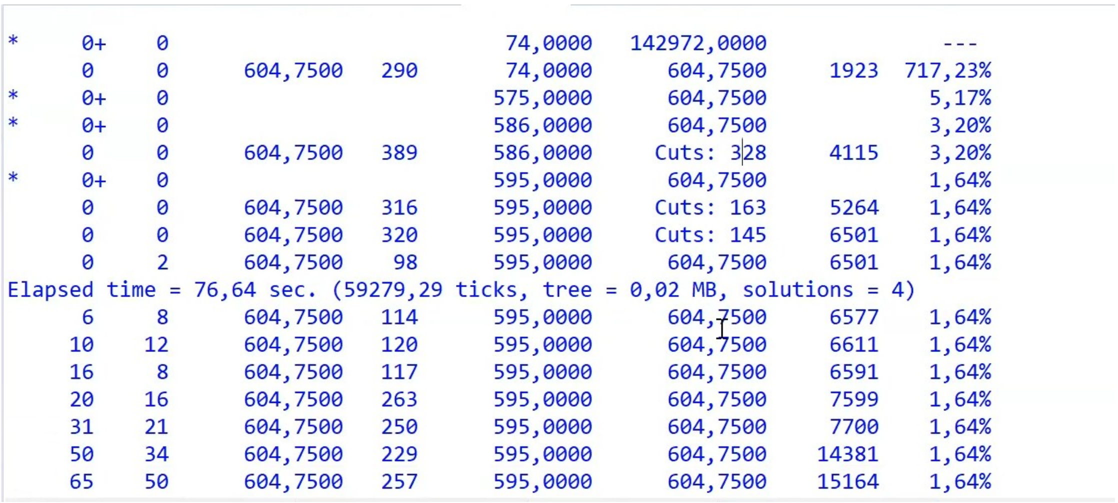
```
MIP Presolve modified 54048 coefficients.
Reduced MIP has 1955 rows, 655775 columns, and 1311550 nonzeros.
Reduced MIP has 655775 binaries, 0 generals, 0 SOSs, and 0 indicators.
Presolve time = 3,72 sec. (1675,54 ticks)
Found incumbent of value 158,000000 after 6,02 sec. (2778,81 ticks)
Tried aggregator 1 time.
Detecting symmetries...
```

A lot of different cuts can exist:



MIP disjunctive cuts switch	Do not generate disjunctive cuts
MIP flow covers cuts switch	Do not generate flow cuts
MIP Gomory fractional cuts switch	Do not generate Gomory fractional cuts
MIP GUB cuts switch	Do not generate GUB cuts
MIP implied bound cuts switch	Do not generate implied bound cuts
Type of Lift and Project cut generation	Do not generate
MIP local implied bound cut generation switch	Do not generate local implied bound cuts
MIP multi-commodity flow cuts switch	Turn off MCF cuts

You see that the linear relaxation cuts away important portions, coming exactly up to the right solution – the best bound of linear relaxation is 604.75 (number of nodes is increasing and the bound stays the same, so good):



```
*     0+    0                        74,0000   142972,0000            ---
      0     0      604,7500   290      74,0000     604,7500     1923  717,23%
*     0+    0                       575,0000     604,7500              5,17%
*     0+    0                       586,0000     604,7500              3,20%
      0     0      604,7500   389     586,0000    Cuts: 328     4115    3,20%
*     0+    0                       595,0000     604,7500              1,64%
      0     0      604,7500   316     595,0000    Cuts: 163     5264    1,64%
      0     0      604,7500   320     595,0000    Cuts: 145     6501    1,64%
      0     2      604,7500    98     595,0000     604,7500     6501    1,64%
Elapsed time = 76,64 sec. (59279,29 ticks, tree = 0,02 MB, solutions = 4)
      6     8      604,7500   114     595,0000     604,7500     6577    1,64%
     10    12      604,7500   120     595,0000     604,7500     6611    1,64%
     16     8      604,7500   117     595,0000     604,7500     6591    1,64%
     20    16      604,7500   263     595,0000     604,7500     7599    1,64%
     31    21      604,7500   250     595,0000     604,7500     7700    1,64%
     50    34      604,7500   229     595,0000     604,7500    14381    1,64%
     65    50      604,7500   257     595,0000     604,7500    15164    1,64%
```

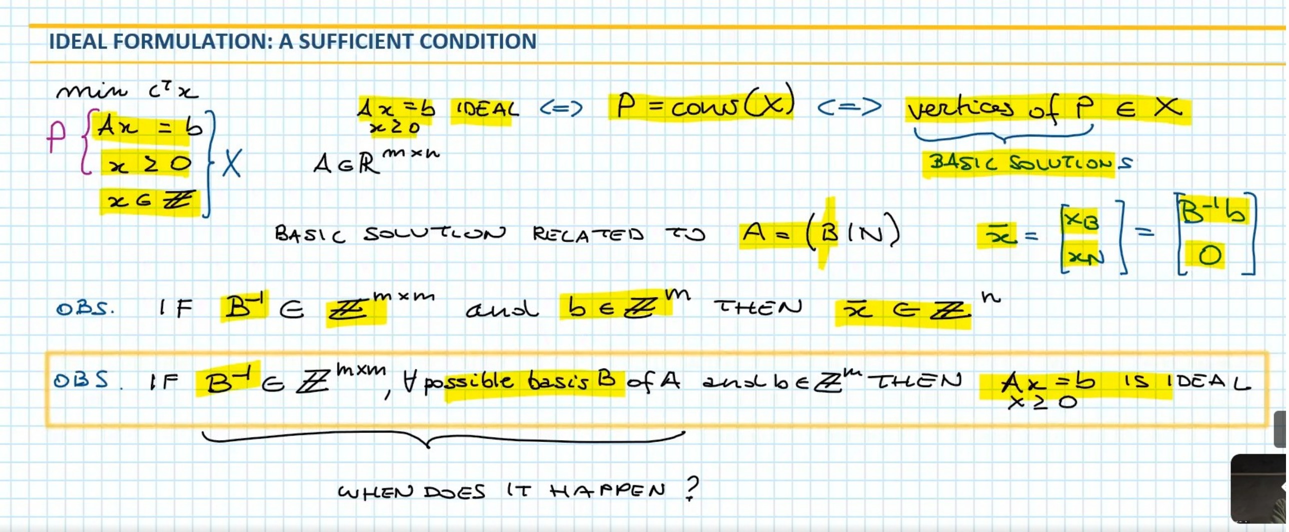
It's better for the bound to be a *smaller* number, which means I branched on the previous bound and got a smaller value.

10 FOR READING - IDEAL FORMULATIONS, ASSIGNMENT PROBLEM AND TOTAL UNIMODULARITY (9)

Note: program was over before this. The professor did not dedicate much time to this, but in order to wrap up properly everything, there are notes properly summarized on the non-relevant parts

In order to solve MILP problems, we need Branch-and-Bound, cutting planes or mixed techniques. Are there cases in which we need simply the simplex method? This is possible when the vertices of the linear relaxation are integer, meaning that the formulation is ideal.

We consider a minimization problem, with integrality requirements and a polyhedron associated to linear relaxation. If formulation is ideal, the polyhedron is a convex hull, and all of these vertices are inside of the feasible region. When does this happen?



IDEAL FORMULATION: A SUFFICIENT CONDITION

$\min c^T x$

$P \begin{cases} Ax = b \\ x \geq 0 \end{cases}$ $\left.\begin{matrix} \\ \\ x \in \mathbb{Z} \end{matrix}\right\} X$ $\quad A \in \mathbb{R}^{m \times n}$

$\begin{matrix} Ax = b \\ x \geq 0 \end{matrix}$ IDEAL $\iff$ $P = \text{conv}(X)$ $\iff$ $\underbrace{\text{vertices of } P}_{\text{BASIC SOLUTIONS}} \in X$

BASIC SOLUTION RELATED TO $A = (B \mid N)$ $\quad \bar{x} = \begin{bmatrix} x_B \\ x_N \end{bmatrix} = \begin{bmatrix} B^{-1}b \\ 0 \end{bmatrix}$

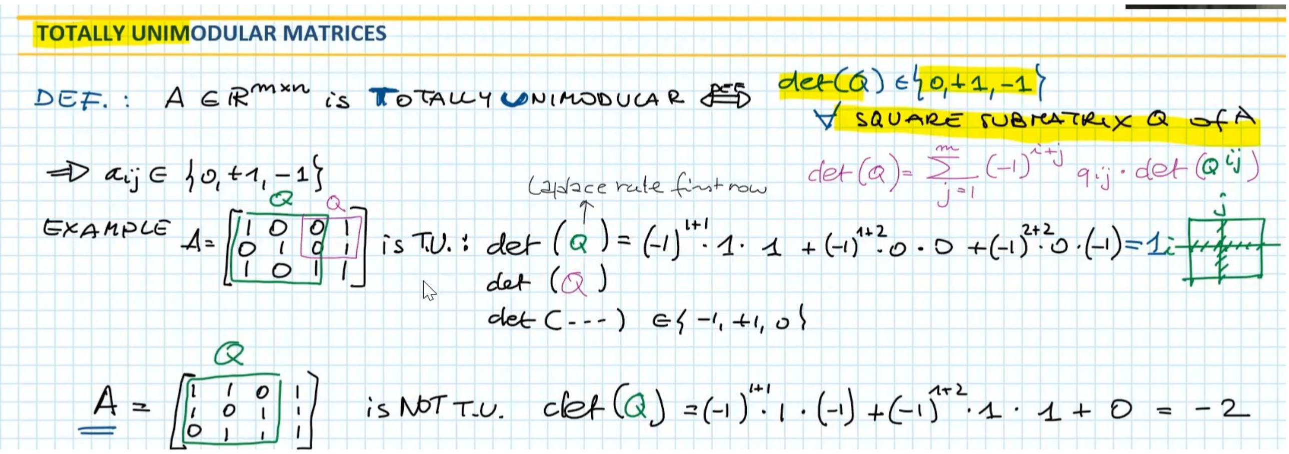
OBS. IF $B^{-1} \in \mathbb{Z}^{m \times m}$ and $b \in \mathbb{Z}^m$ THEN $\bar{x} \in \mathbb{Z}^n$

OBS. IF $\underbrace{B^{-1} \in \mathbb{Z}^{m \times m}, \forall \text{ possible basis } B \text{ of } A}_{\text{WHEN DOES IT HAPPEN ?}}$ and $b \in \mathbb{Z}^m$ THEN $\begin{matrix} Ax = b \\ x \geq 0 \end{matrix}$ IS IDEAL

We get the particular case of totally unimodular matrices, in which for each square submatrix, determinant is either $0, 1, -1$.

- Note that sub-matrices need not be composed of contiguous columns and rows. As a consequence of this definition we have that every element of a totally unimodular matrix must be $0, 1$ or -1, because submatrices of 1×1 must also satisfy the condition on the determinant

Determinant can be computed with Laplace rule, making a cofactor expansion:



TOTALLY UNIMODULAR MATRICES

DEF. : $A \in \mathbb{R}^{m \times n}$ is TOTALLY UNIMODULAR $\overset{\text{DEF}}{\iff}$ $\det(Q) \in \{0, +1, -1\}$ $\forall$ SQUARE SUBMATRIX Q of A

$\Rightarrow a_{ij} \in \{0, +1, -1\}$

Laplace rule first row $\quad \det(Q) = \sum_{j=1}^{m} (-1)^{i+j} q_{ij} \cdot \det(Q^{ij})$

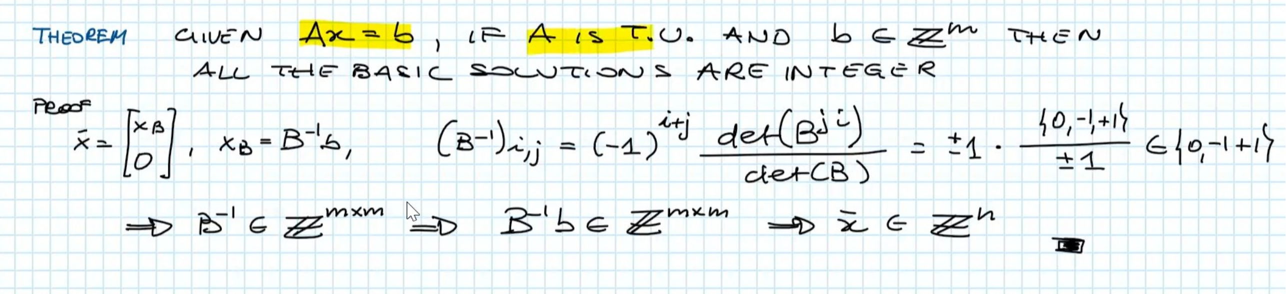
EXAMPLE $A = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix}$ is T.U. : $\det(Q) = (-1)^{1+1} \cdot 1 \cdot 1 + (-1)^{1+2} \cdot 0 \cdot 0 + (-1)^{2+2} \cdot 0 \cdot (-1) = 1$

$\det(Q)$

$\det(\dots) \in \{-1, +1, 0\}$

$\underline{\underline{A}} = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}$ is NOT T.U. $\det(Q) = (-1)^{1+1} \cdot 1 \cdot (-1) + (-1)^{1+2} \cdot 1 \cdot 1 + 0 = -2$

There is an important consequence to this:



THEOREM GIVEN $Ax=b$, IF A IS T.U. AND $b \in \mathbb{Z}^m$ THEN ALL THE BASIC SOLUTIONS ARE INTEGER

PROOF

$$\bar{x} = \begin{bmatrix} x_B \\ 0 \end{bmatrix}, \quad x_B = B^{-1}b, \qquad (B^{-1})_{i,j} = (-1)^{i+j} \frac{\det(B^{ji})}{\det(B)} = \pm 1 \cdot \frac{\{0,-1,+1\}}{\pm 1} \in \{0,-1,+1\}$$

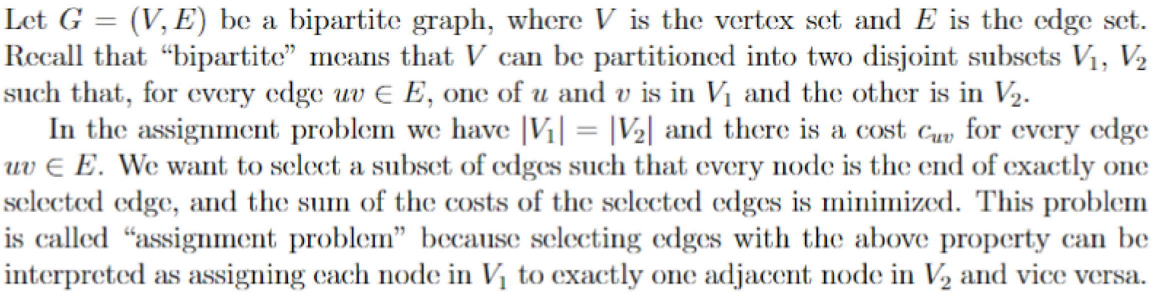
$$\Rightarrow B^{-1} \in \mathbb{Z}^{m \times m} \Rightarrow B^{-1}b \in \mathbb{Z}^{m \times m} \Rightarrow \bar{x} \in \mathbb{Z}^n \quad \blacksquare$$

When A is totally unimodular and b is an integer vector, all basic solutions of the system $Ax = b$ are integer-valued. This means the formulation is ideal - we don't need to explicitly enforce integrality constraints. The proof of this relies on examining how basic solutions are constructed:

- For any basis B of A, the basic solution is given by xB = B^{-1}b, xN = 0
- Since B is a submatrix of a TU matrix, det(B) must be -1 or 1
- The elements of B^{-1} can be expressed as ratios of determinants: $(B^{-1})_{ij} = (-1)^{i+j}$ det($B^j{}_i$)/det(B)
- Since both numerator and denominator are ±1, B^{-1} contains only integers
- Therefore, xB = B^{-1}b must also be integer when b is integer

A key theorem characterizes an important class of TU matrices: If a matrix A has only 0/1 entries with at most two 1s per column, and its rows can be partitioned into two sets such that when a column has two 1s, they occur in rows from different sets, then A is totally unimodular.

10.1 Assignment Problem



Let $G = (V, E)$ be a bipartite graph, where V is the vertex set and E is the edge set. Recall that "bipartite" means that V can be partitioned into two disjoint subsets V_1, V_2 such that, for every edge $uv \in E$, one of u and v is in V_1 and the other is in V_2.

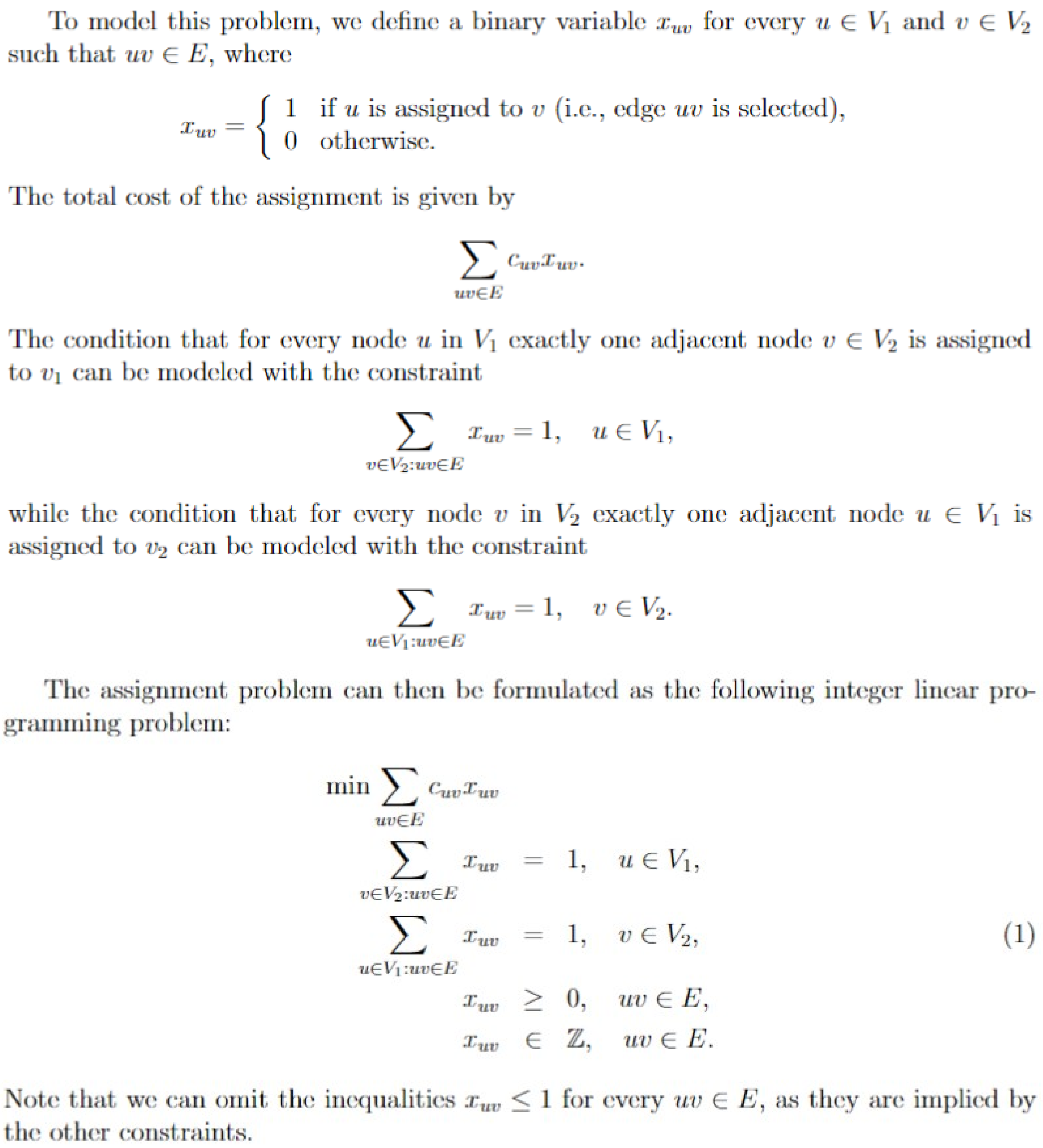
In the assignment problem we have $|V_1| = |V_2|$ and there is a cost c_{uv} for every edge $uv \in E$. We want to select a subset of edges such that every node is the end of exactly one selected edge, and the sum of the costs of the selected edges is minimized. This problem is called "assignment problem" because selecting edges with the above property can be interpreted as assigning each node in V_1 to exactly one adjacent node in V_2 and vice versa.

In a bipartite graph assignment problem, we aim to match elements from two sets while minimizing total cost. The problem's incidence matrix is naturally totally unimodular due to its special structure.

- The incidence matrix of any bipartite graph is totally unimodular
- This follows from the key theorem about 0/1 matrices with at most two 1s per column, as the rows can be partitioned into the two node sets of the bipartite graph
- Each column (representing an edge) has exactly two 1s - one for each endpoint, occurring in different partitions of the nodes

Due to total unimodularity, all basic solutions to the assignment problem's LP relaxation are naturally integer-valued, meaning we can solve the assignment problem using linear programming methods like the simplex algorithm without explicitly enforcing integrality constraints.

More on the problem modeling:



To model this problem, we define a binary variable x_{uv} for every $u \in V_1$ and $v \in V_2$ such that $uv \in E$, where

$$x_{uv} = \begin{cases} 1 & \text{if } u \text{ is assigned to } v \text{ (i.e., edge } uv \text{ is selected),} \\ 0 & \text{otherwise.} \end{cases}$$

The total cost of the assignment is given by

$$\sum_{uv \in E} c_{uv} x_{uv}.$$

The condition that for every node u in V_1 exactly one adjacent node $v \in V_2$ is assigned to v_1 can be modeled with the constraint

$$\sum_{v \in V_2 : uv \in E} x_{uv} = 1, \quad u \in V_1,$$

while the condition that for every node v in V_2 exactly one adjacent node $u \in V_1$ is assigned to v_2 can be modeled with the constraint

$$\sum_{u \in V_1 : uv \in E} x_{uv} = 1, \quad v \in V_2.$$

The assignment problem can then be formulated as the following integer linear programming problem:

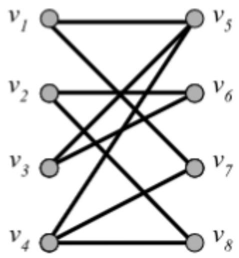
$$\begin{array}{rcll} \min \sum_{uv \in E} c_{uv} x_{uv} & & & \\ \sum_{v \in V_2 : uv \in E} x_{uv} & = & 1, & u \in V_1, \\ \sum_{u \in V_1 : uv \in E} x_{uv} & = & 1, & v \in V_2, \\ x_{uv} & \geq & 0, & uv \in E, \\ x_{uv} & \in & \mathbb{Z}, & uv \in E. \end{array} \qquad (1)$$

Note that we can omit the inequalities $x_{uv} \leq 1$ for every $uv \in E$, as they are implied by the other constraints.

This formulation of the problem is ideal, so I can discard the integrality constraint on the variables and thus I can solve the model using the simplex method. We also have that the obtained constraint matrix is totally unimodular.

This matrix is called the (indirect) graph incidence matrix and is also valid for graphs that are not bipartite. If the graph is bipartite, the incidence matrix is TU. To represent oriented graphs in this way the matter is slightly different. I use -1 in the row for the starting node and $+1$ in the row for the ending node. For a directed degree, the incidence matrix is always TU, even if it is not bipartite.

A bipartite graph and its incidence matrix are shown below.



$$\begin{array}{c|ccccccccc|}
 & v_1v_5 & v_1v_7 & v_2v_6 & v_2v_8 & v_3v_5 & v_3v_6 & v_4v_5 & v_4v_7 & v_4v_8 \\
\hline
v_1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
v_2 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
v_3 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
v_4 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
\hline
v_5 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
v_6 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
v_7 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
v_8 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}$$

It can be easily checked that the assignment problem can be written in the form

$$\begin{gathered}\min c^T x \\ A(G)\,x = \mathbf{1} \\ x \geq 0 \\ x \in \mathbb{Z}^{|E|},\end{gathered}$$

where $\mathbf{1}$ denotes a vector whose components are all equal to 1.

10.2 TU Matrices Properties and Other Problems

Several fundamental properties preserve total unimodularity (TU). For a totally unimodular matrix A:

1. Row and column permutations preserve TU
2. Multiplication of rows/columns by -1 preserves TU
3. Matrix transposition preserves TU: AT is TU
4. Augmentation with identity matrices preserves TU:
    1. (A, I) is TU where I is $m \times m$ identity matrix
    2. $[A; I]$ is TU where I is $n \times n$ identity matrix

The transportation problem naturally extends the assignment problem while maintaining totally unimodular properties:

- Relaxes 1-1 matching to allow multiple units of flow
- Maintains bipartite structure between sources and sinks
- Supply/demand constraints preserve TU properties
- Integer-valued basic solutions when supplies/demands are integer

The network flow framework exploits a maximum flow structure:

- Given directed graph $D = (V, A)$ with capacity constraints c:
    - Flow variables x: x_{uv} represents flow on arc (u, v)
- Conservation constraints:
    - Flow-in equals flow-out at all nodes except s, t
    - Matrix representation $A(D')x = 0$ where D' includes $s - t$ arc
- Capacity constraints: $0 \leq x_{uv} \leq c_{uv}$
- Matrix remains TU through property preservation
- Integer flows guaranteed when capacities are integer

The minimum cost flow has properties:

- Combines flow conservation with arc costs
- TU property ensures integer flows with integer data
- Linear programming solution yields combinatorial optimality
- Polynomial-time solvability despite combinatorial nature

Also, there are circulatory properties

- Special case without source/sink nodes
- All flows must be conserved at every node
- TU properties ensure integer circulation values
- Applications in periodic scheduling and resource allocation

The power of these formulations lies in reducing combinatorial problems to linear programs while maintaining integer solutions through total unimodularity. This enables efficient solution methods that leverage continuous optimization for inherently discrete problems.

This theoretical foundation explains why certain network optimization problems remain tractable despite their combinatorial nature, bridging the gap between discrete and continuous optimization through matrix properties.

On the next part: the TSP cannot be formulated with a totally unimodular constraint matrix (as it's NP-hard). However, certain TSP relaxations/subproblems relate to network flows:

- 1-tree relaxations can be found using max flow techniques
- Assignment problem relaxations (which are TU) provide bounds
- Subtour elimination constraints conceptually relate to flow conservation

11 FOR READING - EXACT METHODS FOR THE TSP – MODELS AND METHODS (10)

Note: The part of "Exact methods for the Traveling Salesman Problem" is considered "For reading" this year of the course, so definitely not gonna touch that into notes

(From Moodle in case you might be interested there are the PDFs, otherwise if you're like me, have a look at the lesson in older Moodle courses. I will provide a helper summary in any case)

TSP arises in two main variants: asymmetric (ATSP) with directed arcs and symmetric (STSP) with undirected edges. Both share the challenge of subtour elimination but require different solution strategies due to their structural differences.

Asymmetric TSP Methods

Constraint Generation Approach

The method builds on a key insight: without subtour elimination constraints, ATSP reduces to an assignment problem with totally unimodular constraints. This leads to an iterative process:

1. Initial Solution Phase:

- Solve pure assignment problem relaxation
- Obtain integer solution efficiently via simplex
- Identify any existing subtours

2. Constraint Addition:

- Add subtour elimination constraints only as needed
- Maintain integrality only on variables in these constraints
- Resolve increasingly constrained problems

3. Convergence:

- Process continues until obtaining Hamiltonian cycle
- Guarantees optimality through exhaustive constraint addition
- Often reaches solution before adding all possible constraints

Branch-and-Bound Strategy

This alternative approach exploits problem structure differently:

1. Relaxation Mechanism:

- Uses assignment problem as base relaxation
- Obtains valid lower bounds efficiently
- Identifies subtours for branching decisions

2. Branching Strategy:

- Selects a subtour in current solution
- Creates branches by prohibiting arcs from this subtour
- Implements prohibitions through cost modification (setting to infinity)

3. Tree Management:

- Maintains problem structure through modified costs
- Avoids explicit constraint handling
- Allows efficient node processing via assignment algorithms

Symmetric TSP Innovations

Advanced Relaxation Methods

The symmetric case permits specialized approaches:

1. Linear Programming Relaxation:

- Works with edge variables (reducing problem size)
- Uses degree constraints as base structure
- Dynamically adds subtour elimination as needed

2. Separation Procedure:

- Converts constraint identification to network flow
- Efficiently handles fractional solutions
- Provides strong cutting planes

Efficient Constraint Generation

The method employs sophisticated separation:

1. Maximum Flow-Based Detection:

- Constructs capacitated network from current solution
- Identifies violated constraints through min-cuts
- Efficiently processes fractional solutions

2. Dynamic Implementation:

- Adds constraints iteratively
- Maintains problem tractability
- Focuses on most violated constraints

Branch-and-Cut: State-of-the-Art Integration

This combines the strengths of previous approaches:

1. Algorithmic Framework:

- Integrates LP relaxation with cutting planes
- Uses intelligent branching on fractional variables
- Inherits cuts through the branch-and-bound tree

2. Implementation Features:

- Efficient linear programming solutions
- Strong valid inequality generation
- Strategic constraint management

3. Practical Performance:

- Solves million-node instances
- Provides provable optimality
- Manages memory and computation trade-offs effectively

4. Advanced Components:

- Multiple cut families beyond subtour elimination
- Sophisticated separation algorithms
- Adaptive cut management strategies

This framework represents the current pinnacle of exact TSP solution methods, successfully bridging theoretical insights and practical computing limitations. Its success demonstrates how deep understanding of mathematical structure can lead to effective algorithms, though significant computational resources may still be required for large instances.

The field continues to advance through:

- New valid inequality classes
- Improved separation algorithms
- Enhanced branching strategies
- More efficient implementation techniques

To end this file – and also my Master/Bachelor notes, done for all courses up to now, below some general interest links.

*Optional reading: sample applications (**free** link from the Department network)*

- *Evolving Neural Networks Through Augmenting Topologies. (K.O. Stanley and R. Miikkulainen, Evolutionary Computation)*
- *Data-driven matheuristic for the Air Traffic Flow Management Problem (L. De Giovanni, C. Lancia and G. Lulli)*
- *A two-level local search heuristic for pickup and delivery problems in express freight trucking (L. De Giovanni, N. Gastaldon and F. Sottovia, Networks)*

12 Last Meeting of the Course

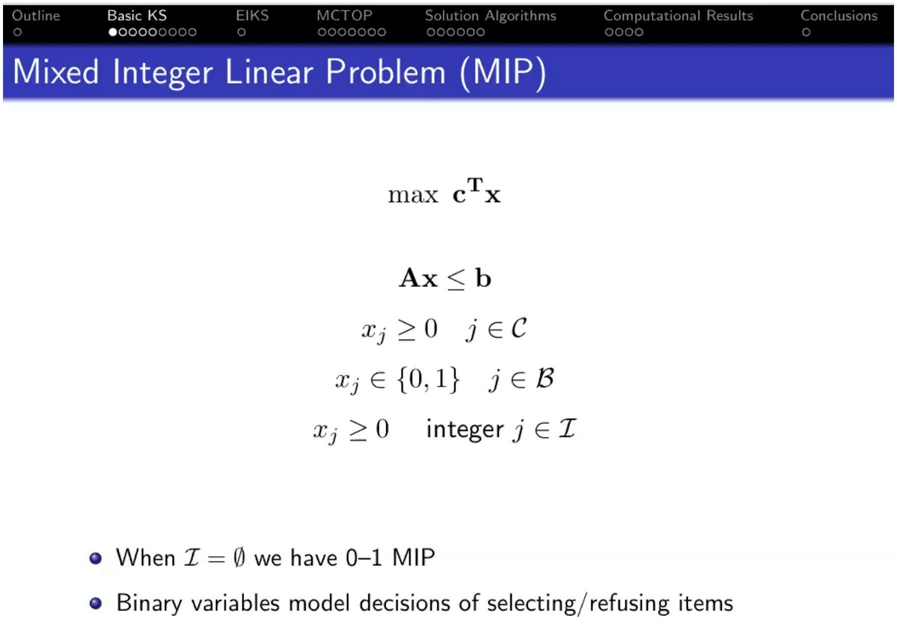
12.1 First Part – Hybrid Metaheuristics

As noted here, the last part of this course revolves around “hybrid metaheuristics” (see here). We would need some basic formal ideas of mathematics in order to get a grasp of the presented concepts. These tools can be “combined” which starts from the statement of the problem (data), providing a good solution. Let’s hybridize then, using the techniques as follows (reading some papers with them:

- We can try to use trajectory methods for example
- Or even use matheuristics – metaheuristics + math
    - Exact methods to get information – then apply heuristics
    - Problem-dependent elements are included only within the lower-level mathematic programming, local search or constructive components

- Another interesting method might be kernel search (helping slides provided by Marsini, researcher of UniTN)
    - As present here by the abstract
        - The central idea is to use some method, for example, the LP-relaxation, to identify a subset (named kernel) of promising decision variables and then to partition the remaining ones into buckets
        - These are concatenated one at a time to the kernel in order to check whether improving solutions can be found

Just to show – since they are internal of this course – some images coming from the presented slides. The problem to be solved is this one:



Mixed Integer Linear Problem (MIP)

$$\max\ \mathbf{c}^{\mathbf{T}}\mathbf{x}$$

$$\mathbf{A}\mathbf{x} \leq \mathbf{b}$$

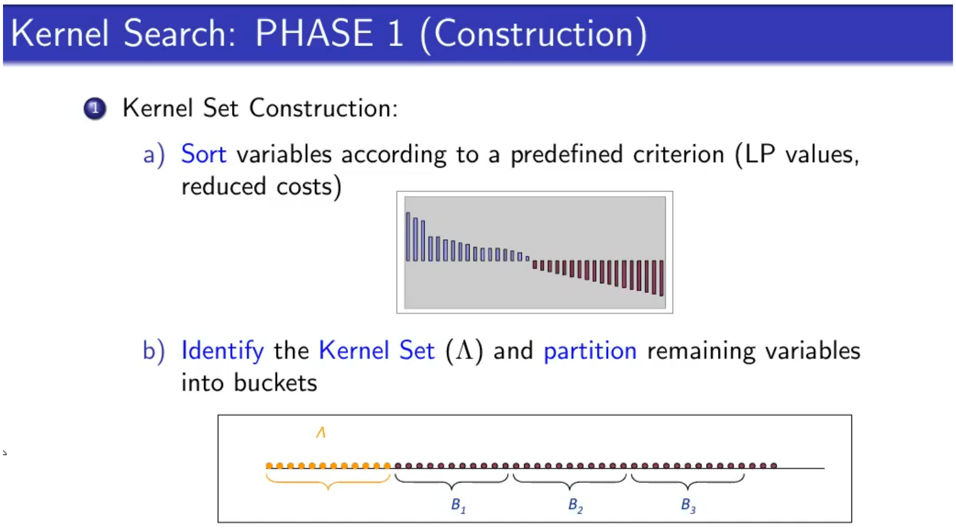
$$x_j \geq 0 \quad j \in \mathcal{C}$$

$$x_j \in \{0,1\} \quad j \in \mathcal{B}$$

$$x_j \geq 0 \quad \text{integer } j \in \mathcal{I}$$

- When $\mathcal{I} = \emptyset$ we have 0–1 MIP
- Binary variables model decisions of selecting/refusing items

Here there is the buckets construction:



Kernel Search: PHASE 1 (Construction)

1. Kernel Set Construction:
   a) Sort variables according to a predefined criterion (LP values, reduced costs)
   b) Identify the Kernel Set (Λ) and partition remaining variables into buckets

- We can use information coming from available data to help with the solution process. Such is called data-driven optimization
  - Data-driven is to use data as a means of production of extracted features through scientific methods and apply them to problems to be solved
  - These methods have certain applicability and advantages in the research of supply chain management
  - Think about AI works, possibly as a way to learn from a part of your solution method
  - See the professor paper to get a grasp much better than this

When there is a problem:

- Spend time to read literature and engineer properly a good method

12.2 Second Part – Talking about the Exam

In this case:

- Homework is ready and given to the professor
- The professor has an idea of the score (between 0 and 10)

First part of the exam is discussion about the homework – if the student is able to justify the final score of the exercise. Once with this, there are questions during the course classes.

This year, as noted above, just the definition of total unimodularity and forward. But please, know the general ideas/definitions or the modeling parts for the TSP (last topic above).

Given the professor nature, you might know at this point questions are made of reasoning, not only exact concepts. It's more of a discussion with technical aspects. For example:

- In a column generation approach, can we use a heuristic method to solve the slave problem?
- Yes, slave problem has a negative reduced cost variable, then we apply heuristics; if no negative cost variable was found, we cannot stop with linear relaxation, but using heuristics

12.3 THIRD PART – TALKING ABOUT THE EXERCISE

Provide a report (unique) for both parts:

- How the model was implemented (not reporting it exactly) but if there were problems/issues on the formulation

On the first part of the exercise:

- Instances generation is important (something to take care of)
- We have TSP in a specific drilling context
- Instances should be coherent with that
- Random components should be present in this generation, depending on ways in which they are generated
- For example, given the matrices sizes, holes are to be distributed uniformly at random distances – this can be a good way, but maybe not the best way for electrical panels
- Holes have to be distributed with some structure (square/circles, shapes even)
- Perhaps this structure can help the finding of the solution of the TSP instance
- If specific choices are made, please, report them (for example, if you don't have time to implement wanted number of instances say it)

On the second part of the exercise:

- Any basic method is OK, but please consider adding things (Local Search with multistart, Tabu Search with intensification, Genetic Algorithm hybridized with Local Search)
- The idea should be simple (but not so much – not basic)
- If you have doubts of any kind, the professor is definitely more than welcome to help you, so in case of doubts, do it before the exam
- Professor is open to discussion

Please deliver the full zip with the parts by mail complete with:

- Code
- Report

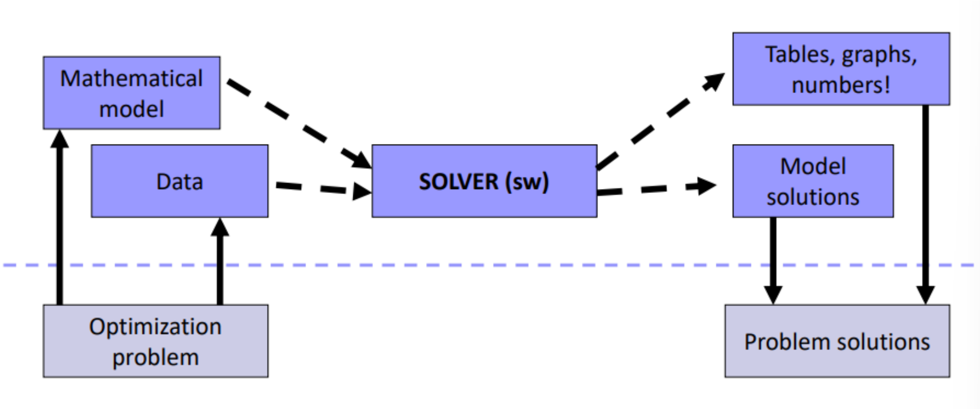
The code **MUST** be compiled inside of the lab machines – before sending the exercise, make sure of this. In case, provide him with instructions on how to execute.

Example: if exam is 30th of January, please deliver up until 27th – but professor may agree with you to have a different date for both submission and examination.

If one delivers the project is outside the exam dates, you can do it whenever you want. Normally, the oral exam is inside of the official date but is to be agreed individually with the professor (as always happens, but anyway).

13 Laboratory 1 - Solvers for Mathematical Programming (Docplex)

We start from the definition of a solver: a software application that takes the description of an optimization problem as input and provides the solution of the model (and related information) as output.



In particular, we're interested in solvers which act on MILP problems, which are the most used in practice:

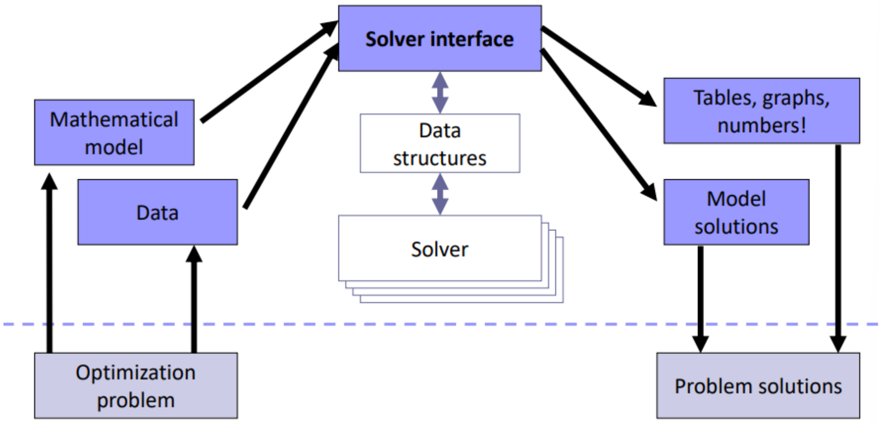
- very efficient
- numerical stability
- easy to use or embed

They have had more than 1 000 000 000 speed-ups in the last 20 years, where thousands of variables may be processed in order to express fast algorithms:

- hardware speed-up: x 1000
- simplex improvements: x 1000
- branch-and-cut improvement: x 1000

Examples may be: Cplex, Gurobi, Xpress, Scip, Lindo, GLPK, Google OR Tools etc.

The core of lab units is not to implement solvers, but solver interfaces, so to build the model and simply call the software.



A commercial software for which we have the license is IBM Ilog Cplex, which is one of the first MILP solvers. Some general features:

- Includes state-of-the-art technology
- One of the best solvers available (Gurobi, Xpress)

A list of possible interfaces is the following:

- Interactive optimizer
- OPL / AMPL / ZIMPL … algebraic modelling language (close to writing models on paper)
- C – API libraries (Callable libraries)
- C++ libraries (Concert technologies)
- Python APIs
- Python (with docplex) / Java / .Net wrapper libraries
- Matlab / Excel plugins

We are introduce to a magical Python interface called DOCplex, which is not compatible with Python 3.9 above versions and has caused in the lab a lot of trouble to different people

Important note: this year, only Docplex was done, while up to now (before 24-25), only OPL was done. So, even in the assignment, you will see that "Docplex Guys" are the "OPL guys" of previous years.

In any case, it's pretty simple and I will share some general info:

- IBM Decision Optimization CPLEX Modeling for Python
- Built upon the Cplex Python APIs
- Exploits Python syntax to provide "easy" and flexible encoding of the mathematical model notation, e.g.:
  - Dictionaries for sets of variables
  - for…in…if… to encode "forall" quantifiers or sum indices
- Ideal for prototyping and integration into "modern" applications
- Documentation: docplex landing pages
  - https://pypi.org/project/docplex/
  - https://ibmdecisionoptimization.github.io/docplex-doc/
    - *Getting started with Docplex*
    - *Mathe Programming Modeling for Python using docplex.mp*
- ■ Installation, e.g.
  - pip install docplex or
  - conda install -c ibmdecisionoptimization docplex

13.1 Make Docplex Run!

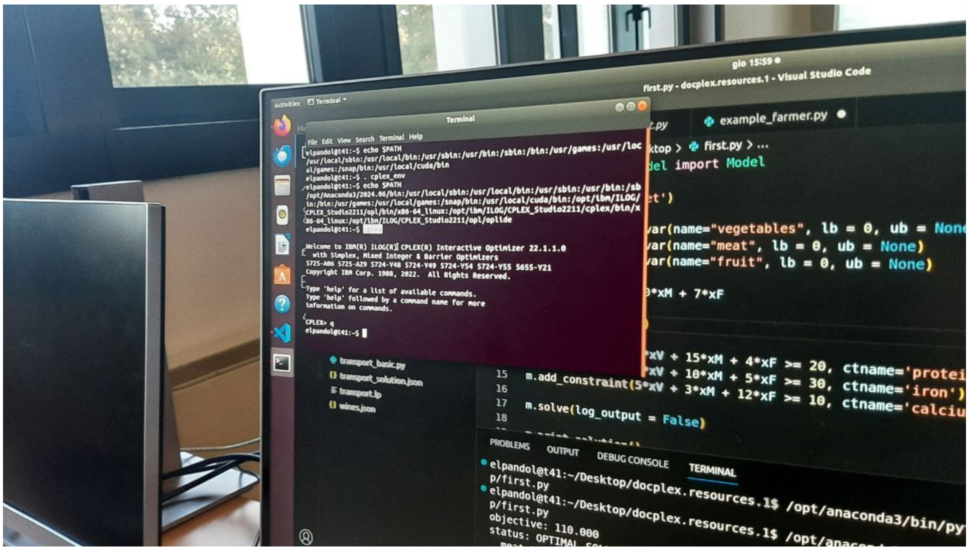
A first crucial step for the PC in labs:

- **To enable Cplex Studio at Lab**: use Linux, run

```
> . cplex_env
```

(notice "dot blank")

Inside of labs, it's important to first enable the cplex environment variable, check the path existence and then execute – we will infact use VS Code here:



```
elpandol@t41:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/loc
al/games:/snap/bin:/usr/local/cuda/bin
elpandol@t41:~$ . cplex_env
elpandol@t41:~$ echo $PATH
/opt/Anaconda3/2024.06/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sb
in:/bin:/usr/games:/usr/local/games:/snap/bin:/usr/local/cuda/bin:/opt/ibm/ILOG/
CPLEX_Studio2211/opl/bin/x86-64_linux:/opt/ibm/ILOG/CPLEX_Studio2211/cplex/bin/x
86-64_linux:/opt/ibm/ILOG/CPLEX_Studio2211/opl/oplide
elpandol@t41:~$ cplex

Welcome to IBM(R) ILOG(R) CPLEX(R) Interactive Optimizer 22.1.1.0
  with Simplex, Mixed Integer & Barrier Optimizers
5725-A06 5725-A29 5724-Y48 5724-Y49 5724-Y54 5724-Y55 5655-Y21
Copyright IBM Corp. 1988, 2022.  All Rights Reserved.

Type 'help' for a list of available commands.
Type 'help' followed by a command name for more
information on commands.

CPLEX> q
elpandol@t41:~$
```

```
del import Model
et')
var(name="vegetables", lb = 0, ub = None
var(name="meat", lb = 0, ub = None)
var(name="fruit", lb = 0, ub = None)
0*xM + 7*xF
*xV + 15*xM + 4*xF >= 20, ctname='protei
*xV + 10*xM + 5*xF >= 30, ctname='iron')
m.add_constraint(5*xV + 3*xM + 12*xF >= 10, ctname='calciu
m.solve(log_output = False)
```

```
elpandol@t41:~/Desktop/docplex.resources.1$ /opt/anaconda3/bin/py
p/first.py
elpandol@t41:~/Desktop/docplex.resources.1$ /opt/anacon
p/first.py
objective: 110.000
status: OPTIMAL
```

In general, as said, it works with Python 3.9 - so:

- Uninstall later versions from Programs
- Before uninstalling, in case for cleanup also remove pip, by opening a terminal and "uninstall pip" (or suggestions given from the prompt)
- Search for 3.9 and download: https://www.python.org/downloads/release/python-390/
- Quickly reinstall the magic "pip" from here: https://phoenixnap.com/kb/install-pip-windows
- (here he himself asks you to add to the path, tell him yes)
- Do "pip install docplex" (once you are sure that "pip" has been installed correctly)

When in doubt:

- Add in the PATH entry environment variables the path where Cplex is installed: C:\Program Files\IBM\ILOG\CPLEX_Studio2211
- That way you can run a terminal by typing "cplex"
- When you run, everything works

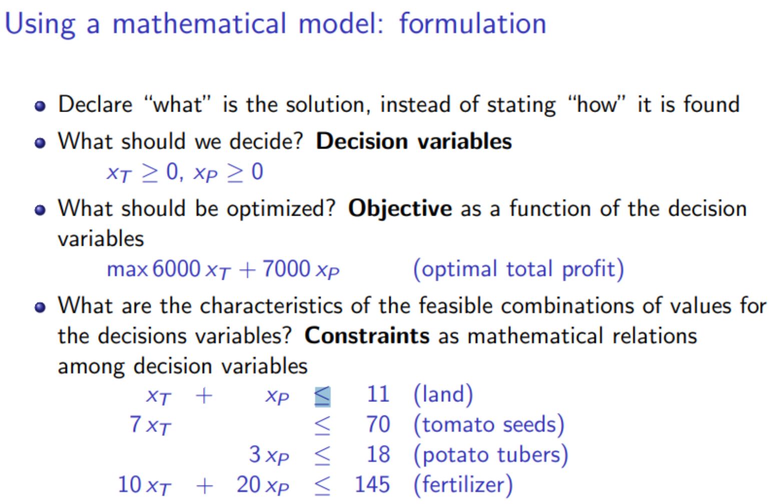
The following is a complete example with comments of all the things done in the lesson, based on the Farmer model, object of the first lesson:

```
#DOcplex basic functions
from docplex.mp.model import Model

# Creating an "empty" model
m = Model(name="Example model")

#Defining a variable
c_var = m.continuous_var()
i_var = m.integer_var()
b_var = m.binary_var()
#Variables can have optional arguments like: name, lb (lower bound), ub (upper bound)
#Default values are: name = x1, x2.. ; lb = 0; ub = +infinity

#Defining expressions: functions of decision and/or usual variables
expr_1 = 6 * c_var + i_var - pow(3,2) * b_var
expr_2 = 10 * c_var

#Creating a constraint
m.add_constraint(expr_1 <= expr_2) #or >= , ==

#Creating the objective function
m.minimize(expr_1) #or maximize

#Solving the model
m.solve()
#* Optional arguments:
#   - log_outputs = True | False
#   - cplex_parameters = ...
#   ...*#

#Checking status of the solution (optimal, infeasible, unounded, ..)
if m.solution == None:
    print("Problems! Status: ", m.get_solve_status())

#Printing the solution:
if m.solution != None:
    m.print_solution() #standard info (status, o.f. and variables values)
    sol = m.solution
    print(sol[c_var]) #value of var c_var
    print(sol.get_objective_value()) #value of the o.f.

#Exporting the model
m.export_as_lp(basename='filename', path='path', hide_user_names=False) #text file (e.g., LP format)

#Exporting the solution in json format
```

```
m.solution.export('filename')

#Exporting the solution in a string:
print(m.solution.to_string())
```

14 Laboratory 2 – Solvers: Docplex (Continuation)

Let’s start from the model we saw last time, so the farmer – given the problems about docplex, we start from the same model:



Using a mathematical model: formulation

- Declare “what” is the solution, instead of stating “how” it is found
- What should we decide? **Decision variables**
  $x_T \geq 0,\ x_P \geq 0$
- What should be optimized? **Objective** as a function of the decision variables
  $\max 6000\, x_T + 7000\, x_P$ (optimal total profit)
- What are the characteristics of the feasible combinations of values for the decisions variables? **Constraints** as mathematical relations among decision variables

$$
\begin{array}{rcrcrl}
x_T & + & x_P & \leq & 11 & \text{(land)} \\
7\, x_T & & & \leq & 70 & \text{(tomato seeds)} \\
 & & 3\, x_P & \leq & 18 & \text{(potato tubers)} \\
10\, x_T & + & 20\, x_P & \leq & 145 & \text{(fertilizer)}
\end{array}
$$

The Python code is the following:

```python
# Starting from scratch, since in the last lab python 3.9 not present in labs conditioned the fact everyone had problems.
# docplex DOES NOT work for versions above

from docplex.mp.model import Model

m = Model(name="Farmer")

#Defining variables

xT = m.continuous_var(name="ht of tomatoes", lb=0) #lb=0 --> should be non-negative
xP = m.continuous_var(name="ht of potatoes", lb=0)

revenue = 6000 * xT + 4000 * xP #objective function
m.maximize(revenue)

m.add_constraint( xP + xT <= 11, ctname='land avail')
m.add_constraint( 7*xT    <= 70)
m.add_constraint( 3*xP    <= 18)
m.add_constraint( 10*xT + 20*xP <= 145 )

m.print_information()
m.export_as_lp('.') #exporting the model as a text file, where lp is the extension for linear programming models

m.solve()
```

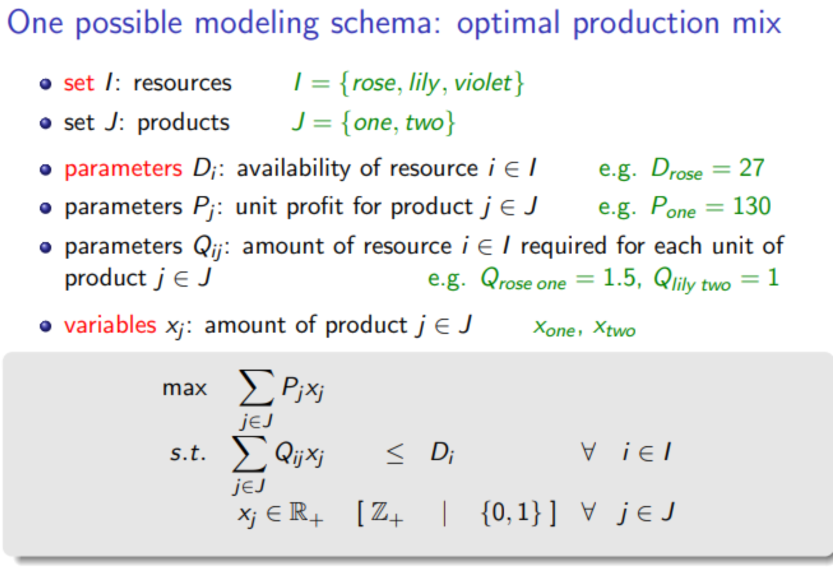
Then, the actual file .lp file is:

```
\ This file has been generated by DOcplex

\ ENCODING=ISO-8859-1

\Problem name: Farmer


Maximize

 obj: 6000 ht_of_tomatoes + 4000 ht_of_potatoes

Subject To

 land_avail: ht_of_tomatoes + ht_of_potatoes <= 11

 c2: 7 ht_of_tomatoes <= 70

 c3: 3 ht_of_potatoes <= 18

 c4: 10 ht_of_tomatoes + 20 ht_of_potatoes <= 145


Bounds

End
```

We check the availability of the solution:

```
# Checking the availability of the solution

if m.solution != None:
  m.print_solution() # print the solution
  print(m.solution(xT)) # press the value of xT
```

We have to generalize the way we create models, like the following optimal production mix:



One possible modeling schema: optimal production mix

- set I: resources $I = \{rose, lily, violet\}$
- set J: products $J = \{one, two\}$
- parameters D_i: availability of resource $i \in I$ e.g. $D_{rose} = 27$
- parameters P_j: unit profit for product $j \in J$ e.g. $P_{one} = 130$
- parameters Q_{ij}: amount of resource $i \in I$ required for each unit of product $j \in J$ e.g. $Q_{rose\ one} = 1.5,\ Q_{lily\ two} = 1$
- variables x_j: amount of product $j \in J$ x_{one}, x_{two}

$$\begin{array}{rlll} \max & \sum_{j \in J} P_j x_j & & \\ s.t. & \sum_{j \in J} Q_{ij} x_j \leq D_i & & \forall\ i \in I \\ & x_j \in \mathbb{R}_+ \quad [\mathbb{Z}_+ \quad \mid \quad \{0,1\}\,] & & \forall\ j \in J \end{array}$$

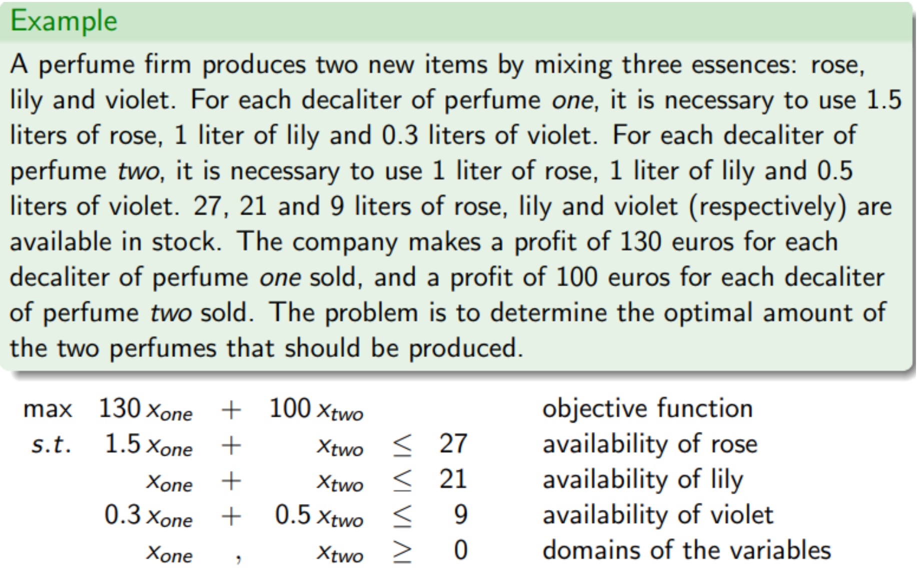
Written by Gabriel R.

This way we can solve several problems and examples; it’s advisable to use dictionaries as structure to represent multiple instances of data.

The complete code actually does basically the same things, but using dictionaries for the data and also getting the status of the solution:

```
from docplex.mp.model import Model

#data
I = ['land', 'tomato seeds', 'potato tubers', 'fertilizer']
J = ['tomatoes', 'potatoes']

#Data parameters
D = {'land': 11, 'tomato seeds':70, 'potato tubers': 18, 'fertilizer': 145}
P = {'tomatoes': 6000, 'potatoes':7000}
Q = {("land", "tomatoes"): 1,
    ("tomato seeds", "tomatoes"): 7,
    ("potato tubers", "tomatoes"): 0,
    ("fertilizer", "tomatoes"): 10,
    ("land", "potatoes"): 1,
    ("tomato seeds", "potatoes"): 0,
    ("potato tubers", "potatoes"): 3,
    ("fertilizer", "potatoes"): 20}

m = Model(name="prod mix")

x = {j: m.continuous_var(name='x({0})'.format(j)) for j in J}

m.maximize(m.sum(P[j] * x[j] for j in J))

for i in I:
   m.add_constraint(m.sum(Q[(i,j)] * x[j] for j in J) <= D[i])

m.print_information()
m.export_as_lp(".")

m.solve()

if m.solution != None:
   m.print_solution()
   for j in J:
      print(m.solution[x[j]], "is the qty of product", j)
else:
   print(m.get_solve_status())
```

If we change only a bit of the domain to try to solve the following problem, integer variables need to be used, changing the sets a bit and the following:



Example

A perfume firm produces two new items by mixing three essences: rose, lily and violet. For each decaliter of perfume *one*, it is necessary to use 1.5 liters of rose, 1 liter of lily and 0.3 liters of violet. For each decaliter of perfume *two*, it is necessary to use 1 liter of rose, 1 liter of lily and 0.5 liters of violet. 27, 21 and 9 liters of rose, lily and violet (respectively) are available in stock. The company makes a profit of 130 euros for each decaliter of perfume *one* sold, and a profit of 100 euros for each decaliter of perfume *two* sold. The problem is to determine the optimal amount of the two perfumes that should be produced.

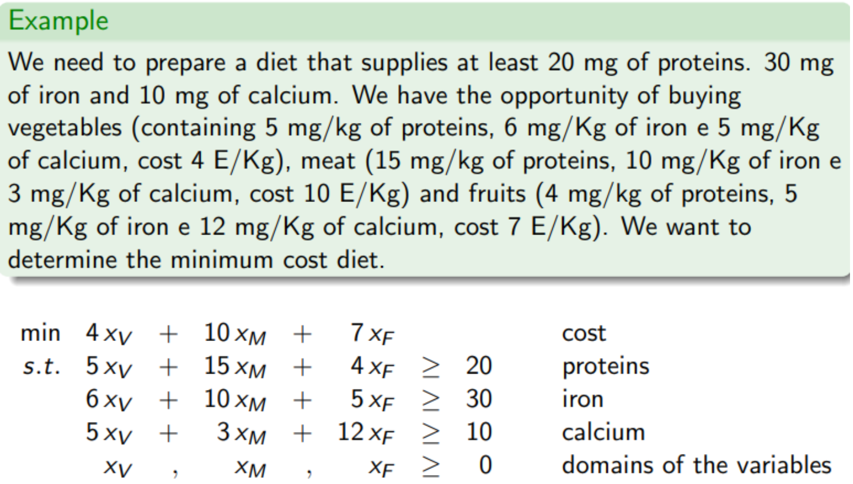
$$\begin{array}{llclcll}
\max & 130\, x_{one} & + & 100\, x_{two} & & & \text{objective function} \\
s.t. & 1.5\, x_{one} & + & x_{two} & \leq & 27 & \text{availability of rose} \\
 & x_{one} & + & x_{two} & \leq & 21 & \text{availability of lily} \\
 & 0.3\, x_{one} & + & 0.5\, x_{two} & \leq & 9 & \text{availability of violet} \\
 & x_{one} & , & x_{two} & \geq & 0 & \text{domains of the variables}
\end{array}$$

In this case we also make checks upon the nature of the actual domain:

```
from docplex.mp.model import Model

### Data ###
I = ['rose', 'lily', 'violet']
J = ['one', 'two']

### Data: parameters ###
D = {'rose': 12.5, 'lily': 21, 'violet': 9}
P = {'one': 130, 'two': 100}
Q = {
  ('rose', 'one'): 1.5, ('lily', 'one'): 1, ('violet', 'one'): 0.3,
  ('rose', 'two'): 1, ('lily', 'two'): 2, ('violet', 'two'): 0.5
}
decision_domain = "integer"

# Model
m = Model(name="prod mix")

if decision_domain == "integer":
  # Decision variables
  x = {j: m.integer_var(name=f'x_{j}') for j in J}
elif decision_domain == "continuous":
  # Decision variables
  x = {j: m.continuous_var(name=f'x_{j}') for j in J}

# Decision variables
x = {j: m.continuous_var(name=f'x_{j}') for j in J}
```

```
# Objective function
m.maximize(m.sum(P[j] * x[j] for j in J))

# Constraints
for i in I:
  m.add_constraint(
    m.sum(Q[i,j] * x[j] for j in J) <= D[i],
    ctname=f'Constraint_{i}'
  )

# Solve and print results
m.print_information()
m.export_as_lp('./prod_mix.lp')
solution = m.solve()

if solution:
  m.print_solution()
  for j in J:
    print(f"{x[j].solution_value:.2f} is the quantity of product {j}")
else:
  print("No solution found.")
  print("Solve status:", m.get_solve_status())
```

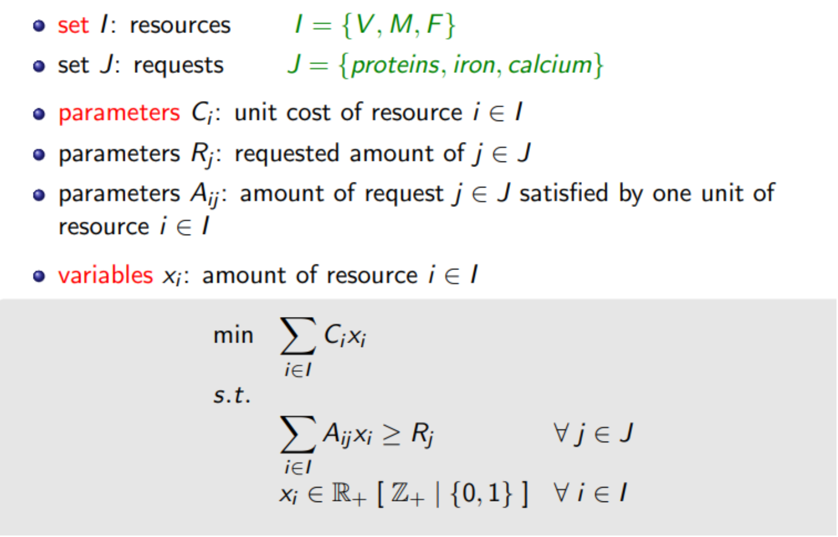
Now we go into the detail of the following model:



Example

We need to prepare a diet that supplies at least 20 mg of proteins. 30 mg of iron and 10 mg of calcium. We have the opportunity of buying vegetables (containing 5 mg/kg of proteins, 6 mg/Kg of iron e 5 mg/Kg of calcium, cost 4 E/Kg), meat (15 mg/kg of proteins, 10 mg/Kg of iron e 3 mg/Kg of calcium, cost 10 E/Kg) and fruits (4 mg/kg of proteins, 5 mg/Kg of iron e 12 mg/Kg of calcium, cost 7 E/Kg). We want to determine the minimum cost diet.

$$\begin{array}{lrcrcrcrl}
\text{min} & 4\,x_V & + & 10\,x_M & + & 7\,x_F & & & \text{cost} \\
s.t. & 5\,x_V & + & 15\,x_M & + & 4\,x_F & \geq & 20 & \text{proteins} \\
 & 6\,x_V & + & 10\,x_M & + & 5\,x_F & \geq & 30 & \text{iron} \\
 & 5\,x_V & + & 3\,x_M & + & 12\,x_F & \geq & 10 & \text{calcium} \\
 & x_V & , & x_M & , & x_F & \geq & 0 & \text{domains of the variables}
\end{array}$$



- set I: resources $I = \{V, M, F\}$
- set J: requests $J = \{proteins, iron, calcium\}$
- parameters C_i: unit cost of resource $i \in I$
- parameters R_j: requested amount of $j \in J$
- parameters A_{ij}: amount of request $j \in J$ satisfied by one unit of resource $i \in I$
- variables x_i: amount of resource $i \in I$

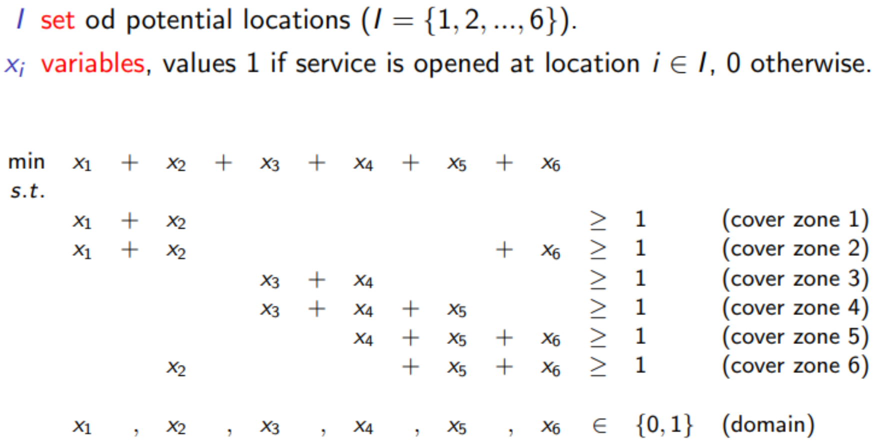
$$\begin{aligned} \min \quad & \sum_{i \in I} C_i x_i \\ s.t. \quad & \\ & \sum_{i \in I} A_{ij} x_i \geq R_j & \forall\, j \in J \\ & x_i \in \mathbb{R}_+ \; [\, \mathbb{Z}_+ \mid \{0,1\} \,] & \forall\, i \in I \end{aligned}$$

The complete version of the code uses the shortcuts able to index the variables present inside of the constraints so to create a coherent model:

```
from docplex.mp.model import Model

### Data ###
I = ['V', 'M', 'F']  # V: vegetables, M: meat, F: fruits
J = ['pro', 'iron', 'cal']  # pro: proteins, iron: iron, cal: calcium

### Data: parameters ###
C = {'V': 4, 'M': 10, 'F': 7}  # Cost per kg
R = {'pro': 20, 'iron': 30, 'cal': 10}  # Required amounts
A = {
  ('V', 'pro'): 5, ('V', 'iron'): 6, ('V', 'cal'): 5,
  ('M', 'pro'): 15, ('M', 'iron'): 10, ('M', 'cal'): 3,
  ('F', 'pro'): 4, ('F', 'iron'): 5, ('F', 'cal'): 12
}
decision_domain = "continuous"  # Can be "continuous", "integer", or "binary"

# Model
m = Model(name="min_cost_diet")

# Decision variables
if decision_domain == "integer":
  x = m.integer_var_dict(keys=I, lb=0, name="x")
elif decision_domain == "binary":
  x = m.binary_var_dict(keys=I, name="x")
else:  # continuous
  x = m.continuous_var_dict(keys=I, lb=0, name="x")

# Objective function
m.minimize(m.sum(C[i] * x[i] for i in I))

# Constraints
```

```
for j in J:
   m.add_constraint(m.sum(A[i,j] * x[i] for i in I) >= R[j], ctname=f"min_{j}")

# Solve and print results
m.print_information()
m.export_as_lp('./min_cost_diet.lp')
solution = m.solve()

if solution:
   m.print_solution()
   for i in I:
      print(f"{x[i].solution_value:.2f} kg of {i}")
   print(f"Total cost: {solution.objective_value:.2f}")
else:
   print("No solution found.")
   print("Solve status:", m.get_solve_status())
```

Another covering schema example is the following:



I set od potential locations ($I = \{1, 2, ..., 6\}$).

x_i variables, values 1 if service is opened at location $i \in I$, 0 otherwise.

$$
\begin{array}{lcccccccccccccll}
\min & x_1 & + & x_2 & + & x_3 & + & x_4 & + & x_5 & + & x_6 & & & \\
s.t. & & & & & & & & & & & & & & \\
 & x_1 & + & x_2 & & & & & & & & & \geq & 1 & \text{(cover zone 1)} \\
 & x_1 & + & x_2 & & & & & & & + & x_6 & \geq & 1 & \text{(cover zone 2)} \\
 & & & & & x_3 & + & x_4 & & & & & \geq & 1 & \text{(cover zone 3)} \\
 & & & & & x_3 & + & x_4 & + & x_5 & & & \geq & 1 & \text{(cover zone 4)} \\
 & & & & & & & x_4 & + & x_5 & + & x_6 & \geq & 1 & \text{(cover zone 5)} \\
 & & & x_2 & & & & & + & x_5 & + & x_6 & \geq & 1 & \text{(cover zone 6)} \\
 & & & & & & & & & & & & & & \\
 & x_1 & , & x_2 & , & x_3 & , & x_4 & , & x_5 & , & x_6 & \in & \{0,1\} & \text{(domain)}
\end{array}
$$

Here, a possible solution is:

```
from docplex.mp.model import Model

m = Model(name="Emergency")

#Defining variables
I = [1, 2, 3, 4, 5, 6]

x = m.binary_var_dict(keys=I, name="x")

# Objective function
m.minimize(m.sum(x[i] for i in I))

# Constraints
m.add_constraint(x[1] + x[2] >= 1)
m.add_constraint(x[1] + x[3] >= 1)
m.add_constraint(x[2] + x[4] >= 1)
```

Written by Gabriel R.

```
m.add_constraint(x[3] + x[4] >= 1)
m.add_constraint(x[5] + x[6] >= 1)

# Solve and print results
m.print_information()
m.export_as_lp('./emergency.lp')
solution = m.solve()

if solution:
    m.print_solution()
    for i in I:
        print(f"{x[i].solution_value:.2f} is the quantity of product {i}")
else:
    print("No solution found.")
    print("Solve status:", m.get_solve_status())
```

Exploiting the fact this is all of a matrix of coverage, we can further minimize the model such as:

```
from docplex.mp.model import Model

# Data
I = range(1, 7)  # Set of potential locations {1, 2, ..., 6}

# Coverage matrix: zones covered by each location
coverage = {
    1: [1, 2],
    2: [1, 2, 6],
    3: [3, 4],
    4: [3, 4, 5],
    5: [4, 5, 6],
    6: [2, 5, 6]
}

# Create the model
m = Model(name="Emergency Location")

# Decision variables
x = m.binary_var_dict(I, name="x")

# Objective function: minimize the number of open locations
m.minimize(m.sum(x[i] for i in I))

# Constraints: ensure each zone is covered
for zone in range(1, 7):
    m.add_constraint(
        m.sum(x[i] for i in I if zone in coverage[i]) >= 1,
        ctname=f"cover_zone_{zone}"
    )

# Solve the model
solution = m.solve()

# Print results
if solution:
    print("Optimal solution found:")
    for i in I:
        if x[i].solution_value > 0.5:
            print(f"Open emergency service at location {i}")
    print(f"Total number of locations: {solution.objective_value}")
else:
    print("No solution found")
    print("Solve status:", m.get_solve_status())

# Optional: export the model to an LP file
m.export_as_lp("emergency_location.lp")
```

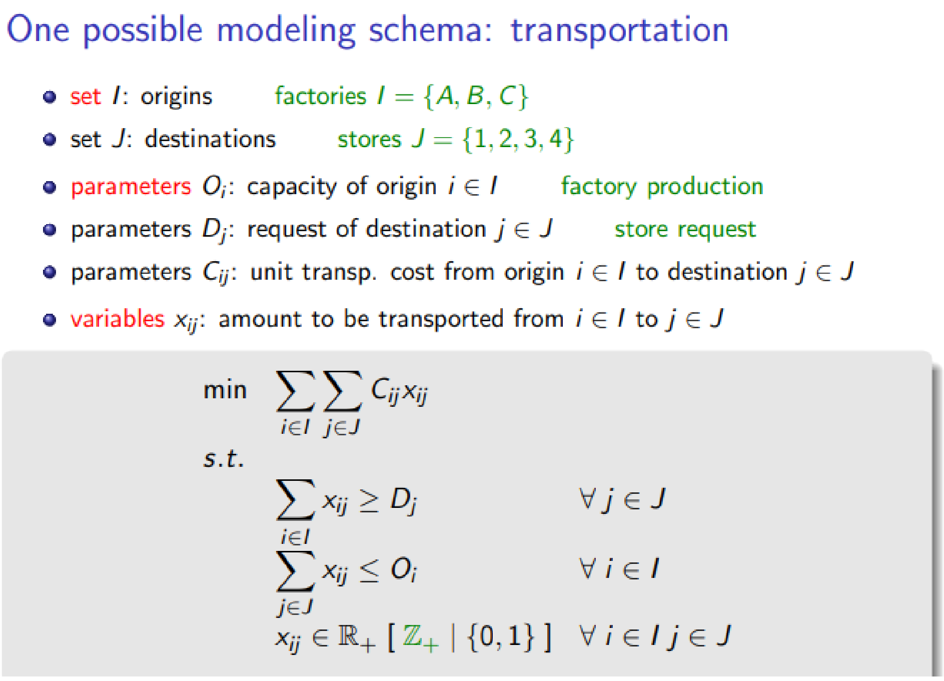
The professor used basically a similar previous approach:

```
from docplex.mp.model import Model

# Data sets
I = [1, 2, 3, 4, 5, 6]  # zones as locations
J = [1, 2, 3, 4, 5, 6]  # zones as people

# Data parameters
C = {i: 1 for i in I}  # Cost of opening a facility at location i
P = {j: 1 for j in J}  # Population (or importance) of zone j

# Coverage matrix A
A = {
    (1,1): 1, (1,2): 1, (1,3): 0, (1,4): 0, (1,5): 0, (1,6): 0,
    (2,1): 1, (2,2): 1, (2,3): 0, (2,4): 0, (2,5): 0, (2,6): 1,
    (3,1): 0, (3,2): 0, (3,3): 1, (3,4): 1, (3,5): 0, (3,6): 0,
    (4,1): 0, (4,2): 0, (4,3): 1, (4,4): 1, (4,5): 1, (4,6): 0,
    (5,1): 0, (5,2): 0, (5,3): 0, (5,4): 1, (5,5): 1, (5,6): 1,
    (6,1): 0, (6,2): 1, (6,3): 0, (6,4): 0, (6,5): 1, (6,6): 1
}

# Create the model
m = Model(name="Emergency Location")

# Decision variables
x = m.binary_var_dict(I, name="x")
```

```
# Objective function: minimize the number of open locations
m.minimize(m.sum(C[i] * x[i] for i in I))

# Constraints: ensure each zone is covered
for j in J:
    m.add_constraint(
        m.sum(A[i,j] * x[i] for i in I) >= 1,
        ctname=f"cover_zone_{j}"
    )

# Solve the model
solution = m.solve()

# Print results
if solution:
    print("OPTIMAL SOLUTION:")
    for i in I:
        if x[i].solution_value > 0.5:
            print(f"1: Open emergency service at location {i}")
        else:
            print(f"0: Do not open emergency service at location {i}")
    print(f"Total cost: {solution.objective_value}")
else:
    print("No solution found")
    print("Solve status:", m.get_solve_status())

# Optional: export the model to an LP file
m.export_as_lp("emergency_location_detailed.lp")
```

15 Laboratory 3 – Transportation and domains constraints

We are going to implement the following schema:



One possible modeling schema: transportation

- set I: origins factories $I = \{A, B, C\}$
- set J: destinations stores $J = \{1, 2, 3, 4\}$
- parameters O_i: capacity of origin $i \in I$ factory production
- parameters D_j: request of destination $j \in J$ store request
- parameters C_{ij}: unit transp. cost from origin $i \in I$ to destination $j \in J$
- variables x_{ij}: amount to be transported from $i \in I$ to $j \in J$

$$
\begin{array}{lll}
\min & \sum_{i \in I} \sum_{j \in J} C_{ij} x_{ij} & \\
s.t. & & \\
 & \sum_{i \in I} x_{ij} \geq D_j & \forall\, j \in J \\
 & \sum_{j \in J} x_{ij} \leq O_i & \forall\, i \in I \\
 & x_{ij} \in \mathbb{R}_+ \; [\, \mathbb{Z}_+ \mid \{0, 1\} \,] & \forall\, i \in I\; j \in J
\end{array}
$$

The complete implementation of the model is:

```
from docplex.mp.model import Model
import json
import ast # for string to dictionary conversion

# we define only the sets, which will be filled with data later
I = [] # set of origins
J = [] # set of destinations

O = {} # set of origins
D = {} # set of destinations
C = {(i,j): 1 for i in I for j in J} # cost of transportation from origin i to destination j
decision_domain = ""

# create one model instance, with a name
m = Model(name='transportation')

if decision_domain == "discrete":
    # dictionary with two indices
    x = m.binary_var_dict(keys1=I, keys2=J, name="x", ub = None, name = "xTR") # transportation quantities from origin i to destination j
elif decision_domain == "cont":
    x = m.continuous_var_dict(keys1=I, keys2=J, name="x", ub = None, name = "xTR")
else:
    x = m.binary_var_dict(keys1=I, keys2=J, name="x", ub = None, name = "xTR")
```

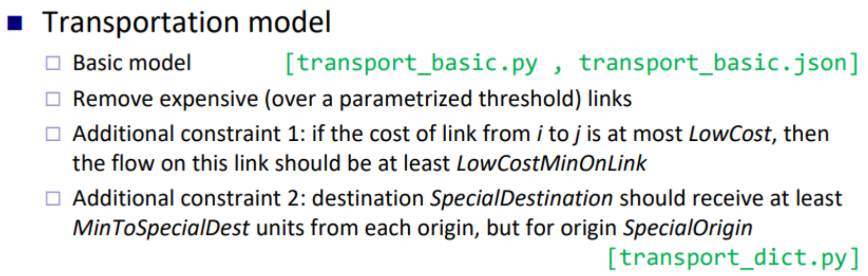
```
# define the objective function
m.minimize(m.sum(C[i,j] * x[i,j] for i in I for j in J))

# constraint for destinations
for j in J:
    m.add_constraint(m.sum(x[i,j] for i in I) >= D[j], ctname="destination_%s"%j)

# constraint for origins
for i in I:
    m.add_constraint(m.sum(x[i,j] for j in J) <= O[i], ctname="origin_%s"%i)

# solve the model
print(m.export_to_string())
m.solve()

# print solution
for i in I:
    for j in J:
        print("x(%s, %s) = %d" % (i, j, x[i,j].solution_value))
```

Then, we import data from a file and then it is being used as control, with variables getting values from files:

```
from docplex.mp.model import Model
import json
import ast # for string to dictionary conversion

# read the input data from the file
with open('refr.json') as f:
    data = json.load(f)

# we define only the sets, which will be filled with data later
I = data["I"] # set of origins
J = data["J"] # set of destinations

o_list = data["o_list"]
O = {I[i]: o_list[i] for i in range(len(I))} # set of origins, read element by element

d_list = data["d_list"]
D = {J[j]: d_list[j] for j in range(len(J))} # set of destinations, read element by element

# read the cost matrix
c_matrix = data["c_matrix"]
C = {(I[i],I[j]): c_matrix[i][j] for i in range(len(I)) for j in range(len(I))}
# indexing by the name and not the position

decision_domain = ""
```

```
# create one model instance, with a name
m = Model(name='transportation')

if decision_domain == "discrete":
  # dictionary with two indices
  x = m.binary_var_dict(keys1=I, keys2=J, name="x", ub = None, name = "xTR") # transportation quantities from origin i to destination j
elif decision_domain == "cont":
  x = m.continuous_var_dict(keys1=I, keys2=J, name="x", ub = None, name = "xTR")
else:
  x = m.binary_var_dict(keys1=I, keys2=J, name="x", ub = None, name = "xTR")

# define the objective function
m.minimize(m.sum(C[i,j] * x[i,j] for i in I for j in J))

# constraint for destinations
for j in J:
  m.add_constraint(m.sum(x[i,j] for i in I) >= D[j], ctname="destination_%s"%j)

# constraint for origins
for i in I:
  m.add_constraint(m.sum(x[i,j] for j in J) <= O[i], ctname="origin_%s"%i)

# solve the model
print(m.export_to_string())
m.solve()

# print solution
for i in I:
  for j in J:
    print("x(%s, %s) = %d" % (i, j, x[i,j].solution_value))
```

Now try to satisfy the model with additional constraints:



- Transportation model
  - □ Basic model `[transport_basic.py , transport_basic.json]`
  - □ Remove expensive (over a parametrized threshold) links
  - □ Additional constraint 1: if the cost of link from *i* to *j* is at most *LowCost*, then the flow on this link should be at least *LowCostMinOnLink*
  - □ Additional constraint 2: destination *SpecialDestination* should receive at least *MinToSpecialDest* units from each origin, but for origin *SpecialOrigin* `[transport_dict.py]`

Now, starting from the JSON file:

```json
{
  "modelname" : "move refrigerators",
  "decision_domain": "discrete",

  "___comment01": "sets of origin factories I and destination stores J (a set is given as ordered list = array)",
  "I": ["A","B","C"],
  "J": [1,2,3,4],

  "___comment02": "arrays of factories' capacity and stores' request (follow the order in the related sets)",
  "o_list": [50,70,30],
  "d_list": [20,60,30,40],

   "___comment03": "factory-to-store cost matrix (matrix indexes follow the order in the related sets)",
  "c_matrix": [
          [6,8,3,2],
          [4,2,1,3],
          [4,2,6,5]
        ],
  "cost_threshold": 0.5,
  "low_cost": 2,
  "low_cost_min": 1
}
```

We write the entire file transport_py:

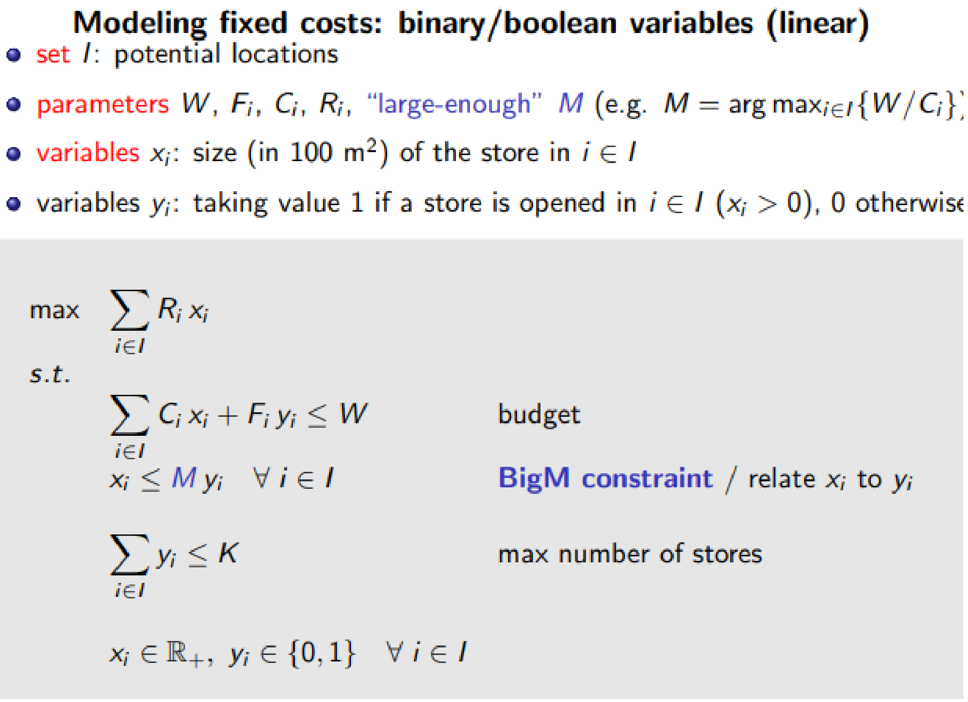
```python
from docplex.mp.model import Model
import json
import ast # for string to dictionary conversion

# read the input data from the file
with open('refr.json') as f:
  data = json.load(f)

# we define only the sets, which will be filled with data later
I = data["I"] # set of origins
J = data["J"] # set of destinations

o_list = data["o_list"]
O = {I[i]: o_list[i] for i in range(len(I))} # set of origins, read element by element

d_list = data["d_list"]
D = {J[j]: d_list[j] for j in range(len(J))} # set of destinations, read element by element

# read the cost matrix
c_matrix = data["c_matrix"]
```

```
C = {(I[i],I[j]): c_matrix[i][j] for i in range(len(I)) for j in range(len(I))}
# indexing by the name and not the position

# OD = origin-destination pairs
cost_threshold_percent = data["cost_threshold_percent"]
ActiveODPairs = [(i,j) for i in I for j in J if C[i,j] <= cost_threshold_percent * max (C[o,d] for o in I for d in J)]

decision_domain = ""

# create one model instance, with a name
m = Model(name='transportation')

if decision_domain == "discrete":
  x = m.integer_var_dict(keys = ActiveODPairs, name="x", lb = 0, ub = None, name = "XD")
elif decision_domain == "cont":
  x = m.continuous_var_dict(keys = ActiveODPairs, name="x", lb = 0, ub = None, name = "XC")
else:
  x = m.binary_var_dict(keys = ActiveODPairs, name="x", name = "XB")

# define the objective function
m.minimize(m.sum(C[i,j] * x[i,j] for i,j in ActiveODPairs))

# constraint for destinations
for j in J:
  m.add_constraint(m.sum(x[i,j] for i in I if(i,j) in ActiveODPairs) >= D[j], ctname="destination_%s"%j)

m.add_constraints(m.sum(x[i,j] for j in J if(i,j) in ActiveODPairs) <= O[i] for i in I)

L = data["LowCost"]
T = data["LowCostMinOnLink"]

m.add_constraints(x[i,j] >= L[i,j] for i,j in ActiveODPairs if C[i,j] <= T)
#x_ij >= L if c_ij <= T, forall i,j in act

sD = data["SpecialDestination"]
sO = data["SpecialOrigin"]
minSD = data["MinSpecialDestination"]
# destination SpecialDestination should receive at least MinToSpecialDest units from each origin, but for origin SpecialOrigin

# x_i, sD >= minSD forall i in I \ {sO}
m.add_constraints(x[i,sD] >= minSD for i in I if i != sO and (i,sD) in ActiveODPairs) # this means that the pair (i,sD) is active

m.print_information()
m.export_as_lp(path='transport.lp')

# solve the model
```

```
m.solve()

# print the solution

for i,j in ActiveODPairs:
   print("x_%s_%s = %d" % (i, j, x[i,j].solution_value))
```

16 Laboratory 4 – Fixed costs model and Efficient structures

We want to implement this model:



Modeling fixed costs: binary/boolean variables (linear)

- set I: potential locations
- parameters W, F_i, C_i, R_i, "large-enough" M (e.g. $M = \arg\max_{i \in I}\{W/C_i\}$
- variables x_i: size (in 100 m^2) of the store in $i \in I$
- variables y_i: taking value 1 if a store is opened in $i \in I$ ($x_i > 0$), 0 otherwise

$$\max \sum_{i \in I} R_i\, x_i$$

s.t.

$$\sum_{i \in I} C_i\, x_i + F_i\, y_i \le W \qquad \text{budget}$$

$$x_i \le M\, y_i \quad \forall\, i \in I \qquad \text{BigM constraint / relate } x_i \text{ to } y_i$$

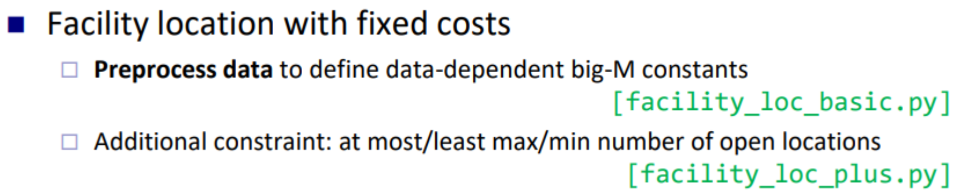
$$\sum_{i \in I} y_i \le K \qquad \text{max number of stores}$$

$$x_i \in \mathbb{R}_+,\; y_i \in \{0,1\} \quad \forall\, i \in I$$

We use the following JSON file:

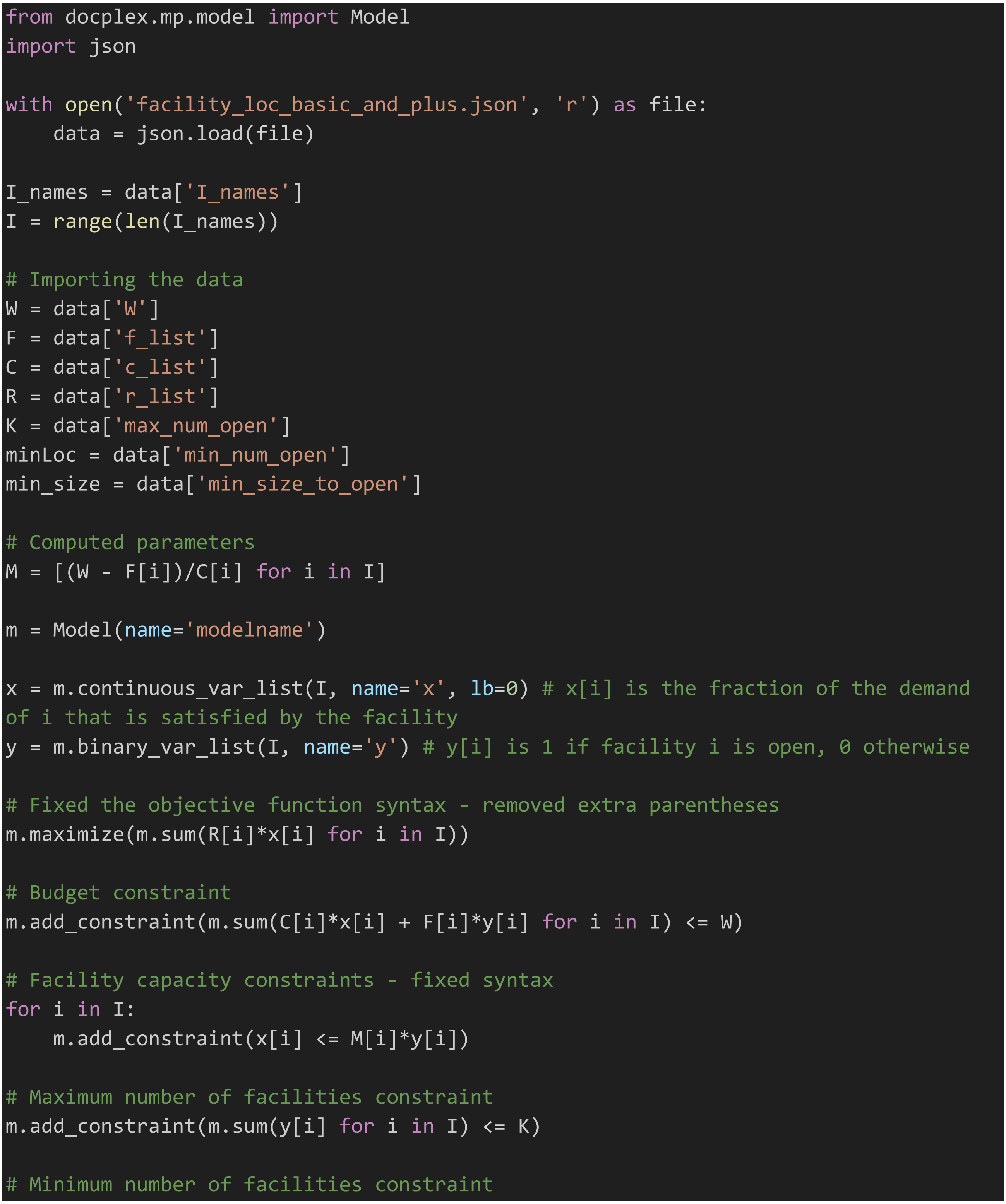
```
{
  "modelname" : "fixed cost location",

  "___comment01": "set of potential locations' names. Each location is identified
by its position, first position is 0",
  "I_names":
["loc0","loc1","loc2","loc3","loc4","loc5","loc6","loc7","loc8","loc9"],

  "___comment02": "available budget",
  "W": 1e6,

  "___comment03": "lists of fixed costs, variable costs and revenues for each
location (euro per 100sqm)",
  "f_list": [1000,1210,2000,1500,1350,1560,1450,2100,1720,1110],
  "c_list": [ 250, 230, 190, 210, 200, 210, 260, 255, 220, 270],
  "r_list": [3000,4000,6600,5000,6000,6500,3500,2500,2600,4700],

  "___comment04": "PLUS: maximum and minimum number of open locations and their
minimum extension",
  "max_num_open": 5,
  "min_num_open": 3,
  "min_size_to_open": 15
}
```

The model we are trying to implement is present here:



- Facility location with fixed costs
  - □ **Preprocess data** to define data-dependent big-M constants [facility_loc_basic.py]
  - □ Additional constraint: at most/least max/min number of open locations [facility_loc_plus.py]

The actual model follows:

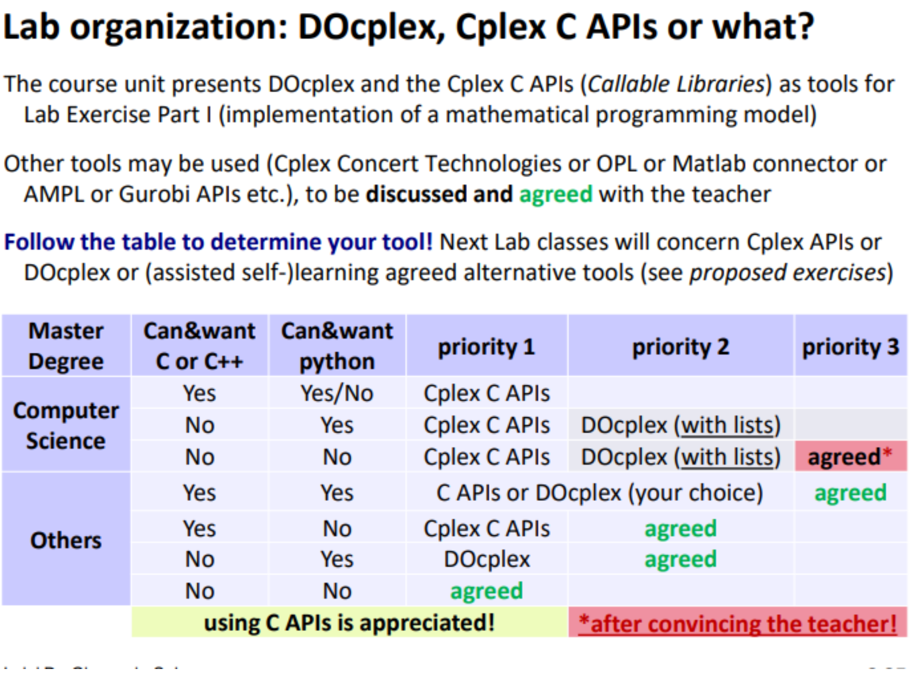


```
from docplex.mp.model import Model
import json

with open('facility_loc_basic_and_plus.json', 'r') as file:
    data = json.load(file)

I_names = data['I_names']
I = range(len(I_names))

# Importing the data
W = data['W']
F = data['f_list']
C = data['c_list']
R = data['r_list']
K = data['max_num_open']
minLoc = data['min_num_open']
min_size = data['min_size_to_open']

# Computed parameters
M = [(W - F[i])/C[i] for i in I]

m = Model(name='modelname')

x = m.continuous_var_list(I, name='x', lb=0) # x[i] is the fraction of the demand of i that is satisfied by the facility
y = m.binary_var_list(I, name='y') # y[i] is 1 if facility i is open, 0 otherwise

# Fixed the objective function syntax - removed extra parentheses
m.maximize(m.sum(R[i]*x[i] for i in I))

# Budget constraint
m.add_constraint(m.sum(C[i]*x[i] + F[i]*y[i] for i in I) <= W)

# Facility capacity constraints - fixed syntax
for i in I:
    m.add_constraint(x[i] <= M[i]*y[i])

# Maximum number of facilities constraint
m.add_constraint(m.sum(y[i] for i in I) <= K)

# Minimum number of facilities constraint
```

```
m.add_constraint(m.sum(y[i] for i in I) >= minLoc)

# Minimum size constraints
for i in I:
    m.add_constraint(x[i] >= min_size/100 * y[i])  # Convert min_size from sqm to
fraction

m.print_information()

if m.solve():
    m.print_solution(print_zeros=True)
    citytoshow = "loc01"
    posOfCity = I_names.index(citytoshow)
    print("the size of", citytoshow, "is", m.solution[x[posOfCity]])
else:
    print("no sol", m.get_solve_status())
```

Remember that operations done here should use the correct data structure (which are not dictionaries, indexes, etc.), considering evaluating solutions might be very time-consuming. So, moral of the story: Python is not good for efficiency.

Herem the first part of the exercise is implementing a model, while the second part is implementing heuristics. For example, Computer Scientists are required basically to use C++, according to the following:



Lab organization: DOcplex, Cplex C APIs or what?

The course unit presents DOcplex and the Cplex C APIs (*Callable Libraries*) as tools for Lab Exercise Part I (implementation of a mathematical programming model)

Other tools may be used (Cplex Concert Technologies or OPL or Matlab connector or AMPL or Gurobi APIs etc.), to be **discussed and** agreed with the teacher

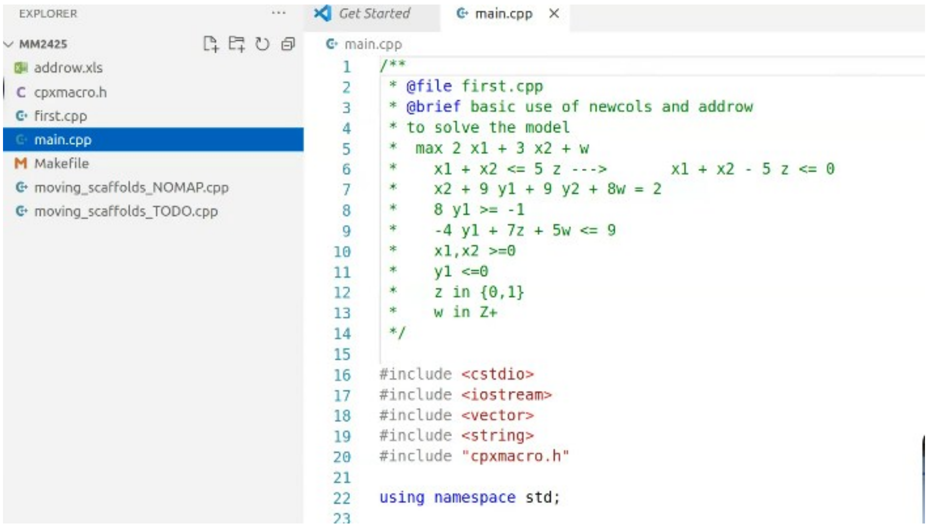
Follow the table to determine your tool! Next Lab classes will concern Cplex APIs or DOcplex or (assisted self-)learning agreed alternative tools (see *proposed exercises*)

Master Degree	Can&want C or C++	Can&want python	priority 1	priority 2	priority 3
Computer Science	Yes	Yes/No	Cplex C APIs		
	No	Yes	Cplex C APIs	DOcplex (with lists)	
	No	No	Cplex C APIs	DOcplex (with lists)	**agreed***
Others	Yes	Yes	C APIs or DOcplex (your choice)		agreed
	Yes	No	Cplex C APIs	agreed	
	No	Yes	DOcplex	agreed	
	No	No	agreed		
	using C APIs is appreciated!			*after convincing the teacher!	

17 LABORATORY 5 – CPLEX APIS – INTRO, CONSTRAINTS AND MODEL EXAMPLE

Note: for people using Windows, you have to use Visual Studio 2022, with C++ installed. In this case, you will execute everything safely, using also this chapter as entire reference.

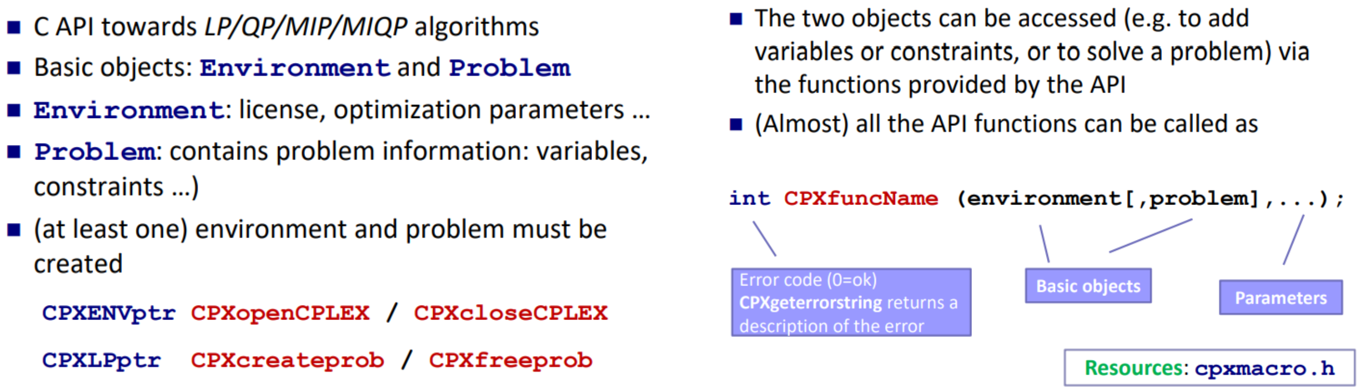
The professor provided us with an example folder, complete with a file called “cpxmacro.h” and also some other files useful for compilation. It’s necessary to have a main() method, using for example something like this:



```
/**
 * @file first.cpp
 * @brief basic use of newcols and addrow
 * to solve the model
 *  max 2 x1 + 3 x2 + w
 *     x1 + x2 <= 5 z --->       x1 + x2 - 5 z <= 0
 *     x2 + 9 y1 + 9 y2 + 8w = 2
 *     8 y1 >= -1
 *     -4 y1 + 7z + 5w <= 9
 *     x1,x2 >=0
 *     y1 <=0
 *     z in {0,1}
 *     w in Z+
 */

#include <cstdio>
#include <iostream>
#include <vector>
#include <string>
#include "cpxmacro.h"

using namespace std;
```

Note: consider the existing *Italian* notes (found it myself and put on MEGA); on this part, they are made really well.

The following are basic objects which are to be used inside of Cplex files:



- C API towards *LP/QP/MIP/MIQP* algorithms
- Basic objects: **Environment** and **Problem**
- **Environment**: license, optimization parameters …
- **Problem**: contains problem information: variables, constraints …)
- (at least one) environment and problem must be created

  CPXENVptr CPXopenCPLEX / CPXcloseCPLEX

  CPXLPptr CPXcreateprob / CPXfreeprob

- The two objects can be accessed (e.g. to add variables or constraints, or to solve a problem) via the functions provided by the API
- (Almost) all the API functions can be called as

int CPXfuncName (environment[,problem],...);

Consider the Cplex APIs are inside of the file “cpxmacro.h”, present inside of the directory structure of the professor. The following is its content:

```
/**
 * @file cpx_macro.h
 * Cplex Helper Macros
 *
 */

#ifndef CPX_MACRO_H
```

```
#define CPX_MACRO_H

#include <cstring>
#include <string>
#include <stdexcept>
#include <ilcplex/cplex.h>

#define STRINGIZE(something) STRINGIZE_HELPER(something)
#define STRINGIZE_HELPER(something) #something

/**
 * typedefs of basic Callable Library entities,
 * i.e., environment (Env) and problem pointers (Prob).
 */

typedef CPXENVptr Env;
typedef CPXCENVptr CEnv;
typedef CPXLPptr Prob;
typedef CPXCLPptr CProb;

/* Cplex Error Status and Message Buffer */

extern int status;

const unsigned int BUF_SIZE = 4096;

extern char errmsg[BUF_SIZE];

/* Shortcut for declaring a Cplex Env */
#define DECL_ENV(name) \
Env name = CPXopenCPLEX(&status);\
if (status){\
    CPXgeterrorstring(NULL, status, errmsg);\
    int trailer = std::strlen(errmsg) - 1;\
    if (trailer >= 0) errmsg[trailer] = '\0';\
    throw std::runtime_error(std::string(__FILE__) + ":" + STRINGIZE(__LINE__) + ":
" + errmsg);\
}

/* Shortcut for declaring a Cplex Problem */
#define DECL_PROB(env, name) \
Prob name = CPXcreateprob(env, &status, "");\
if (status){\
    CPXgeterrorstring(NULL, status, errmsg);\
    int trailer = std::strlen(errmsg) - 1;\
    if (trailer >= 0) errmsg[trailer] = '\0';\
    throw std::runtime_error(std::string(__FILE__) + ":" + STRINGIZE(__LINE__) + ":
" + errmsg);\
}
```

```
/* Make a checked call to a Cplex API function */
#define CHECKED_CPX_CALL(func, env, ...) do {\
status = func(env, __VA_ARGS__);\
if (status){\
    CPXgeterrorstring(env, status, errmsg);\
    int trailer = std::strlen(errmsg) - 1;\
    if (trailer >= 0) errmsg[trailer] = '\0';\
    throw std::runtime_error(std::string(__FILE__) + ":" + STRINGIZE(__LINE__) + ":
" + errmsg);\
} \
} while(false)

#endif /* CPX_MACRO_H */
```

As you can see, it allows you to call Cplex with specific functions you will see soon. In particular, consider the CHECKED_CPX_CALL, which allows you to craft new constraints (ADD_ROWS) or add variables (ADD_COLS). Consider the following calls as examples to be used in general (this is the first exercise given by the professor, but is present so you can see how these functions work):

~~~~

1. First CHECKED_CPX_CALL - Creating x variables:
```cpp
CHECKED_CPX_CALL(CPXnewcols, env, lp, 1, &obj, &lb, &ub, &xtype, &xname);
// Parameters:
// env    - CPLEX environment pointer
// lp     - CPLEX problem pointer
// 1      - Number of columns (variables) to create (1 at a time)
// &obj   - Pointer to objective coefficient (0.0 for x variables as they don't
appear in objective)
// &lb    - Pointer to lower bound (0.0 for x variables)
// &ub    - Pointer to upper bound (CPX_INFBOUND as x ∈ R+)
// &xtype - Pointer to variable type ('C' for continuous)
// &xname - Pointer to variable name ("x_i_j")
```


2. Second CHECKED_CPX_CALL - Creating y variables:
```cpp
CHECKED_CPX_CALL(CPXnewcols, env, lp, 1, &obj, &lb, &ub, &ytype, &yname);
// Parameters:
// env    - CPLEX environment pointer
// lp     - CPLEX problem pointer
// 1      - Number of columns (variables) to create (1 at a time)
// &obj   - Pointer to objective coefficient (C[i][j] for y variables)
// &lb    - Pointer to lower bound (0.0 for y variables)
// &ub    - Pointer to upper bound (1.0 as y is binary)
// &ytype - Pointer to variable type ('B' for binary)
// &yname - Pointer to variable name ("y_i_j")
```


3. CHECKED_CPX_CALL for Constraint (10) - Flow Conservation:
~~~~

~~~
```cpp
CHECKED_CPX_CALL(CPXaddrows, env, lp, 0, 1, idx.size(), &rhs, &sense, &matbeg,
&idx[0], &coef[0], NULL, NULL);
// Parameters:
// env          - CPLEX environment pointer
// lp           - CPLEX problem pointer
// 0            - Number of new columns (0 as we're just adding constraints)
// 1            - Number of new rows (1 constraint at a time)
// idx.size()   - Number of nonzero coefficients in the constraint
// &rhs         - Pointer to right hand side value (1.0 for flow conservation)
// &sense       - Pointer to constraint sense ('E' for equality)
// &matbeg      - Pointer to beginning position in the constraint matrix (0)
// &idx[0]      - Pointer to array of variable indices in this constraint
// &coef[0]     - Pointer to array of coefficients for those variables
// NULL         - No new column names
// NULL         - No new row names
```
~~~

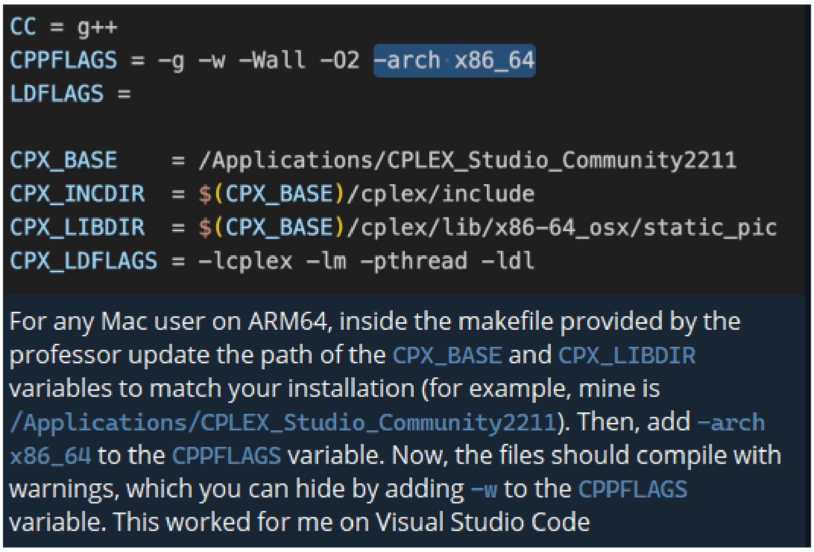
A problematic line is the following one:

#include <ilcplex/ilocplex.h>

This tells you that you have to include Cplex in order to make it work, it is the trickiest part.

Jump here or in case follow this video (both mine) to know more, particularly for Windows users.

- Linux users are only required to activate . cplex_env and you should be good to go
- Mac users might apply the following solution



```
CC = g++
CPPFLAGS = -g -w -Wall -O2 -arch x86_64
LDFLAGS =

CPX_BASE    = /Applications/CPLEX_Studio_Community2211
CPX_INCDIR  = $(CPX_BASE)/cplex/include
CPX_LIBDIR  = $(CPX_BASE)/cplex/lib/x86-64_osx/static_pic
CPX_LDFLAGS = -lcplex -lm -pthread -ldl
```

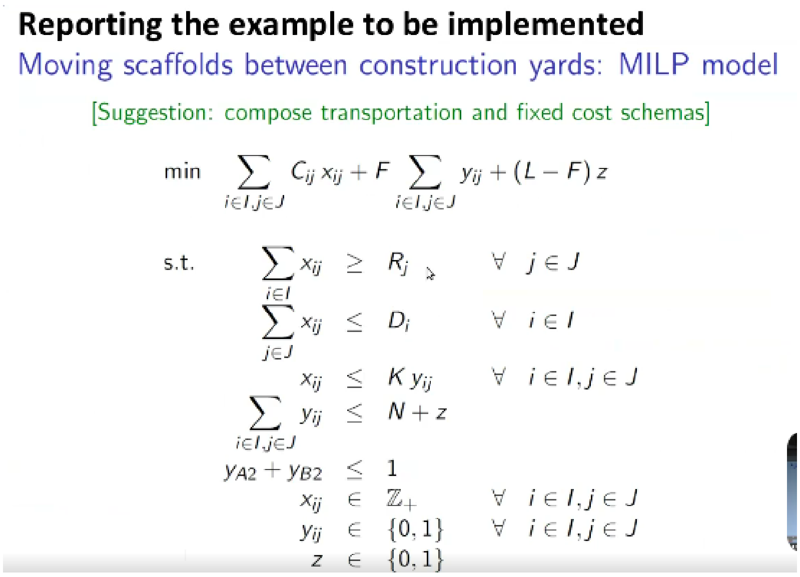
For any Mac user on ARM64, inside the makefile provided by the professor update the path of the CPX_BASE and CPX_LIBDIR variables to match your installation (for example, mine is /Applications/CPLEX_Studio_Community2211). Then, add -arch x86_64 to the CPPFLAGS variable. Now, the files should compile with warnings, which you can hide by adding -w to the CPPFLAGS variable. This worked for me on Visual Studio Code

To run the code examples in this guide, you will need:

- Cplex Optimization Studio installed with a valid license
- A C++ compiler (e.g. GCC)

- Cplex included library paths configured in your development environment. For detailed instructions on setting up Cplex and configuring your environment, refer to the official Cplex documentation.

Coming back to the actual lab, the model to be implemented here is the following:



Reporting the example to be implemented

Moving scaffolds between construction yards: MILP model

[Suggestion: compose transportation and fixed cost schemas]

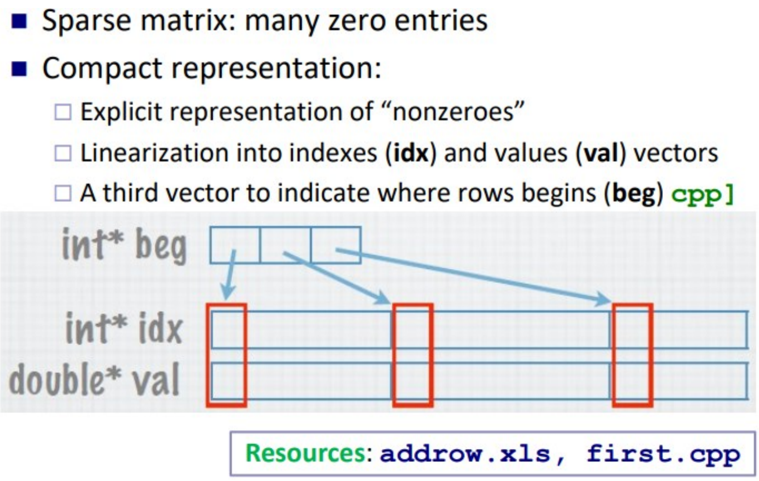
$$
\begin{array}{lrcll}
\min & \multicolumn{4}{l}{\sum_{i \in I, j \in J} C_{ij}\, x_{ij} + F \sum_{i \in I, j \in J} y_{ij} + (L - F)\, z} \\
\text{s.t.} & \sum_{i \in I} x_{ij} & \geq & R_j & \forall \; j \in J \\
 & \sum_{j \in J} x_{ij} & \leq & D_i & \forall \; i \in I \\
 & x_{ij} & \leq & K\, y_{ij} & \forall \; i \in I, j \in J \\
 & \sum_{i \in I, j \in J} y_{ij} & \leq & N + z & \\
 & y_{A2} + y_{B2} & \leq & 1 & \\
 & x_{ij} & \in & \mathbb{Z}_+ & \forall \; i \in I, j \in J \\
 & y_{ij} & \in & \{0, 1\} & \forall \; i \in I, j \in J \\
 & z & \in & \{0, 1\} &
\end{array}
$$

The objective is to minimize total cost while satisfying all supply, demand, and truck availability constraints. The linking constraints ensure that y[i][j] takes a value of 1 if there is any flow on the route from i to j.

A generic model is to be represented with *sparse matrices* since many elements can be evaluated to be zero. We will be using three main vectors here_

- val = values of matrix in a compact way
- idx = index of column which value is in the same position for vector "val"
- beg = indexes for "val" vector where matrix rows begin

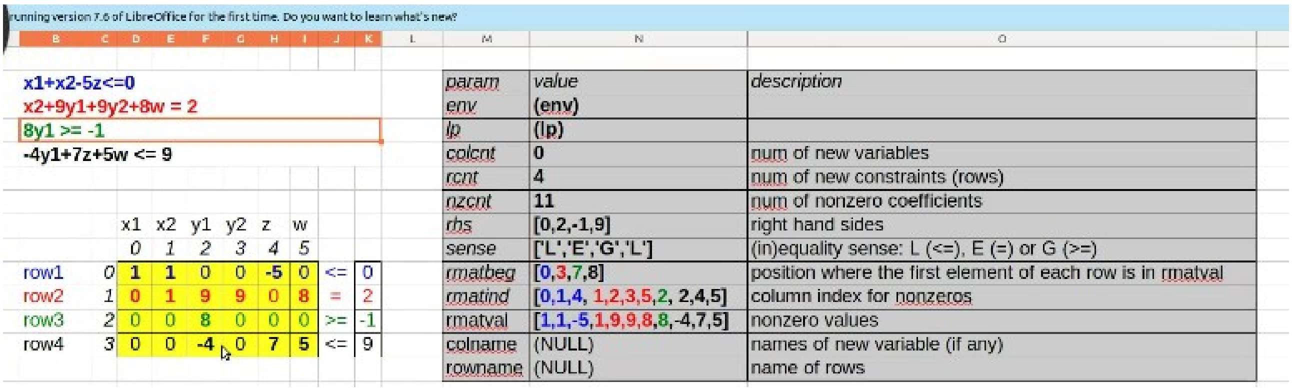
Consider normally you can have 100000 constraints and 2000000 variables, so many will be null; this requires a sparse matrix data structure, where normally most of the entries may be zero.



- Sparse matrix: many zero entries
- Compact representation:
  - □ Explicit representation of "nonzeroes"
  - □ Linearization into indexes (**idx**) and values (**val**) vectors
  - □ A third vector to indicate where rows begins (**beg**) `cpp]`

Resources: `addrow.xls, first.cpp`

Inside of the addrow.xls file, provided within the same folder between the examples, there is the representation to be used by both the model and the solution of the actual model:


running version 7.6 of LibreOffice for the first time. Do you want to learn what's new?
B
C
D
E
F
G
H
I
J
K
L
M
N
O

x1+x2-5z<=0
x2+9y1+9y2+8w = 2
8y1 >= -1
-4y1+7z+5w <= 9

		x1	x2	y1	y2	z	w		
		0	1	2	3	4	5		
row1	0	1	1	0	0	-5	0	<=	0
row2	1	0	1	9	9	0	8	=	2
row3	2	0	0	8	0	0	0	>=	-1
row4	3	0	0	-4	0	7	5	<=	9

param	value	description
env	(env)	
lp	(lp)	
colcnt	0	num of new variables
rcnt	4	num of new constraints (rows)
nzcnt	11	num of nonzero coefficients
rhs	[0,2,-1,9]	right hand sides
sense	['L','E','G','L']	(in)equality sense: L (<=), E (=) or G (>=)
rmatbeg	[0,3,7,8]	position where the first element of each row is in rmatval
rmatind	[0,1,4, 1,2,3,5,2, 2,4,5]	column index for nonzeros
rmatval	[1,1,-5,1,9,9,8,8,-4,7,5]	nonzero values
colname	(NULL)	names of new variable (if any)
rowname	(NULL)	name of rows

For example, looking at row1 in the matrix:

```
x1 + x2 - 5z <= 0

In sparse format:

rmatbeg[0] = 0   // Row 1 starts at position 0

rmatind = [0,1,4] // Column indices for x1,x2,z

rmatval = [1,1,-5] // Coefficient values
```

This representation:

- Saves memory by only storing nonzero elements
- Makes computation more efficient
- `beg` array lets you quickly find where each row starts
- `idx` and `val` arrays work in parallel to store location and value of nonzeros

Before moving on, let's clarify:

- Variables are to be invoked with CPXnewcols with following syntax:

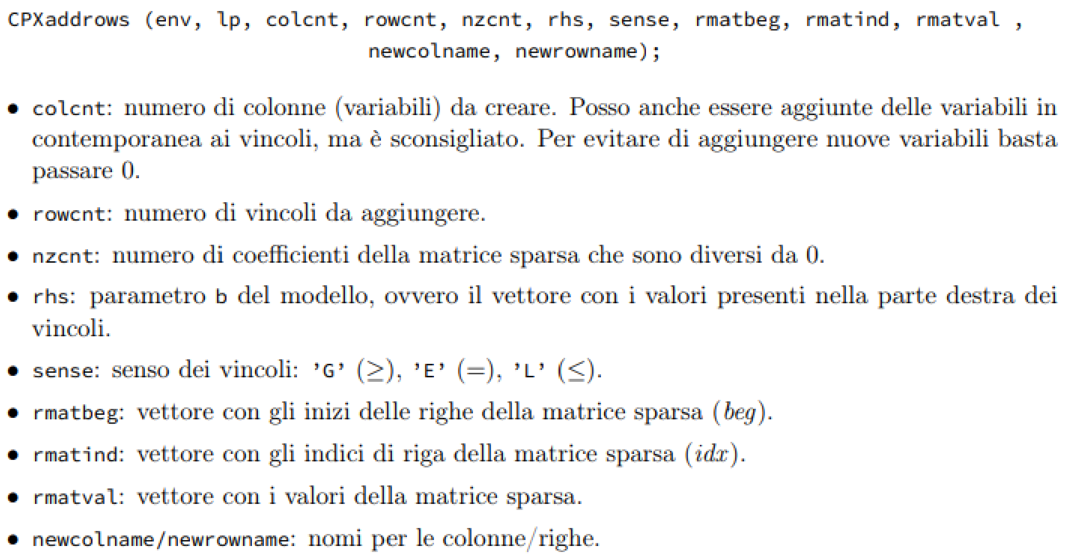
```
CPXnewcols (env, lp, ccnt, obj, lb, ub, xctype, colname);
```

- `env`: l'ambiente CPLEX;
- `lp`: il problema;
- `ccnt`: il numero di variabili da aggiungere al problema;
- `obj`: i coefficienti delle variabili all'interno della funzione obiettivo;
- `lb/ub`: i lower bound e upper bound del dominio delle variabili;
- `xctype`: i tipi delle variabili:
  - `'C'`: variabile continua (reale);
  - `'B'`: variabile binaria;
  - `'I'`: variabile intera.
- `colname`: nomi delle variabili. Se viene passato `NULL`, CPLEX assegnerà dei nomi di default.

As taken from the Italian notes, we say in summary:

- Environment
- Linear Problem
- Count of variables
- Objective value
- Lower Bound/Upper Bound
- Type of variables
- Name of variables

- Constraints are to be inserted with CPXaddrows



```
CPXaddrows (env, lp, colcnt, rowcnt, nzcnt, rhs, sense, rmatbeg, rmatind, rmatval ,
                                newcolname, newrowname);
```

- `colcnt`: numero di colonne (variabili) da creare. Posso anche essere aggiunte delle variabili in contemporanea ai vincoli, ma è sconsigliato. Per evitare di aggiungere nuove variabili basta passare 0.
- `rowcnt`: numero di vincoli da aggiungere.
- `nzcnt`: numero di coefficienti della matrice sparsa che sono diversi da 0.
- `rhs`: parametro `b` del modello, ovvero il vettore con i valori presenti nella parte destra dei vincoli.
- `sense`: senso dei vincoli: `'G'` ($\geq$), `'E'` ($=$), `'L'` ($\leq$).
- `rmatbeg`: vettore con gli inizi delle righe della matrice sparsa (*beg*).
- `rmatind`: vettore con gli indici di riga della matrice sparsa (*idx*).
- `rmatval`: vettore con i valori della matrice sparsa.
- `newcolname/newrowname`: nomi per le colonne/righe.

As taken from the Italian notes, we say in summary:

- Environment
- Linear Problem
- Number of columns (variables) to create
- Number of rows (constraints) to add
- Number of coefficients not zero (nz)
- Right hand side (so, after the sense sign)
- Sense (<, =, >)
    - Constraint greater than X
        - c1 > x (sense = G, rhs = x)
- Lower Bound/Upper Bound
- Starting values of the rows (beg)
- Starting rows for the index rows (idx)
- Names of columns/rows

The code to be examined is the following (commented below), which is the less efficient way, made to benchmark and to call the Cplex model once so to make it work:

```
/**
 * @file first.cpp
 * @brief basic use of newcols and addrow
 * to solve the model
 *  max 2 x1 + 3 x2 + w
 *    x1 + x2 <= 5 z --->       x1 + x2 - 5 z <=
0
 *    x2 + 9 y1 + 9 y2 + 8w = 2
 *    8 y1 >= -1
 *    -4 y1 + 7z + 5w <= 9
 *    x1,x2 >=0
 *    y1 <=0
 *    z in {0,1}
 *    w in Z+
 */

#include <cstdio>
#include <iostream>
#include <vector>
#include <string>
#include "cpxmacro.h"

using namespace std;


// error status and messagge buffer
int status;
char errmsg[BUF_SIZE];

int main (int argc, char const *argv[])
{
    try
    {
    ///////////////////////// init
    DECL_ENV( env );
    DECL_PROB( env, lp );

    ///////////////////////// create variables with newcols
    //
    //    status =      CPXnewcols (env, lp, ccnt, obj, lb, ub, xctype, colname);
    //
    // all variables will be created in an array and each variable will be
identified by the related index
    //  => we assume that variables are SORTED as ** x1, x2, y1, y2, z, w **
    int ccnt = 6;
    double objCost[6] = {   2.0, 3.0, 0.0, 0.0, 0.0, 1.0 };
      // cost in the objective function;
    double lb[6]      = {   0.0       , 0.0         , -CPX_INFBOUND,   -
CPX_INFBOUND, 0.0, 0.0          };
    double ub[6]      = { CPX_INFBOUND, CPX_INFBOUND,   0.0          , CPX_INFBOUND
, 1.0, CPX_INFBOUND };
      // variable lower and upper bounds
    char xtype[6]     = { 'C' , 'C' , 'C' , 'C' , 'B' , 'I' };
      // variable types 'C' or 'B' or 'I' ...
    char ** xname = NULL;
      // no names
    CHECKED_CPX_CALL( CPXnewcols, env, lp, ccnt, &objCost[0], &lb[0], &ub[0],
&xtype[0], xname );

    ///////////////////////// create constraints
    //
    //    status = CPXaddrows (env, lp, colcnt, rowcnt, nzcnt, rhs, sense, rmatbeg,
rmatind, rmatval , newcolname, newrowname);
    //
    int colcount = 0;
      // no new columns
    int rowcount = 4;
      // number of constraints
    int nzcnt = 11;
      // number of NON-ZERO coefficients
    double rhs[4] = {0,2,-1,9};
      // right-hand-sides: notice that constraints are rewritten so that NO
variable appears in the RHS!!!
    char sense[4] = {'L','E','G','L'};
      // constraint type 'L' or 'E' or 'G' ...
    // the coefficient matrix will be linearized in a vector and ONLY NON-ZERO
coefficients
    //  need to be stored: the following three vectors are used
    double rmatval[11] = { 1.0, 1.0, -5.0,   1.0, 9.0, 9.0, 8.0,     8.0,      -
4.0, 7.0, 5.0 };
      // the linearized vector of non-zero coefficients
    int rmatbeg[4]   = {0,3,7,8};
      // one element for each constraint (row), reporting the index where each row
of the coefficient matrix starts
    int rmatind[11]    = {0,1,4, 1,2,3,5, 2, 2,4,5};
      // one element for each zon-zero coefficient, reporting its column index
        char ** newcolnames = NULL;
        char ** rownames = NULL;
          // no names
        CHECKED_CPX_CALL( CPXaddrows, env, lp, colcount, rowcount, nzcnt, &rhs[0],
&sense[0], &rmatbeg[0], &rmatind[0], &rmatval[0], newcolnames , rownames );

    CHECKED_CPX_CALL( CPXchgobjsen, env, lp, CPX_MAX );
        // change to MAXimize (default is MINimize)

    ///////////////////////// print (debug)
    CHECKED_CPX_CALL( CPXwriteprob, env, lp, "first.lp", NULL );
    ///////////////////////// optimize
    CHECKED_CPX_CALL( CPXmipopt, env, lp );

    ///////////////////////// print
    double objval;
    CHECKED_CPX_CALL( CPXgetobjval, env, lp, &objval );
    // get the objective function value into objval
    std::cout << "Objval: " << objval << std::endl;
    int n = CPXgetnumcols(env, lp);
    // get the number of variables (columns) into n (simple routine, no need for
return status);
    /////// get the value of the variables using
    //
    //    status = CPXgetx (env, lp, varVals, fromIdx, toIdx);
    //
    std::vector<double> varVals;
    varVals.resize(n);
    int fromIdx = 0;
    int toIdx = n - 1;
    // we prepare a vector to get n values from index 0 to index n-1
    CHECKED_CPX_CALL( CPXgetx, env, lp, &varVals[0], fromIdx, toIdx );
      // get the value of the variables from index fromIdx to index toIdx into an
array having (toIdx - fromIdx + 1) open positions
    for ( int i = 0 ; i < n ; ++i ) {
      std::cout << "var in position " << i << " : " << varVals[i] << std::endl;
      /// to get variable name, use the RATHER TRICKY "CPXgetcolname"
      /// status = CPXgetcolname (env, lp, cur_colname, cur_colnamestore,
cur_storespace, &surplus, 0, cur_numcols-1);
    }
    double value_of_zed_var;
    CHECKED_CPX_CALL( CPXgetx, env, lp, &value_of_zed_var, 4, 4 );

    CHECKED_CPX_CALL( CPXsolwrite, env, lp, "first.sol" );
    // write the solution to a text file

    // free
    CPXfreeprob(env, &lp);
    CPXcloseCPLEX(&env);
    }
    catch(std::exception& e)
    {
        std::cout << ">>>EXCEPTION: " << e.what() << std::endl;
    }
    return 0;
}
```

Creating variables and constraints requires one step at a time, so we use something like the following, in which variables are called “columns”:

```
CPXnewcols(env, lp, ccnt, obj, lb, ub, xctype, colname)
// - ccnt: number of variables
// - obj: objective coefficients (Cij, F, L-F)
// - lb: lower bounds (0 for all)
// - ub: upper bounds (capacity K for xij, 1 for yij and z)
// - xctype: variable types ('C' for continuous xij, 'B' for binary yij and z)
```

In Cplex a single list of variables will be created each time, where each variable will be in the relative correct position.

Double vectors or maps are basically the same thing of using lists or dictionaries, then we map iteratively each variable in a straightforward way.

Now, one by one, we implement the constraints and the parts of the model – First, let's handle the y variables and map (similar to how x was handled):

```
/*MAP FOR y VARS: initial memory allocation for map vector*/
map_y.resize(I);
for ( int i = 0 ; i < I ; ++i ) {
    map_y[i].resize(J);
    for ( int j = 0 ; j < J ; ++j ) {
        map_y[i][j] = -1;
    }
}

// add y vars [in o.f.: ... + F sum{ij} y_ij + ... ]
for (int i = 0; i < I; i++) {
    for (int j = 0; j < J; j++) {
        if ( C[i][j] > od_cost_max ) continue; // EXT1

        char xtype = 'B'; // Binary variable
        double lb = 0.0;
        double ub = 1.0;
        snprintf(name, NAME_SIZE, "y_%c_%c", nameI[i], nameJ[j]);
        char* xname = (char*)(&name[0]);

        CHECKED_CPX_CALL( CPXnewcols, env, lp, 1, &F, &lb, &ub, &xtype,
&xname );


        map_y[i][j] = current_var_position++;

    }

}
```

Let’s go on completing the code, the objective function - min ∑ C_ij x_ij + F ∑ y_ij + (L-F)z:

```
// add z var [in o.f.: ... + (L-F) z ]

char xtype = 'B';  // Binary variable

double lb = 0.0;

double ub = 1.0;

double obj = L-F;  // Coefficient in objective function

snprintf(name, NAME_SIZE, "z");

char* xname = (char*)(&name[0]);

CHECKED_CPX_CALL( CPXnewcols, env, lp, 1, &obj, &lb, &ub, &xtype, &xname
);

map_z = current_var_position++;
```

Now the constraints match the mathematical model:

```
// add capacity constraints (origin) [ forall i, sum{j: x_ij exists} x_ij
<= D_j ]

for (int i = 0; i < I; i++) {

    std::vector<int> idx;

    std::vector<double> coef;

    char sense = 'L';

    for (int j = 0; j < J; j++) {

        if ( map_x[i][j] < 0 ) continue;


        idx.push_back(map_x[i][j]);

        coef.push_back(1.0);

    }

    int matbeg = 0;
    CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &D[i],
&sense, &matbeg, &idx[0], &coef[0], NULL, NULL );
}

// add linking constraints (x_ij - K y_ij <= 0)
for (int i = 0; i < I; i++) {
    for (int j = 0; j < J; j++) {
        if ( map_x[i][j] < 0 ) continue;

        std::vector<int> idx;
        std::vector<double> coef;
        double rhs = 0.0;
        char sense = 'L';

        idx.push_back(map_x[i][j]);
        coef.push_back(1.0);
        idx.push_back(map_y[i][j]);
        coef.push_back(-K);

        int matbeg = 0;
        CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &rhs,
&sense, &matbeg, &idx[0], &coef[0], NULL, NULL );
    }
}

// add counting constraint (sum_ij y_ij - z <= N)
{
    std::vector<int> idx;
    std::vector<double> coef;
    char sense = 'L';

    for (int i = 0; i < I; i++) {
        for (int j = 0; j < J; j++) {
            if ( map_y[i][j] < 0 || C[i][j] > od_cost_low ) continue;

            idx.push_back(map_y[i][j]);
            coef.push_back(1.0);
        }
    }
    idx.push_back(map_z);
    coef.push_back(-1.0);

    int matbeg = 0;
    CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &N, &sense,
&matbeg, &idx[0], &coef[0], NULL, NULL );
}

// add condition constraint (y_A2 + y_B2 <= 1)
{
    std::vector<int> idx;
    std::vector<double> coef;
    double rhs = 1.0;
    char sense = 'L';

    idx.push_back(map_y[0][1]); // A2
    coef.push_back(1.0);
    idx.push_back(map_y[1][1]); // B2
    coef.push_back(1.0);

    int matbeg = 0;
    CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &rhs,
&sense, &matbeg, &idx[0], &coef[0], NULL, NULL );
}
```

Finally, getting the solution values:

```
// print values of decision variables
```

```
std::vector<double> varVals;
varVals.resize(current_var_position);
CHECKED_CPX_CALL( CPXgetx, env, lp, &varVals[0], 0, current_var_position-1 );

// Print x variables
for (int i = 0; i < I; i++) {
    for (int j = 0; j < J; j++) {
        if (map_x[i][j] >= 0) {
            std::cout << "x_" << nameI[i] << "_" << nameJ[j] << " = "
                     << varVals[map_x[i][j]] << std::endl;
        }
    }
}

// Print y variables
for (int i = 0; i < I; i++) {
    for (int j = 0; j < J; j++) {
        if (map_y[i][j] >= 0) {
            std::cout << "y_" << nameI[i] << "_" << nameJ[j] << " = "
                     << varVals[map_y[i][j]] << std::endl;
        }
    }
}

// Print z variable
std::cout << "z = " << varVals[map_z] << std::endl;
```

18 Laboratory 6 – CPLEX APIs – Concluding Scaffolds Modeling

Continuing what we have established up to now, we want to add constraints to the previous model formulation.

- *Request/Demand Constraints*: For each destination j, the total amount transported from all origins must be at least the required demand R[j]. Mathematically: $\forall j \in J : \sum_{i=1}^{N} x_{ij} \geq R_j$

- *Capacity/Supply Constraints*: For each origin i, the total amount transported to all destinations cannot exceed the available supply D[i]. Mathematically: $\forall i \in I : \sum_{j=1}^{M} x_{ij} \leq D_i$

- *Linking Constraints*: If any amount is transported from origin i to destination j (i.e., if $x_{ij} > 0$), then the corresponding y_{ij} variable must be 1. This is enforced using a BigM constraint. Mathematically: $\forall i \in I, j \in J : x_{ij} - M \cdot y_{ij} \leq 0$

Here's how we implement these in our code using the CPLEX API's CPXaddrows function, in order to use all of the sets present, adding 1 constraint to the model for all parts of the vector following the model starting from 0 in the sparse matrix.

The point of using the vectors is to have all of the indices of coefficients of the corresponding variables, pushing back one after the other all of the variables involved:

```
// add request constraints (destinations)

for (int j = 0; j < J; j++) {

  std::vector<int> idx;

  std::vector<double> coef;

  char sense = 'G';

  for (int i = 0; i < I; i++) {

    if ( map_x[i][j] < 0 ) continue;

    idx.push_back(map_x[i][j]);

    coef.push_back(1.0);

  }

  int matbeg = 0;

  CHECKED_CPX_CALL( CPXaddrows, env, lp, 0, 1, idx.size(), &R[j], &sense,
&matbeg, &idx[0], &coef[0], NULL, NULL);

}


// add capacity constraints (origin)

for (int i = 0; i < I; i++) {
  std::vector<int> idx;

  std::vector<double> coef;

  char sense = 'L';

  for (int j = 0; j < J; j++) {

    if (map_x[i][j] < 0) continue;

    idx.push_back(map_x[i][j]);

    coef.push_back(1.0);

  }

  int matbeg = 0;

  CHECKED_CPX_CALL(CPXaddrows, env, lp, 0, 1, idx.size(), &D[i], &sense,
&matbeg, &idx[0], &coef[0], NULL, NULL);

}


// add linking constraints

for (int i = 0; i < I; i++) {

  for (int j = 0; j < J; j++) {

    if (map_x[i][j] < 0 || map_y[i][j] < 0) continue;

    std::vector<int> idx(2);

    std::vector<double> coef(2);

    char sense = 'L';

    idx[0] = map_x[i][j];

    idx[1] = map_y[i][j];

    coef[0] = 1.0;

    coef[1] = -K;

    double rhs = 0;

    int matbeg = 0;

    CHECKED_CPX_CALL(CPXaddrows, env, lp, 0, 1, idx.size(), &rhs, &sense,
&matbeg, &idx[0], &coef[0], NULL, NULL);

  }

}
```

The logic is similar to what we did for the other constraints:

- We loop through the relevant dimensions (origins i and destinations j)

- We build up idx and coef vectors with the variable indices and coefficients for the non-zero terms
- We specify the right-hand side value (R[j], D[i] or 0), sense ('G' for ≥, 'L' for ≤) and matrix start index matbeg
- We invoke CPXaddrows to add the fully constructed constraint to CPLEX

The only new aspect is handling the BigM constraint, where we need two terms in the expression x_{ij} - M·y_{ij}. We account for this by having idx and coef vectors of size 2 and setting their elements accordingly.

With this, our model implementation is complete! We can now invoke the solver, retrieve the solution, and interpret the results.

The code for solving the model remains the same as before:

```
// Optimize the model
CHECKED_CPX_CALL( CPXmipopt, env, lp );
// Get the solution status
int solstat = CPXgetstat(env, lp);
if (solstat == CPXMIP_OPTIMAL) {
  std::cout << "Optimal solution found!\n";
} else {
  std::cout << "No optimal solution found.\n";
  // Handle other statuses ...
}
```

CPXmipopt invokes the CPLEX solver on our mixed integer programming model. CPXgetstat retrieves the solution status, which we check to determine if an optimal solution was found.

Assuming a solution exists, we can query CPLEX for the variable values:

```
// Get the objective value
double objval;
CHECKED_CPX_CALL( CPXgetobjval, env, lp, &objval );
std::cout << "Objective value: " << objval << std::endl;


// Get the variable values
std::vector<double> xval(I*J);
CHECKED_CPX_CALL( CPXgetx, env, lp, &xval[0], 0, I*J-1 );
```

```
for (int i = 0; i < I; i++) {
  for (int j = 0; j < J; j++) {
    if ( map_x[i][j] >= 0 ) {
      std::cout << "x_" << nameI[i] << "_" << nameJ[j] << " = "
                << xval[map_x[i][j]] << std::endl;
    }
  }
}
// Similarly get y and z values ...
```

CPXgetobjval retrieves the value of the objective function at the optimal solution.

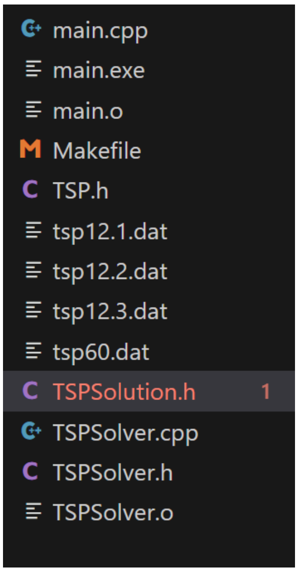
CPXgetx retrieves the values of the decision variables. Since CPLEX stores all variables in a single indexed array, we use our map_x to go from the logical 2D representation (origins i and destinations j) to the actual indices of the x variables. We then print out the non-zero x values.

The same approach can be used for the y and z variables. With this, we have completed the end-to-end process of implementing, solving and interpreting the results of an optimization model using the CPLEX APIs.

19 Laboratory 7 – Neighborhood Search for the Symmetric TSP (Tabu Search)

Note: for people using Windows: make is not natively installed – follow this one.

The goal is to implement a local search for the TSP (folder of the same name) with the following files (this comes from 0.skeleton) – differently from Cplex APIs lessons, here VS Code is enough:



main.cpp
main.exe
main.o
Makefile
TSP.h
tsp12.1.dat
tsp12.2.dat
tsp12.3.dat
tsp60.dat
TSPSolution.h 1
TSPSolver.cpp
TSPSolver.h
TSPSolver.o

We are trying to compile one file of the dat present like this:

```
PS C:\Users\roves\OneDrive\Documenti\GitHub\Computer-Science-
UniPD\Courses\Other elective\MeMoCO\Labs\Lab
7\l03.heur.ls.tsp\0.skeleton> make

g++ -g -Wall -O2 -c TSPSolver.cpp -o TSPSolver.o

TSPSolver.cpp: In member function 'bool TSPSolver::solve(const TSP&,
const TSPSolution&, TSPSolution&)':

TSPSolver.cpp:15:10: warning: unused variable 'iter' [-Wunused-variable]

   15 |      int  iter = 0;

      |           ^~~~

g++ -g -Wall -O2 -c main.cpp -o main.o

g++ -g -Wall -O2 TSPSolver.o main.o -o main

PS C:\Users\roves\OneDrive\Documenti\GitHub\Computer-Science-
UniPD\Courses\Other elective\MeMoCO\Labs\Lab
7\l03.heur.ls.tsp\0.skeleton> ./main tsp12.1.dat

number of nodes n = 12

### 0 8 2 7 10 4 9 1 6 5 11 3 0  ###

FROM solution: 0 8 2 7 10 4 9 1 6 5 11 3 0 (value : 135.7)

TO   solution: 0 8 2 7 10 4 9 1 6 5 11 3 0 (value : 135.7)
```

```
in 0.000942945 seconds (user time)

in 0.001 seconds (CPU time)
```

To avoid problems/long logs for people in labs/using Linux, comment line 14 inside of TSPSolution.h:

```
// #define uint unsigned int // for people in lab, comment it
```

Let's read for example the file:

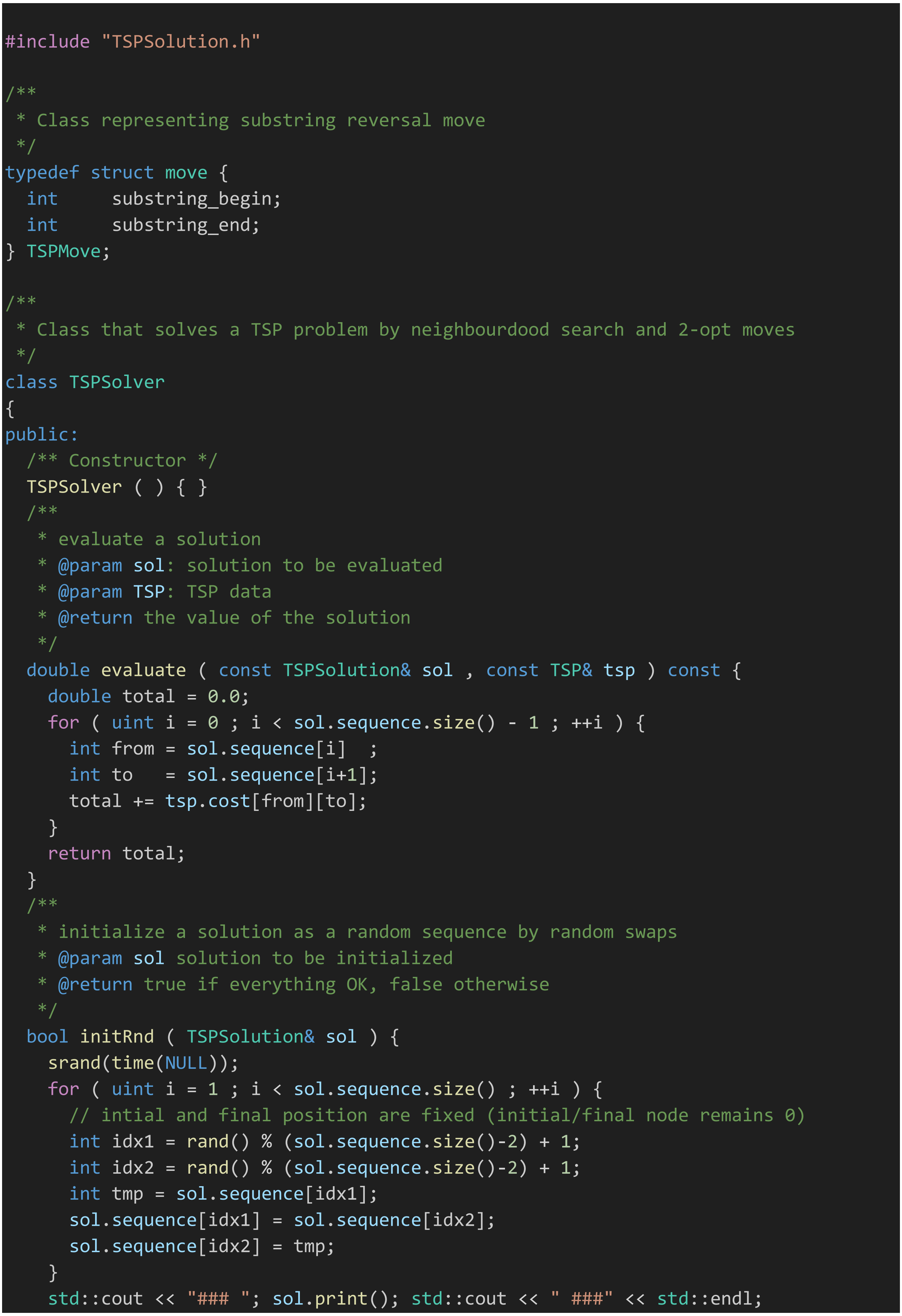
```
/**
* @file TSPSolution.h
* @brief TSP solution
*
*/

#ifndef TSPSOLUTION_H
#define TSPSOLUTION_H

#include <vector>

#include "TSP.h"

#define uint unsigned int // for people in lab, comment it

/**
* TSP Solution representation: ordered sequence of nodes (path representation)
*/
class TSPSolution
{
public:
  std::vector<int>    sequence;
public:
  /** Constructor
  * build a standard solution as the sequence <0, 1, 2, 3 ... n-1, 0>
  * @param tsp TSP instance
  * @return ---
  */
  TSPSolution( const TSP& tsp ) {
    sequence.reserve(tsp.n + 1);
    for ( int i = 0; i < tsp.n ; ++i ) {
      sequence.push_back(i);
    }
    sequence.push_back(0);
  }
  /** Copy constructor
  * build a solution from another
  * @param tspSol TSP solution
  * @return ---
  */
  TSPSolution( const TSPSolution& tspSol ) {
    sequence.reserve(tspSol.sequence.size());
    for ( uint i = 0; i < tspSol.sequence.size(); ++i ) {
      sequence.push_back(tspSol.sequence[i]);
    }
  }
public:
  /** print method
  * @param ---
  * @return ---
  */
  void print ( void ) {
    for ( uint i = 0; i < sequence.size(); i++ ) {
      std::cout << sequence[i] << " ";
    }
  }
  /** assignment method
  * copy a solution into another one
  * @param right TSP solution to get into
  * @return ---
  */
  TSPSolution& operator=(const TSPSolution& right) {
    // Handle self-assignment:
    if(this == &right) return *this;
    for ( uint i = 0; i < sequence.size(); i++ ) {
      sequence[i] = right.sequence[i];
    }
    return *this;
  }
};

#endif /* TSPSOLUTION_H */
```

This is the header file that defines the `TSPSolution` class and related structures. It contains:

- The sequence of the cities as a vector
- N+1 cities to have an Hamiltonian cycle

We'll also be commenting `TPSSolver.h`:

```
/**
 * @file TSPSolver.h
 * @brief TSP solver (neighborhood search)
 *
 */

#ifndef TSPSOLVER_H
#define TSPSOLVER_H

#include <vector>
```



```cpp
#include "TSPSolution.h"

/**
 * Class representing substring reversal move
 */
typedef struct move {
  int     substring_begin;
  int     substring_end;
} TSPMove;

/**
 * Class that solves a TSP problem by neighbourdood search and 2-opt moves
 */
class TSPSolver
{
public:
  /** Constructor */
  TSPSolver ( ) { }
  /**
   * evaluate a solution
   * @param sol: solution to be evaluated
   * @param TSP: TSP data
   * @return the value of the solution
   */
  double evaluate ( const TSPSolution& sol , const TSP& tsp ) const {
    double total = 0.0;
    for ( uint i = 0 ; i < sol.sequence.size() - 1 ; ++i ) {
      int from = sol.sequence[i]  ;
      int to   = sol.sequence[i+1];
      total += tsp.cost[from][to];
    }
    return total;
  }
  /**
   * initialize a solution as a random sequence by random swaps
   * @param sol solution to be initialized
   * @return true if everything OK, false otherwise
   */
  bool initRnd ( TSPSolution& sol ) {
    srand(time(NULL));
    for ( uint i = 1 ; i < sol.sequence.size() ; ++i ) {
      // intial and final position are fixed (initial/final node remains 0)
      int idx1 = rand() % (sol.sequence.size()-2) + 1;
      int idx2 = rand() % (sol.sequence.size()-2) + 1;
      int tmp = sol.sequence[idx1];
      sol.sequence[idx1] = sol.sequence[idx2];
      sol.sequence[idx2] = tmp;
    }
    std::cout << "### "; sol.print(); std::cout << " ###" << std::endl;
    return true;
  }
  /**
   * search for a good tour by neighbourhood search
   * @param TSP TSP data
   * @param initSol initial solution
   * @param bestSol best found solution (output)
   * @return true id everything OK, false otherwise
   */
  bool solve ( const TSP& tsp , const TSPSolution& initSol , TSPSolution& bestSol
);

protected:
  //TODO: declare here any "internal" method

#endif /* TSPSOLVER_H */
```

The move is represented by a substring reversal, where given a sequence you take a solution reversing a substring (taking X arcs and removing them so to understand the actual starting/ending position).

There is then the solution evaluation, starting from the o.f., starting from 0 and finishing into the second-last position, adding cost from each city. The solution is initialized starting from random swaps, so to create an instance of the TSP.

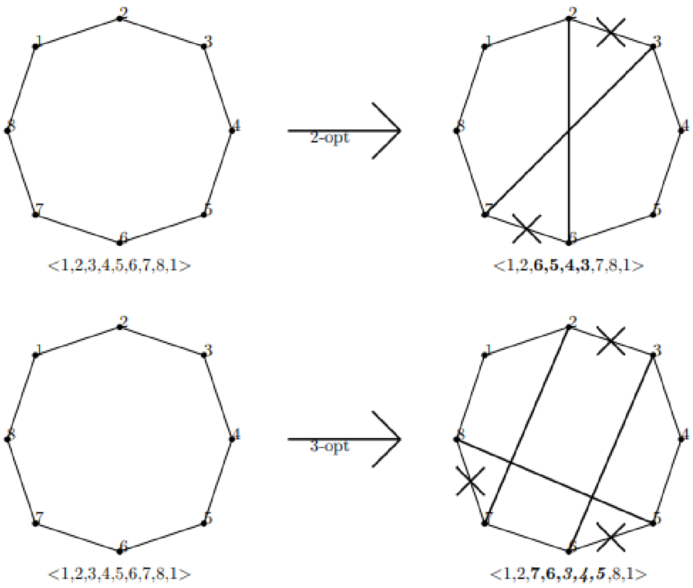
Then, we are commeting TSPSolver.cpp:

```
/**
 * @file TSPSolver.cpp
 * @brief TSP solver (neighborhood search)
 *
 */

#include "TSPSolver.h"
#include <iostream>

bool TSPSolver::solve ( const TSP& tsp , const TSPSolution& initSol , TSPSolution&
bestSol )
{
  try
  {
    bool stop = false;
    int  iter = 0;

    TSPSolution currSol(initSol);

    while ( ! stop ) {

      /// TODO: replace the following by the local search iteration
      //  that updates currSol if an improving neighbor exists and
      //  stops otherwise
      stop = true;

    }
    bestSol = currSol;
  }
  catch(std::exception& e)
  {
    std::cout << ">>>EXCEPTION: " << e.what() << std::endl;
    return false;
  }
  return true;
}

//TODO: "internal methods” if any
```

This is the implementation file containing the actual neighborhood search algorithm. Currently, the `solve()` method has a placeholder implementation that needs to be replaced with a complete local search iteration that:

1. Searches for improving neighbor solutions using 2-opt moves
2. Updates the current solution when a better neighbor is found
3. Stops when no improving neighbor exists

The main functionality that needs to be implemented here is the neighborhood exploration and solution improvement logic. The logic is not so simple, according to the professor; consider we have to swap each nodes by “k”, as you can see by the image here;



We would need some function to transform the current solution to a neighbor solution (take a better solution with the local search); one way to start is to write a small function into the solver, for example inside the protected part (present in the TSPSolver header file).

I propose the following implementation:

TSPSolver:

```
protected:
    /**
     * Performs a 2-opt move by reversing a subsequence
     * @param sol solution to modify
     * @param move the 2-opt move to perform
     * @return true if move was performed successfully
     */
    bool make2OptMove(TSPSolution& sol, const TSPMove& move) const;

    /**
     * Performs a 3-opt move by reversing two subsequences
     * @param sol solution to modify
     * @param move1 first subsequence to reverse
     * @param move2 second subsequence to reverse
     * @return true if move was performed successfully
     */
    bool make3OptMove(TSPSolution& sol, const TSPMove& move1, const TSPMove& move2)
const;

    /**
     * Evaluates the improvement of a 2-opt move without actually performing it
     * @param sol current solution
     * @param tsp problem instance
     * @param move the move to evaluate
     * @return cost improvement (negative if move improves solution)
     */
    double evaluate2OptMove(const TSPSolution& sol, const TSP& tsp, const TSPMove&
move) const;

    /**
     * Evaluates the improvement of a 3-opt move without actually performing it
     * @param sol current solution
     * @param tsp problem instance
     * @param move1 first move to evaluate
     * @param move2 second move to evaluate
     * @return cost improvement (negative if move improves solution)
     */
    double evaluate3OptMove(const TSPSolution& sol, const TSP& tsp,
                            const TSPMove& move1, const TSPMove& move2) const;
```

TSPSolver.cpp

```
bool TSPSolver::make2OptMove(TSPSolution& sol, const TSPMove& move) const {
    // Reverse subsequence from begin to end
    int i = move.substring_begin;
    int j = move.substring_end;
    while(i < j) {
        std::swap(sol.sequence[i], sol.sequence[j]);
        i++; j--;
    }
    return true;
}

bool TSPSolver::make3OptMove(TSPSolution& sol, const TSPMove& move1, const TSPMove&
move2) const {
    // Perform two subsequence reversals
    make2OptMove(sol, move1);
    make2OptMove(sol, move2);
    return true;
}

double TSPSolver::evaluate2OptMove(const TSPSolution& sol, const TSP& tsp, const
TSPMove& move) const {
    int i = move.substring_begin;
    int j = move.substring_end;

    // Calculate cost difference by looking at edges that would be removed/added
    double removed_cost = tsp.cost[sol.sequence[i-1]][sol.sequence[i]] +
                         tsp.cost[sol.sequence[j]][sol.sequence[j+1]];
    double added_cost = tsp.cost[sol.sequence[i-1]][sol.sequence[j]] +
                       tsp.cost[sol.sequence[i]][sol.sequence[j+1]];

    return added_cost - removed_cost;
}

double TSPSolver::evaluate3OptMove(const TSPSolution& sol, const TSP& tsp,
                                 const TSPMove& move1, const TSPMove& move2) const
{
    // Evaluate both moves combined
    TSPSolution temp_sol = sol;
    make2OptMove(temp_sol, move1);
    double improvement = evaluate2OptMove(temp_sol, tsp, move2);
    return improvement;
}
bool TSPSolver::solve(const TSP& tsp, const TSPSolution& initSol, TSPSolution&
bestSol) {
    try {
        TSPSolution currSol(initSol);
        double currCost = evaluate(currSol, tsp);
        bool improved = true;

        while(improved) {
            improved = false;

            // Try 2-opt moves first
            for(uint i = 1; i < currSol.sequence.size()-2 && !improved; i++) {
                for(uint j = i+1; j < currSol.sequence.size()-1; j++) {
                    TSPMove move = {(int)i, (int)j};
                    double improvement = evaluate2OptMove(currSol, tsp, move);

                    if(improvement < 0) {  // Improving move found
                        make2OptMove(currSol, move);
                        currCost += improvement;
                        improved = true;
                        break;
                    }
                }
            }

            // If no 2-opt improvement, try 3-opt moves
            if(!improved) {
                for(uint i = 1; i < currSol.sequence.size()-4 && !improved; i++) {
                    for(uint j = i+2; j < currSol.sequence.size()-2; j++) {
                        TSPMove move1 = {(int)i, (int)j};

                        // Try second reversal after first segment
                        for(uint k = j+1; k < currSol.sequence.size()-1; k++) {
                            TSPMove move2 = {(int)j+1, (int)k};
                            double improvement = evaluate3OptMove(currSol, tsp,
move1, move2);

                            if(improvement < 0) {  // Improving move found
                                make3OptMove(currSol, move1, move2);
                                currCost += improvement;
                                improved = true;
                                break;
                            }
                        }
                        if(improved) break;
                    }
                }
            }
        }

        bestSol = currSol;
        return true;
    }
    catch(std::exception& e) {
        std::cout << ">>>EXCEPTION: " << e.what() << std::endl;
        return false;
    }
}
```

This implementation:

1. Uses a first-improvement strategy but explores both 2-opt and 3-opt neighborhoods
2. Tries 2-opt moves first since they're simpler, then moves to 3-opt if no improvement is found
3. Evaluates moves efficiently by only calculating the cost difference of affected edges
4. Maintains the fixed start/end node (0) by not including it in moves
5. Uses helper methods to keep the code organized and maintainable
6. Includes proper exception handling

The solver follows a hierarchical neighborhood structure - it first tries simpler moves (2-opt) before attempting more complex ones (3-opt), which is generally more efficient than always exploring the full 3-opt neighborhood.

Remember what we are doing:

- In the context of the Traveling Salesman Problem (TSP), 2-opt and 3-opt moves are fundamental techniques for improving an existing solution through local search.

- The 2-opt move involves selecting two nonadjacent arcs of the current path and swapping them with two new arcs in order to obtain a new valid path. Operationally, this results in reversing a sub-sequence of the path. For example, if we have the path <1,2,3,4,5,6,7,8,1> and select the arcs (2,3) and (6,7), after the 2-opt move we will get the path <1,2,6,5,4,3,7,8,1>. This operation is equivalent to “uncrossing” two intersecting arcs in the path design.

- The 3-opt move is a generalization of 2-opt involving three arcs instead of two. In this case, three nonadjacent arcs are selected, and the path is reorganized by considering all possible ways of reconnecting the resulting segments. In practice, this is equivalent to performing two sub-sequence reversals. Going back to the previous example, a 3-opt move could transform the path <1,2,3,4,5,6,7,8,1> into <1,2,7,6,3,4,5,8,1>, reversing two distinct segments of the path.

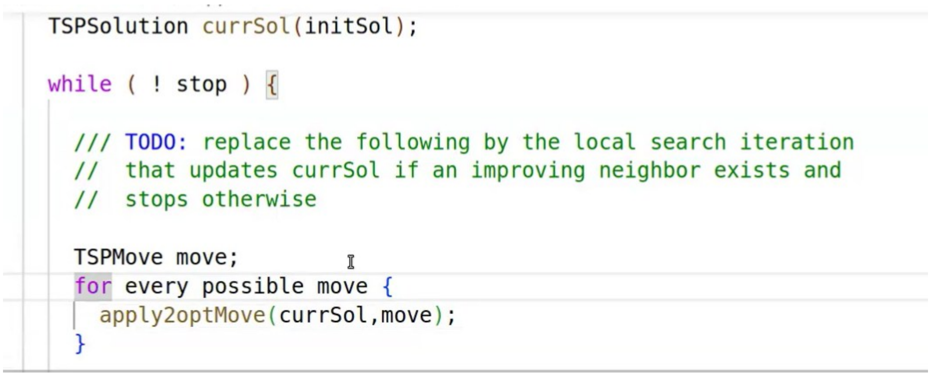
- The combined use of these moves allows the exploration of a wider neighborhood than the current solution. The 2-opt is simpler and faster to implement, while the 3-opt can find improvements that the 2-opt cannot identify, but it requires more computational time since it explores a larger number of possible changes. In practice, we often start by looking for improvements with 2-opt moves and, only if none are found, move on to the more complex 3-opt moves.

- It is important to note that both moves maintain path validity: the result is always a Hamiltonian cycle that visits all nodes exactly once, changing only the order in which they are visited.

Let's see the professor solution instead – first define the 2-opt-move:

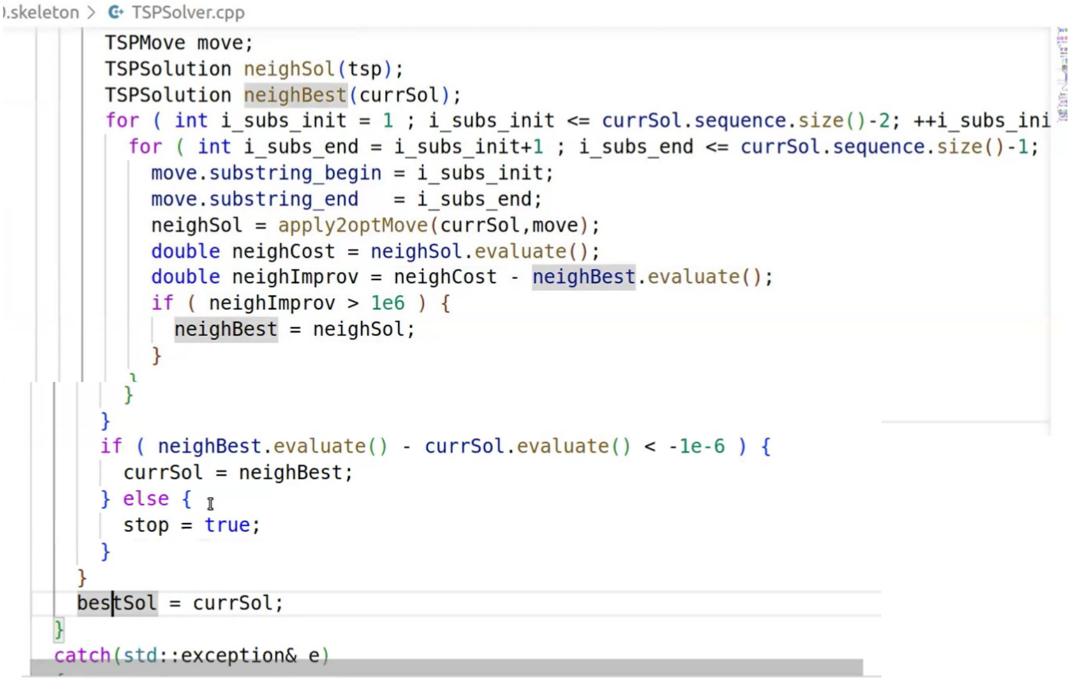
```
TSPSolution& TSPSolver::apply2OptMove(TSPSolution& tspSol, const TSPMove& move)
const {
    for(int i = move.substring_begin, i = move.substring_end; i++) {
        tspSol.sequence[i] = tspSol.sequence[move.substring_end - (i -
move.substring_begin)]; // Reverse subsequence
    }
    return tspSol;
}
```

Then, inside the `solve()` method, we would have to apply this move for every possible move, generating all of the possible pairs for the substrings combinations:



```
TSPSolution currSol(initSol);

while ( ! stop ) {

  /// TODO: replace the following by the local search iteration
  //  that updates currSol if an improving neighbor exists and
  //  stops otherwise

  TSPMove move;
  for every possible move {
    apply2optMove(currSol,move);
  }
```

We would need to apply for the move for every possible pair of city so to update the cost accordingly and then reverse the actual string:



```
      TSPMove move;
      TSPSolution neighSol(tsp);
      TSPSolution neighBest(currSol);
      for ( int i_subs_init = 1 ; i_subs_init <= currSol.sequence.size()-2; ++i_subs_ini
        for ( int i_subs_end = i_subs_init+1 ; i_subs_end <= currSol.sequence.size()-1;
          move.substring_begin = i_subs_init;
          move.substring_end   = i_subs_end;
          neighSol = apply2optMove(currSol,move);
          double neighCost = neighSol.evaluate();
          double neighImprov = neighCost - neighBest.evaluate();
          if ( neighImprov > 1e6 ) {
            neighBest = neighSol;
          }
        }
      }
      if ( neighBest.evaluate() - currSol.evaluate() < -1e-6 ) {
        currSol = neighBest;
      } else {
        stop = true;
      }
    }
    bestSol = currSol;
  }
  catch(std::exception& e)
```

The professor told us this is a very “bad” implementation, to be found inside of the Internet with: “how can i implement local search for tsp”. Consider in case sources like this one to understand more.

```
bool TSPSolver::solve ( const TSP& tsp , const TSPSolution& initSol , TSPSolution&
bestSol )
{
  try
  {
    bool stop = false;
    int  iter = 0;

    TSPSolution currSol(initSol);

    while ( ! stop ) {
      if ( tsp.n < 20 ) currSol.print(); std::cout << '\n';

        /// TODO: replace the following by the local search iteration
      //  that updates currSol if an improving neighbor exists and
      //  stops otherwise

      TSPMove move;
      TSPSolution neighSol(tsp);
      TSPSolution neighBest(currSol);
      for ( int i_subs_init = 1 ; i_subs_init < currSol.sequence.size()-2;
++i_subs_init ) {
        for ( int i_subs_end = i_subs_init+1 ; i_subs_end <
currSol.sequence.size()-1; ++i_subs_end ) {
          move.substring_begin = i_subs_init;
          move.substring_end   = i_subs_end;
          neighSol = apply2optMove(currSol,move);
          //if ( i_subs_init == 1 && i_subs_end == i_subs_init+1 )
neighSol.print();
          double neighCost = this->evaluate(neighSol,tsp);
          double bestCost = this->evaluate(neighBest,tsp);
          double neighImprov = neighCost - bestCost;
          if ( neighImprov < -1e-6 ) {
            neighBest = neighSol;
          }
        }
      }
      double currCost = this->evaluate(currSol,tsp);
      double bestCost = this->evaluate(neighBest,tsp);
      if ( bestCost - currCost < -1e-6 ) {
        currSol = neighBest;
      } else {
        stop = true;
      }
    }
    bestSol = currSol;
  }
  catch(std::exception& e)
  {
    std::cout << ">>>EXCEPTION: " << e.what() << std::endl;
    return false;
  }
  return true;
}
```

Specifically:

1. In the solve method, the code iterates over all possible 2-opt moves by considering all pairs of substring start and end positions.
2. For each move, it creates a new neighSol solution by applying the 2-opt move to the current solution using the apply2optMove method.
3. It evaluates the cost of the neighbor solution using the evaluate method.
4. It compares the cost of the neighbor solution with the best neighbor solution found so far and updates neighBest if the current neighbor is better.
5. After considering all possible moves, it updates currSol with neighBest if an improvement is found, otherwise, it stops the search.

The inefficiency in this implementation lies in the fact that it applies the 2-opt move and evaluates the entire solution for each possible move. This involves unnecessary computations and memory allocations.

Inside the “1-essential” folder, there is a more efficient implementation, with code as follows:

1. In the solve method, it calls the `findBestNeighborDecrement` method to find the best 2-opt move and its corresponding cost decrement.
2. The `findBestNeighborDecrement` method iterates over all possible 2-opt moves, but instead of applying the move and evaluating the entire solution, it calculates the cost variation directly.
    - It retrieves the cities before and after the substring (h, i, j, l) based on the current solution's sequence.
    - It calculates the cost variation by subtracting the costs of the removed edges (h-i and j-l) and adding the costs of the new edges (h-j and i-l).
    - It updates `bestDecrement` and the corresponding move if a better cost decrement is found.
3. If a move with a negative cost decrement is found, it applies the move using apply2optMove and continues the search. Otherwise, it stops the search.

The efficiency in the second implementation comes from the following:

1. It avoids creating new solution objects and evaluating the entire solution for each move. Instead, it calculates the cost variation solely based on the cities involved in the move.
2. It uses the findBestNeighborDecrement method to find the best move and its cost decrement in a single pass, reducing redundant computations.
3. It applies the move only when an improvement is found, avoiding unnecessary solution modifications – just when needed.

Inside the header file:

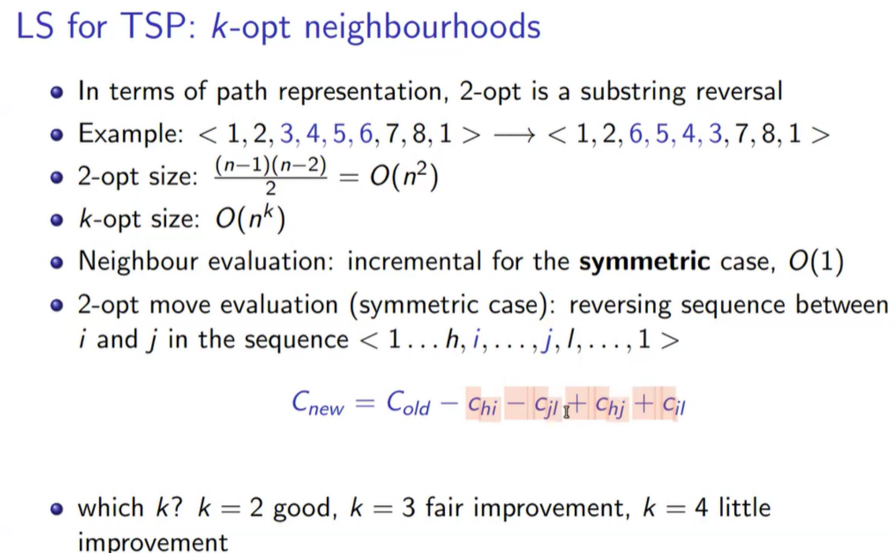
```
  double findBestNeighborDecrement ( const TSP& tsp , const TSPSolution& currSol ,
TSPMove& move );
```

Inside the cpp file:

```
bool TSPSolver::solve ( const TSP& tsp , const TSPSolution& initSol , TSPSolution&
bestSol )
{
  try
  {
    bool stop = false;
    int  iter = 0;

    TSPSolution currSol(initSol);

    TSPMove move;
    while ( ! stop ) {
                if ( tsp.n < 20 ) currSol.print(); //log current solution (only
small instances)
      double bestDecrement = findBestNeighborDecrement(tsp,currSol,move);
                std::cout << "(" << ++iter << "ls) move " << move.substring_init <<
" , " << move.substring_end << " improves by " << bestDecrement << std::endl;
      if ( bestDecrement < -1e-6 ) {
        currSol = apply2optMove(currSol,move);
        stop = false;
      } else {
        stop = true;
      }
    }
    bestSol = currSol;
  }
  catch(std::exception& e)
  {
    std::cout << ">>>EXCEPTION: " << e.what() << std::endl;
    return false;
  }
  return true;
}
```

Written by Gabriel R.

This is based on this move present inside the slide (focused formula) – compute the best decrement without needing the cost, only the cost variation:



LS for TSP: k-opt neighbourhoods

- In terms of path representation, 2-opt is a substring reversal
- Example: $< 1, 2, 3, 4, 5, 6, 7, 8, 1 > \longrightarrow < 1, 2, 6, 5, 4, 3, 7, 8, 1 >$
- 2-opt size: $\frac{(n-1)(n-2)}{2} = O(n^2)$
- k-opt size: $O(n^k)$
- Neighbour evaluation: incremental for the **symmetric** case, $O(1)$
- 2-opt move evaluation (symmetric case): reversing sequence between i and j in the sequence $< 1 \dots h, i, \dots, j, l, \dots, 1 >$

$$C_{new} = C_{old} - c_{hi} - c_{jl} + c_{hj} + c_{il}$$

- which k? $k = 2$ good, $k = 3$ fair improvement, $k = 4$ little improvement

An output example might be the following:

```
### 0 11 1 9 5 6 10 8 3 7 2 4 0  ###

0 11 1 9 5 6 10 8 3 7 2 4 0

0 1 11 9 5 6 10 8 3 7 2 4 0

0 1 11 9 5 6 10 8 7 3 2 4 0

0 1 11 9 2 3 7 8 10 6 5 4 0

0 1 11 9 10 8 7 3 2 6 5 4 0

0 1 11 9 10 8 7 3 4 5 6 2 0

0 1 11 9 10 8 7 6 5 4 3 2 0

0 1 11 10 9 8 7 6 5 4 3 2 0

FROM solution: 0 11 1 9 5 6 10 8 3 7 2 4 0 (value : 126)

TO   solution: 0 1 11 10 9 8 7 6 5 4 3 2 0 (value : 72)

in 0.0159369 seconds (user time)

in 0.017 seconds (CPU time)
```

20 LABORATORY 8 – COLUMN-GENERATION BASED HEURISTIC FOR 1D-CUTTING STOCK PROBLEM

Downloading from the Moodle the same name folder (TODO version ofc), we have two environments, in order to create the master and the slave problem. We start from the main file here.

The code sets up a master Linear Programming (LP) problem and iteratively generates new columns (cutting patterns) through the following key components:

- The main data structures and initialization:

```
DECL_ENV( env );  // CPLEX environment

DECL_PROB( env, lp ); // CPLEX problem

CS1D cs1dSolver(env, lp); // Custom solver class

Data data; // Problem data

data.read(argv[1]); // Read input
```

cs1dSolver.initMaster(data); // Initialize master problem

- The core column generation loop:

```
while(newcol) {

    // Key components here:

    // 1. Solve master LP to get dual values

    // 2. Solve pricing subproblem to find new columns

    // 3. Add new column if found or terminate if none found

}
```

The master problem and pricing subproblem work together in this way:

1. The master problem starts with a subset of cutting patterns and solves the LP relaxation (continuous) to get:

- Primal solution (x): How many times to use each pattern
- Dual solution (u): Shadow prices for the demand constraints, information to get the new column

2. The pricing subproblem uses these dual values to:

- Search for a new cutting pattern with negative reduced cost
- Add this pattern as a new column to the master if found
- Return false if no negative reduced cost pattern exists

Once the column generation loop terminates, we have solved the LP relaxation optimally. The code then uses two different methods to find integer solutions:

1. Simple rounding:

```
// Round up each variable value

for (unsigned int i = 0; i < x.size(); i++) {

    x[i] = (x[i] > 1e-5) ? ceil(x[i]) : 0.0;

}
```

2. Branch-and-bound on generated columns:

```
cs1dSolver.branchAndBoundOnThePartialModel(x, INTobjval2);
```

This applies integer programming techniques on the restricted set of columns found during column generation.

The key insight is that column generation allows us to solve large problems by dynamically generating only the "good" cutting patterns as needed, rather than enumerating all possible patterns upfront. The pricing subproblem efficiently finds these good patterns by solving a knapsack problem using the dual values from the master problem.

This implements what's known as a "price-and-cut" approach:

- Price: Generate new columns through the pricing subproblem
- Cut: Solve the master LP with the current columns
- Repeat until no negative reduced cost columns remain

The integer solutions found at the end are heuristic since we may have missed some patterns that could be useful in the integer optimal solution but weren't needed for the LP relaxation.

First of all, we solve the master and then call the slave:

```
while(newcol){
        //TODO...
        //     - solve master obtaining dual information
        cs1dSolver.solveMasterLP(x, u, objval);

        std::cout << "*** IT " << it++ << " *** " << " LPobj: " << objval << " x: ";
        if (x.size() < 10) for (unsigned int j = 0; j < x.size(); j++)
                std::cout << setw(7) << x[j] << " ";

std::cout << std::endl;

                        //     - call the slave [price] with dual information (the slave
also adds a variable to the master, if any, otherwise it returns false)
                cs1dSolver.price(env2, data, u);

}
```

This is a sophisticated example of *decomposition* - breaking down a complex problem into more manageable master and subproblems that work together through the dual values to find an optimal solution.

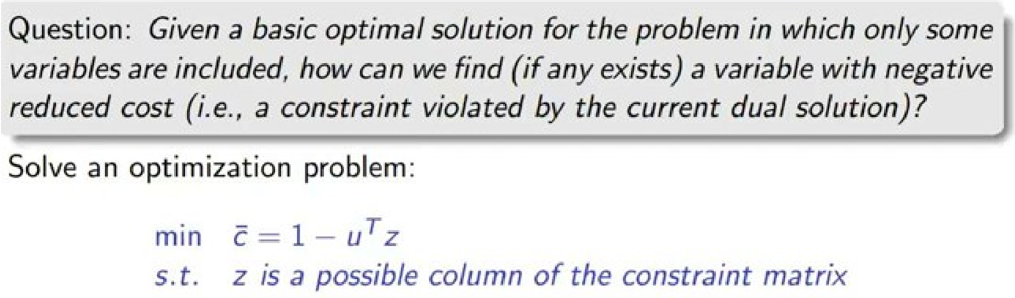
Inside of cs1d file, we will complete the method to solve the master LP (which is restricted) as follows:

```
void CS1D::solveMasterLP(std::vector<double>& x, std::vector<double>& u, double&
objval)
{
	//TODO
	// solve using CPX*lp*opt
	CHECKED_CPX_CALL(CPXlpopt, env, lp);
	// get current LP obj value (reference objval)
	CHECKED_CPX_CALL(CPXgetobjval, env, lp, &objval);
	// get current RESTRICTED LP PRIMAL solution (reference x)
	int n = CPXgetnumcols(env, lp);
	x.resize(n);
	CHECKED_CPX_CALL(CPXgetx, env, lp, &x[0], 0, n - 1);
	// get current RESTRICTED LP DUAL solution using *CPXgetpi* (reference u)
	int m = CPXgetnumrows(env, lp);
	u.resize(m);
	CHECKED_CPX_CALL(CPXgetpi, env, lp, &u[0], 0, m - 1);

}
```

Above, we:

- solves the restricted master problem, which contains only the columns (cutting patterns) generated so far
- retrieves the objective value of the current solution - in the cutting stock context, this represents the total number of stock pieces needed with the current set of patterns.
- retrieves the primal solution - how many times each cutting pattern should be used. The vector is resized to match the current number of patterns,
- retrieves the dual values (or shadow prices) associated with the demand constraints.

Now we go into the detail of the pricing procedure; find the minimum possible reduced cost (maximum violation in dual terms) – a column s.t. we find the master problem:



Question: *Given a basic optimal solution for the problem in which only some variables are included, how can we find (if any exists) a variable with negative reduced cost (i.e., a constraint violated by the current dual solution)?*

Solve an optimization problem:

$$\begin{array}{ll} \min & \bar{c} = 1 - u^T z \\ s.t. & z \text{ is a possible column of the constraint matrix} \end{array}$$

The knapsack problem is solved efficiently with dynamic programming; we will use the exec function from the knapsack.h file present in the same folder.

This function determines if there are any beneficial new cutting patterns to add to the master problem. The pricing problem needs to solve a knapsack problem where:

- The "weights" are the item lengths
- The "capacity" is the stock length
- The "values" are the dual values from the master problem
- The solution indicates how many of each item to include in a new cutting pattern

In column generation terms:

- The reduced cost tells us if a new pattern would improve the master solution
- A negative reduced cost means the pattern would help decrease the objective value
- If no negative reduced cost pattern exists, we've reached optimality

If we find a beneficial pattern, we need to add it to the master problem and solve that to optimality. The final implementation would be:

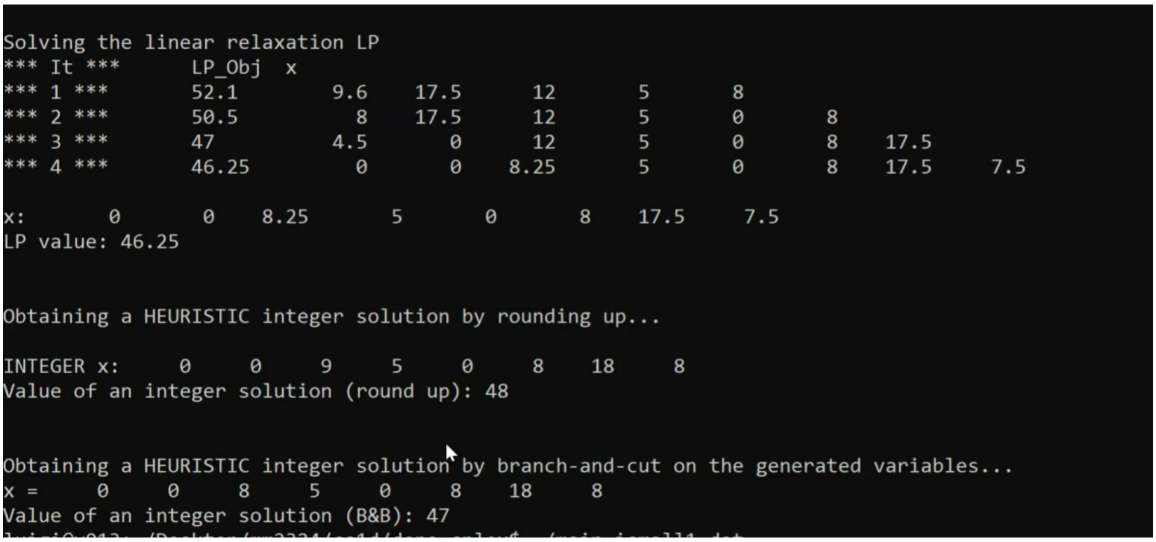
```
bool CS1D::price(Env pricerEnv, const Data& data, const std::vector<double>& u)
{
    KPSolver kp(pricerEnv);
    std::vector<double> z;
    double value;


    //CALL kp.exec to solve the right knapsack problem and get
    //    the related objective function into < value >
    //TODO...
    kp.exec(data.L, u, data.W, z, value);

    //TODO...
    //"return false" if NO negative reduced cost variable exists
    if (value <= 1 + 1e-5) return false; // 1 + 1e-5 is the tolerance for numeric
issues
    // could be evem if(1 - value > -ZERO_EPS) return false;
    // if the valye is 1.000001, the reduced cost is - 0.000001, which is negative.
    // Because of possible numerical issues, we consider this number as 0.

    // Add one column to RMP:
    // prepare parameters for following *CPXaddcols*
    //TODO...
    //   - the vector idx of the indexes of the rows in which the variable appears
with (nonzero) oefficient
    //   - the vector coef of the (nonzero) coefficients related to the row indexes
above
    //   - the coefficient obj in the objective function
    //...

    std::vector<int> idx;
    std::vector<double> coef;
    int m = z.size();
    for (int i = 0; i < m; i++)
    {
        if (z[i] > 1e-5)
        {
            idx.push_back(i);
```

```
                        coef.push_back(z[i]);
                }
        }
        double obj = 1.0;
        int matbeg = 0;

        // add the variable to the model
        CHECKED_CPX_CALL( CPXaddcols, env, lp, 1, idx.size(), &obj, &matbeg, &idx[0],
&coef[0], NULL, NULL, NULL );
        //status = CPXaddcols (env, lp, ccnt, nzcnt, obj, cmatbeg, cmatind, cmatval, lb,
ub, newcolname);
        return true;
}
```

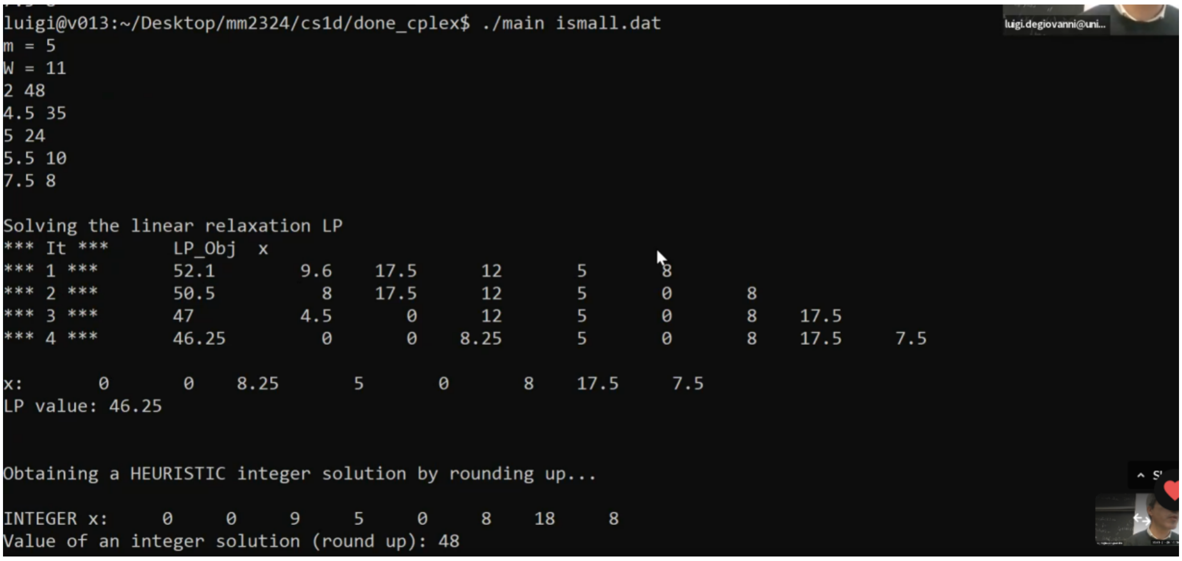
This implementation illustrates the elegant interplay between the master and pricing problems in column generation.

- By solving a knapsack problem that uses the current dual values as profits and piece lengths as weights, it efficiently identifies cutting patterns that could improve the overall solution
- The function handles numerical precision issues through careful tolerance checks and sparse data structures, ensuring both reliability and computational efficiency
- Each time it finds a beneficial pattern, it expands the master problem's solution space, gradually building towards the optimal cutting strategy through an iterative process that only generates patterns as they become potentially useful

Some of the results appear here:



```
Solving the linear relaxation LP
*** It ***      LP_Obj  x
*** 1 ***       52.1          9.6    17.5       12        5       8
*** 2 ***       50.5            8    17.5       12        5       0       8
*** 3 ***       47            4.5       0       12        5       0       8    17.5
*** 4 ***       46.25           0       0     8.25        5       0       8    17.5     7.5

x:        0       0    8.25       5       0       8    17.5     7.5
LP value: 46.25


Obtaining a HEURISTIC integer solution by rounding up...

INTEGER x:     0     0     9     5     0     8    18     8
Value of an integer solution (round up): 48


Obtaining a HEURISTIC integer solution by branch-and-cut on the generated variables...
x =     0     0     8     5     0     8    18     8
Value of an integer solution (B&B): 47
```



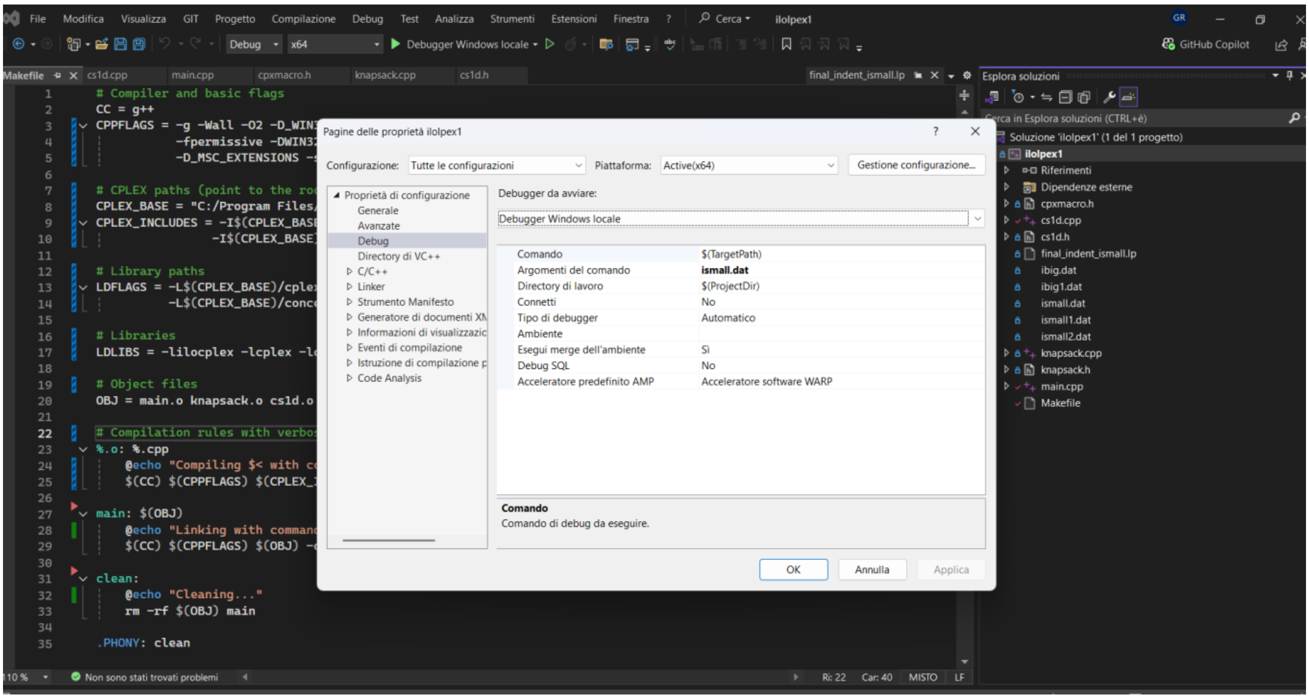
```
luigi@v013:~/Desktop/mm2324/cs1d/done_cplex$ ./main ismall.dat
m = 5
W = 11
2 48
4.5 35
5 24
5.5 10
7.5 8

Solving the linear relaxation LP
*** It ***      LP_Obj  x
*** 1 ***       52.1        9.6    17.5      12       5       8
*** 2 ***       50.5          8    17.5      12       5       0       8
*** 3 ***       47          4.5       0      12       5       0       8    17.5
*** 4 ***       46.25         0       0    8.25       5       0       8    17.5     7.5

x:       0       0    8.25       5       0       8    17.5     7.5
LP value: 46.25


Obtaining a HEURISTIC integer solution by rounding up...

INTEGER x:     0     0     9     5     0     8    18     8
Value of an integer solution (round up): 48
```

This implementation demonstrates the effectiveness of column generation for solving large cutting stock problems:

1. The master problem starts with basic single-item patterns and progressively generates more complex patterns only as needed.
2. The pricing mechanism efficiently identifies beneficial new patterns by solving knapsack problems using the dual values from the master problem.
3. The final integer solutions are obtained through two different approaches:
   - Simple rounding provides a quick but potentially loose bound
   - Branch-and-bound on the restricted problem provides a more refined solution

The relatively small gap between the LP relaxation (46.25) and the best integer solution (47) suggests that the column generation approach was effective at identifying good cutting patterns.

<u>WARNING</u> – I was NOT able to compile this lab using the ilolpex import method as done before; there is no actual way to make this project work even when modifying the Makefile and the cpxmacro. What I did on Windows was to simply click “Build” (green Play button) above and to give as argument in debug window one .dat file and it works! You can see below how (works with others too).



```
# Compiler and basic flags
CC = g++
CPPFLAGS = -g -Wall -O2 -D_WIN
           -fpermissive -DWIN3
           -D_MSC_EXTENSIONS -

# CPLEX paths (point to the ro
CPLEX_BASE = "C:/Program Files
CPLEX_INCLUDES = -I$(CPLEX_BAS
                 -I$(CPLEX_BASE

# Library paths
LDFLAGS = -L$(CPLEX_BASE)/cple
          -L$(CPLEX_BASE)/conc

# Libraries
LDLIBS = -lilocplex -lcplex -l

# Object files
OBJ = main.o knapsack.o cs1d.o

# Compilation rules with verbo
%.o: %.cpp
	@echo "Compiling $< with c
	$(CC) $(CPPFLAGS) $(CPLEX_

main: $(OBJ)
	@echo "Linking with comman
	$(CC) $(CPPFLAGS) $(OBJ) -

clean:
	@echo "Cleaning..."
	rm -rf $(OBJ) main

.PHONY: clean
```

Pagine delle proprietà ilolpex1

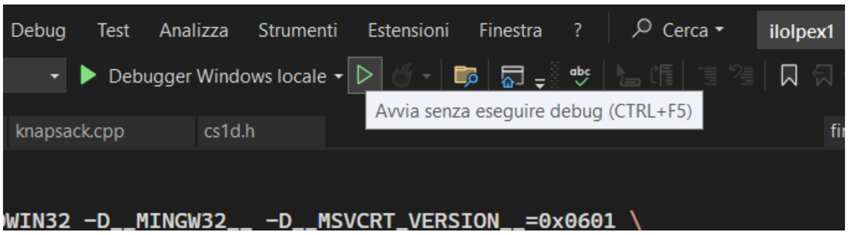
Configurazione: Tutte le configurazioni Piattaforma: Active(x64)

Debugger da avviare: Debugger Windows locale

Comando	$(TargetPath)
Argomenti del comando	**ismall.dat**
Directory di lavoro	$(ProjectDir)
Connetti	No
Tipo di debugger	Automatico
Ambiente	
Esegui merge dell'ambiente	Sì
Debug SQL	No
Acceleratore predefinito AMP	Acceleratore software WARP

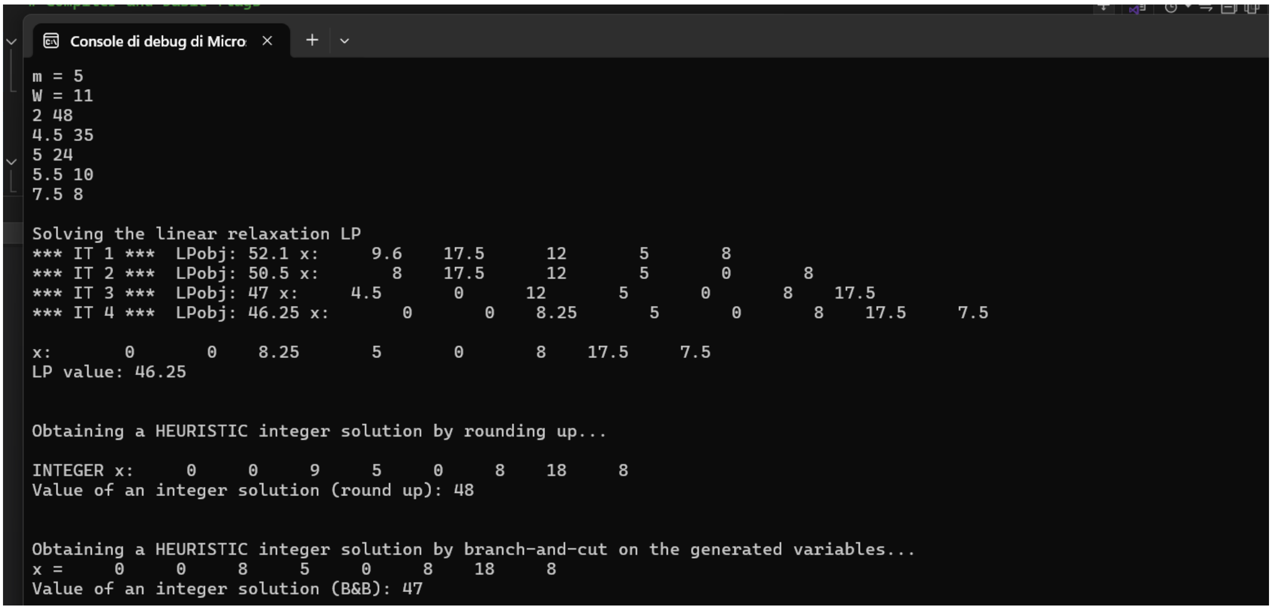
Comando
Comando di debug da eseguire.

File Modifica Visualizza GIT Progetto Compilazione Debug Test Analizza Strumenti Estensioni Finestra ? Cerca ilolpex1
Debug x64 Debugger Windows locale
Makefile cs1d.cpp main.cpp cpxmacro.h knapsack.cpp cs1d.h final_indent_ismall.lp
Proprietà di configurazione
Generale
Avanzate
Debug
Directory di VC++
C/C++
Linker
Strumento Manifesto
Generatore di documenti XM
Informazioni di visualizzazio
Eventi di compilazione
Istruzione di compilazione p
Code Analysis
Gestione configurazione...
OK
Annulla
Applica
Esplora soluzioni
Soluzione 'ilolpex1' (1 del 1 progetto)
ilolpex1
Riferimenti
Dipendenze esterne
cpxmacro.h
cs1d.cpp
cs1d.h
final_indent_ismall.lp
ibig.dat
ibig1.dat
ismall.dat
ismall1.dat
ismall2.dat
knapsack.cpp
knapsack.h
main.cpp
Makefile
110 %
Non sono stati trovati problemi
Ri: 22 Car: 40 MISTO LF



Debug Test Analizza Strumenti Estensioni Finestra ? Cerca ilolpex1
Debugger Windows locale
Avvia senza eseguire debug (CTRL+F5)
knapsack.cpp cs1d.h

```
WIN32 -D__MINGW32__ -D__MSVCRT_VERSION__=0x0601 \
```



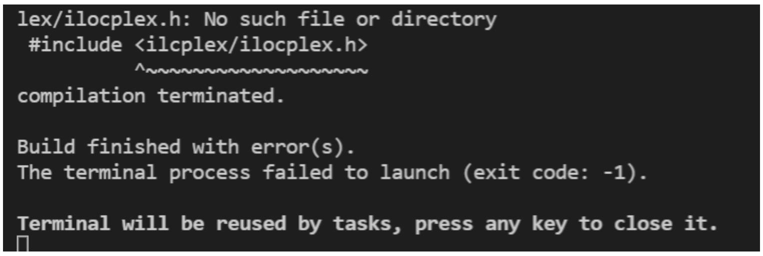
```
m = 5
W = 11
2 48
4.5 35
5 24
5.5 10
7.5 8

Solving the linear relaxation LP
*** IT 1 ***  LPobj: 52.1 x:     9.6    17.5      12       5       8
*** IT 2 ***  LPobj: 50.5 x:       8    17.5      12       5       0       8
*** IT 3 ***  LPobj: 47 x:     4.5       0      12       5       0       8    17.5
*** IT 4 ***  LPobj: 46.25 x:       0       0    8.25       5       0       8    17.5     7.5

x:       0       0    8.25       5       0       8    17.5     7.5
LP value: 46.25


Obtaining a HEURISTIC integer solution by rounding up...

INTEGER x:      0      0      9      5      0      8     18      8
Value of an integer solution (round up): 48


Obtaining a HEURISTIC integer solution by branch-and-cut on the generated variables...
x =      0      0      8      5      0      8     18      8
Value of an integer solution (B&B): 47
```

21 EXTRA: WINDOWS CPLEX COMPILATION – INFO & INSTRUCTIONS

To execute the code, we need in order:

- Visual Studio IDE (note: this is different from Visual Studio Code)
    - Installing the Community Version from here
- A C++ compiler
    - You can either use MinGW (here) or MSVC (done when selecting "Develop C++ applications" when installing Visual Studio
- CPLEX Studio installed on your machine (current version is 22.11)
    - All of the info present in the Moodle of the course here

The problem on Windows is evidenced by the fact that fatal errors might occur, like:



```
lex/ilocplex.h: No such file or directory
 #include <ilcplex/ilocplex.h>
          ^~~~~~~~~~~~~~~~~~~~
compilation terminated.

Build finished with error(s).
The terminal process failed to launch (exit code: -1).

Terminal will be reused by tasks, press any key to close it.
```

As seen here, a normal execution of Cplex would include:

`C:\Program Files\IBM\ILOG\CPLEX_Studio_Community201\cplex\include`

`C:\Program Files\IBM\ILOG\CPLEX_Studio_Community201\concert\include`

These folder need to be configured inside of the additional inclusions and also additional dependencies in the form of files and directives to the compiler.

The most recent versions (up to 2019, current is 2022) do not allow complete editing of the compilation options as you might see here.

As found within the internal group of the course, the main problem seems to be that there is not cpxmacro.h file inside of Windows installations, so if we try to use it in our project we have problem with functions for declaration of env, adding variables and constraints.

- Also, after doing all the stuff that are mentioned inside the files (this or the above one) we need to import cpxmacro.h in order to work properly and that we can use all APIs and just everything has to be imported before running all the code
- It is easier to copy cpxmacro.h in the cplex/include folder and you can include it just with that small adjustment

21.1 Windows CPLEX Compilation – Solution 1

The solution is actually the following:

- Once you have installed Visual Studio and CPLEX on your machine, you should take some ready-made examples, so to import a Solution file (basically, a configuration file which needs to be imported in order to make the execution work) and then an executable file with a main()

The paths to consider for solution files are the following:

- C++ files:

```
C:\Program
Files\IBM\ILOG\CPLEX_Studio2211\cplex\examples\x64_windows_msvc14\stat_mda
```

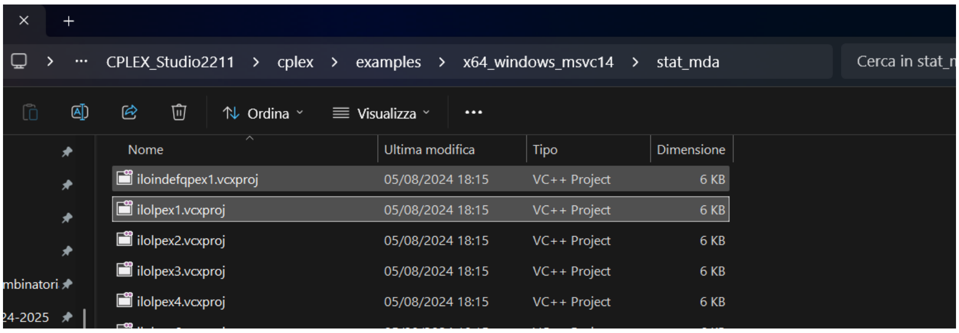
- C files:

```
C:\Program
Files\IBM\ILOG\CPLEX_Studio2211\cplex\examples\x64_windows_msvc14\stat_mdd
```

These folders report a lot of different files which are the "Solution" files; we need to consider files with extension .vcxproj. The goal here would be to first select a Solution file and then select a C/C++ file of the same name. So:

- If you want to execute a C example, go the "mda"
- If you want to execute a C++ example, go to "mdd"

For instance, let's consider ilolpex1.vcproj, which is a C++ file:


CPLEX_Studio2211 › cplex › examples › x64_windows_msvc14 › stat_mda
Cerca in stat_r
Ordina
Visualizza
mbinatori
24-2025

Nome	Ultima modifica	Tipo	Dimensione
iloindefqpex1.vcxproj	05/08/2024 18:15	VC++ Project	6 KB
ilolpex1.vcxproj	05/08/2024 18:15	VC++ Project	6 KB
ilolpex2.vcxproj	05/08/2024 18:15	VC++ Project	6 KB
ilolpex3.vcxproj	05/08/2024 18:15	VC++ Project	6 KB
ilolpex4.vcxproj	05/08/2024 18:15	VC++ Project	6 KB

We then need the actual source codes, which are to be linked with the respective vcxproj files of before. Once again, it's different for both formats:

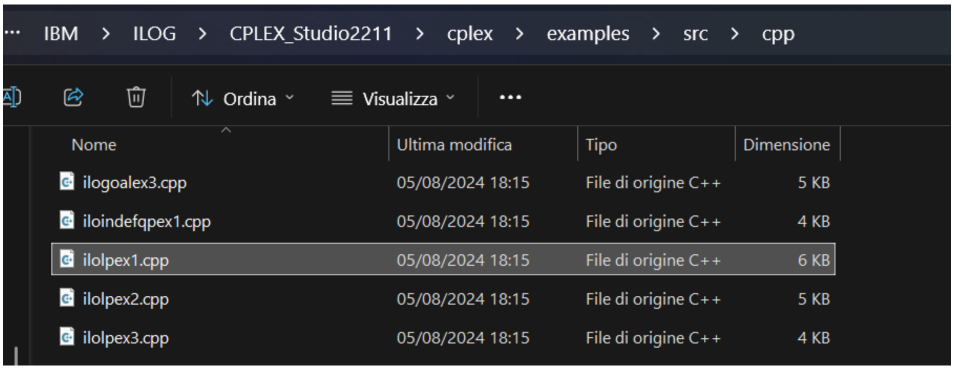
- C files:

```
C:\Program Files\IBM\ILOG\CPLEX_Studio2211\cplex\examples\src\c
```

- C++ files:

```
C:\Program Files\IBM\ILOG\CPLEX_Studio2211\cplex\examples\src\cpp
```

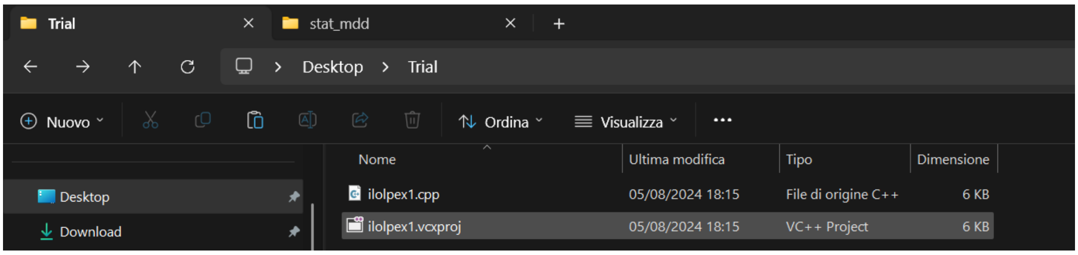
We then take the file of the same name as before:



··· IBM > ILOG > CPLEX_Studio2211 > cplex > examples > src > cpp
Ordina
Visualizza
···

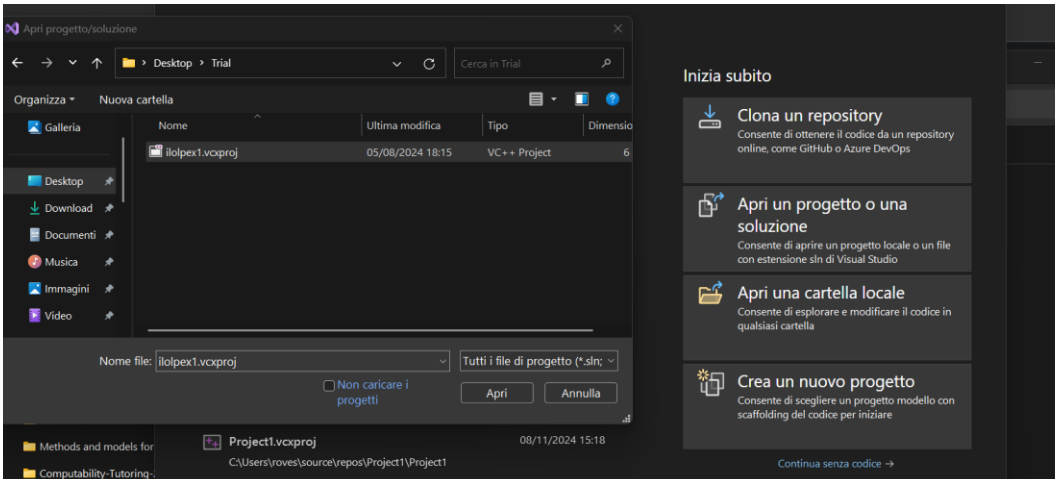
Nome	Ultima modifica	Tipo	Dimensione
ilogoalex3.cpp	05/08/2024 18:15	File di origine C++	5 KB
iloindefqpex1.cpp	05/08/2024 18:15	File di origine C++	4 KB
ilolpex1.cpp	05/08/2024 18:15	File di origine C++	6 KB
ilolpex2.cpp	05/08/2024 18:15	File di origine C++	5 KB
ilolpex3.cpp	05/08/2024 18:15	File di origine C++	4 KB

We need both files in order to import them into Visual Studio and then customize the code of the actual source file (C/C++) so to make the code work fine. We then create a folder with a custom name on a custom location with both files, like the following:



Nome	Ultima modifica	Tipo	Dimensione
ilolpex1.cpp	05/08/2024 18:15	File di origine C++	6 KB
ilolpex1.vcxproj	05/08/2024 18:15	VC++ Project	6 KB

We then open Visual Studio loading the vcxproj file on “Open a project or a solution” file.



Nome	Ultima modifica	Tipo	Dimensio
ilolpex1.vcxproj	05/08/2024 18:15	VC++ Project	6

Nome file: ilolpex1.vcxproj

Tutti i file di progetto (*.sln;

□ Non caricare i progetti

Project1.vcxproj 08/11/2024 15:18
C:\Users\roves\source\repos\Project1\Project1

Inizia subito

Clona un repository
Consente di ottenere il codice da un repository online, come GitHub o Azure DevOps

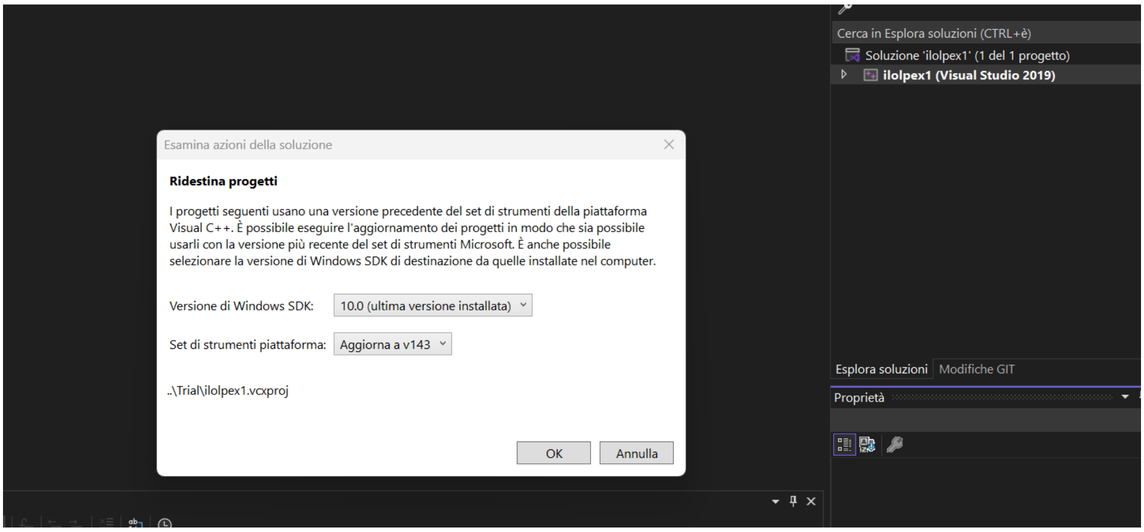
Apri un progetto o una soluzione
Consente di aprire un progetto locale o un file con estensione sln di Visual Studio

Apri una cartella locale
Consente di esplorare e modificare il codice in qualsiasi cartella

Crea un nuovo progetto
Consente di scegliere un progetto modello con scaffolding del codice per iniziare

Continua senza codice →

Once the prompt is open, select the Windows SDK version and multiplatform by default and continue.



Esamina azioni della soluzione

Ridestina progetti

I progetti seguenti usano una versione precedente del set di strumenti della piattaforma Visual C++. È possibile eseguire l'aggiornamento dei progetti in modo che sia possibile usarli con la versione più recente del set di strumenti Microsoft. È anche possibile selezionare la versione di Windows SDK di destinazione da quelle installate nel computer.

Versione di Windows SDK: 10.0 (ultima versione installata)

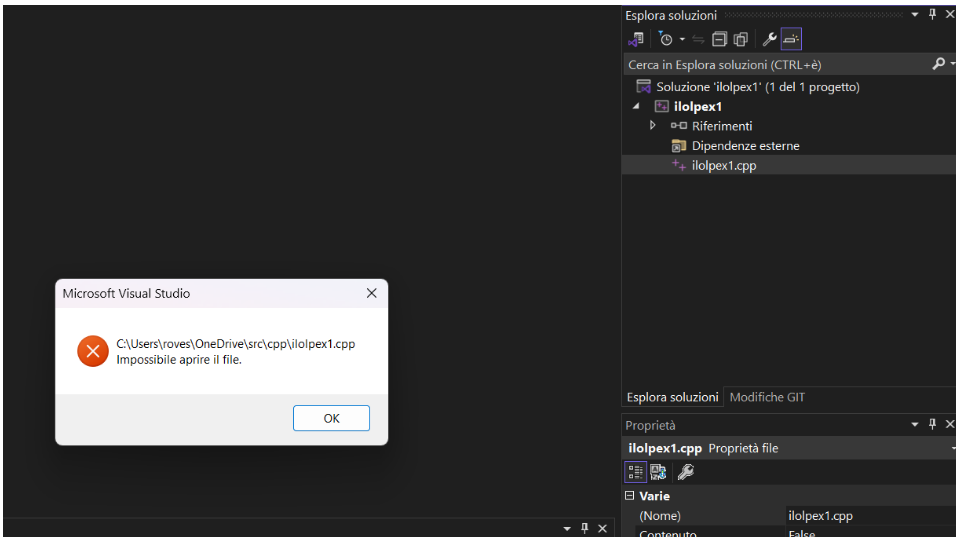
Set di strumenti piattaforma: Aggiorna a v143

..\Trial\ilolpex1.vcxproj

OK
Annulla
Cerca in Esplora soluzioni (CTRL+è)
Soluzione 'ilolpex1' (1 del 1 progetto)
ilolpex1 (Visual Studio 2019)
Esplora soluzioni
Modifiche GIT
Proprietà

WARNING

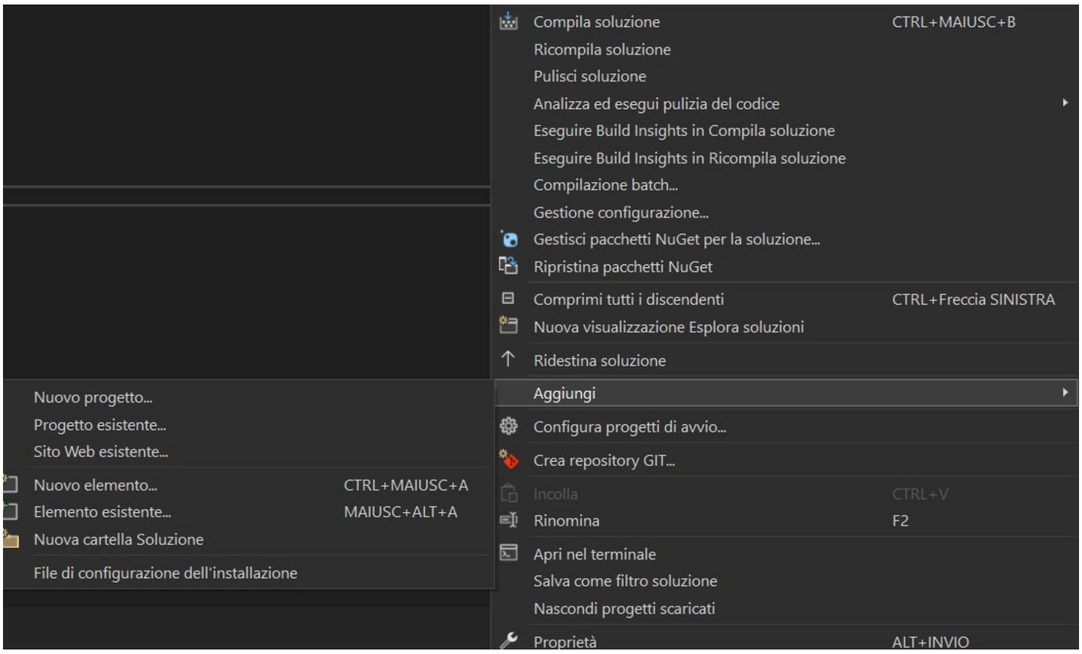
At this point, since the vcxproj file points to files present in the previous path (so inside of the Cplex path), it will tell you “Impossible to open file”, since it does not see the local path:



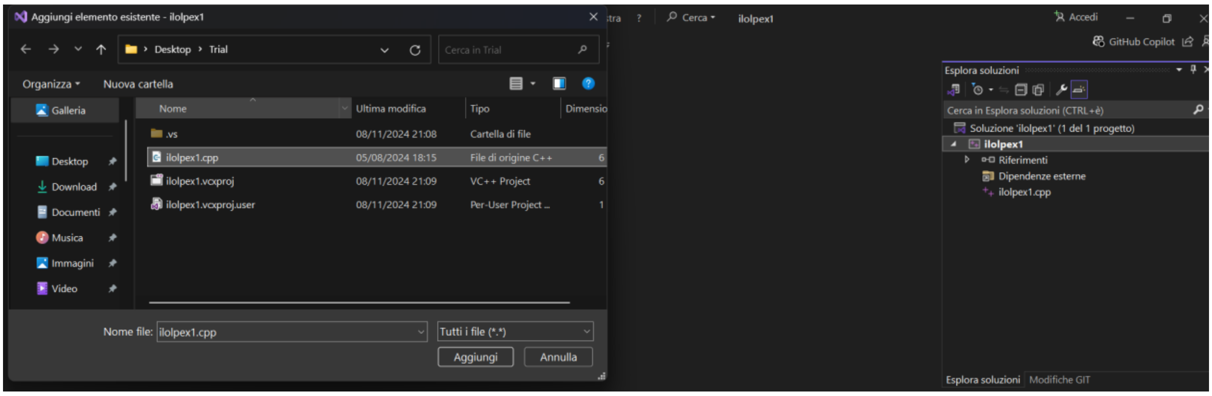
Microsoft Visual Studio

C:\Users\roves\OneDrive\src\cpp\ilolpex1.cpp
Impossibile aprire il file.

What you will do to solve this problem, is to right-click the name of project in the right menu present (in this case where there is ilolpex1) and then click "Add" (Aggiungi) and then click on "Existing element" (Elemento esistente):


Compila soluzione CTRL+MAIUSC+B
Ricompila soluzione
Pulisci soluzione
Analizza ed esegui pulizia del codice
Eseguire Build Insights in Compila soluzione
Eseguire Build Insights in Ricompila soluzione
Compilazione batch...
Gestione configurazione...
Gestisci pacchetti NuGet per la soluzione...
Ripristina pacchetti NuGet
Comprimi tutti i discendenti CTRL+Freccia SINISTRA
Nuova visualizzazione Esplora soluzioni
Ridestina soluzione
Aggiungi
Configura progetti di avvio...
Crea repository GIT...
Incolla CTRL+V
Rinomina F2
Apri nel terminale
Salva come filtro soluzione
Nascondi progetti scaricati
Proprietà ALT+INVIO
Nuovo progetto...
Progetto esistente...
Sito Web esistente...
Nuovo elemento... CTRL+MAIUSC+A
Elemento esistente... MAIUSC+ALT+A
Nuova cartella Soluzione
File di configurazione dell'installazione

Here we will select the actual C/C++ file:

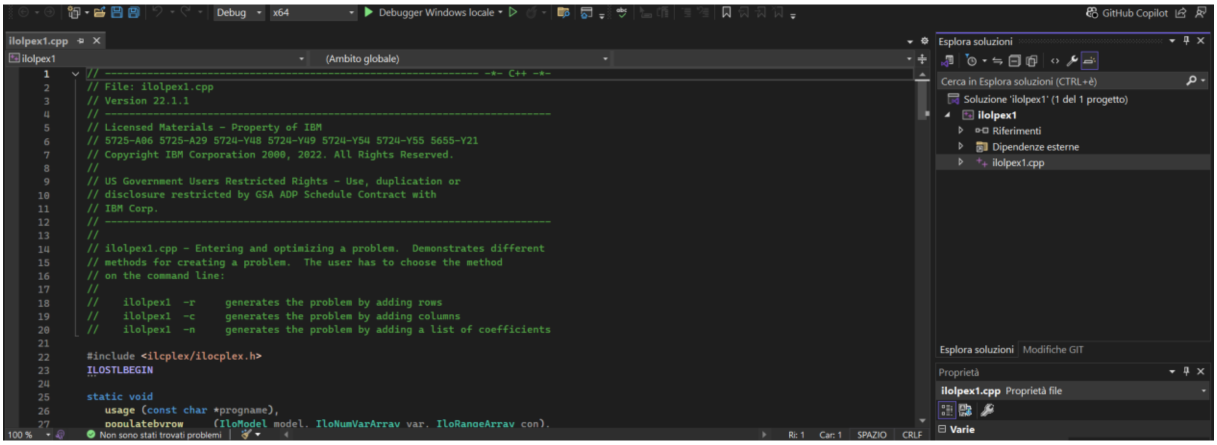


Nome	Ultima modifica	Tipo	Dimensio
.vs	08/11/2024 21:08	Cartella di file	
ilolpex1.cpp	05/08/2024 18:15	File di origine C++	6
ilolpex1.vcxproj	08/11/2024 21:09	VC++ Project	6
ilolpex1.vcxproj.user	08/11/2024 21:09	Per-User Project ...	1

Nome file: ilolpex1.cpp

Tutti i file (*.*)

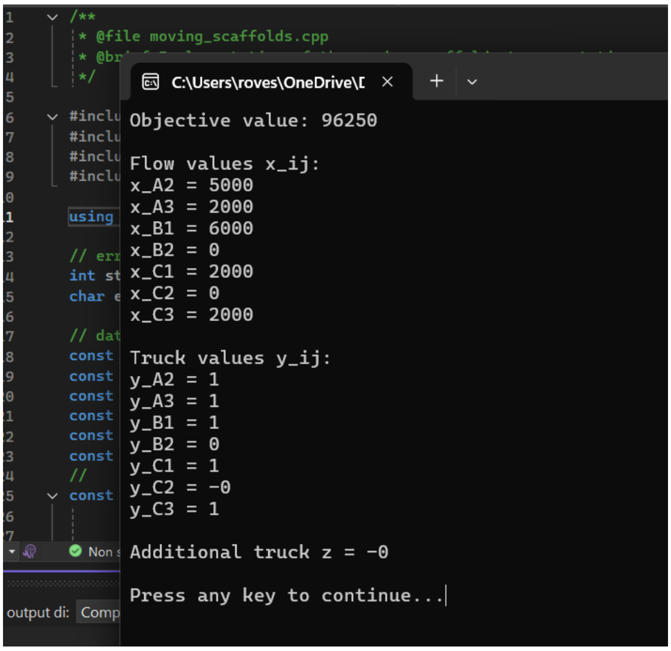
Please remove the old file, which is not to be found, so you have only one, the correctly imported file. You should see something like this:



```
// -------------------------------------------------------------- -*- C++ -*-
// File: ilolpex1.cpp
// Version 22.1.1
// --------------------------------------------------------------------------
// Licensed Materials - Property of IBM
// 5725-A06 5725-A29 5724-Y48 5724-Y49 5724-Y54 5724-Y55 5655-Y21
// Copyright IBM Corporation 2000, 2022. All Rights Reserved.
//
// US Government Users Restricted Rights - Use, duplication or
// disclosure restricted by GSA ADP Schedule Contract with
// IBM Corp.
// --------------------------------------------------------------------------
//
// ilolpex1.cpp - Entering and optimizing a problem.  Demonstrates different
// methods for creating a problem.  The user has to choose the method
// on the command line:
//
//    ilolpex1  -r     generates the problem by adding rows
//    ilolpex1  -c     generates the problem by adding columns
//    ilolpex1  -n     generates the problem by adding a list of coefficients

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

static void
   usage (const char *progname),
   populatebyrow     (IloModel model, IloNumVarArray var, IloRangeArray con),
```

Written by Gabriel R.

We then build the actual file, and a command prompt window will feedback the right execution here (this is a different execution, the example run in the laboratory, so you have an idea):



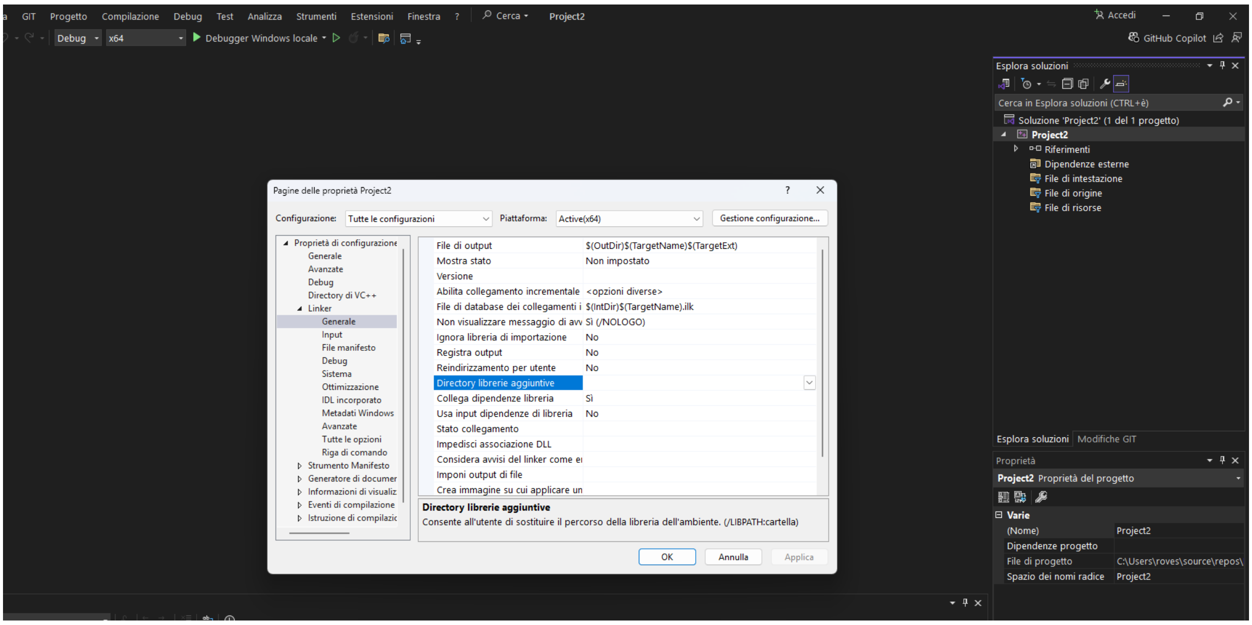
```
Objective value: 96250

Flow values x_ij:
x_A2 = 5000
x_A3 = 2000
x_B1 = 6000
x_B2 = 0
x_C1 = 2000
x_C2 = 0
x_C3 = 2000

Truck values y_ij:
y_A2 = 1
y_A3 = 1
y_B1 = 1
y_B2 = 0
y_C1 = 1
y_C2 = -0
y_C3 = 1

Additional truck z = -0

Press any key to continue...
```

This way, any kind of project works. This was tested both on C and C++ files.

21.2 WINDOWS CPLEX COMPILATION – SOLUTION 2

Another way to make this work is to create a C++ project from scratch and then right click on the right side menu on Properties so to open the following window – adapted from 4-5 page of this.

Then, one goes to “Linker” > “General” > “Additional library directories”:

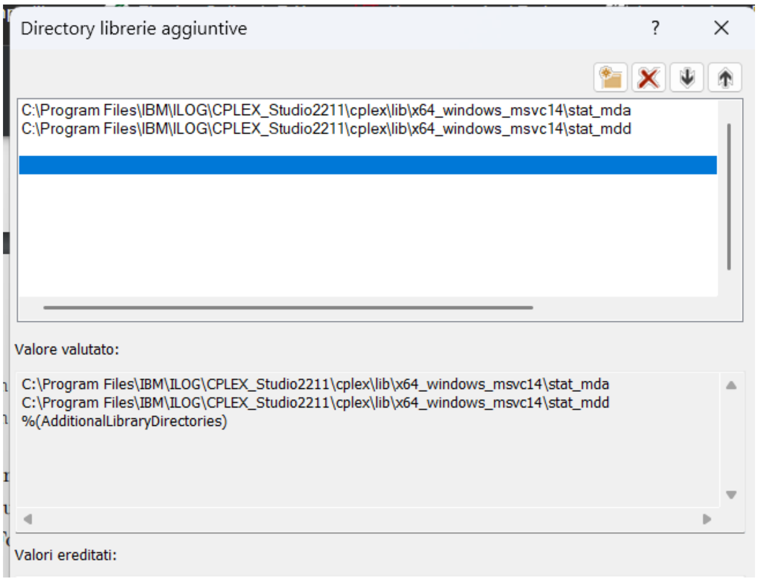


Configurazione: Tutte le configurazioni   Piattaforma: Active(x64)   Gestione configurazione...

File di output	$(OutDir)$(TargetName)$(TargetExt)
Mostra stato	Non impostato
Versione	
Abilita collegamento incrementale	<opzioni diverse>
File di database dei collegamenti i	$(IntDir)$(TargetName).ilk
Non visualizzare messaggio di avv	Sì (/NOLOGO)
Ignora libreria di importazione	No
Registra output	No
Reindirizzamento per utente	No
Directory librerie aggiuntive	
Collega dipendenze libreria	Sì
Usa input dipendenze di libreria	No
Stato collegamento	
Impedisci associazione DLL	
Considera avvisi del linker come er	
Imponi output di file	
Crea immagine su cui applicare un	

Directory librerie aggiuntive
Consente all'utente di sostituire il percorso della libreria dell'ambiente. (/LIBPATH:cartella)

Here, one then adds the mda/mdd folders as path:



```
C:\Program Files\IBM\ILOG\CPLEX_Studio2211\cplex\lib\x64_windows_msvc14\stat_mda
C:\Program Files\IBM\ILOG\CPLEX_Studio2211\cplex\lib\x64_windows_msvc14\stat_mdd
```

Valore valutato:

```
C:\Program Files\IBM\ILOG\CPLEX_Studio2211\cplex\lib\x64_windows_msvc14\stat_mda
C:\Program Files\IBM\ILOG\CPLEX_Studio2211\cplex\lib\x64_windows_msvc14\stat_mdd
%(AdditionalLibraryDirectories)
```

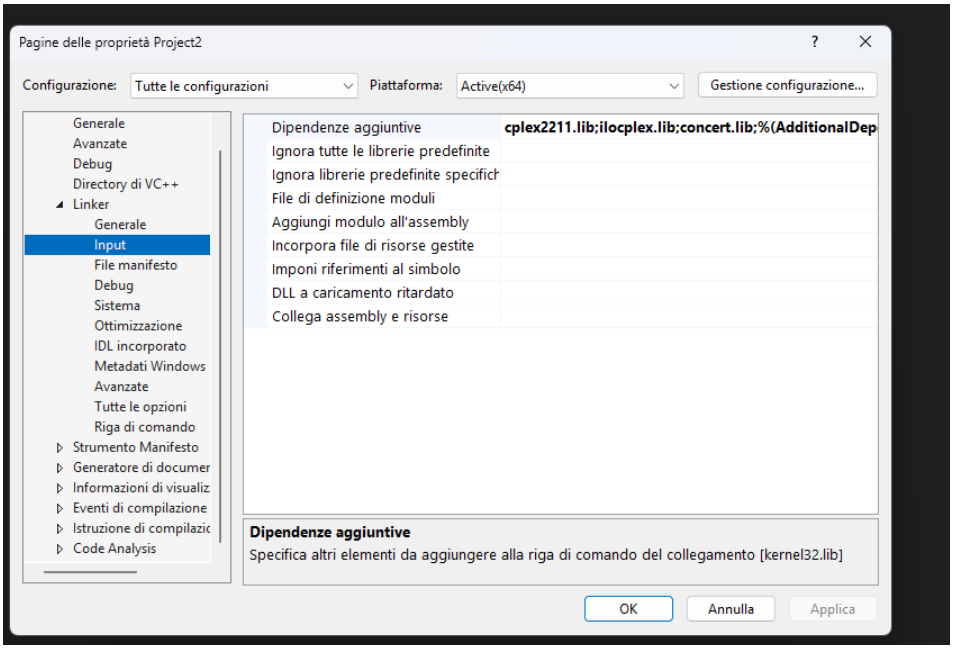
Valori ereditati:

which are:, I remember:

- `C:\Program Files\IBM\ILOG\CPLEX_Studio2211\cplex\lib\x64_windows_msvc14\stat_mdd`
- `C:\Program Files\IBM\ILOG\CPLEX_Studio2211\cplex\lib\x64_windows_msvc14\stat_mda`

Written by Gabriel R.

Then, in the “Linker” > “Input” tab, click on “Additional dependencies”:



Pagine delle proprietà Project2

Configurazione: Tutte le configurazioni

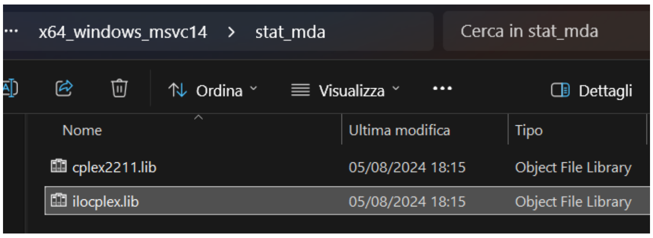
Piattaforma: Active(x64)

Dipendenze aggiuntive	**cplex2211.lib;ilocplex.lib;concert.lib;%(AdditionalDep**
Ignora tutte le librerie predefinite	
Ignora librerie predefinite specifich	
File di definizione moduli	
Aggiungi modulo all'assembly	
Incorpora file di risorse gestite	
Imponi riferimenti al simbolo	
DLL a caricamento ritardato	
Collega assembly e risorse	

Dipendenze aggiuntive

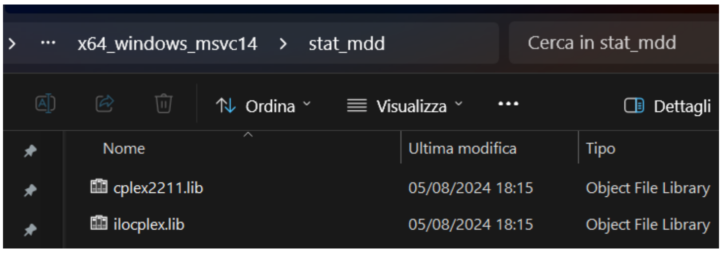
Specifica altri elementi da aggiungere alla riga di comando del collegamento [kernel32.lib]

Add all of the files which are .lib files inside of the mda/mdd folders:



··· x64_windows_msvc14 > stat_mda

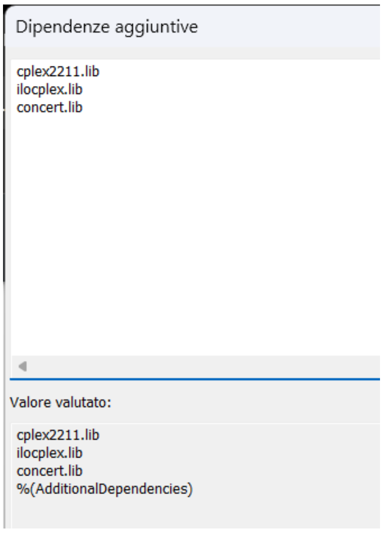
Nome	Ultima modifica	Tipo
cplex2211.lib	05/08/2024 18:15	Object File Library
ilocplex.lib	05/08/2024 18:15	Object File Library



> ··· x64_windows_msvc14 > stat_mdd

Nome	Ultima modifica	Tipo
cplex2211.lib	05/08/2024 18:15	Object File Library
ilocplex.lib	05/08/2024 18:15	Object File Library

They are concert.lib (concert directory of before and the two above files), separated with a semicolon when inserted:



Dipendenze aggiuntive

```
cplex2211.lib
ilocplex.lib
concert.lib
```

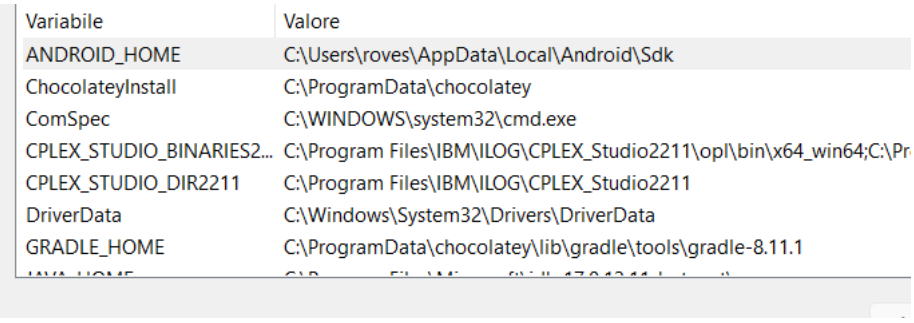
Valore valutato:

```
cplex2211.lib
ilocplex.lib
concert.lib
%(AdditionalDependencies)
```

22 What to include in a CPLEX Project to make it work on Windows

<u>Requirement</u>

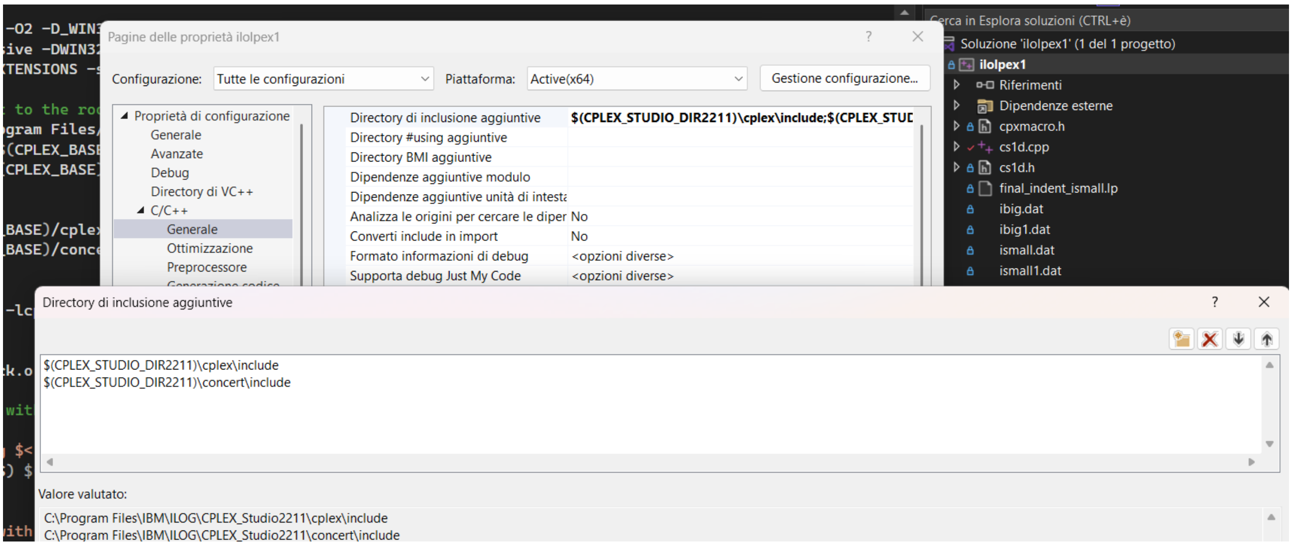
Set as env variable CPLEX to be easily found (test with terminal too):



Variabile	Valore
ANDROID_HOME	C:\Users\roves\AppData\Local\Android\Sdk
ChocolateyInstall	C:\ProgramData\chocolatey
ComSpec	C:\WINDOWS\system32\cmd.exe
CPLEX_STUDIO_BINARIES2...	C:\Program Files\IBM\ILOG\CPLEX_Studio2211\opl\bin\x64_win64;C:\Pr
CPLEX_STUDIO_DIR2211	C:\Program Files\IBM\ILOG\CPLEX_Studio2211
DriverData	C:\Windows\System32\Drivers\DriverData
GRADLE_HOME	C:\ProgramData\chocolatey\lib\gradle\tools\gradle-8.11.1

<u>Step 1: Library Directories</u>

- In Properties, navigate to:
    - Configuration Properties > C++ > General
- Find "Additional Library Directories"
- Add these paths:



Pagine delle proprietà ilolpex1

Configurazione: Tutte le configurazioni Piattaforma: Active(x64)

Directory di inclusione aggiuntive	**$(CPLEX_STUDIO_DIR2211)\cplex\include;$(CPLEX_STUD**
Directory #using aggiuntive	
Directory BMI aggiuntive	
Dipendenze aggiuntive modulo	
Dipendenze aggiuntive unità di intesta	
Analizza le origini per cercare le diper	No
Converti include in import	No
Formato informazioni di debug	<opzioni diverse>
Supporta debug Just My Code	<opzioni diverse>

Directory di inclusione aggiuntive

```
$(CPLEX_STUDIO_DIR2211)\cplex\include
$(CPLEX_STUDIO_DIR2211)\concert\include
```

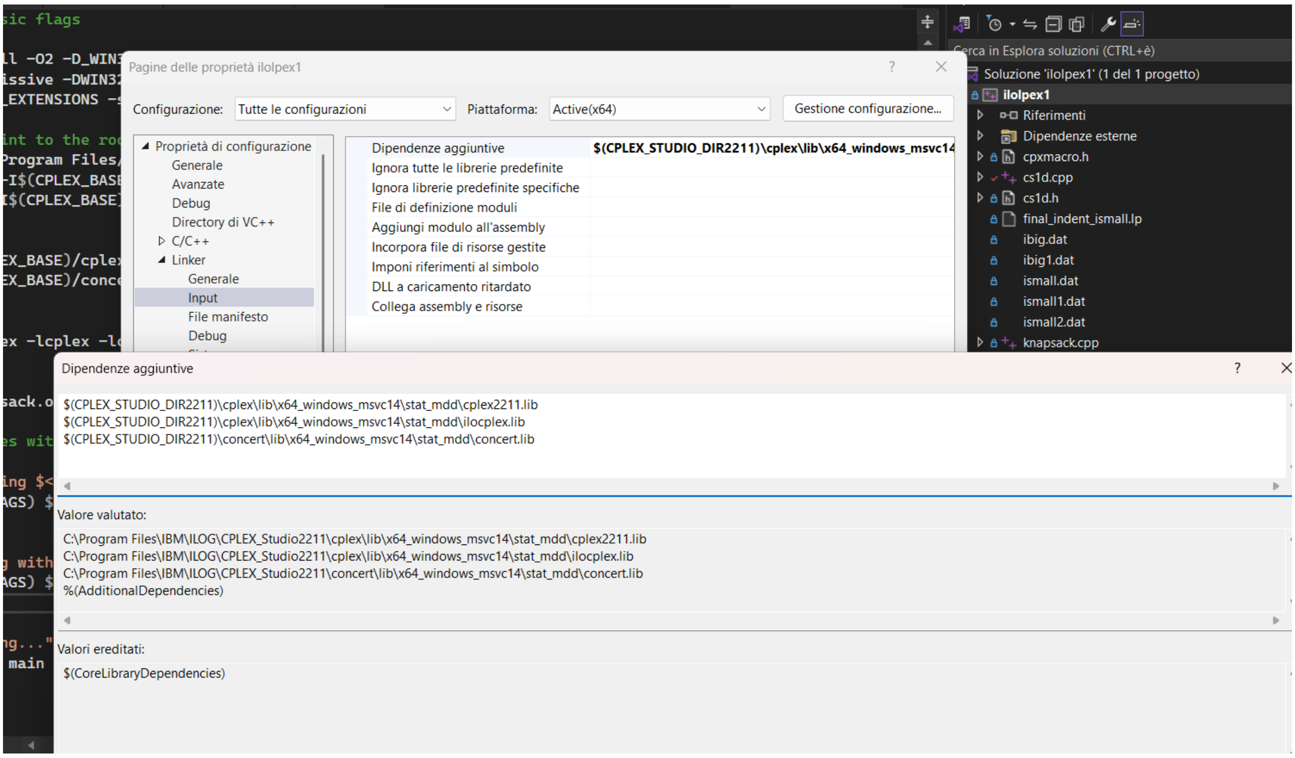
Valore valutato:

```
C:\Program Files\IBM\ILOG\CPLEX_Studio2211\cplex\include
C:\Program Files\IBM\ILOG\CPLEX_Studio2211\concert\include
```

<u>Step 2: Library Dependencies</u>

- In Properties, navigate to:
    - Configuration Properties > Linker > Input
- Find "Additional Dependencies"

- Add these libraries:



Pagine delle proprietà ilolpex1

Configurazione: Tutte le configurazioni   Piattaforma: Active(x64)   Gestione configurazione...

Dipendenze aggiuntive	$(CPLEX_STUDIO_DIR2211)\cplex\lib\x64_windows_msvc14
Ignora tutte le librerie predefinite	
Ignora librerie predefinite specifiche	
File di definizione moduli	
Aggiungi modulo all'assembly	
Incorpora file di risorse gestite	
Imponi riferimenti al simbolo	
DLL a caricamento ritardato	
Collega assembly e risorse	

Dipendenze aggiuntive

```
$(CPLEX_STUDIO_DIR2211)\cplex\lib\x64_windows_msvc14\stat_mdd\cplex2211.lib
$(CPLEX_STUDIO_DIR2211)\cplex\lib\x64_windows_msvc14\stat_mdd\ilocplex.lib
$(CPLEX_STUDIO_DIR2211)\concert\lib\x64_windows_msvc14\stat_mdd\concert.lib
```

Valore valutato:

```
C:\Program Files\IBM\ILOG\CPLEX_Studio2211\cplex\lib\x64_windows_msvc14\stat_mdd\cplex2211.lib
C:\Program Files\IBM\ILOG\CPLEX_Studio2211\cplex\lib\x64_windows_msvc14\stat_mdd\ilocplex.lib
C:\Program Files\IBM\ILOG\CPLEX_Studio2211\concert\lib\x64_windows_msvc14\stat_mdd\concert.lib
%(AdditionalDependencies)
```

Valori ereditati:

```
$(CoreLibraryDependencies)
```