Foundations of Operations Research

Elia Ravella

September 1, 2021

Contents

1 Graphs

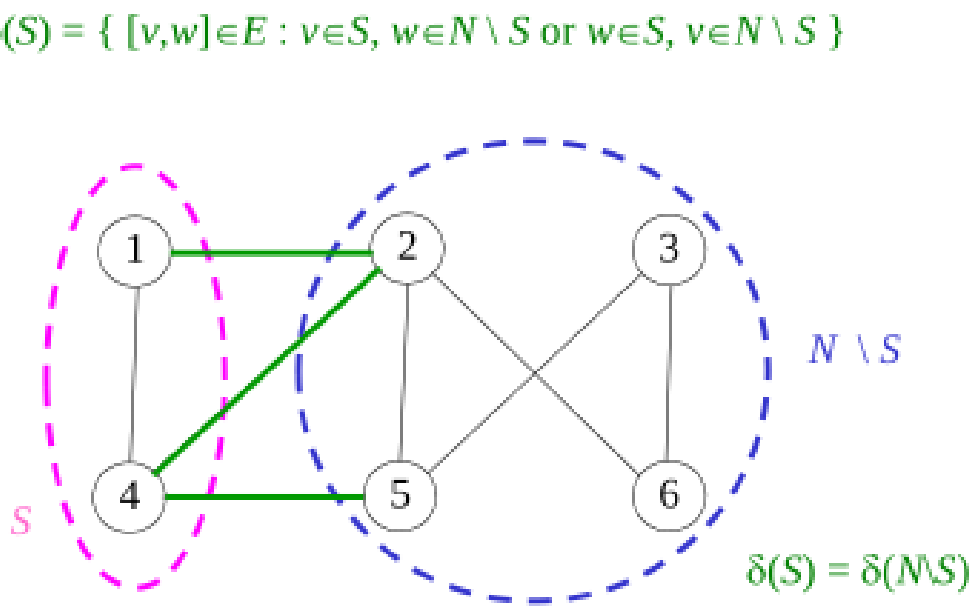
1.1 Definitions

A graph is a pair of sets. Graph $G = (N, E)$ is composed of a set N of nodes and a set E of edges, that are themselves pair of nodes. The intuitive representation is the classic web of interconnected nodes.
Two nodes are said to be *adjacent* if there's an edge connecting them.
An edge is *incident* to a node if it has that node at an endpoint.
The *degree* of a node is the number of incident edges to it.
A *path* is a subset of the E set composed of consecutive edges (the endpoint of an edge is the "start"point of the next one[1]) that connects some edges in the graph.
Two nodes are *connected* if exists a path that links them. A graph con be *connected* to, if and only if there exists a path the connects **all its nodes**.
A graph is *directed* if some of the edges can be traversed in only one way. These edges are called *arcs*.
A *cycle* or *circuit* is a directed path that end where it starts.
A graph is *complete* if for all pair of nodes there's an edge connecting them. Reasoning on this line, we can define the upper bounds for the number of edges given the nodes:

$$\begin{cases} e \leq \frac{n \times (n-1)}{2} \text{ if undirected} \\ e \leq n \times (n-1) \text{ if directed} \end{cases} \tag{1}$$

where intuitively "e" is the number of edges and "n" the number of nodes. Given this two inequalities, we can redefine completeness: a graph is complete if and only if $e = \frac{n \times (n-1)}{2}$ or $e = n \times (n-1)$ depending on the graph being directed.

Cuts Given an undirected graph $G = (N, E)$ we can take $S \subset G$ subsets of nodes; we then observe $\delta(S)$ the *cut* induced by S is the subsets of **edges** that connects S to N \S.



Same definition holds for directed graphs, but this time we can distinguish among *outgoing* and *incoming* cuts, wheter the edges are all exiting S or ending in it.
A graph is said to be *bipartite* if we find a partition $N = (N_1, N_2)$ such as $\delta(N_i) = \emptyset$. So **the two subsets are disconnected**.

[1] not in a directional way: we are still talking about non directed graphs, the two consecutive edges just in fact share an endpoint

A bit of foreshadowing: if a cut from a partial tree to another portion of the graph holds a minimum cost arc, then *there exists a spanning tree with that arc.* This is called the cut property.

Data Representations for Graphs Depending one the edges / nodes ratio, two different data structures can hold a valid representation for a graph. When $e \approx n^2$ (a dense graph) then an adjacency matrix is the best choice. Otherwise (sparse graphs) linked lists of successors for each node.

1.2 Reachability

The reachability problem can be formulated as "given a graph, determine all the nodes that are reachable from a starting one, decided a priori. A node is reachable is there's a path from the starting one and itself".

1.3 Subgraphs, Trees, Spanning Trees

We call $G' = (N', E')$ a subgraph of $G = (N, E)$ if and only if

$$\begin{cases} N' \subseteq N \\ E' \subseteq E \\ \text{all edges inside } E' \text{ connects nodes in } N' \end{cases} \tag{2}$$

A *tree* $G_t = (N', E')$ is a subgraph of a network that's both **connected** and **acyclic**. A tree is *spanning* if it contains all the nodes of the original graph. The *leaves* of a tree are the nodes with unitary degree.

Properties of trees:

- Every tree with a number of nodes greater than 2 has at least 2 leaves
- All trees have number of arcs equal to number of nodes minus 1
- Every pair of nodes in a tree is connected by a unique path (acyclic connected graph)
- By adding an edge to a tree, we create a *unique cycle.* This also mean that if we now remove *any* of the edges of this new unique cycle we go back to a spanning tree
- A complete Graph with n nodes has exactly n^{n-2} spanning trees

1.4 Minimum Cost Spanning Tree

Problem: find in a graph $G = (N, E)$ with *weighted arcs*[2] a subgraph that is a tree and has minimum total cost of edges. The "minimum total cost" condition is verified when the sum of the costs along edges cannot be decreased by swapping edges in and out from the tree.

Prim's Algorithm The Prim's algorithm is a simple iterative algorithm to build a spanning tree starting from an initial node.
The algorithm adopts an incremental greedy strategy: it adds to the (initally empty) tree the nearest node every time (nearest = connected with the minimum cost outgoing edge) until the tree has the same number of nodes as the original graph.
The full algorithm:

1. initialize the output: $S = (N', T)$ where N' is composed only of the initial chosen node and $T = \emptyset$ is the set of edges
2. add to S's nodes the "nearest" node from the ones connected to it, and add to T the lowest-cost edge. Careful: we are watching at *all the nodes reachable from the outgoing cut of S*, not only the ones from a single node
3. repeat step 2 until the graph is covered

Prim's algorithm is exact[3].
This algorithm exploits the **cut property**.

[2] numerated arcs with a "cost" associated to them
[3] Formally proven to provide an optimal solution for each instance of a problem

1.5 Shortest Path and Shortest Path Trees

Finding the shortest path on a graph from node A to node B is another classical problem. The most used algorithm is the famous Dijkstra's algorithm.

Dijkstra Algorithm Dijkstra algorithm is similar to the Prim's one, but applied in a different direction. Dijkstra starts from a subset of the nodes (called the "unvisited nodes") and considers themone at a time. At each iteration, a node is selected from the unvisited set (the first is chosen a priori, will be the *initial* node) and for all his neighbours is updated the distance. When all neighbours have been inspected, the node is marked as "visited"[4]. Next node to be selected is the *closest* and the algorithm restarts. It finishes when the "unvisited" set is empty. More schematically:

1. Select the closest node (first one a priori)
2. Check his neighbours and update their distance
3. When all neighbours have been inspected, go to point one. If no more unvisited nodes exists, end.

At the end of the algorithm we have a *shortest path tree*[5] that highlights all the shortest paths from the initial node to all other nodes in the graph. Taking the *closest* node at each iteration checking the outgoing cut from the "visited nodes" set ensures the exactness of Dijkstra algorithm.

Floyd Warshall Algorithm Negative-cost edges, if present, do not allow to apply Dijkstra algorithm. But they are a further threat to shortest path problems: if a *negative cost cycle* exists, there's no minimum cost path that's also finite between two nodes that traverse that cycle. This is a ill-defined problem, and the Floyd Warshall algorithm detects it. This algorithm uses two matrixes: a distance and a predecessor one. At each iteration of the algorithm, a *triangular check* is performed: for all his neighbours, an undirect path is searched (a multi step path, generally with a single intermediate node) to shorten the distance wrt the direct path.

1. initialize a counter h at 1, it will be our index for the nodes
2. for each value of the counter from 1 to the number of nodes, perform the check $d_{ij} > d_{ih} + d_{hj}$ to evaluate if there is an alternative two-steps path that reduces the distance between nodes i and j.
3. if a loop with negative cost is detected (so a node can go from itself to itself with less than 0 distance passing through another node) that the problem is flagged as ill and the algorithm terminates

At the end, combining the two matrixes, we have a shortest path tree of the original graph. This time, it can be calculated with negative cost arches.

1.5.1 Direct Acyclic Graphs and Dynamic Programming

If we add the hypothesis that the graph we're working on (to find the shortes path or the minimum cost spanning tree) does not contain cycles, we can study some more interesting solutions to such a problem. Moreover, DAGs[6] are well suited to model a wide range of problems.

Topological Ordering DAGs can be *topologically ordered*, which means that we can label the nodes as if the graph represents an order relation between them (in the algebraic sense). The formal definition is that

$$\forall (i,j) \in A \,|\, i < j \tag{3}$$

where A is the set of edges. This looks like a trivial additional property to a graph, but having a *strictly monotone ordering of nodes* is super helpful.
To order nodes in such a way, however, the algorithm is quite simple:

1. find a node with no incoming edges[7] and assign the smallest index available

[4] and never checked again
[5] that's **different** from a minimum cost spanning tree
[6] directed acyclic graphs
[7] the existence of such a node is proved by the acyclicity condition

2. remove the node from the graph and return to point 1

Dynamic Programming The "dynamic programming" approach has a bit of a confusing name. The underlying idea is that we can exploit a recursive relation between costs along a shortest path, using the "shortest subpath" property: given π_{ij} shortest path $i \to j$ we can divide it in $\pi_{it} + c_{tj}$ where π_{it} is the *shortest subpath from i to t* and the second term is the cost of the last step. If we define $L(i)$ as the cost of the shortest path from the initial node to node i we have that

$$L(t) = min\{L(i)_{i \text{ predecessor of } t} + c_{it}\} \tag{4}$$

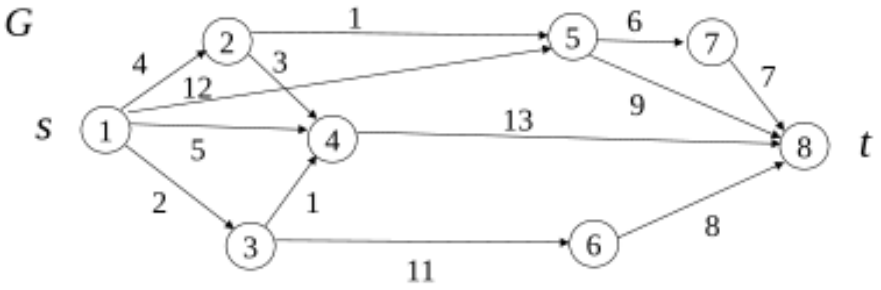
This is a recursive relation that can be expanded to all nodes in the path. I'll put here the example by the professor:

$$L(1) = 0 \qquad \text{pred(1)=1}$$

$$L(2) = L(1) + c_{12} = 0 + 4 = 4 \qquad \text{pred(2)=1}$$

$$L(3) = L(1) + c_{13} = 0 + 2 = 2 \qquad \text{pred(3)=1}$$

$$L(4) = \min_{i=1,2,3}\{L(i) + c_{i4}\} = \min\{0 + 5, 4 + 3, 2 + 1\} = 3 \qquad \text{pred(4)=3}$$



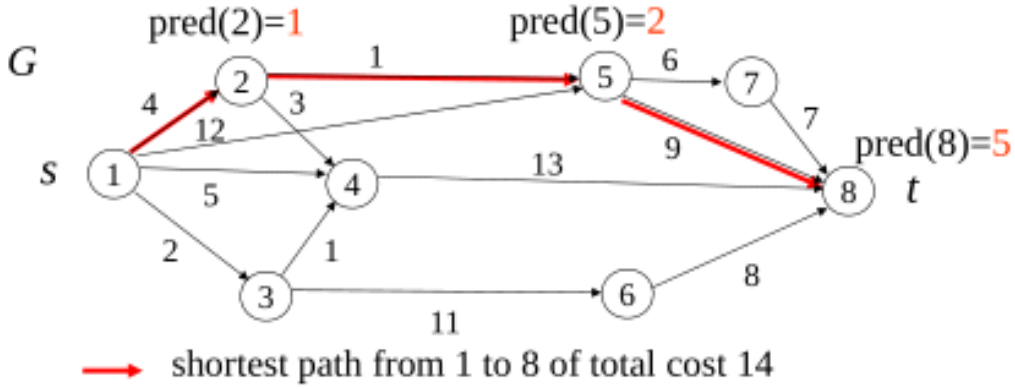
[.] indicates the predecessor of node i in a shortest path from 1 to i

$$L(5) = \min_{i=1,2}\{L(i) + c_{i5}\} = \min\{\ 0 + 12,\ 4 + 1\ \} = 5 \qquad \text{pred}(5)=2$$

$$L(6) = \min_{i=3}\{L(i) + c_{i6}\} = \min\{\ 2 + 11\ \} = 13 \qquad \text{pred}(6)=3$$

$$L(7) = L(5) + c_{57} = 5 + 6 = 11 \qquad \text{pred}(7)=5$$

$$L(8) = \min_{i=4,5,6,7}\{L(i) + c_{i8}\} = \min\{\ 3+13,\ 5+9,\ 13+8,\ 11+7\ \} = 14 \qquad \text{pred}(8)=5$$



1.5.2 Project Planning

As shown in the Examples, we can use graphs to express dependency relationships. A direct application of this model is used when organizing various activitie inside a project: each phase will depend from some previous phases, there will be an "initial" phase with no predecessors and a final phase with no successors. The model adopted here is the one that represents

- *activities* as arcs
- *activity duration* with arcs weight
- *checkpoints* as nodes

This formulation leads to a weighted directed acyclic graph. It's acyclic *by nature*: it would be meaningless (and also logically incorrect) if activities in a project had circular relationships.
This model helps in finding the *minimum overall project duration*: being directed and acyclic, this will be the duration of the *longest path* from the initial node to the final (we cannot compress the project time beyond that).

Critical Path Method The CPM aims to provide an optimal schedule for a project, with allotted time for each activity and the possible slack of each one, given the DAG of a project activities precedences. The algorithm is:

1. Construct the DAG of activities and precedences
2. Topologically sorts the nodes
3. For each node, calculate
   - The earliest start time starting from inital activity. This would be (at the final node) the minimum overall project duration
   - The latest time an activity can be started to NOT increase the minimum project duration (starting from the end, this time)

4. For each activity now we can calculate the *slack*: $T_{max} - T_{min} - d$ where d is the duration of the activity itself

The slack of each activity indicates if it could be *deliberately delayed* without affecting the overall project duration. There exist activities with null slack: these are called critical activities and cannot be delayed in any case. They compose the critical path from the beginning to the end of the project.

1.5.3 Network Flows

Network flows problems are the ones that involves "distribution of a given resource over a network". Needless to say, they boils down to find the optimal throughput of a network given its *Directed Weighted Graph* representation. In this kind of problems, we can remove the acyclicity hypothesis. The graph model we're dealing with is like:

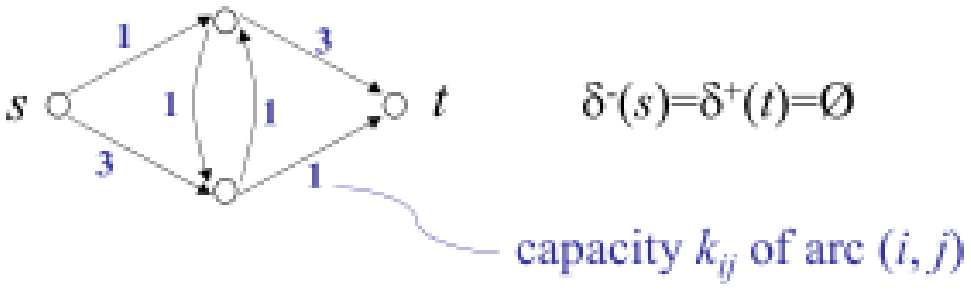


Figure 1: Arcs weights assume a different meaning here: they are not more a cost or a lenght, but instead a *width* or *capacity* as shown in the blue note

We introduce the *feasible flow vector*: it's a tuple of values representing the **occupated fraction** of each edge of the network; it should respects two major constraints:

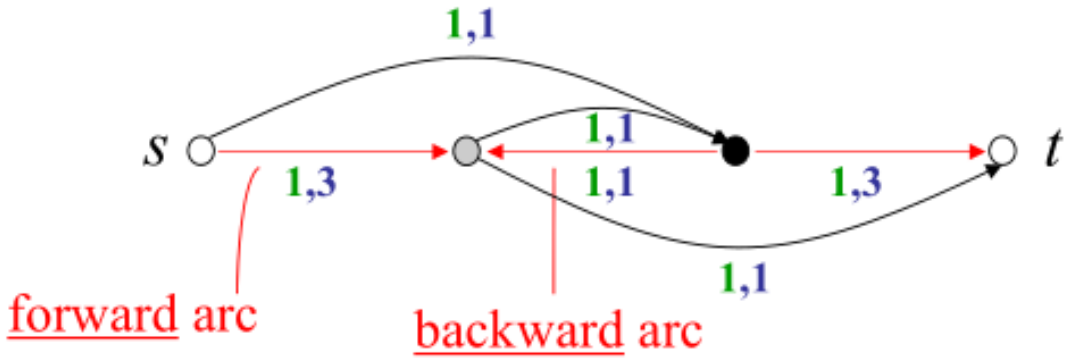
1. Capacity constraints: $\forall (i,j) \in E \,|\, (0 \leq x_{ij} \leq k_{ij})$ where x_{ij} is the value of the flow in the feasible flow vector, k_{ij} the maximum capacity for that arc and E the set of edges;
2. Flow balance constraints: $\forall h \in N \,|\, (\sum(x_{ih}) = \sum(x_{hj}))$ that simply states that for each node, the amount that enters the node is the same amount that exits it

We can then calculate the *value of flow* $\phi = \sum(x_{sj})$, where s is the initial node and j all the nodes reached by the *outgoing cut* of s; this represents the actual maximum flow that can go through the network.[8]

Ford Fulkerson's Algorithm Another algorithm, this time to resolve a network flow problem. Again, this is just the algorithmic version of the intuitive solution for this problem: keep sending stuff on non-saturated channels until the network is maxed out.
Said so, we need some additional definition to handle easily this algorithm:

- a *backward arc* is an edge of the network that "sends units backward" with regard to a simil-topological ordering of the nodes. We can "flat out" the graph of the network to visually understand what a backward arc is:

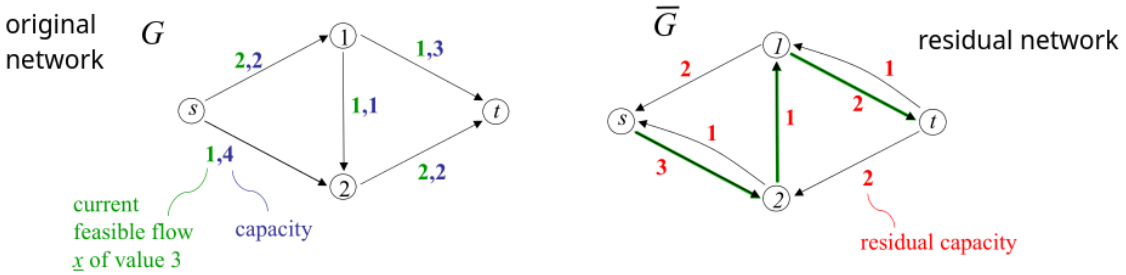
[8]The maximum flow is directly associated with the minimum capacity of a cut. Will see later how these, in fact, are mathematically correlated: *they're dual problems*, or two representations of the same problem.



- an *augmenting path* (wrt the current ϕ) is a path through all the network (so start to end) where

$$\begin{cases} x_{ij} < k_{ij} \text{ for all forward arcs} \\ x_{ij} > 0 \text{ for all backward arcs} \end{cases} \tag{5}$$

- the *residual network* of a graph is a graph itself buil keeping track of the possible changes that can be applied to the flows on the arcs. This means, if we have a saturated channel with capacity 3 we will have in the residual network the same channel but *reversed*: this signals that we have a residual capacity of 3 units *in the opposite direction*.



So, the algorithm is structured in 2 main phases:

1. Search for an augmenting path on the actual graph, then rebuild the residual graph. Repeat this point until no more trivial augmenting path can be found
2. Look for an augmenting path *on the residual graph*. Using the second graph is useful when some edges could be desaturated in order to maximize the flow. The algorithm stops when on the residual graph there are no more paths from s to t

2 Complexity

An *algorithm* is a set of instructions that allows a machine to solve all instances of a given problem. Its executions time depends on the instance itself (usually its dimension) and the computer carrying out the operations. The complexity of an algorithm is the "number of elementary operations[9] needed to complete it" in the worst case (so big-O scenario / notation).
Complexity is used to estabilish two kind of things:

- The complexity of a given algorithm to solve a given problem
- The inherent difficulty of a problem

2.1 Problems and Instances

If we want to estimate the performance of *alternative* algorithm on a given problem, we should not forget that performances are conditioned by the instance size in a significative way. The formal definition for "size of an instance" is the number of bits needed to encode it. Quick example (take a look at indices)

$$I = \begin{cases} m \\ c_1 ... c_m \end{cases} \tag{6}$$

both m and c_j are integers. The computation of the instance size is:

$$|I| \leq \lceil \log_2(m) \rceil + m \times \lceil \log_2(c_{max}) \rceil \tag{7}$$

Where c_{max} is the maximum integer value among c_j. We take the max because we want to encode all numbers the same way. *m is exactly the number of c values, pay attention.*

2.2 Complexity Classes

When looking at an algorithm we usually want to calculate its time complexity: we're looking for a function that is an upper bound of the time needed to complete, dependently from the instance size. This function is usually expressed with asymptotic terms, with the big-O notation. This said, we can divide algorithms in classes of complexity, wheter they are asymptotically similar. The two classes of complexity of interest are

- Polynomial: $O(n^d)$
- Exponential: $O(2^n)$

where n,as usual, indicates the instance size.

2.2.1 Complexity Classes for *Problems*

We can identify the *problems* with the algorithm[10] that solves them. In this way, we can distinguish between "easy" and "hard" problems. We classify problems in the two infamous classes

- $\mathcal{P}$ problems: the recognition[11] problems that can be *solved in polynomial time*
- $\mathcal{NP}$ problems, or *nondeterministic polynomial* problems; these problems CAN'T be solved in polynomial time[12], but a solution can be verified in polynomial time.

Speaking in Turing machine terms, the problems in class $\mathcal{P}$ can be solved by a standard Turing machine, while the ones in $\mathcal{NP}$ by a nondeterministic one. N.B.: the nondeterministic Turing machine is just a Turing machine that *verifies*[13] a *guessed* solution, ence the name "nondeterministic polynomial".
Millennium prize problem: is $\mathcal{P} \equiv \mathcal{NP}$?

[9] memory access, arithmetic ops, conrontations
[10] the most efficent so far, generally
[11] estabilishing if a property is verified, a true/false response.
[12] so far
[13] in polynomial time

Reductions Reducing a problem to another means, more or less, solving a problem with another one. We say that P_1 *reduces in polynomial time* to P_2 if we can solve P_1 using an algorithm that solves P_2 as subroutine, calling it a *polynomial number of times*.
It's straightforward to notice that if the algorithm used to solve P_2 has constant complexity, then P_1 will have a overall polynomial complexity. A reduction where the second algorithm is called only once is a *transformation*.

$\mathcal{NP}$-complete problems A problem is said to be $\mathcal{NP}$-complete $\Leftrightarrow$

- it belongs to $\mathcal{NP}$ and
- every other problem in $\mathcal{NP}$ can be reduced to it in polynomial time

So these problems are

1. nondeterministic polynomial, so hard to solve but simple to verify (hard means above-polynomial, simple means polynomial or less)
2. complete, because we can use them to represents a whole set of problems

$\mathcal{NP}$-hard problems A problem is $\mathcal{NP}$-hard when "every other problem in $\mathcal{NP}$ can be reduced to it in polynomial time". This problematic definition is cleared by this observation, I think: all recognition problems that are $\mathcal{NP}$-complete have their optimization version that's $\mathcal{NP}$-hard.

3 Linear Programming

3.1 Linear Programming Problems

A linear programming problem is an *optimization* problem in the form

$$\begin{cases} \min\ f(\underline{x}) \\ \text{s.t. } \underline{x} \in X \subseteq \mathbb{R}^n \end{cases} \tag{8}$$

where

- $f : X \to \mathbb{R}$ is a *linear function*
- the *feasible region* $X \subseteq \mathbb{R}^n$ is a combination of linear functions.

We then define the *optimal solution* of a linear programming problem as $\{\bar{\underline{x}} \,|\, f(\bar{\underline{x}}) \leq f(\underline{x})\}$ for all other vectors in the feasible region.
LP problems can also be formulated with matrixes:

$$\begin{cases} \min\ z = \underline{c}^t\, \underline{x} \\ A\underline{x} \geq \underline{b} \\ \underline{x} \geq \underline{0} \end{cases} \tag{9}$$

Assumptions of LP Models LP models and the algorithms operating on them work under several assumptions:

1. Linearity of the objective function and constraints
2. Divisibility of the variables: there are no "undivisible units"
3. Constant parameters in the model. Also, accurate parameters

3.2 LP Problems Forms

The most general for for a linear programming model we can define is:

$$\begin{cases} \min/\max\ z = \underline{c}^t\, \underline{x} \\ A_1\underline{x} \geq \underline{b_1} \\ A_2\underline{x} \leq \underline{b_2} \\ A_3\underline{x} = \underline{b_3} \\ x_j \geq 0 \text{ for some } j \end{cases} \tag{10}$$

This form is usually referred to as *canonical form.*

Standard Form The standard form is the "most restrictive" one, and also one of the easiest to work with

$$\begin{cases} \min\ z = \underline{c}^t\, \underline{x} \\ A\underline{x} = \underline{b} \\ \underline{x} \geq \underline{0} \end{cases} \tag{11}$$

In this form, all constraints must be equalities, and all variables must be non negative. Pay attention: in the original general model, we have the non-negativity onstraints only on some variables.

Transforming to Standard Form We can easily transform a general formulation of a LP problem in standard one by applying these simple transformations:

- $\max(\underline{c}^t\underline{x}) = -\min(-\underline{c}^t\underline{x})$ to change objective function

- all inequalities are transformed adding a "slack variable":

$$\underline{a}^t \underline{x} \leq \underline{b} \Rightarrow \begin{cases} \underline{a}^t \underline{x} + s = \underline{b} \\ s \geq 0 \end{cases} \tag{12}$$

  or a surplus variable in case it's a $\geq$ inequality

$$\underline{a}^t \underline{x} \geq \underline{b} \Rightarrow \begin{cases} \underline{a}^t \underline{x} - s = \underline{b} \\ s \geq 0 \end{cases} \tag{13}$$

- when a variable is unrestricted in sign it's "splitted" in its positive and negative part:

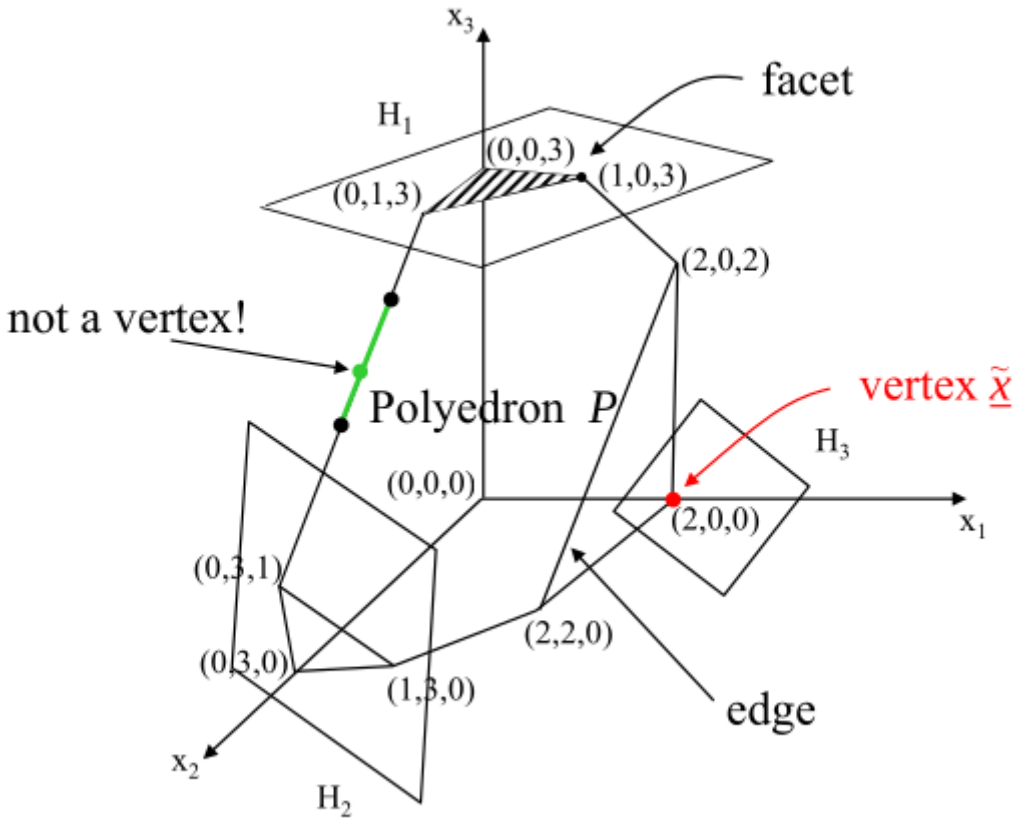
$$x_j \in \mathbb{R} \Rightarrow \begin{cases} x_j = x'_j - x''_j \\ x'_j \geq 0 \\ x''_j \geq 0 \end{cases} \tag{14}$$

3.3 Geometry of a LP Problem

We assume known the notions of "level curve" for a function (a set of points where a multi variable function is constant) and gradient[14].

3.3.1 Feasible Region as an Hyperplane

By definition, $\{x \in \mathbb{R}^n \,|\, \underline{a}^t x = \underline{b}\}$ is an hyperplane, while $\{x \in \mathbb{R}^n \,|\, \underline{a}^t x \geq \underline{b}\}$ an affine half space, defined as a combination of all inequality constraints.
So, the *feasible region* of an LP problem, defined as *a region delimited by hyperplanes*, is called a *polyhedron*. A polyhedron (so a polygon in two dimension, a 3D figure in three dimension and so on) is a convex set of $\mathbb{R}$ as generated by convex sets.



[14] Also its geometric interpretation as the *direction of maximum growth*.

We call a *convex combination* a linear combination where all coefficients are non negative and sum to 1. We define a *vertex* of a polyhedron as a point in the ph that cannot be expressed as the convex combination of two other dintinct points in the ph. Don't worry if you're confused: all "polyhedra theory" has a lot of dark spots[15]. Non-empty polyhedra representing an LP problem (in both canonical or standard form) **has a finite number of vertices**, greater or equal to one[16]. We can represent *every point of a polyhedron* as a convex combination of its *vertices* plus (if needed) a vector that never ends inside the polyhedron (for unbounded polyhedron).

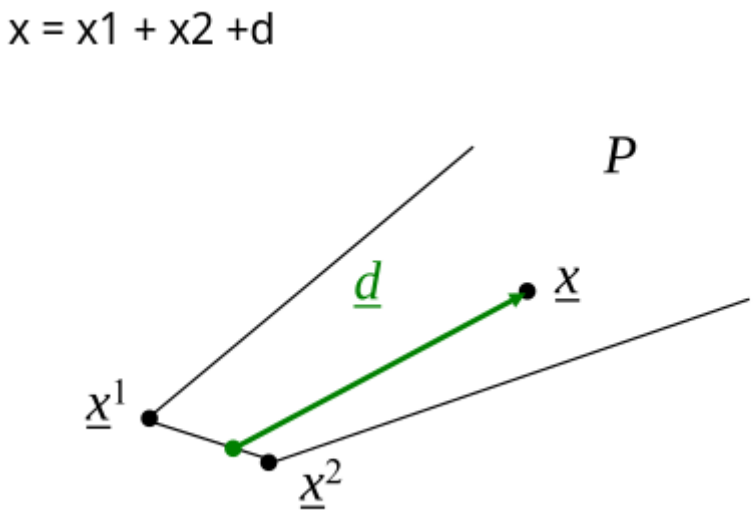


Figure 2: Every point of this unbounded polyhedron can be represented as a combination of x_1 and x_2 and $\underline{d}$, where $\underline{d}$ is called *unbounded feasible direction* of the polyhedron.

3.3.2 Fundamental Theorem of Linear Programming

Consider a LP $\{\min(\underline{c}^t x\}$ where $x \in P$, which is a non-empty polyhedron of the feasible solutions (in standard or canonical form). Then either

1. exists at least a single optimal vertex
2. the value of the objective function is unbounded below on P

This foggy theorem has a simple intuitive interpretation, in my mind: the feasible region is represent as a subset of $\mathbb{R}$ and reprents *all the solutions of the system of constraints*. To minimize the objective function, we must "move around" this subspace in a well-defined direction, that is the opposite of the gradient of the objective function itself. Being this a *straight line*, we will eventually hit a boundary of the polyhedron. We could then "slide" onto it until we are "stuck" and every move we make is not anymore bettering the solution. This can happen *only* on the intersection of boundaries (so the vertices), because it's the only point where a "direction" to better the solution finds his boundary[17].
This theorem allows us to do determine a priori which are our solutions, just looking at the way the feasible region is defined. Also, being *finite*, we can cycle through the solutions to "easily" find the right one.

3.4 Algebraic Characterization of the LP Problem

We've seen we can exploit the geometry of the feasible region to find the solutions. We need an algebraic formulation to do that algorithmically.

[15] check out they wikipedia page
[16] a polyhedron with only one vertex is a quarter of plane, for example
[17] I find this mumbo jumbo more clear than the slides definition, ok? In my head it's clear this way.

3.4.1 The Feasible Vector Space

Classic LP problem: $P = \{x \in \mathbb{R} \mid Ax = b, x \geq \underline{0}\}$ in standard form. We assume from now on that A is full rank. This leads to two possible scenarios:

- A is a square matrix $\Rightarrow$ there exists only a *unique* solution to the system: $x = A^{-1}b$.
- A is a rectangular matrix with more columns than rows: this leads to infinite solutions of the system.

Obviously, having the exact number of constraints equal to the variables one is not common. We focus on the second case.
If we partition the matrix $A = [B \mid N]$, where B is a basis[18] and N the "remaining" vectors, we can redefine the solutions' vector as:

$$x = (x_B,\, x_N) \tag{15}$$

where the two components of the vector will be long as the rank of the matrix and the lenght of the remaining matrix. We then substitute into the feasible region definition

$$Bx_B + Nx_N = \underline{b} \tag{16}$$

and so

$$x_B = B^{-1}\underline{b} - B^{-1}Nx_N \tag{17}$$

from this last equation, we can define

- a basic solution is a solution setting all non base variables x_N to zero, so $x_B = B^{-1}\underline{b}$
- a basic *feasible* solution is a basic solution that has all components greater or equal to zero
- the variable in x_B are called basic variables and the ones in x_N non basic variables

Now we link the algebraic formulation with the geometrical interpretation:

Theorem $x \in \mathbb{R}^n$ is a *basic feasible solution* $\Leftrightarrow$ x is a *vertex* of $P = \{x \in \mathbb{R}^n \mid Ax = \underline{b}, x \geq \underline{0}\}$

3.5 The Simplex Method

Dantzig's simplex method is an algorithm to find the best basic feasible solution of a LP problem: it examines a sequence of BFSs with *non increasing objective function values* until an optimal solution is reached, or the problem is found to be *unbounded*.

3.5.1 The Algorithm

1. check for infeasibility of the problem
2. find an initial vertex
3. move from a current vertex to a *better adjacent vertex* (or estabilish that the problem is unbounded)
4. determine if the current vertex is *optimal*

Infeasibility Check To check if a problem is infeasible, we create an auxiliary problem with artificial variables (from the original problem) as:

$$P = \begin{cases} \min z = \underline{c}^t x \\ Ax = \underline{b} x \geq \underline{0} \end{cases} \tag{18}$$

$$P_{aux} = \begin{cases} \min v = \sum_{i=1}^{m} y_i \\ Ax + Iy = \underline{b} \\ x \geq \underline{0}, y \geq \underline{0} \end{cases} \tag{19}$$

obviously, there exists a BFS composed by all elements in y. We face two cases now:

[18] set of linearly indipendent vectors

1. $v > 0$ the problem is ***infeasible***
2. if $v = 0$, the problem is feasible and we must manipulate the matrix in order to obtain a BFS that only depends on the original x_i variables

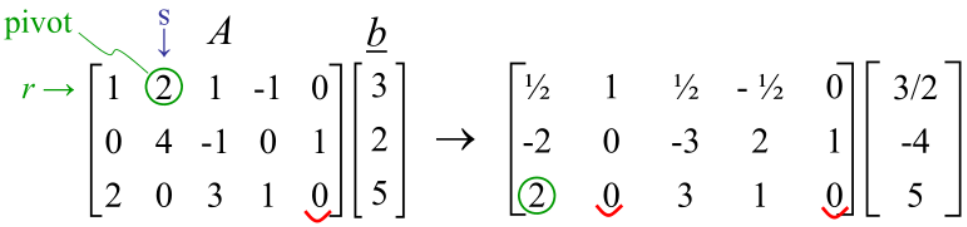
Don't forget we have to turn to the original problem and recompute the objective function.

Initial Vertex Selection The selection of a starting initial BFS is a problem per se. If the problem is easy enough we'll have the initial matrix already in a $A = [N \mid I]$ form, where I represents the identity matrix $\Rightarrow$ a basic feasible solution. In that case, we have already a perfect indication of which variables are in the basis and at which cost, so we can directly go to the "movement" phase (that's the one that really optimizes the objective function value).
In the other case (that's obviously more realistic) we have to resort again at the "auxiliary problem" with artificial variables: the initial basis we find is the one determined by the y_i vectors.

Movement Across Vertices To "move" from a vertex to another means to swap some of the basic vertices out of the solution and "bring in" some of the vectors that were in the N matrix. *Algebrically*, this means to *expressing the basic variables in terms of the non basic ones.* **Remember**: non basic variables are set to **zero**. Algorithmically, this is done through a simple pivoting operation on the combined $[B|N]b$ matrix.
Given $Ax = \underline{b}$

1. select a coefficient $a_{rs} \neq 0$ of table A where r is the row index and s is the column index
2. divide row r for a_{rs} (also in order to have $a_{rs} = 1$)
3. for alla other rows, subtract row r multiplied by the coefficient on column s: that is, a_{is}

This will render the s column all zeroes except the 1 on row r. Don't forget we're applying this procedure also the the $\underline{b}$ vector.



The choice of r and s are crucial: in fact, they select which variable enters the basis (through the pivot column) and which leaves it (through the pivot row). This choice can be taken

- randomly
- using heuristics
- usign **Bland's rule**, that also avoid cycles in the algorithm

Bland's Rule Bland's rule dictate a choice for rows and columns to enter/leave the basis. It states:

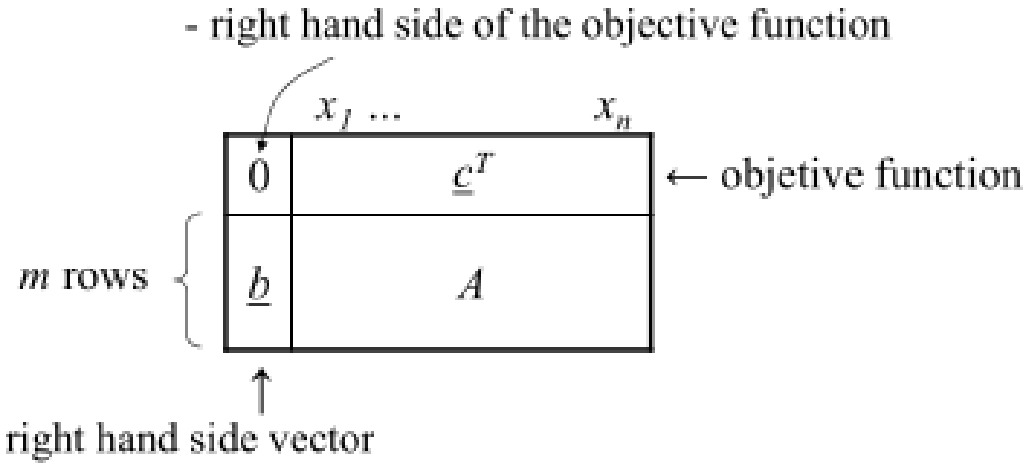
1. select s the *first column with negative reduced cost*
2. then select r the row with the *smallest* $\frac{b_r}{a_{rs}}$ ratio **among the positive ones**

This simple criterion avoids cycles in the Simplex algorithm, that can be stuck in a vertex for some degenerate LP problems. In practice, due to the additional complexity added and the rarity of such problems, Bland's rule is never used.

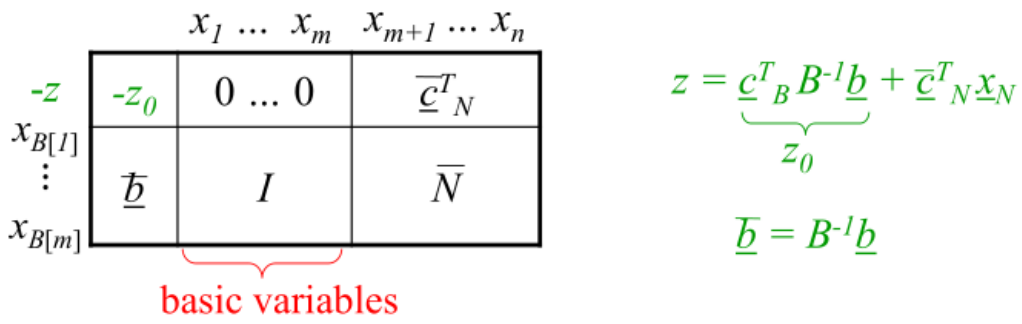
Tableau Representation As spoiled when talking about movement across vertices, the Simplex method has a peculiar way to represent the matrixes and vectors it uses to ease the computing problem.
Given

$$P = \begin{cases} z = \underline{c}^t x \\ Ax = \underline{b} \end{cases} \tag{20}$$

We initialize the tableau of values as



Then we proceed (through the pivoting operation) to put the tableau in canonical form:



Optimality Check To check if a solution is optimal we cannot only watch the feasible region, we must take into account also the objective function. We repeat the reasoning made for finding the expressions of BFSs: given $x_B = B^{-1}\underline{b} - B^{-1}N\underline{x_N}$ basic solution and posing x_N equal to zero to find the basic feasible solution, we can express the $\{\min(\underline{c}^t x)\}$ objective function as

$$\underline{c}^t x = (c_B, c_N) * \begin{pmatrix} x_B \\ x_N \end{pmatrix} = \begin{pmatrix} B^{-1}\underline{b} - B^{-1}N\underline{x_N} \\ x_N \end{pmatrix} \tag{21}$$

we can now separate

$$\underline{c}^t x = \underbrace{c_B^t B^{-1} b}_{z_0} + \underbrace{(c_N^t - c_B^t B^{-1} N) x_N}_{\overline{c_N}} \tag{22}$$

where

- z_0 is a *constant value* and is called cost of the basic feasible solution

- the second term $\overline{\underline{c_N}}$ is a function *only of the non basic variables* and is the *reduced cost* o the non basic variables

We can also define de vector of reduced costs wrt the basis itself: $\overline{\underline{c}} = \underline{c}^t - \underline{c}_B^t B^{-1} A$ that is

$$\begin{pmatrix} c_B^t - c_B^t B^{-1} B \\ c_N^t - c_B^t B^{-1} N \end{pmatrix} = \begin{pmatrix} 0 \\ \overline{\underline{c_N}} \end{pmatrix} \tag{23}$$

The reduced costs gives a measure on how much the objective function changes wrt a change in the value of the variable. Also, they provide a sufficent (but not usually necessary) condition for **optimality**: if all reduced costs of non basic variables are non negative, then the basic feasible solution associated to that non negative variables is optimal. So, more formally:
Given $P = \{\underline{c}^t x : [B|N]x = \underline{b}, x \geq \underline{0}\}$

$$\overline{\underline{c_N}} \geq \underline{0} \Rightarrow (x_B, x_N) \text{ s.t. } x_B = B^{-1}\underline{b} \geq \underline{0} \wedge x_N = \underline{0}, x_B \text{ is optimal} \tag{24}$$

3.6 Duality

The **duality** concept for linear programming can be explained as: to any minimization (maximization) LP problem we can associate a closely related maximization (minimization) LP problem *based on the same parameters*. For example, the famous maximum network flow problem is the dual of the minimum capacity cut problem.

3.6.1 Building the Dual Problem

Given a maximization problem $P = \{\max \underline{c}^t x, Ax \leq \underline{b}\}$ any feasible solution provides a *lower bound* of the objective function value. "Which one is the best lower bound"? Can we "flip" the problem to a minimization one?
The basic idea is to find a *linear combination of the constraints* in such a way that the obtained value for the constraints *dominates*[19] the objective function. So, a new set of variables should be introduced (the coefficents of the linear combination) and should be linked to the *coefficents of the objective value*. These new constraints will create the new feasible region. The objective function the must be changed also: it has to tune each new constraint with the original value of such equation. An example will do the trick.

Example Given

$$\begin{cases} \max z = 4x_1 + x_2 + 5x_3 + 3x_4 \\ \begin{pmatrix} 1 & -1 & -1 & 3 \\ 5 & 1 & 3 & 8 \\ -1 & 2 & 3 & -5 \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} \leq \begin{pmatrix} 1 \\ 55 \\ 3 \end{pmatrix} \\ x \geq \underline{0} \end{cases} \tag{25}$$

If we just multiply the second row of matrix A for $\frac{5}{3}$, we obtain a disequality that dominates the objective function. We also obtain such a constraint if we add the second and the third row of the matrix. This two operations generate a *valid constraint* that also *brings information about the objective function*: it gives the obj an upper bound.
Generalizing this reasoning, we can define a linear combination of the constraints, and also linarly combine them with the right hand side vector b, to have actually a whole new set of constraints that is also consistent

$$y_1 * (\text{ first line of A }) + y_2 * (\text{ second line of A }) + y_3 * (\text{ third line of A }) \leq y_1 + 55y_2 + 3y_3 \tag{26}$$

Now we can factorize the above equation in order to isolate the x_i variables. Then, we turn back to imposing that the *coefficient for x_i must be greater or equal to the one of the objective function* in order to dominate it.

$$\begin{pmatrix} 1 & 5 & -1 \\ -1 & 1 & 2 \\ -1 & 3 & 3 \\ 3 & 8 & -5 \end{pmatrix} \times \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} \geq \begin{pmatrix} 4 \\ 1 \\ 5 \\ 3 \end{pmatrix} \tag{27}$$

[19] has a higher value

we can see as the last vector is just the $\underline{c}$ vector of the original problem. Given that we're now looking for an upper bound of z, and also the *lowest* upper bound, the dual problem will be a minimization problem; also, every "constraint coefficient" variable will contribute to the objective function proportionally to their right hand side coefficient of the original problem. We can now formulate the full dual problem:

$$\begin{cases} \min v = y_1 + 55y_2 + 3y_3 \\ \begin{pmatrix} 1 & 5 & -1 \\ -1 & 1 & 2 \\ -1 & 3 & 3 \\ 3 & 8 & -5 \end{pmatrix} \times \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} \geq \begin{pmatrix} 4 \\ 1 \\ 5 \\ 3 \end{pmatrix} \\ y \geq \underline{0} \end{cases} \tag{28}$$

3.6.2 General Form of the Dual Problem

As could be derived from the example, the relation between th **primal** and **dual** problem is

$$(P) = \begin{cases} \max z = \underline{c}^t x \\ Ax \leq \underline{b} \\ x \geq \underline{0} \end{cases} \tag{29}$$

$$(D) = \begin{cases} \min v = \underline{b}^t y \\ A^t y \geq \underline{c} \\ y \geq \underline{0} \end{cases} \tag{30}$$

The duality relation is symmetric: if D is the dual of P, then the dual of D is P.
When dealing with problems in standard form, the non negativity constraint on the vector y is not more needed.
Another formulation for this problem that keeps into account the basic and nonbasic solutions is:

$$(P) = \begin{cases} \min z = c_B x_B + c_N x_N \\ Bx_b + Nx_N = b \\ x_B,\, x_N \geq \underline{0} \end{cases} \tag{31}$$

$$(D) = \begin{cases} \max v = yb \\ yB \leq c_B \\ yN \leq c_N \end{cases} \tag{32}$$

3.6.3 Weak Duality Theorem

Given the classical formulation for a linear programming problem and its dual:

$$(P) = \begin{cases} \max z = \underline{c}^t x \\ Ax \leq \underline{b} \\ x \geq \underline{0} \end{cases} \tag{33}$$

$$(D) = \begin{cases} \min v = \underline{b}^t y \\ A^t y \geq \underline{c} \\ y \geq \underline{0} \end{cases} \tag{34}$$

the *weak duality theorem* states that for every feasible solution $\overline{x} \in X$ of (P) and every feasible solution $\overline{y} \in Y$ of (D) we have

$$\underline{b}^t \overline{y} \leq \underline{c}^t \overline{x} \tag{35}$$

This is easy to visualize if we imagine the two problems (and in particular, the two feasible regions) as opposite but with the same objective: one is trying to reach the optimal solution by increasing the objective function value, the other doing the opposite reducing it. This leads us to the next important property of dual problems:

3.6.4 Strong Duality Theorem

Given primal and dual problem as in (31) and (32), and given that they are both *bounded* and *feasible*, then if x^* is an optimal solution for (P) and y^* is an optimal solution for (D) we have that

$$\underline{c}^t x^* = \underline{b}^t y^* \tag{36}$$

Again, this is just an extension of what we've said for the weak duality theorem: the visualization is also very simple, as shown in the figure

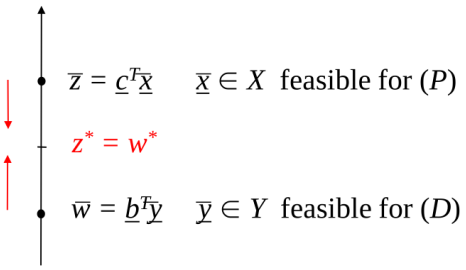


Figure 3: Relation between feasible regions and optimal solutions of dual problems.

in the end, we have 4 possible cases out of 9 possible combination of problems, that are explained in the table below:

	∃ an optimal solution	unbounded problem	infeasible problem
∃ an optimal solution	Granted by the strong duality theorem	Not possible (SDT)	Not possible (SDT)
unbounded problem		not possible (WDT)	consequnce of weak duality theorem
infeasible problem			only possible combination for empty feasible region

where obviously columns and rows represent the possible condition of the primal and dual problem. As said, the relation is symmetric (so only half of the matrix is shown) and SDT and WDT means Strong Duality Theorem and Weak Duality Theorem.

3.6.5 Optimality Conditions

The strong duality theorem gives us a powerful tool to prove the optimality of a solution of the linear problem: if we find x^* optimal solution for the primal problem and y^* optimal solution for the dual, it must hold that $\underline{c}^t x^* = \underline{b}^t y^*$. This can be easily proven by substituting the definition of the vectors inside the equation given:

$$\begin{cases} Ax^* = \underline{b} \\ A^t y^* = \underline{c} \\ \Rightarrow \ y^{*t}\underline{b} = y^{*t}Ax^* = \underline{c}^t x^* \end{cases} \tag{37}$$

3.6.6 Complementary Slackness

Following along the reasoning that gaves us the new Optimality Conditions, we can see a lso a relation between *slack variables* of optimal solution. Let's say $x^* \in X$ and $y^* \in Y$ optimal solutions for the primal and the dual problem respectively. We can notice that

$$\begin{cases} \forall i \,|\, y_i^*(a_i^t x^* - \underline{b}_i) = 0 \\ \forall j \,|\, (\underline{c}_i^t - y^{*t}a_j)x_j^* = 0 \end{cases} \tag{38}$$

Where i and j represents respectively the row index and the column index of the A table. We can see that at *optimality*, the product of each variable with the corresponding slack variable of

the constraint of the relative dual is null. This *again* gives us a way to easily detect optimality in a solution and to find the dual one: it's enough to check the complementary slackness equations once found a solution.

3.7 Sensitivity Analisys

"How much does an optimal solution change wrt the variations of some of its parameters?"
This is sensitivity analysis, so calculate and quantify the effect produced on the optimal objective function value by a variation in the parameter value. This is useful when adjustments must be made, and we have to choose which parameter ensures the maximum or minimum variation overall. Remember: in sensitivity analysis *everything* is a parameter, from the optimal solution values to the right hand side vector of the feasible region definition.

3.7.1 Optimality Intervals

A direct application of sensitivity analysis is to find the interval in which a basis B remains optimal[20]. In this case, we can vary the cost coefficents and the right hand side termns in order to verify the intervl of validity of a base.

Tweaking the $\underline{b}$ Vector We have the optimal solution $\underline{x}^* = \begin{pmatrix} B^{-1}(\underline{b}) \\ \underline{0} \end{pmatrix}$, we modify *slightly* the solutions vector, we obtain:

$$x^* = \begin{pmatrix} B^{-1}(\underline{b} + \delta\underline{e}) \\ \underline{0} \end{pmatrix} \tag{39}$$

where as usual the nullifiedpart of the vector is the one associated to the non basic variables, and e represents a column of the identity matrix. In systesys, we're modifyin just a single entry of the solutions vector, by δ. We just apply the definition of optimality for a basis and have that B *remains optimal as long as*

$$B^{-1}(\underline{b} + \delta\underline{e}) \geq \underline{0} \Rightarrow B^{-1}\underline{b} \geq -\delta B^{-1}\underline{e} \tag{40}$$

Obviously, changing the value of the basic vectors impacts the value of the objective function, which goes from $\{\underline{c}^t B^{-1}\underline{b}\}$ to $\{\underline{c}^t B^{-1}(\underline{b} + \delta\underline{e})\}$.

Tweaking the $\underline{c}$ Vector We increment just as before the cost coefficient vector as $\underline{c}' = \underline{c} + \delta\underline{e}$. The basis remains optimal as long as

$$\underline{c}'^t_N - \underline{c}'^t_B B^{-1} N \geq \underline{0} \tag{41}$$

In this case, we're changing the objective function value, but *differently from the previous case* the x^* vector remain unchanged.

[20] so that $B^{-1}\underline{b}$ remains non negative with x_N set to zero, and th reduced costs of the variables in the basis remain non negative too.

4 Integer Linear Programming

An ILP problem is a Linear Programming problem with the additional constraint $\underline{x} \in \mathbb{Z}^n$. Obiously we could add the integer constraints only to some varibales: in that case, the problem is said *mixed* integer LP. An assumption we make (that does not influence the reasoning or the outcome) is that both A and $\underline{b}$ are also composed of integers values.

4.1 Linear Relaxation and Relation with Linear Programming

If we remove the integer constraint froman ILP problem, weare left with a simple linear problem that shares some properties with the starting integer one:

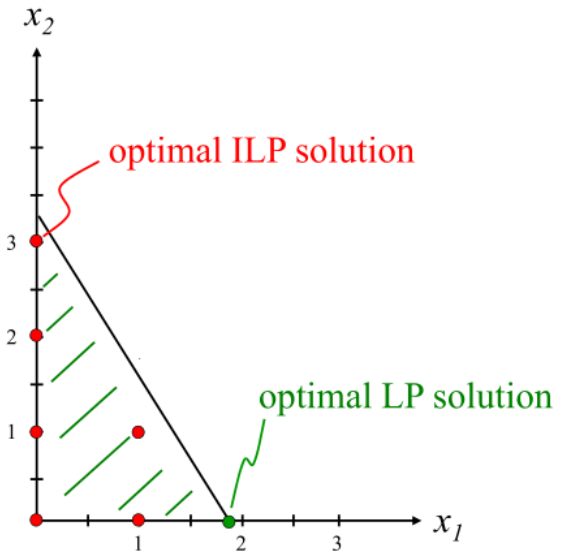
$$ILP: \begin{cases} \max \underline{c}^t x \\ Ax \leq \underline{b} \\ x \geq \underline{0} \\ x \in \mathbb{Z}^n \end{cases} \tag{42}$$

$$LP: \begin{cases} \max \underline{c}^t x \\ Ax \leq \underline{b} \\ x \geq \underline{0} \end{cases} \tag{43}$$

The second one is called *linear continuous relaxation* of the integer problem. For each ILP problem, we have that

- The objective function value of the optimal solution of the integer problem is always *smaller in module* wrt the one of the linear relaxation
- The feasible region of the ILP problem is fully contained inside the one of the continuous relaxation: $X_{ILP} \subseteq X_{LP}$

These two properties are easily explained by the graphical representation of the feasible regions: it's easy to see how the "discreteness" of the ILP problem produces a "loss" of information (as the fractional part of solutions) and thus "reduces" the dimension of the feasible region.



4.2 Solving an ILP Problem

The first naive approach to solve an ILP problem could be to find the solutions of the associated linear relaxation, the round them to the "nearest" integer value that respects the boundaries. This method is not perfect because often the rounded solutions are *infeasible* for the ILP problem or even *useless*. The best application of this approach is when the objective function value is optimal for "large" numbers, so rounding does not affect in a significative way the actual OFV.
Instead of focusing on the linear relaxation, we can directly solve the ILP problem exploiting its peculiarities:

- Implicit enumeration techniques explore all the feasible solutions by "traversing" them. In this category fall the "branch and bound" method but also dynamic programming techniques

- Cutting planes strategy aims to exploit the geometry of an integer problem, tuning the constraints of the feasible region in order to achieve an integer-bound feasible region to isolate easily a solution
- all sorts of heuristics can be used. These are not exact methods thogh.

4.2.1 Branch and Bound Method

The idea behind branch and bound method is the classic "divide et impera" approach in algorithm design. In this particular problem, "divide" becomes "partition the feasible region into subregions" and "impera" is "resolve the problem locally in the subregion".

Branching Given the classic minimization problem $\{\min(c(x)) \,|\, x \in X\}$ we partition the feasible region as $X = \bigcup X_n$ with $X_i \bigcap X_j = \emptyset$, $i \neq j$. We can define $z_i = \min(c(x)) \,|\, x \in X_i$, so to have that the *global* minimum is the minimum among all these *local* z_i minima. In an ILP problem, partitioning is really straightforward: we initially find a candidate solution for the linear relaxation: if such solution $\underline{x}$ is fractional, we divide the problem in the "greater or equal then" and "less or equal then" the integer approximation of the solution $\underline{x}$

Bounding For each subproblem defined by the branching phase, we can do three things:

- prove that the selected subregion of X is empty
- prove that the objective function value won't decrease if we search in this subregion (so skipping the analysis for this particular subregion)
- determine a local optimal solution and update the value of the objective function found so far

this is a recursive method: we can also branch subregions and repeat the reasoning inside these subsubregions and so on.
In an ILP, we just look for the solution of the linear relaxation of each subregion.

B&B Algorithm Summary

1. Find the solution of the linear relaxation of the ILP considered
   - if the solution is integer, return this value
   - if it's fractional, go to step 2
2. x_{LR} the solution at step 1, divide the feasible region in $x \leq \lfloor x_{LR} \rfloor$ and $x \geq \lfloor x_{LR} \rfloor + 1$ and return at step 1

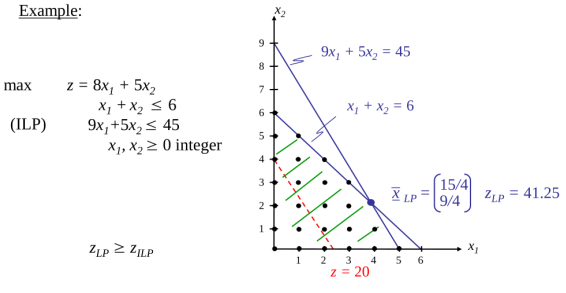


Figure 4: The problem

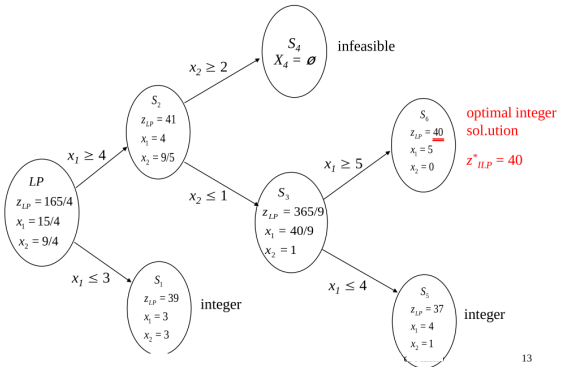


Figure 5: Branch and bound decision tree with local solution values

The choice of "which subnode to evaluate next" can be done following the "deeper path" (so trying to fix the major number of nodes sequentially, the simplest version of the algorithm) or following the most promising nodes (the greedy approach).

4.2.2 Cutting Plane Method and Gomory Fractional Cut

The B&B method is a good method to find integer solutions of an ILP problems, but has the common problem of recursive tree-based algorithms: overall temporal complexity, difficult to evaluate single nodes etc. The cutting planes method aims, as already said, to exploit the peculiarity of the "integer geometry" of the ILP feasible region.
The idea is: the polyhedron described by the feasible region could be "readjusted" in order to costraint *exactly* the integer solutions, and not also the fractional part of the relaxation. It's easily explained by an example: if the "rightmost" constraint of our two-dimensional feasible region is $x \leq 2.15$, we can readjust the constrainf to $x \leq 2$ without losing solutions of the ILP problem. Also, it would be simpler to find an exact optimal solution along this constraint. If we repeat the reasoning for all rows of matrix A, we could "shrink" the feasible region down to have just the integer solutions on the vertices. We define an *ideal formulation* of the feasible region as the one that describes the *convex hull* of X_{ILP}, so the *smallest convex set that encloses* X_{ILP}. REMEMBER: X_{ILP} is a discrete set, while the hull is a continuous polyhedra. This definition is useful when paired when the theorem:
for any feasible region X_{ILP} of an integer linear programming problem, there exists an ideal formulation involving a finite number of linear constraints.
Usually, the number of constraints of the ideal formulation is exponential wrt the original formulation: additional constraints are usually needed to "encapsule" the original discrete set of feasible solutions.

Gomory's Cutting Plane Method The idea is: we don't need to calculate the *entire* convex hull, we just focus on the optimal solution "neighbourhood" and progressively "cut away" fractional parts until we reach an integer optimal solution. We define a Gomory cut as:

$$\sum_{j:x_j \in N} ((a_{rj} - \lfloor a_{rj} \rfloor)x_j) \geq (b_r - \lfloor b_r \rfloor) \tag{44}$$

Where r represents a row in the matrix A, N is the set of non basic variables, a and b are respectively the left hand side and right hand side coefficients of that equation in the system. The Gomory cut is a **cutting plane**: it is violated by the optimal solution of the linear relaxation, but it is satisfied by all the integer solutions of the original problem.

5 Examples

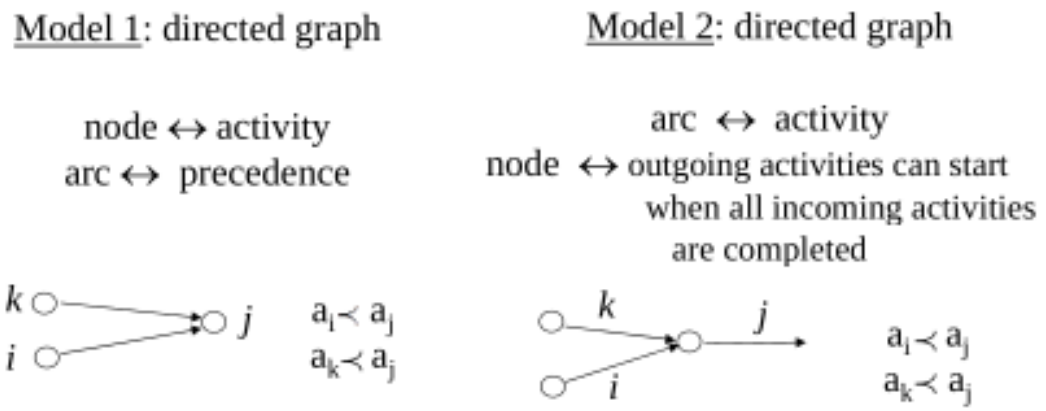
This section collects all the example, following the course contents schema

5.1 Graphs

Cuts and incompatibility relations Problem to model: assigning tasks to the right-qualified engineers. We create a set N of node composed of N_t the tasks and N_e the engineers. We then observe the cuts between the two subsets (obviously, $N_t = N \setminus N_e$) to see if we have enough engineers for the tasks. We will return on this example later.

Directionality and precedence We can use the directionality of graphs to model precedence (among tasks, activities, phases...). Two alternative representations are usually used:

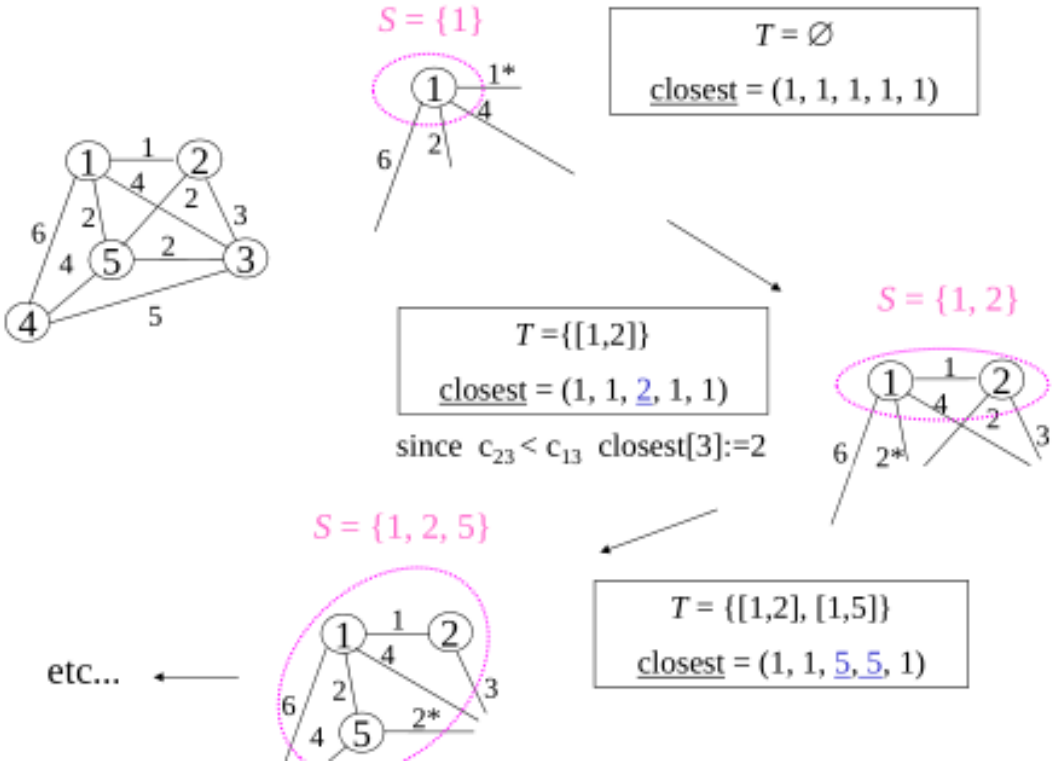
1. Nodes as activities, arcs as precedence relations
2. Arcs as activities, nodes as checkpoints



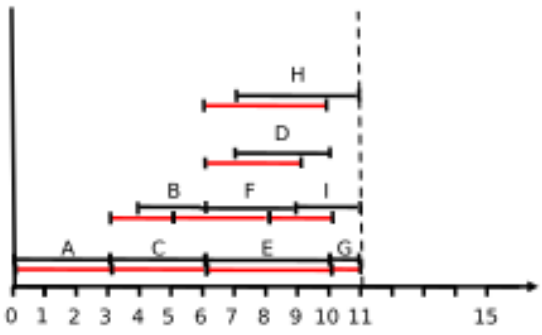
Minimum spanning trees and optimal message passing Wecan model a network of peers as a weighted graph where each node is a peer and each edge has associated (as the weight) the probability of interception of a message along that channel. The problem of finding a broadcasting tree with the minimum interception probability is a minimum spanning tree problem. We call $p_{i\,j}$ the probability of interception between nodes i and j, and the objective function of the problem is simply

$$max \prod_{i,j \in T} (1 - p_{i\,j}) \tag{45}$$

O(n^2) Prim's algorithm formulation This formulation of the algorithm uses as data structure a vector that's long as the number of nodes. Each entry in the vector is numbered as the associated node, so that this vector represents *the predecessors* at the end of the algorithm. At startup, it will be filled with the ID of the starting node. At each iteriation, the vector is updated where a path is found that decreases the cost of a node (if node 3 is reachable through node 2 with a decreasing-cost edge, the enntry in position 3 will be updated to 2).



DAGs and Gantt charts Given the output of the Critical Path Method, we can build a visual chart that represents the actual parallelism and slack of each activity in a more intuitive way. Example:



(i,j)	d_{ij}	Tmin[i]	Tmax[j]
A	3	0	3
B	2	3	6
C	3	3	6
D	3	6	10
E	4	6	10
F	3	5	9
G	1	10	11
H	4	6	11
I	2	8	11

Network flow and matching problems The already discussed task-engineers problem can be seen as a network flow problem, where we're trying to maximize the flow between the first half (the engineers) and the second half of the nodes (the tasks). Note that the graph must be *bipartite*. The idea is to add a fake initial and final node to have a connected graph (the weights of the additional edges must be all equal) and then finde the maximum possible flow through the network: this will highlight the best combination of assignements for the engineers.

5.2 Integer Linear Programming

Knapsack problem We have to determine a subset of the carriable objects that fit into the knapsack and maximize the value.

$$\begin{cases} n \text{ objects} \\ p_j \text{ profit for object } j \\ v_j \text{ volume for object } j \\ b \text{ maximum sack capacity} \end{cases} \tag{46}$$

we define binary variable x_j that indicates if we chose to pick object j. The problem has a very easy to understand objective function and feasible region:

$$\begin{cases} \max(\sum_{j=1}^{n} p_j x_j) \text{ maximize profit} \\ \sum_{j=1}^{n} v_j x_j \leq b \text{ maximum volume constraint} \\ x_j \in \{0,1\} \forall j \text{ decision variable constraint} \end{cases} \tag{47}$$

Transportation problem Classic offer-demand-constraint problem, characterized by:

$$\begin{cases} m \text{ production plants} \\ n \text{ clients} \\ c_{ij} \text{ transportation cost from } i \text{ to } j \\ p_i \text{ production capacity for plant } i \\ d_j \text{ demand of client } j \\ q_{ij} \text{ maximum amount that can be transported on that road} \end{cases} \tag{48}$$

Again a very intuitive problem to solve: we must minimize the total transportation cost satisfying the demands and without exceeding the transportation limit of roads. We use decision variable x_{ij} to describe the amount of product transported from plant to clinet.

$$\begin{cases} \min(\sum_J (\sum_I (c_{ij} \times x_{ij}))) \\ \sum_I (x_{ij}) \geq d_j \\ \sum_J (x_{ij}) \leq p_i \\ 0 \leq x_{ij} \leq q_{ij} \forall (i,j) \end{cases} \tag{49}$$