Foundations of operations research

Niccoló Didoni

September 2021

Contents

Part I

Introduction to operations research

Chapter 1

Operations research

Operations research is the discipline that tries to optimise some complex problems.

Such problems include everyday tasks like finding the best route between to points, finding the best route that crosses a certain number of intermediate points or finding the best route taking into consideration traffic and departure time.

1.1 Models in operations research

Decision making in operations research is made with the aid of both mathematics and computer science, for instance OR uses

- **Mathematical models**.
- **Quantitative models**.
- **Efficient algorithms**.

1.2 Decision making

Decision making in OR is divided into 3 different types

1. The **strategic level** that takes into consideration a longer span of time and a big effort.
2. The **tactical level** that focuses on mid-term decisions.
3. The **operational level** that put emphasis on short-term decisions.

For instance consider the railroad network.

- The design of the railroad is a strategic level decision because a lot of time is needed to designing the railroad.
- The decision on how many trains to use on that railroad is a tactical level decision because it does not require as much time as designing the network but its effect are durable in time.
- The schedule of the trains is an operational level decision because it's effects are temporary.

Part II

Modelling

Chapter 2

Introduction

Modelling is the process to reduce a real world problem to a mathematical level. Models depend on the use, in fact there's no one fits all model. It is also important to understand that modelling and solving a problem are two different and separated tasks (that can be in relation though).

A model is made of three main ingredient

1. **Decisions** that are translated in **variables**.
2. **Rules** that translated in **constraints**.
3. **Objective** that is translated in a **function**.

Chapter 3

Variables

3.1 Non negative variables

Definition 1 (Non negative variable). *A non negative variable (i.e. non negative absolute quantity) is a variable that only allows positive (or null) values (in other words no negative value is allowed).*

$$x \geq 0$$

Non negative variables can be used when modelling something that we possess. For example we cannot posses a negative value of some object (either we have none, one or more objects).

3.2 Unrestricted variables

Definition 2 (Unrestricted variable). *An unrestricted variable (i.e. unrestricted absolute quantity) allows both positive and negative values.*

Unrestricted variables (i.e. unrestricted absolute quantities) allow both positive and negative values.

Unrestricted variables can be used to model perturbations like delays (if a train could be running late the variable would have a negative value, if it is running early the variable would have a positive value) or deltas.

3.2.1 Relation with non negative variables

An unrestricted variable con always be converted in a non negative value. Let π be an unrestricted variable. We can always write π as the difference of two positive values π^+ and π^-. If π^+ is greater that π^- than π will be positive, otherwise it'll be negative

$$\pi = \pi^+ - \pi^- |\; \pi^+, \pi^- \geq 0$$

3.3 Relative variables

Definition 3 (Relative variable). *Relative variables are variables that can only take a value in between 0 and 1 (both included)*

$$0 \leq x \leq 1$$

3.3.1 Capital investment problem

As an example consider the following problem. We can invest in 5 different types of investments ($e_i: \ i = 1, ..., 5$) and for each investment we can only invest a maximum amount of money a_i. Each investment returns an amount of money computed as follows

$$r_i = c_i \cdot \frac{x_i}{a_i}$$

where c_i is the coupon for the investment i and x_i is the total amount of money invested in the investment i. Given a budget we want to maximise the financial return.

Model

Before solving this problem we have to model it. Let x_i be the amount of money invested in the bond (investment) i then

1. The maximum amount invested should not be greater than the budget B
$$\sum_i x_i \leq B$$
2. We can only invest a positive amount of money (in other words x is a non negative variable)
$$x_i \geq 0 \quad \forall i = 1, ..., 5$$
3. For each investment the amount invested cannot be greater then the maximum amount a_i for that investment
$$x_i \leq a_i \quad \forall i = 1, ..., 5$$
4. We have to maximise the return of the investment
$$\max \left(\prod_i c_i \cdot \frac{x_i}{a_i} \right)$$

To simplify the model we can introduce a relative variable y_i that computes the relative quantity of the investment i

$$y_i = \frac{x_i}{a_i}$$

Because of the third constraint y_i cannot be greater than 1, and because of the second constraint it cannot be smaller than 0 either.

We can now use this variable to redefine the fourth constraint as follows

$$\max \left(\prod_i c_i \cdot y_i \right)$$

Reversing the definition of y_i we can also rewrite the first constraint to obtain

$$\sum_i y_i a_i \leq B$$

The simplified model is therefore

$$\sum_i y_i a_i \leq B \tag{3.1}$$

$$\max\left(\prod_i c_i \cdot y_i\right) \tag{3.2}$$

$$0 \leq y_i \leq 1 \quad \forall i = 1, ..., 5 \tag{3.3}$$

Solution

To solve this problem we can compute the return of a unitary investment (i.e. how much money we get in return when we invest $x_i = 1$)

$$\frac{c_i}{a_i}$$

Now we have to invest the maximum amount allowed in the investment with the maximum unitary return (i.e. the most convenient bond) and repeat the process with the next most convenient investment until we have no more money.

We can write this procedure in an algorithmic way as

```
sort indices so that if i < j then ci/ai >= cj/aj
i ← 1;
repeat
    yi ← min {1, b/ai};
    b ← b-yi ai;
    i←i+1;
until b=0 or i>n
```

3.4 Logical variables

Definition 4 (Logical variable)**.** *A logical variable is a variable that only allows the values 0 and 1.*

Logical variables are useful when we have to model as problem in which we can either take an item or not (we cannot chose to take an arbitrary amount of an item).

3.4.1 Burglar problem

Consider a problem similar to the capital investment problem in which a burglar breaks into a jewellery and has to steal some jewels. Every jewel has a certain weight and the backpack of the burglar has a limited weight capacity. This problem is a little bit different from the other one because the burglar has either to take or leave a certain jewel (the investment in the other example). In this example the previous solution could work, but it is not assured.

```
sort indices so that if i < j then ci/ai >= cj/aj
i ← 1;
repeat
    b ← b-yi ai;
    i←i+1;
until b=0 or i>n
```

Difficulty of the problem

The burglar problem has a finite number of possible combinations (x_i can only be 1 or 0, so the list of all possible combination of jewels stolen is the list of all the strings of 1s and 0s whose length is the same as the number of jewels in the jewellery). On the other hand the investment problem has infinite solutions because we can take any quantity between 0 and the max quantity allowed for every investment. That said, the second problem is much easier to solve in an algorithmic way than the first (even if the space of feasible solutions is much larger in the first scenario).

3.5 Discrete value variables

3.5.1 Integer variables

Definition 5 (Integer variable)**.** *Integer variables are variables that can only assume integer values*

$$x \in \mathbf{Z}_*$$

3.5.2 Finite set variables

Definition 6 (Finite set variables)**.** *Finite set variables are variables that can assume a value within a finite set of possible values*

$$x \in \{v_1, ..., v_n\}$$

This type of variable can be expressed using flag variables. A flag variable is a logic variable y_i that has value 1 if the finite set variable x equals to the i-th value, 0 otherwise

$$y_i = \begin{cases} 1 \text{ if } x = v_i \\ 0 \text{ otherwise} \end{cases}$$

We can use the flag variables to express x as

$$x = \sum_{i=1}^{n} v_i y_i$$

This definition isn't enough, in fact we also have to specify that only one of the variables y_i has to be 1. The represent this property we can say that the sum of all the variables y_i has to be 1

$$\sum_{i=1}^{n} y_i = 1$$

The set variable x can also be represented, in an alternative way, removing the constraint that forces only one y_i variable to be 1. In this case if $x = v_k$ then all the variables from y_1 to y_k have to be 1 while the other have to be 0.

$$y = [1, 1, 1, 1, 0, 0, 0, \dots, 0]$$

In this situation the variable x can be written as

$$x = v_1 y_1 + (v_2 - v_1) y_2 + \cdots + (v_n - v_{n-1} y_n) = v_1 y_1 + \sum_{i=2}^{n} (v_n - v_{n-1}) y_n$$

In other words the variable x is the sum of v_1 and the sum of the differences between one variable and the next one.

Chapter 4

Constraints

4.1 Availability constraints

We write an availability constraint when we have a limited number of resources. An availability constraint has

- The number of resource we cannot exceed on the right side.
- How much of that resource we consume on the left hand side.

The two sides are separated by a $\leq$.

$$x_A + 3y_B \leq 22$$

4.2 Requirement constraints

The requirement constraint is the reverse of the availability constraint, in fact it is used when we want to say that we have to use at least some amount of a resource. This constraint has the same structure of the availability constraint but the two sides are separated by a $\geq$ symbol.

$$x_A + 3y_B \geq 22$$

4.2.1 Transforming constraints

We can always transform an availability constraint in a requirement constraint, and vice versa, by multiplying both sides by -1

$$Ax \leq b \longleftrightarrow -Ax \geq -b$$

While transforming a $\leq$ in a $\geq$ is quite easy, it becomes trickier when we have to transform an inequality in a equality. To achieve this transformation we have to introduce a non negative slack variable s. The slack variable represent the amount we have to sum to Ax to reach the value b. For instance consider the constraint $Ax \leq b$. The value of Ax cannot be greater of b, in other words we should add some value to Ax to reach b. This value is the slack variable.

$$Ax \leq b \longrightarrow Ax + s = b$$

$$Ax \geq b \longrightarrow Ax - s = b$$

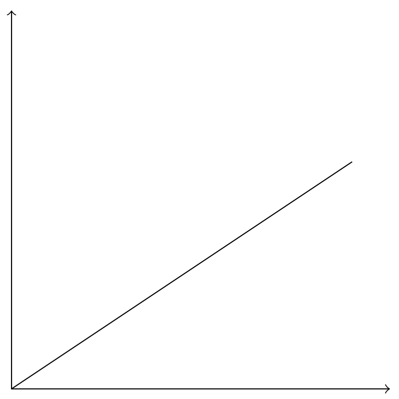


Figure 4.1: A graphical representation of the transformation between an equation and an inequation.

To do the opposite transformation (from an equality to a inequality) we have to look at the geometric meaning. The equation $ax = b$ represents a line that can be seen as the intersection of the planes planes $ax \geq b$ and $ax \leq b$.

$$Ax = b \longrightarrow \begin{cases} Ax \geq b \\ Ax \leq b \end{cases}$$

4.3 Blending constraints

The blending constraint allows to describe requirements on the percentage of presence of certain components.

For instance in a diet we could ask that at least 30% of the calories eaten during the day comes from fruits and vegetables. This type of constrain can be expressed using the ratio between the calories given by fruits and vegetables and the total amount of calories eaten

$$\frac{C_{fruits} \cdot x_{fruits} + C_{veg} \cdot x_{veg}}{\sum_{f \in Food} C_f \cdot x_f} \geq 0.3 \tag{4.1}$$

where x_f is the amount of food f consumed and C_f are the calories given by a kilogram of the food f.

The equation 4.1 isn't linear because it is expressed as the ratio of two variables but we can make it linear by multiplying both sides by $\sum_{f \in Food} C_f \cdot x_f$

$$C_{fruits} \cdot x_{fruits} + C_{veg} \cdot x_{veg} \geq 0.3 \cdot \left(\sum_{f \in Food} C_f \cdot x_f \right) \tag{4.2}$$

4.4 Logical constraints

Logical constraints allow to express logical preposition using logical variables. Consider two logical variables x and y

- The logical union $x \vee y$ can be expressed as
$$x + y \geq 1$$
- The logical intersection $x \wedge y$ can be expressed as
$$x \geq 1, y \geq 1$$
- The logical implication $x \Rightarrow y$ can be expressed as
$$x \leq y$$
- The logical double implication $x \Leftrightarrow$ can be expressed as
$$x = y$$

4.4.1 Logical constraints extended to real variables

Logical constraints can also be applied to real variables. Consider for instance two real variables x_p and x_r of which only one can be different from 0 (the variables represent the quantity of pasta and rice one wants to eat in a canteen). This constraint can be expressed, in words, as

$$\text{if } x_p \neq 0 \text{ then } x_r = 0$$

To write in a formal way this constraint we could multiply the variables and ask the result to be 0 (if at least one of the two has to be 0, then we are multiplying a number for 0, which always returns 0)

$$x_p \cdot x_r = 0$$

This constraint is correct but it is not linear.

To solve this problem we can introduce two flag variables

$$y_p = \begin{cases} 1 \text{ if } x_p > 0 \\ 0 \text{ otherwise} \end{cases}$$

and

$$y_r = \begin{cases} 1 \text{ if } x_r > 0 \\ 0 \text{ otherwise} \end{cases}$$

These variables can be used to express the constraint by saying that only one of the two has to be one

$$y_p + y_r \leq 1$$

Now the problem is that the variables y_p and y_r aren't linked to the variables x_p and x_r. To bind these variables we can use linking constraints that state that the quantity x_p doesn't have to be greater than the maximum allowed quantity M of x_p (for instance M could be the maximum quantity of food a plate can contain) multiplied by the relative flag variable y_p.

$$x_p \leq My_p$$

The same constraint is applied to x_r.

$$x_r \leq My_r$$

In this way if $y_p = 0$ then x_p has to be 0 (if $y_p = 0$ then $My_p = 0$ and the only value of x_p that satisfies the constraint is 0), and x_r can have any value from 0 to M.

4.5 Flow conservation constraints

To introduce this type of constraint we can use an example. Consider an oil field in which oil can flow in some direction between the components of the field (for example from a well to a refinery or from an well to another). For this model we can introduce a variable x_{ij} that represents the flow on the pipe from the component i to the component j.

If a well 1 can only extract M_1 litres of oil and is connected to component 2 and 3 we can write

$$x_{12} + x_{13} \leq M_1$$

Consider now well 2 that can extract at most M_2 litres of oil, receives oil from well 1 and transfers oil to components 3 and 4. In this case the constraint is

$$x_{23} + x_{24} - x_{12} \leq M_2$$

Chapter 5

Objective functions

5.1 Minimum to maximum transformations

Consider an objective function that has to minimise a certain function $f(x_{ij}) = \sum_{(i,j)\in P} c_{ij}x_{ij}$

$$min f(x_{ij}) = min \sum_{(i,j)\in P} c_{ij}x_{ij}$$

To transform the objective function in a maximum we can simply multiply by -1 the function $f(x_{ij}$

$$min f(x_{ij}) \longrightarrow max - f(x_{ij}) = max - \sum_{(i,j)\in P} c_{ij}x_{ij}$$

The same can be done the transform a maximum in a minimum.

5.2 No objective function

In some cases we simply have a system of inequalities

$$A_1x \leq b_1 A_2x \leq b_2 A_3x \leq b_3 \tag{5.1}$$

but no objective function $mincx$. To solve this problem we can use one of the inequalities and use it as an objective function. For instance we could use the first equality objective function

$$minA_1x$$

When we obtain the solution x^* we have to check if the solution satisfies the constraint

$$A_1x^* \leq b_1$$

If the constraint is satisfied then x^* is the solution, otherwise there's no solution to the system.

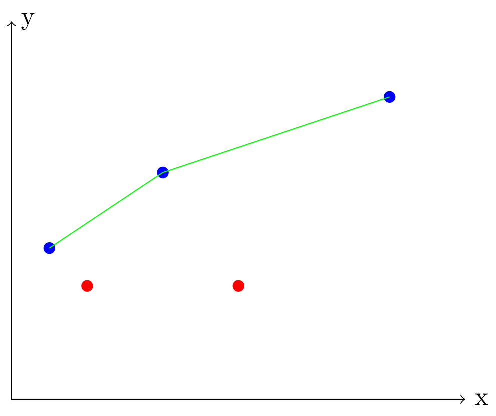


Figure 5.1: An example of multi-criteria analysis.

5.3 Multi-criteria analysis

Some problems require to maximise a certain function (y) and at the same time minimise another function (x). In other words these problems have to be modelled with two objective functions. In these cases we can find a solution to the system using a Cartesian's plane in which we put on the axes the variables of the two objective functions.

The plane is populated with all the possible solutions that satisfy the constraints of the system. These solutions aren't the optimal solutions but are just a set of acceptable values. For each solution we have to check if it is dominated by another solution. A solution $S_1 = (x_1, y_1)$ dominates another solution $S_2 = (x_2, y_2)$ if

- S_1 has the same x as S_2 but a bigger y or,
- S_1 has the same y as S_2 but a smaller y.

All the non dominated solutions are called **Pareto optimal solutions** and are the solutions we have to choose among. The line that connects all the optimal solutions is called **efficiency frontier**.

Part III

Graphs

Chapter 6

Introduction to graphs

Graphs are a mathematical language that can be used to model a problem in the same way we did with variables and constraints. In other words graph are abstract representation (i.e. a model) of a real situation.

Definition 7 (Graph)**.** *A graph is a mathematical entity made of* ***nodes*** *that are connected by* ***arcs****. Each arc can also have an attribute called* ***weight****.*

Formal definition Formally a graph G is a couple (N, A) where

- N is the set of nodes of the graph G.
- A is the set of arcs of the graph G. An arc is a pair of nodes (i.e. the nodes connected by the arc).

For instance the graph 6.1 can be represented as

$$G = (\{A, B, C, D\}, \{\{A, B\}, \{A, C\}, \{B, C\}, \{C, D\}\})$$

Directed and undirected graphs A graph can be

- **Directed** if the arcs have a direction. In this case an arc from node A to node B is written as

$$(A, B)$$

  and it's represented using an arrow (graph 6.3).

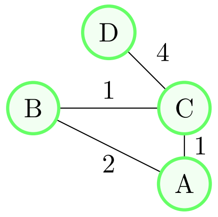


Figure 6.1: A graph with weighted arcs.

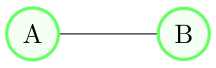


Figure 6.2: An undirected graph containing just two nodes.

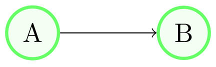


Figure 6.3: A directed graph containing just two nodes.

- **Undirected** if the arcs don't have a direction. In this case an arc from node A to node B is written as
$$\{A, B\}$$
and it's represented using a line (graph 6.2).

6.1 Graph classification

6.1.1 Eulerian graph

Before explaining what an Eulerian graph is we have to define the degree of a node.

Definition 8 (Degree of a node)**.** *The degree of a node N is the number of arcs incident to N (an arc a is incident to a node N if one of the two nodes the arc connects is N).*

For directed graph we can define a indegree that only counts the number of incoming arcs, and an outdegree that only counts the number of outgoing arcs.

Thanks to the definition of degree we can define an Eulerian graph as follows

Definition 9 (Eulerian graph)**.** *A graph G is an Eulerian graph if and only if the degree of each node is even.*

6.1.2 Paths

Definition 10 (Path)**.** *A path is a sequence of consecutive arcs (i.e. the head of an arc a_1 coincides with the tail of another arc a_2).*

This definition of path includes path with repeated arcs.

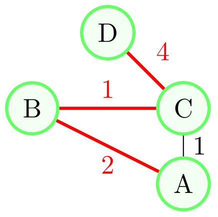


Figure 6.4: An elementary path (coloured in red) in a graph.

Simple path A simple path is a path with no repeated arcs.

Elementary path An elementary path is a simple path without repeated nodes. To be more precise, an elementary path has no repeated nodes and no repeated arcs.

6.1.3 Cycles

Definition 11 (Cycle)**.** *A cycle is a path that starts and ends in the same node.*

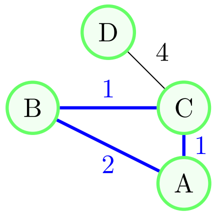


Figure 6.5: A graph with a cycle (coloured in blue).

Elementary cycle An elementary cycle is a cycle with no repeated node.

Hamiltonian cycle An Hamiltonian cycle is a cycle that touches every node of the graph.

Simple cycle A simple cycle is a cycle without repeated arcs.

Eulerian tour An Eulerian tour is a cycle in which all the arcs of the graph are visited only once. The Eulerian tour is based on the problem of the Seven Bridges of Königsberg.

6.1.4 Subsets

Definition 12 (Cut)**.** *A graph's cut is defined by a partition of the set of nodes N into two sub-graphs N_1 and N_2 so that the two sub-graphs contain all the nodes of the graph*

$$N = N_1 \cup N_2$$

and have no node in common

$$N_1 \cap N_2 = \emptyset$$

The arcs of the cut (N_1,N_2) are defined as

- The arcs that go from one node of N_1 to a node of N_2 (arcs in accordance with the cut).
-
- The arcs that go from one node of N_2 to a node of N_1 (arcs not in accordance with the cut).

Removing the arcs of the cut, an initially connected graph is partitioned into at least two connected components.

Definition 13 (Forward star)**.** *The forward star $FS(s)$ of a node s in the graph $G = (N, A)$ is the set of nodes that have tail in s.*

$$FS(s) = (s, i) : i \in A$$

Definition 14 (Backward star)**.** *The backward star $BS(s)$ of a node s in the graph $G = (N, A)$ is the set of nodes that have head in s.*

$$BS(s) = (i, s) : i \in A$$

The star of a node s is the union of its forward and backward star.

$$S(s) = BS(s) \cup FS(s)$$

6.1.5 Properties

Definition 15 (Bipartite graph)**.** *A graph $G = (N, A)$ is bipartite if the set of nodes N can be partitioned into two subsets N_1 and N_2 such that for each arc (i, j) of A*

- *either $i \in N_1$ and $j \in N_2$,*
- *or $i \in N_2$ and $j \in N_1$.*

Chapter 7

Problems

7.1 Minimum spanning tree

Consider an undirected weighted graph

$$G = (N, A)$$

in which each arc from node i to node j has weight

$$w_{ij} \geq 0 \quad \forall (i, j) \in A$$

The goal is to find a subset of arcs $A' \subset A$ so that for every pair of nodes there exists one path (i.e. every pair of nodes has to be connected by one path). The subset of arcs have to be the subset with the minimum value

$$G' = (N, A') : A' \subseteq A$$

$$\min \sum_{(i,j) \in A'} w_{i,j}$$

The graph

$$G' = (N, A')$$

is called **minimum spanning tree** (MST) because it

- Is the graph with the minimum cost.
- It has no cycles so it's a tree. If there is a cycle, one of the arcs of the cycle can be removed because it's an useless cost. For instance consider three nodes $A \to B$, $B \to C$ and $C \to A$ and consider the last connection the most expensive one. In this conditions the last connection can be removed because A is connect to C via B.
- It connects all the nodes of G.

Properties The most important properties of a minimum spanning tree are

- It has no cycles.
- The number of arcs is equal to the number of nodes minus 1.
- G' is connected (every node of N is connected).

- If we add a new arc to G' we get a cycle.
- If we remove an arc from G' we obtain two isolated subsets of nodes. This happens because there are no cycles, so there is only one path from a node to another and if such path is interrupted the nodes aren't connected anymore.
- If we add an arc to the solution the cost of the arc added is greater of the cost of any other arc in the cycle formed by the new arc. In other words if we add a new arc and remove a different arc from the cycle created by the new arc, we get a non-optimal solution.

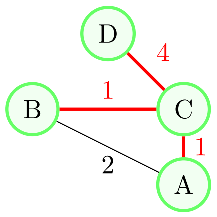


Figure 7.1: The minimum spanning tree (in red).

7.1.1 Model

This problem can be modelled using variables, constraints and the objective function.

Sets The set used in this problems are

- The set of nodes N.
- The set of arcs A.

Parameters The parameters used in this problem are the weight of the arcs

$$w_{ij} \geq 0 \quad \forall (i,j) \in A$$

Variables To model the fact that A' is a subset of A (i.e. that we only take some arcs to build the MST) we need a boolean variable x_{ij} to decide if we have to take a certain arc or not

$$x_{ij} = \begin{cases} 1 & (i,j) \in A' \\ 0 & \text{otherwise} \end{cases}$$

Objective function The objective function has to minimise the weights of the arcs.

$$\min \sum w_{ij} x_{ij}$$

Constraints Expressing the constraints for this problem is a little bit trickier. To help us in this process we can express the constraint in natural language. The constraints have to say that *between any two nodes there exist a path.* This formulation is tricky to convert in a formal way so we have to find an equivalent one.

In particular we can say that *at least one arc has to cross the border (also known as cut) of any two subset of arcs.* In other words if we take any two subsets of nodes have to exist an arc that exits from one subset and reaches the other subset.

This expression can be translated in a formal way

$$\sum_{(i,j)\in A, i\in S\wedge j\in N\setminus S} x_{ij} \geq 1 \qquad \forall S \subset N \wedge S \neq \emptyset$$

7.1.2 Algorithms

Kruskal's algorithm

The following algorithm was invented by Joseph Kruskal in 1959 to find the minimum spanning tree in a graph.

1. Sort all arcs in increasing order of weight w_{ij}.
2. Initialise the set of arcs A' as an empty set $A' = \emptyset$.
3. repeat until $|A'| = |N| - 1$
   (a) consider the next arc $\{i, j\}$ in the order.
   (b) if $A' \cup i, j$ doesn't include a cycle then add i, j to A'

Prim's algorithm

The most expensive part of this algorithm is checking if a graph contains a cycle every time a new arc has to be added. To solve this problem, Robert Prim has proposed an algorithm that doesn't have to look for cycles. In particular the algorithm starts from a node N_1 and adds to A' the arc a_1 with the smallest weight. Call N_2 the node connected to a_1, the nodes N_1 and N_2 are condensed in a single node M that has as boundaries the boundaries of both nodes. The same procedure is repeated on the node M until all nodes are added to M.

Boruvska's algorithm

The algorithm proposed by Otakar Boruvska to solve the MST problem is similar to Prim's but it considers all nodes at a time, in fact for each node we select the best arc (without merging the node). This process is iterated until all nodes are connected.

7.2 Minimum Hamiltonian cycle

A problem similar to the minimum spanning tree is quite relevant in telecommunications. The idea is to build a reliable network, that is a graph G' in which between any two nodes there are at leas two disjoint (i.e. without shared arcs) paths with the minimum cost in terms of total weight of the arcs.

The graph G' is called minimum Hamiltonian cycle (MHC) because the best solution to this problem is to create a cycle that touches every node of the graph (i.e. a ring).

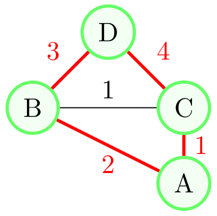


Figure 7.2: The minimum Hamiltonian cycle (in red).

7.2.1 Model

This problem can be modelled like the MSP problem.

Sets The set used are

- The set of nodes N.
- The set of arcs A.

Parameters The parameters used in this problem are the weight of the arcs

$$w_{ij} \geq 0 \quad \forall (i,j) \in A$$

Variables As for the MST problem we can use a boolean variable x_{ij} to decide if we have to take a certain arc or not

$$x_{ij} = \begin{cases} 1 & (i,j) \in A' \\ 0 & \text{otherwise} \end{cases}$$

Objective function The objective function has to minimise the weights of the arcs.

$$\min \sum w_{ij} x_{ij}$$

Constraints The main differences with the minimum spanning tree problem are the constraints, in fact not only all the nodes have to be connected

$$\sum_{(i,j) \in A, i \in S \wedge j \in N \setminus S} x_{ij} \geq 1 \qquad \forall S \subset N \wedge S \neq \emptyset$$

, but they also have to form a cycle. To express such property we can say that for each node the number of selected arcs is 2.

$$\sum_{j:(i,j) \in A} x_{i,j} = 2$$

Without the constraints used for the MST we will allow cyclic subsets of nodes but not connected between them. In other words we would have had two cyclic subsets S_1 and $S_2 = N \setminus S_1$ without arcs form S_1 to S_2.

7.3 Connection problems

Connection problems try to answer two different questions. In particular we may have to

- Find if there exist a path between two nodes s and t.
- Find if it doesn't exist a path between two nodes s and t.

Such problems have different solutions and complexities.

7.3.1 Path existence

Longest path For a graph $G = (N, A)$ with n nodes (i.e. $|N| = n$), the longest path from a node s to a node t is $n-1$ arcs long (to demonstrate it consider the nodes connected to to another without loops). To put it in a complexity way, the path has a size of

$$\mathcal{O}(n)$$

7.3.2 Path nonexistence

To demonstrate that doesn't exist a path between two nodes s and t, we have to use a different approach than the one used for demonstrating path existence.

Algorithm

Procedure To demonstrate that it doesn't exist a path between two nodes s and t we have to

1. Mark the starting node s as visited and visit all the nodes in the forward star of s.
2. Every time we visit a node
   - If the node is marked as visited we do nothing.
   - If the node is not marked as visited we mark it and visit all the nodes in the forward star.
3. We stop when we visited all the nodes. If we haven't visited the node t than it doesn't exist a path between s and t.

Algorithm This algorithm can be executed in polynomial time. To verify such statement we can write the procedure in a more algorithmic way

```
graph_search(G, s, P)
    for i in N do P[i] <- 0;        // initiate P with n zeros
    P[s] <- s;                      // add s as the predecessor of itself
    Q <- {s};                       // initialise the set of nodes to visit with s

    repeat
        select i from Q;
        Q <- Q - {i};                // remove i from Q

        foreach (i, j) in FS(i) do
            if P[j] = 0 then
                P[j] <- i;
                Q <- Q + {j};        // add j to Q nodes yet to visit

    until Q = {}
```

Lets explain the algorithm. The input parameters are

- The graph $G = (N, A)$.
- The starting node s.
- A list of predecessors P. Every cell of the list represents a node. In particular the element in $P[i]$ is the predecessor node of i. If $P[i] = 0$ we have never visited i. We can use this list to determine which nodes have been visited starting from s (i.e. the nodes such that $P[i] \neq 0$).

Here explained some important phases of the algorithm

- At line 2 and 3 we initialise the P list with all zeros and we set the predecessor of s as itself. This way we say that for now we have only visited s.
- At line 8 the node i is removed from the list of nodes to visit because we are visiting it.
- At lines 11, 12 and 13, if the predecessor of node j is 0 it means that we have never visited node j, so we add i as predecessor of j (we mark j) and we add j to the nodes yet to visit.

Complexity To analyse the time complexity of this algorithm we can focus on the single operations. In particular

- The for loop at line 2 has linear complexity $\mathcal{O}(n)$ because it iterates on all the nodes.
- The foreach loop at lines 10 to 14 has linear time complexity $\mathcal{O}(n)$ because a node can have at least $n - 1$ successors. Such loop is executed n times, in fact the loop that starts at line 6 is repeated for each node in G (in worst case scenario).

Summing all up we obtain that the loop dominates the for loop's complexity (it's a foreach loop inside a loop against a single loop) so the time complexity of such algorithm is quadratic (hence polynomial)

$$\mathcal{O}(n^2)$$

Such complexity can be reduce to the total number of arcs m, in fact the algorithm just needs to verify at most all the arcs. In such case the complexity would be linear in the number of arcs m

$$\mathcal{O}(m)$$

An important property

Given a directed graph $G = (N, A)$ and a couple of nodes $s, t \in N$

- either there is a path p that connects the two nodes.
- or there is a cut

  $$(N_s, N_t), s \in N_s, t \in N_t$$

  such that each arc of the cutset has head in N_t and tail in N_s

  $$(i, j) : i \in N_t, j \in N_s$$

  In other words it is possible only to travel from a node in N_t to a node in N_s but not vice versa.

7.4 Shortest path problem

Given a directed graph $G = (N, A)$ with costs c_{ij} and two nodes s, t, the shortest path problem aims to find the elementary path p from s to t minimising its cost

$$\sum_{(i,j)\in p} c_{ij}$$

7.4.1 Model

Sets The sets used in this problem are

- The set of nodes N.
- And the set of arcs A.

Parameters The parameters used in this problems are

- The cost c_{ij} of each arc $(i, j) \in A$.
- The starting and ending nodes s and t.

Variables To specify if a certain arc (i, j) belongs to the path p we can use Boolean variables x_{ij}. In particular the variable x_{ij} is

- 1 if the arc (i, j) belongs to the path.
- 0 otherwise

$$x_{ij} = \begin{cases} 1 & \text{if the arc (i, j) belongs to the path} \\ 0 & \text{otherwise} \end{cases}$$

Objective function The objective function has to minimise the cost of the path

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

Constraints The constraints are the most challenging part of the model. In fact firstly we have to specify that s is the starting node. To specify such property we can say that we path is formed by one and only one arc that has tail in s (i.e. that exists from s). To say that we can write that the sum of all x_{ij} related to the arcs in the forward star of s have to be 1 (so that only one variable is 1)

$$\sum_{(s,j)\in FS(s)} x_{sj} = 1$$

We also have to specify that t is the final node. In this case we can use an approach similar to the one used for s. The only difference is that we use the backward star because we want to say that one and only one arc with head in t has to belong to the path

$$\sum_{(j,t)\in BS(t)} x_{jt} = 1$$

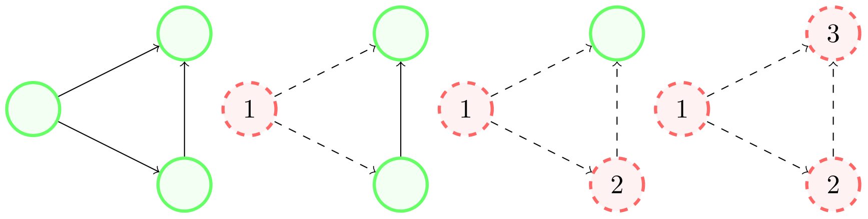


Figure 7.3: Steps to verify if a graph is cyclic and topological ordering

Finally we also have to say that all the nodes between s and t have to be connected one to the other. In other words we have to specify that a directed path exists between s and t. We can use the variables x to specify such constraint, in fact for each node (but s and t) of the path we have to say that if an arrow enters in a node i (i.e. $x_{ai} = 1$) than an arrow has to exit from node i (i.e. $x_{ib} = 1$). In other words the sum of all entering and exiting arrows have to be 0 (considering the entering arrows with a negative sign)

$$\sum_{(j,i)\in BS(i)} x_{ji} - \sum_{(i,j)\in FS(i)} x_{ij} = 0 \quad \forall i \in N \setminus \{s,t\}$$

7.4.2 Acyclicity check

Some algorithms to find the shortest path require the graph to be acyclic thus it's useful to describe a method to find out if a graph is cyclic.

The algorithm to check if a graph G is cyclic works as follows

1. Initialise a counter $i = 1$.
2. Look for the start node of the graph, i.e. the node that has no incoming arcs.
   (a) If we can't find the start and the graph is not empty, then the graph is cyclic.
   (b) If the graph is empty the graph is acyclic.
   (c) Otherwise continue.
3. Remove the start node and mark it using the counter i.
4. Increment i.
5. Go back to instruction 2.

Topological order The indices are used to sort the graph in a topological order. This order is very important and will be used in minimum path search algorithms.

7.4.3 Shortest-path tree algorithm

To find the shortest path of an acyclic graph G we can use the shortest-path tree algorithm. This algorithm assigns to every node a label d that tells the minimum length of the path to reach the starting node. In particular

- The label of the starting node is 0.

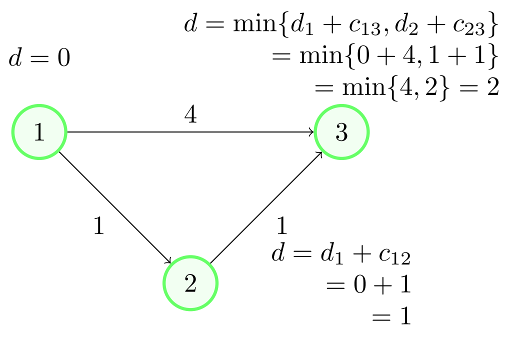

$d = 0$
$d = \min\{d_1 + c_{13}, d_2 + c_{23}\} = \min\{0 + 4, 1 + 1\} = \min\{4, 2\} = 2$
4
1
1
$d = d_1 + c_{12} = 0 + 1 = 1$
1
2
3

Figure 7.4: Labels for the MST algorithm applied to an acyclic graph.

- A node m is labelled with the minimum value in the set

$$\{d_i + c_{im} : (i, m) \in BS(m)\}$$

The nodes are visited in topological order to assign the labels.

```
1   SPT_acyclic(G, P, d)
2       for i <- 2, ..., n do
3           P[i] <- 0
4           d[i] <- infinity
5
6       P[1] = 1
7       d[1] = 0
8
9       for i <- i, ..., n-1 do
10          foreach (i, j) in FS(i) do
11              if d[i] + c[i][j] < d[j] then
12                  P[j] <- i
13                  d[j] <- d[i] + c[i][j]
```

Where

- `G` is the graph.
- `P` is the list where all the predecessors on the shortest-path tree are written. In other words the predecessor of node i in the spanning tree can be found accessing `P[i]`.
- `d` is the list of labels. The label of node i is stored in `d[i]`.

The shortest path from one node i to another node f can be found following iteratively the predecessors in the `P` array starting from `P[f]`.

Complexity Lets consider the complexities of the various segments of the shortest path algorithm

- The first for loop (lines 2-4) is repeated n times so has linear time complexity with respect to the number of nodes.

$$\mathcal{O}(n)$$

- The second loop might look like having quadratic complexity with the number of states but if we analyse it better we can notice that the two nested loops visit every arc of the graph one, thus such piece of code has linear complexity with respect to the number of arcs m

$$\mathcal{O}(m)$$

The total complexity of the algorithm is

$$\mathcal{O}(n+m)$$

Such complexity becomes

$$\mathcal{O}(m)$$

if we consider that usually the number of arcs is much higher than the number of states.

Advantages The advantages of the shortest path algorithm are

- Each node is visited only once.
- Each arc is visited only once, in fact for every node we visit all the nodes in the forward start, but if all nodes are visited once then the outgoing arcs are visited once. For this reason the algorithm is linear.
- After visiting the forward star of a node, the labels of that node are fixed. This happens because the graph is acyclic.

Maximum approach

Until now we considered the shortest path problem as a minimum problem. Actually it is possible to model the same problem so that the objective function's goal is to maximise a certain value.

Level Before introducing the new model we have to consider the variable Π_i that expresses the level of the node i.

Objective function The objective function maximises the distance between levels of any two nodes f and i

$$\max \Pi_f - \Pi_i$$

Constraints To get this problem to work we have to add some constraints. In particular we have to say that the difference between the levels of any two nodes i and j can't be bigger than the cost c_{ij} of the path that connect the two nodes.

$$\Pi_j - \Pi_i \leq c_{ij} \quad \forall (i,j) \in A$$

Path An arc (i,j) belongs to the path p if the distance between the levels of nodes i and j is equal to the cost of the path that connect the two nodes

$$\Pi_j - \Pi_i = c_{ij} \Rightarrow (i,j) \in p$$

Activity	Description	Time
A	Crush biscuits	3
B	Melt butter	5
C	Mix butter and biscuits	2
D	Soak the jelly	10
E	Whip cheese, sugar and cream	7
F	Join cheese and jelly and let cool	15
G	Cut an orange	2

Table 7.1: The table containing the actions with the corresponding duration.

7.4.4 Project planning

Graphs can be used also to plan activities that require a certain amount of time to complete and that have to be done respecting some precedence constraints. Let us consider for instance the recipe for a cake. The cake is made of a cookie base, the main part and a garnish. These three components can be cooked separately (in parallel) but considering a single element, we have some constraints regarding the order of operations. For instance we have to crush the cookies and then mix them with the butter, but not vice versa.

In this situation we want to minimise the time from the beginning to the end of the process and satisfy the constraints.

Graph To model this problem we can use a graph in which each node represents an action. When we find a dependence between an action B and another action D we write an arc from node B to D. The weight of an arc (i, j) represents how much time we need to end the task at node i. Two dummy states have to be added

- An initial state S to begin the process. All outgoing arcs from S have cost 0.
- A final state E to end the process. All incoming arcs in E have cost 0.

Algorithm The idea is to assign to each action (i.e. to each node) a timestamp T_i that satisfies the constraints. The timestamps represent the order in which action can be executed.

The dummy start node has timestamp 0 so as all its successors. For a generic node n we can apply the following rules

- For each incoming arc find the arc with the **highest** value of $T_i + c_{in}$ cost. We have to consider the highest value of $T_i + c_{in}$ because all previous actions must have finished so we have to wait the slowest one (i.e. the one that needs the most time, that is the most $T_i + c_{in}$).
- Assign such value as T_n.

Model The objective function we use for this problem minimises the time needed to complete all the actions (i.e. the difference between the ending and initial timestamps).

$$\min T_{end} - T_{start}$$

The only constraint is that difference between the timestamps of any two nodes i and j has to be bigger (or equal) to the cost to get from node i to node j.

$$T_j - T_i \geq c_{ij}$$

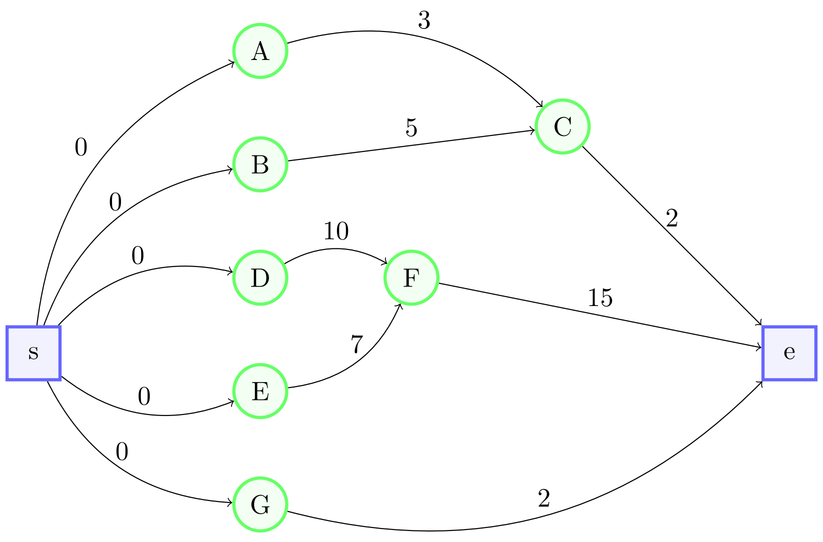


Figure 7.5: The graph that models table 7.1

Algorithm The algorithm to solve the project planning problem is the following

```
1   LongestPT_acyclic(G, P, d)
2       for i <- 2, ..., n do
3           P[i] <- 0
4           d[i] <- -infinity
5
6       P[1] = 1
7       d[1] = 0
8
9       for i <- i, ..., n-1 do
10          foreach (i, j) in FS(i) do
11              if d[i] + c[i][j] > d[j] then
12                  P[j] <- i
13                  d[j] <- d[i] + c[i][j]
```

We can notice that the algorithm is very similar to the SPT's one apart from two differences, in fact

- At line 4 we have initialised every element with $-\infty$ while in SPT algorithm we used $+\infty$.
- At line 11 we check if `d[i] + c[i][j]` is greater than `d[j]` while before we checked if it was smaller.

Longest path The longest path, also called **critical path**, from a node I to a node F is obtained starting from F and going backward using the array `P` (`P[i]` is the predecessor of node i on the path).

Starting time The starting time for each activity i is saved in the array `d`. In particular the activity i starts at time $d[i]$.

Figure 7.6: Dijkstra's algorithm

7.4.5 Dijkstra's algorithm

If a graph is acyclic then the shortest path tree algorithm can find the shortest path in linear time. In many cases graphs have cycles so it's not possible to apply the SPT algorithm, instead we can use Dijkstra's algorithm to compute the shortest path.

Assumptions To introduce the Dijkstra's algorithm we have to assume that all costs are non negative

$$c_{ij} \geq 0 \ \ \forall (i,j) \in A$$

Algorithm Dijkstra's algorithm works similarly to SPT algorithm but nodes are accessed in decreasing order of label. In particular we have to

1. Label the initial node with 0

   ```
   d[1] = 0
   ```

2. Consider the initial node as the current node.
3. Calculate the labels of the nodes adjacent to the current node.
4. Choose the node with the smallest label and repeat point 2 considering the smallest labelled node as current node.

```
SPT_Dijkstra(G, P, d)
    for i <- 2, ..., n do
        P[i] <- 0
        d[i] <- infinity

    P[1] = 1
    d[1] = 0
    Q <- {r}

    repeat
        select i from Q with minimum d[i]
        Q <- Q - {i}
        foreach (i, j) in FS(i) do
            if d[i] + c[i][j] < d[j] then
                P[j] <- i
                d[j] <- d[i] + c[i][j]
                Q <- Q + {j}
    until Q empty
```

The variable `Q` is the label in which we save the labelled nodes yet to expand.

Complexity

The critical parts in the Dijkstra algorithm are the operations to handle the list `Q`. Thus to analyse the complexity of this algorithm we have to analyse the complexity of the operations and how many times they are repeated. In particular we are going to analyse two possible implementation for `Q`

- An heap, i.e. a binary tree in which the key of the parent node has to be bigger than the keys of both nodes.
- A list.

The main operations performed on `Q` are

- Initialisation. This operation is repeated 1 time.
- Selection of the node with minimal value. This operation is repeated $|N| = n$ times.
- Insertion. This operation is repeated $|A| = m$ times.

Heap In a heap

- **Initialisation** can be done in **constant** time
$$\mathcal{O}(1)$$
- **Selection** can be done in **logarithmic** time
$$\mathcal{O}(\log n)$$
- **Insertion** can be done in **logarithmic** time
$$\mathcal{O}(\log n)$$

Thus the total complexity is
$$\mathcal{O}(m \cdot \log n)$$
because the number of arcs m is in general much bigger than the number of nodes n (hence $m \log n >> n \log n$).

List In a list

- **Initialisation** can be done in **constant** time
$$\mathcal{O}(1)$$
- **Selection** can be done in **linear** time
$$\mathcal{O}(n)$$
- **Insertion** can be done in **constant** time
$$\mathcal{O}(1)$$

Thus the total complexity is
$$\mathcal{O}(n^2)$$

Confront In general it's not possible to say which of the two implementation is better, in fact it depends on the type of graph.

OPERATIONS	NUMBER OF TIMES	HEAP COST	LIST COST
Initialisation	1	1	1
Selection	n	$\log n$	n
Insertion and update	m	$\log n$	1
Total complexity		$\mathcal{O}(m \log n)$	$\mathcal{O}(n^2)$

Table 7.2: A summary of complexities of various implementation.

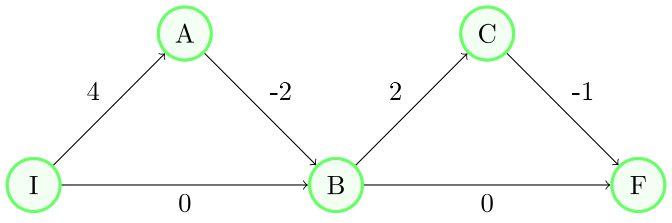


Figure 7.7: A graph with negative costs that makes Dijkstra's algorithm very slow

7.4.6 Generalisation for all graph

Initially we assumed that the graph had only non negative costs to apply Dijkstra. Now we want to generalise the algorithm to work with graph that have no constraints on the cost of each transition

$$c_{ij} \lesseqgtr 0 \quad \forall (i,j) \in A$$

Example If we execute Dijkstra's algorithm as in 7.4.5 on graph 7.7 we'll find out that nodes are inserted multiple times in `Q` and the computation has a time complexity of

$$\mathcal{O}(2^n)$$

Naive solution To generalise the Dijkstra algorithm we can apply the piece of code at lines 13-16 of 7.4.5 to every arc in A and repeat the iteration multiple times. This solution has polynomial complexity.

7.4.7 Label correction algorithm

If a graph G is cyclic and some arcs have a negative cost then we can neither apply Dijkstra nor add a constant value to all the arcs so that all the costs are non negative. To solve this problem we have to apply the label correction algorithm.

Procedure The label correction algorithm works like Dijkstra but the nodes are obtained from the queue Q using a FIFO policy, this means that the first element added is the first removed. This means that the oldest node is removed.

In particular the label correction algorithm is the following

```
1 SPT_label_correction(G, P, d, r)
2     for i in N do
```

```
        P[i] <- 0
        d[i] <- infinity

    P[r] = r
    d[r] = 0
    Q <- {r}

    repeat
        select i from Q with FIFO policy
        Q <- Q - {i}
        foreach (i, j) in FS(i) do
            if d[i] + c[i][j] < d[j] then
                P[j] <- i
                d[j] <- d[i] + c[i][j]
                Q <- Q + {j}
    until Q empty
```

Complexity This algorithm has a complexity of

$$\mathcal{O}(m \cdot n)$$

where

- m is the number of arcs.
- n is the number of nodes.

Proof For all arcs (i, j) in the Shortest Path Tree the label of node j is the sum of the label of node j and the cost of the arc

$$d[i] + c_{ij} = d[j] \tag{7.1}$$

This is true because at line 16 of 7.4.7. For equation 7.1 we can define a reduced cost

$$\bar{C_{ij}} = c_{ij} + d[i] - d[j] = 0$$

The reduced cost of the arcs in the SPT is always 0.

On the other hand for all arcs (i, j) not in the Shortest Path Tree the label of node j is bigger than the sum of the label of node j and the cost of the arc

$$d[i] + c_{ij} = d[j] \tag{7.2}$$

As for the arcs in the SPT we can calculate the reduced cost

$$\bar{C_{ij}} = c_{ij} + d[i] - d[j] \geq 0$$

that is always non negative.

Thanks to the reduced cost all the arcs have a non negative cost and the SPT on the graph with costs $\bar{C_{ij}}$ is the same of the former graph with negative costs. What changes between the reduced version and the original version is the total cost of the graph. In particular if we compute the cost on a reduced graph we obtain a cost that depends only on the label of the starting node, the label of the ending node and the cost on the former graph

$$\bar{C}_{s \to t} = d[s] - d[t] + c_{s \to t}$$

This certifies that the SPT generated by the graph with reduced costs is the same as the one generated by the original graph.

Dijkstra after label correction After applying label correction it's possible to apply Dijkstra (starting from and going to whatever node) on the graph with reduced costs, in fact the new graph only has non negative costs. We usually apply Dijkstra because it's faster.

7.4.8 Choosing an algorithm

Given a graph G we have to decide what algorithm to apply in order to find the SPT depending on the characteristics of the graph G. In particular

1. If the graph G is acyclic we can apply `SPT_acyclic` (7.4.3). This algorithm has a linear comlpexity
$$\mathcal{O}(n)$$
2. Otherwise, if the all costs are non negative we can apply Dijkstra (7.4.5) using a list for dense graphs and a tree for sparse graphs. This algorithm has a complexity of
$$\mathcal{O}(n^2)$$
or
$$\mathcal{O}(m \log n)$$
depending on the implementation.
3. Otherwise we have to apply the label correction algorithm (7.4.7). This algorithm has a complexity of
$$\mathcal{O}(m \cdot n)$$

Notice that the complexity of the algorithm to choose increases with graphs that don't have good properties.

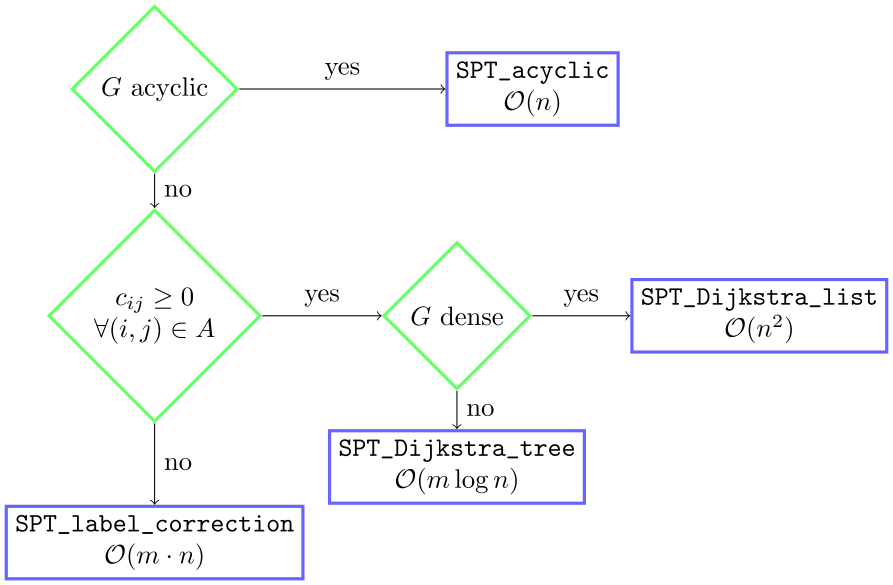


Figure 7.8: How to decide which algorithm to use to compute the SPT.

Chapter 8

Maximum flow

Maximum flow problems consist in finding the maximum flow that can be sent from a node s to a node t in a graph G with weighted arcs.

Evacuation plan example We can use the evacuation plan example to better understand what a maximum problem is. Let us consider a floor of a building made of many rooms connected by corridors. Each corridor can communicate with a garden using an exit. When the floor has to be evacuated all the people in the various rooms have to reach the exits using the corridors. Each corridor and each exit has a maximum capacity, meaning that only a limited number of people can use them. We have to decide if every person can reach an exit without exceeding the capacity of each corridor and exit. In other words we have to find if the maximum flow of people from the offices to the exit allow everyone to reach the exits.

8.1 Model

Consider a graph G with nodes N and arcs A.

$$G = (N, A)$$

Each arc (i, j) has a positive weight.

$$u_{ij} \geq 0$$

Sets

The sets we are going to use are

- The set of **nodes** N.
- The set of **arcs** A.

Parameters

To model a maximum flow problem we need the following parameters

- A **starting node** $s \in N$.

- A **final node** $t \in N$.
- The **maximum flow** (i.e. the weight) of the arcs $u_{ij} \geq 0 \ \ \forall (i,j) \in A$.

Variables

Modelling a maximum flow problem only requires one set of variables x_{ij} that represents the amount of flow on arc (i,j)

$$x_{i,j} \geq 0 \ \ \forall (i,j) \in A$$

Constraints

Modelling a maximum flow problem requires two constraints

- A **capacity constraint**. The capacity constraint limits the flow x_{ij} on arc (i,j) to the maximum flow u_{ij} of that arc. In other words we have to ensure that for every arc (i,j) the amount of flow is not grater than the maximum flow for that arc.

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

- A **flow conservation constraint**. The flow constraints ensures that the for every node, the incoming flow is equal to the outgoing flow. In other words we don't want to neither create nor lose flow. In some way we have to write a Kirchhoff Current Law.

$$\sum_{(k,i) \in BS(i)} x_{ki} - \sum_{(i,k) \in FS(i)} x_{ik} = 0 \qquad \forall i \in N \setminus \{s,t\}$$

Objective function

The objective function can be written in two equivalent ways. In particular we can

- Maximise the outgoing flow from s.

$$\max \sum_{(s,i) \in FS(s)} x_{si}$$

- Maximise the incoming flow in t.

$$\max \sum_{(i,t) \in BS(t)} x_{it}$$

These functions are equivalent because of the flow conservation constraint, in fact if we maximise the outgoing amount of flux from s and no flux is lost in the nodes between s and t, then the flux that arrives in t has to be the same as the one that left from s.

8.2 Minimum cost cut

Another interesting problem is the minimum cost cut problem. Consider a graph like the one we used to describe the maximum flow problem (i.e. with capacities on each arc). We want to cut the some arcs on the graph (i.e. $x_{ij} = 0$) so that the flow from s to t is interrupted, this means that the flow incoming in t has to be 0. The only constraint is that cutting an arc has a cost that is proportional to the capacity u of the arc. This means that we have to cut the arcs in an optimal way (i.e. minimising the cost).

Cut If we want to have a flow equal to 0, then we have to define a cut that creates two partitions N_s and N_t of the graph so that

- The initial node s belongs to N_s.
$$s \in N_s$$
- The terminal node t belongs to N_t.
$$t \in N_t$$
- A node is either in one or in the other partition (i.e. the intersection of the two partitions is empty).
$$N_s \cap N_t = \emptyset$$
- All nodes are in either one or the other partition (i.e. the union of the two partitions is the set of all nodes N).
$$N_s \cup N_t = N$$

Capacity of the cut The capacity of the cut $U(N_s, N_t)$ is the sum of the capacities of the arcs that start from a node in N_s and arrive in a node in N_t.

$$U(N_s, N_t) = \sum_{(i,j) \in A : i \in N_s, j \in N_t} u_{ij}$$

Flow traversing the cut The flow traversing the cut $x(N_s, N_t)$ is the difference between the flow from N_s to N_t and the flow from N_t to N_s.

$$x(N_s, N_t) = \sum_{(i,j) \in A : i \in N_s, j \in N_t} x_{ij} - \sum_{(j,i) \in A : j \in N_t, i \in N_s} x_{ji}$$

If we call $A^+(N_s, N_t)$ the arcs outgoing from N_s and $A^-(N_s, N_t)$ the arcs incoming to N_s from N_t, then the flow traversing the cut can be written as

$$x(N_s, N_t) = \sum_{(i,j) \in A^+(N_s, N_t)} x_{ij} - \sum_{(j,i) \in A^-(N_s, N_t)} x_{ji}$$

8.2.1 Properties

Flow traversing the cut The flow traversing a cut is always the same, in particular it's the flow that goes from s to t.

Maximum flow Given any cut (N_s, N_t), the flow traversing the cut is always smaller than or equal to the capacity of the cut.
$$x(N_s, N_t) \leq U(N_s, N_t)$$

Optimal flow The flow traversing a cut is equal to the capacity of the cut if and only if the flow is optimal.
$$x(N_s, N_t) = U(N_s, N_t) \iff (N_s, N_t) \text{ optimal}$$

Important property Given a directed graph $G = (N, A)$ and two nodes $s, t \in N$, either there is a path between s and t or there is a cut (N_s, N_t) such that each arc of the cutset is

$$(i, j) : i \in N_t, j \in N_s$$

8.2.2 Flow augmentation algorithm

The flow augmentation algorithm allows us to maximise the flow from s to t, given a graph with capacity and flow for each arc. Let us consider graph 8.1 to analyse the flow augmentation algorithm. For each arc we can increase the flow or decrease the flow. In particular

- For those arcs in which the flow is less than the capacity we can increase by $u-x$ and decrease by x the flow.
- For those arcs in which the flow is equal to the capacity we can only decrease the flow by x.

where u is the capacity and x is the flow.

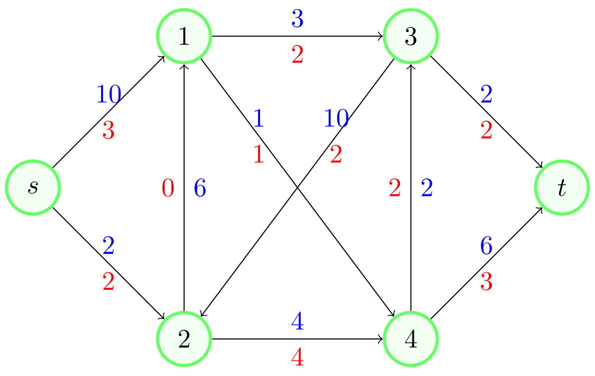


Figure 8.1: Graph for the flow augmentation algorithm.

The graph has to be re-written according to the actions that can be done, in particular consider an arc (i, j)

- If we can increase the flow from i to j then we have to draw an arc from i to j (usually coloured green to identify an increase of the flow).
- If we can decrease the flow from i to j then we have to draw an arc from j to i (usually coloured red to identify a decrease of the flow from i to j).

Residual capacities For each arc (i, j) we can define a residual capacity, in particular

- If the flow x_{ij} is less then the capacity u_{ij} then the residual capacity of the increasing arc (in green) is $u_{ij} - x_{ij}$

  $$x_{ij} < u_{ij} \Rightarrow u_{ij} - x_{ij}$$

  If the flow is grater than 0 then we can also define the residual capacity for the decreasing arc (in red) is x_{ij}

  $$x_{ij} \geq 0 \Rightarrow x_{ij}$$

- If the flow x_{ij} is equal to the capacity u_{ij} then the residual capacity for the decreasing arc (in red) is x_{ij}

$$x_{ij} = u_{ij} \Rightarrow x_{ij}$$

The residual capacities specify how much we can increase (if on a green arc) or decrease (if on a red arc) the flow of each arc.

The graph in our example with increasing and decreasing arcs is shown in figure 8.2.

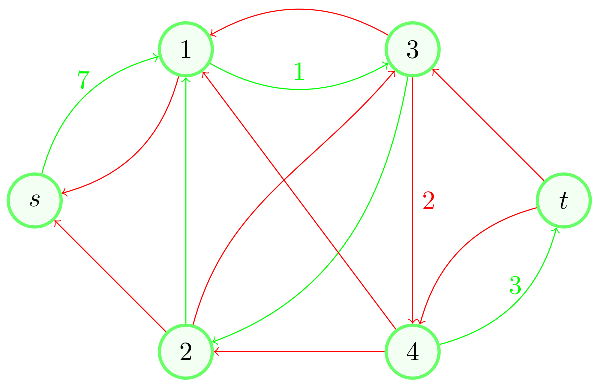


Figure 8.2: Graph with increasing and decreasing arcs (normal arcs are omitted).

Increasing and decreasing the flow To increase the flow we have to use the residual capacities. In particular we have to find a path from s to t using increasing and decreasing arcs. In our example a possible path from s to t is $s \to 1 \to 3 \to 4 \to t$. Given the path from s to t we have to consider the residual capacities on the arcs of the path. Such residual capacities specify how much we can increase or decrease the flow on the path. To decide how much to increment the flow we have to satisfy all constraints. In particular we have to satisfy the flow conservation constraint, thus we can increment the flow only by the minimum of the residual capacities. This is true because if we increment the flow by more than the minimum residual capacity then the sum of entering the flows is less than the sum of outgoing flows. In our example the minimum residual capacity is 1, thus we can increment the flows on the arcs $(s, 1)$, $(1, 3)$, $(4, t)$ by 1 and decrease the arc $(4, 3)$ by the same amount.

The resulting graph is shown in figure 8.3.

Algorithm iterations The algorithm has to be iterated until it's not possible to find a path from s to t. In our example after the first iteration the algorithm stops because we can't find a path from s to t in the new increase graph.

Notice that in graph 8.3 we can identify a cut

$$(N_s, N_t) = (\{s, 1\}, \{2, 3, 4, t\})$$

so that the flow of all arcs from N_s to N_t is maximum. In other words the flow traversing the cut is equal to the capacity of the cut.

$$x(N_s, N_t) = U(N_s, N_t)$$

$$\sum_{(i,j)\in\{(1,4),(1,3),(2,4)\}} x_{i,j} = \sum_{(i,j)\in\{(1,4),(1,3),(2,4)\}} u_{i,j}$$

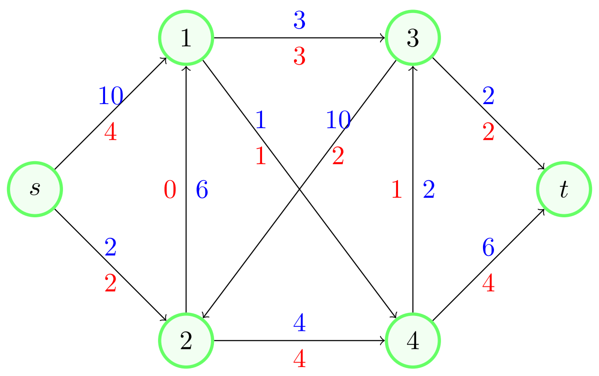


Figure 8.3: Graph with augmented flow.

Algorithm

The pseudocode for the flow augmentation algorithm is shown in listing 8.1.

```

Residual_capacity(P, x)
    increases = [u[i, j] - x[i, j] foreach (i, j) in intersection(A_plus, P)]
    decreases = [x[j, i] foreach (i, j) in intersection(A_minus, P) ]
    return min( increases + decreases )

Augment_flow(P, x)
    omega <- Residual_capacity(P, x)
    foreach (i, j) in P do
        if (i, j) in A_plus
            x[i, j] <- x[i, j] + omega
        else
            x[j, i] <- x[j, i] - omega

Augmenting_paths(G, s, t, x)
    foreach (i, j) in A do x[i, j] <- 0

    repeat
        Gr <- residual_graph(x)
        graph_search(Gr, s, P)
        if P[t] != 0 then Augment_flow(P, x)
    until P[t] = 0
```

Listing 8.1: Flow augmentation algorithm.

Edmonds and Karp complexity If we select the augmenting path with the minimum number of arcs and we implement the path search using breath first search, then the complexity of the augmenting flow algorithm is

$$\mathcal{O}(m^2 n)$$

where n is the number of nodes and m is the number of arcs.

8.3 Minimum cost flow problem

The minimum cost flow problem is a maximum flow problem in which arcs (i, j) have both capacities u_{ij} and costs c_{ij}, thus we have to find the maximum flow while minimising the cost (since moving a unit of flow over an arc costs).

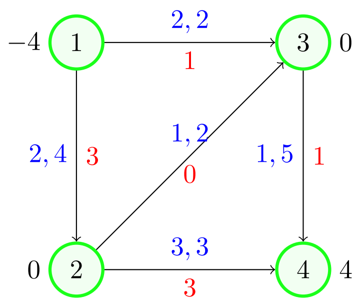


Figure 8.4: Minimum cost flow problem graph (costs and capacities in blue, flow in red).

Node capacity In minimum cost flow problems we also have to consider capacities on the nodes. In particular a capacity on a node is

- **Positive** if the node can receive some flow (**needing node**). In maximum flow problems the terminal node t is a needing node.
- **Negative** if the node can give some flow (**offering node**). In maximum flow problems the starting node s is an offering node.
- **Null** if the node is just used to transfer the flow (**transfer node**). In maximum flow problems the inner nodes are transfer nodes.

Let us assume that the sum of nodes capacities b_i is 0.

$$\sum_i b_i = 0$$

8.3.1 Model

Let us consider a graph $G = (N, A)$. Formally the minimum cost flow problem can be modelled as follows.

Sets Two sets are required to model a minimum cost flow problem

- The nodes of the arcs N. Let us call n the cardinality of N.
- The arcs of the graph A. Let us call m the cardinality of A.

Parameters Three parameters are required to model a minimum cost flow problem

- The **amount** that can be removed from or added to the nodes

$$b_i \gtreqless 0 \quad \forall i \in N$$

- The **cost** of each arc

$$c_{ij} \quad \forall (i,j) \in A$$

- The **capacity** of each arc

$$u_{ij} \quad \forall (i,j) \in A$$

Variables To model a minimum cost flow problem only one variable is required, namely the flow x_{ij} on arc (i,j).

$$x_{ij} \geq 0 \quad \forall (i,j) \in A$$

Constraints Minimum cost flow problems require the same constraints of maximum flow problems but have to be slightly modified to take into account the fact that nodes can be offering or needing nodes. In particular

- The **flow conservation constraint** has to ensure that the sum of entering and exiting flow is equal to the capacity of the node.

$$\sum_{(j,i)\in BS(i)} x_{ji} - \sum_{(i,j)\in FS(i)} x_{ij} = b_i \quad \forall i \in N$$

- The maximum flow constraint has to ensure that the flow doesn't exceed the capacity of the arc.

$$0 \leq x_{ij} \leq u_{ij} \quad \forall (i,j) \in A$$

Objective function The objective function has to minimise the cost of the flow.

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

8.3.2 Flow optimisation

Given a flow on a graph G we can improve it (i.e. optimising it, decrease its cost) using a residual graph similar to the one we used for maximum flow problems.

Residual graph In building a residual graph we have to add

- A green (increasing) arc where the flow can be increased. The cost represented on the arc is the cost of the arc. This cost is positive because if we add some unit of flow to an arc we increase the total cost of the flow.
- A red (decreasing) arc where the flow can be decreased. The cost represented on the arc is the negative cost of the arc. This cost is negative because if we remove some unit of flow from an arc we decrease the total cost of the flow. The decreasing arc representing the reduction in arc (i,j) is represented as an arc from j to i.

Let us call

- A^+ the set of increasing arcs of the residual graph.

$$A^+ = \{(i,j) \in A : x_{ij} < u_{ij}\}$$

- A^- the set of decreasing arcs of the residual graph.

$$A^- = \{(i,j) \in A : x_{ij} > 0\}$$

The residual graph obtained from graph 8.4 is shown in figure 8.5.

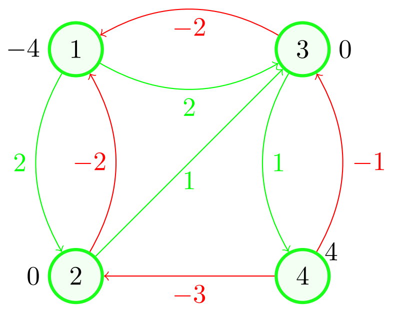


Figure 8.5: The residual graph obtained from graph 8.4.

Cycle search The optimised flow has to be feasible with respect to the capacities of the nodes and the arcs (i.e. we have to satisfy the constraints). In maximum flow problems we have looked for paths in the graph, but in this case, having to satisfy the flow conservation constraints with capacities on nodes, we have to look for negative **cycles**. The cycles have to be negative because we need to decrease the cost to reach an optimal solution.

A graph can have multiple cycles, thus we have to choose one. This operation is very important, in fact we have to choose the cycle with a specific criteria to obtain a polynomial complexity.

Residual capacities After finding a negative cost cycle in the residual graph we have to compute the residual capacities. Let us consider a cycle C ($1 \to 3 \to 4 \to 2 \to 1$ in the graph 8.5). The residual capacity γ_{ij} for each arc in the cycle is

- The difference between the capacity and the flow if the arc is an increasing arc (green).

$$\gamma_{ij} = u_{ij} - x_{ij} \iff (i,j) \in A^+$$

- The flow if the arc is a decreasing arc (red).

$$\gamma_{ji} = x_{ij} \iff (i,j) \in A^-$$

Given the residual capacities for all the arcs (red and green) in the cycle C, the flow θ that can be inserted in the cycle is the minimum residual capacity

$$\theta = \min\{\gamma_{ij} : (i,j) \in C, (i,j) \in A^+ \vee (i,j) \in A^-\}$$

Graph update After computing θ we have to

- Increase by θ the flow on increasing arcs.
- Decrease by θ the flow on decreasing arcs (considering the arc in the opposite direction with respect to the direction of the decreasing arc). In other words if we have a decreasing arc (a, b) then we have to decrease arc (b, a).

Iteration The algorithm has to be iterated until we can't find a negative cost cycle, in fact it's possible to demonstrate that if there are no negative cost cycles in the residual graph, then the flow is optimised (i.e has minimal cost).

Pseudocode The algorithm that we have analysed can be written in pseudocode as in listing 8.2.

```
Update_flow(C, theta, x)
    foreach (i,j) in C
        if (i,j) in A_plus
            x[i,j] <- x[i,j] + theta
        else
            x[j,i] <- x[j,i] - theta

Minimum_cost_flow(C):
    x <- Feasible_flow
    GR <- Residual graph(x)
    while Negative_cost_cycle(C, GR)
        theta <- Residual_capacity(C, x)
        Update_flow(theta, C, x)
        Update(GR)
```

Listing 8.2: The pseudocode to solve the minimum cost flow problem.

Complexity Selecting the cycles in the best possible way the algorithm needs

$$\mathcal{O}(m^2 n \log n)$$

iterations.

Each iteration has a complexity of

- $\mathcal{O}(mn)$ given by the `SPT_L_queue` to find the cycle.
- $\mathcal{O}(n)$ given by the graph update.

8.4 Minimum cost multicomodity flow problem

The minimum cost multicomodity flow problem is a variant of the minimum cost flow problem. In particular in this case the flow can be divided in different types and each node can offer or be in need of a certain type of flow. The arcs accept all types of flows and the capacity takes into account the sum of all the types of flows.

A practical example Here is a practical example to better understand this problem. Let us consider a delivery company that delivers two types of products (electronics and food) that are two types of flow. The products are taken from different origins (the offering nodes) and have to be delivered to different places (electronics to repair shops, food to supermarkets) but the lorries used to transport them can travel on any street (the arcs) without distinction between lorries that are transporting electronic components or food. The goal of the delivery company is to deliver all the goods at minimum cost.

8.4.1 Model

The model of this problem is mainly based on the model of the minimum cost flow problem and some minor adjustments have to be made to consider the fact that multiple types of flows travel the graph.

Sets Three sets are required to model a minimum cost multicomodity flow problem

- The nodes of the arcs N. Let us call n the cardinality of N.
- The arcs of the graph A. Let us call m the cardinality of A.
- The set of flows K.

Parameters Three parameters are required to model a minimum cost multicomodity flow problem

- The **amount** that can be removed from or added to the nodes. Because we have multiple flows we have to consider multiple amounts, one for each type of flow $k \in K$

$$b_i^k \gtreqless 0 \quad \forall i \in N, k \in K$$

- The **cost** of each arc

$$c_{ij} \quad \forall (i,j) \in A$$

- The **capacity** of each arc

$$u_{ij} \quad \forall (i,j) \in A$$

Variables To model a minimum cost multicomodity flow problem we need one variable for each type of flow. In particular the variable x_{ij}^k represents the flow of type $k \in K$ on arc (i,j).

$$x_{ij}^k \geq 0 \quad \forall (i,j) \in A, k \in K$$

Constraints Minimum cost flow problems require the same constraints of maximum flow problems but have to be slightly modified to take into account the fact that nodes can be offering or needing nodes. In particular

- The **flow conservation constraint** has to ensure that the sum of entering and exiting flow of type k is equal to the capacity of the node for flow of type k.

$$\sum_{(j,i) \in BS(i)} x_{ji}^k - \sum_{(i,j) \in FS(i)} x_{ij}^k = b_i^k \quad \forall i \in N, k \in K$$

- The **maximum flow constraint** has to ensure that the sum of all types of flows doesn't exceed the capacity of the arc.

$$0 \leq \sum_{k \in K} x_{ij}^k \leq u_{ij} \quad \forall (i,j) \in A$$

Objective function The objective function has to minimise the cost of the flow considering the different types of flows.

$$\min \sum_{(i,j) \in A} \left(c_{ij} \sum_{k \in K} x_{ij}^k \right)$$

Part IV

Linear programming

Chapter 9

Introduction

9.1 Model

Linear programming problems are part of a family of problems that can be expressed in the following form

$$\begin{aligned} &\max f(x) \\ &g_i(x) \leq b_i \\ &x_j \in \mathbb{R} \end{aligned} \tag{9.1}$$

When $g_i(x)$ and $f(x)$ are linear functions then the problem is a linear programming problem.

9.1.1 Matrix form

The model 9.1 can be rewritten in a matrix form. In particular

- x is a vector.

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

- The function $f(x)$ is a linear combination of the elements of x, thus we can use a row vector C so that $f(x) = C \cdot x$, where $\cdot$ is the matrix product.

$$C = \begin{bmatrix} c_1 & \cdots & c_n \end{bmatrix}$$

  The number of elements in C has to be the same of the number of elements in x.

- b is a vector.

$$b = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}$$

- The functions g_i can be represented as a matrix A of coefficients, so that $g_i(x)$ is the i-th row of the matrix product (row-column) between A and x. The matrix A has to be a $m \times n$ (rows $\times$ columns) matrix otherwise it wouldn't be possible to compute the matrix product.

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \dots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$$

Using the elements described above we can rewrite the model 9.1 as

$$\begin{aligned} &\max Cx \\ &Ax \le b \end{aligned} \tag{9.2}$$

or more explicitly as

$$\begin{gathered} \max \begin{bmatrix} c_1 & \cdots & c_n \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \\ \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \dots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \le \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix} \end{gathered} \tag{9.3}$$

9.1.2 Graphical representation

If we consider the linear programming problem in two dimensions we can easily represent it on a Cartesian plane.

Model To get things straight let us rewrite the model in two dimensions.

$$\begin{aligned} &\max Cx = \max \begin{bmatrix} c_1 & c_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &Ax \le b = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \le \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{cases} a_{11}x_1 + a_{12}x_2 \le b_1 \\ a_{21}x_1 + a_{22}x_2 \le b_2 \end{cases} \end{aligned} \tag{9.4}$$

Let us consider the two constraints

$$\begin{cases} a_{11}x_1 + a_{12}x_2 \le b_1 \\ a_{21}x_1 + a_{22}x_2 \le b_2 \end{cases}$$

If we replace the inequality with an equality we obtain a couple of lines. To decide what part of the plane satisfies the constraints we have to compute the gradient ∇ of both the constraints

$$\begin{cases} \nabla_1 = (a_{11}, a_{12}) \\ \nabla_2 = (a_{21}, a_{22}) \end{cases}$$

To define where the inequality is satisfied we have to go towards smaller values, thus we have to go in the opposite direction with respect to the gradient (because ∇ represents the direction in which the function grows faster).

If we want to maximise the function Cx then we have to follow the gradient of Cx until we can't find a new point in the direction of the gradient. We can also write a set of parallel lines each of which represents the points that have the same value with respect to the function to maximise. To maximise Cx we have to move from line to line in the direction of the gradient until it's not possible to find a new line with a point in the solution space. For instance the line $Cx = 4$ is the set of points x for which Cx is 4.

Example To better visualise the graphical representation of the constraints let us consider the following model

$$\begin{aligned} 1x_1 + 0x_2 &\leq 4 \\ 0x_1 + 1x_2 &\leq 2 \\ 3x_1 + 2x_2 &\leq 14 \end{aligned} \tag{9.5}$$

From this model we can obtain the gradient for each equation

$$\begin{cases} \nabla_1 = (1, 0) \\ \nabla_2 = (0, 1) \\ \nabla_3 = (3, 2) \end{cases}$$

The yellow area in figure 9.1 represents the possible solutions for the problem, i.e. the space of solutions that satisfy all the constraints. Among all these points we have to find the one that maximises the function Cx.

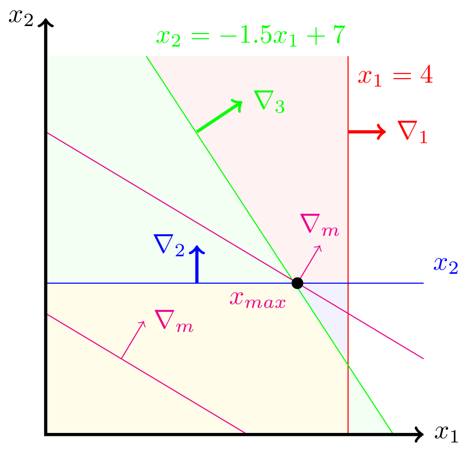


Figure 9.1: A visual representation of constraints 9.5.

Having found the solutions space we have to find the solution that maximises Cx. Let us consider

$$Cx = 3x_1 + 5x_2$$

The gradient of this function is

$$\nabla_m = (3, 5)$$

thus we have to move in this direction until we can't find a point in the solution space. If we consider

$$3x + 5x = 8$$

we can move in the direction of ∇_m and find a point in the solution space (yellow region) but if we consider

$$r : 3x + 5x = 20$$

we find out that there is a point $x_{max} = (3.3, 2)$ that belongs to r and there is no other line bigger than r with a point in the solutions region, thus

$$x_{max} = (3.3, 2)$$

is the solution to the problem.

9.1.3 Convex sets definitions

Linear programming is based on convex sets, thus its useful to remember some definitions about convex sets.

Definition 16 (Convex set)**.** *A set S is convex if and only if given any two points x and y in the set, the segment that joins them is completely contained in S. In other words S is convex if and only if, given $\lambda \in [0, 1]$ the segment $\lambda x + (1 - \lambda)y$ is completely contained in S.*

$$S \subset \mathbb{R}^n \textit{ convex } \iff \lambda x + (1 - \lambda)y \in S \quad \forall x, y \in S, \lambda \in [0, 1]$$

Definition 17 (Convex polyhedron)**.** *A polyhedron $P \subset \mathbb{R}^n$ is convex if it contains the segments that connects any two points in the polyhedron.*

$$P = \{x : A_i x \leq b_i, i = 1, ..., n\}$$

Definition 18 (Cone)**.** *A polyhedron cone C is the set of points x so that λx is in C, for every non negative λ.*

$$C = \{x : x \in C \Rightarrow \lambda x \in C, \forall \lambda \geq 0\}$$

Definition 19 (Convex cone)**.** *A cone C is convex if, given any two points x and y, their linear combination is always contained in C.*

$$C \textit{ convex } \iff \lambda x + \mu y \in C \quad \forall x, y \in C, \mu, \lambda \geq 0$$

Definition 20 (Polyhedral cone)**.**

$$P = \{x : A_i x \leq 0, i = 1, ..., n\}$$

Definition 21 (Finitely generated cone)**.** *A finitely generated cone is a cone that can be generated by a finite number of rays.*

$$x_1, ..., x_r \in \mathbb{R}^n$$

The cone can be written as

$$C = \{x : x = \prod_{i=1}^{r} \lambda_i x_i\}$$

9.2 Optimal points

Let us call P the convex polyhedron formed by the intersection of the inequalities. In other words P is the space of solutions. Intuitively the points on the border are those who are more probable to be the optimal solutions, thus it's important to define such points.

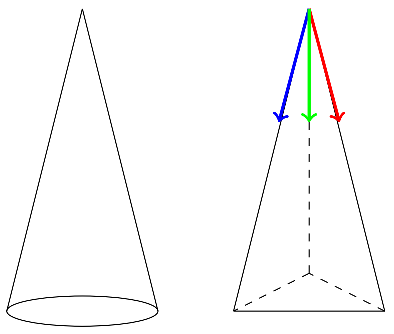


Figure 9.2: An infinitely generated cone (left) and a finitely generated cone (right).

9.2.1 Extreme point

A point x is an extreme point of a polyhedron (space of solution) P if and only if we can't find any two different points y and z in P so that x lies on the segment that unites y and z

$$x \text{ extreme point} \iff \nexists y, z : y \neq z \wedge x = \lambda y + (1-\lambda)z \text{ for some } \lambda \in [0,1]$$

Using this definition is rather difficult, in fact given a point it's hard to prove that there doesn't exist the points y, z of the definition.

9.2.2 Vertex

A point x is a vertex of a polyhedron P if and only if for every point y in P there exists a line r that passes for point y in P that is smaller than the line parallel to r that passes in x.

$$x \text{ vertex} \iff \forall y \in P \exists C : y \neq x \wedge Cx > Cy$$

As before this definition is hard to verify, in fact we have to check whatever vector C (i.e. whatever line).

9.2.3 Basic solution

Let us consider a point x in polyhedron P. x can be uniquely defined by the intersection of two lines.

In general in a n-dimensional space x is a basic solution if and only if there exists n linear independent equations. More formally let us call P the polyhedron of solutions (i.e. the intersection of all the constraints)

$$P = \{x : A_i x \leq b_i, i = 1, ..., m\}$$

and $I(x^*)$ the set of indices of the constraints

$$I(x^*) = \{i : A_i x^* = b_i, x^* \in P\}$$

The point x^* is a solution if among all $A_i : i \in I(x^*)$ (where A_i is a row of matrix A) there are n independent constraints.

If we call $A_B x = b_B$ such equation, then the solution x is

$$x = A_B^{-1} b_B$$

In our example constraints 2 and 3 are linear independent (i.e. not parallel in the graphical representation) thus we can write the equation

$$\begin{gathered} A_B x = b_B \\ \begin{bmatrix} 0 & 1 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 14 \end{bmatrix} \end{gathered}$$

thus the solution of this system is

$$\begin{aligned} x &= A_B^{-1} b_B \\ &= \begin{bmatrix} x_1 = 3.3 \\ x_2 = 2 \end{bmatrix} \end{aligned}$$

9.3 Dual problems

In linear programming, given the model of a problem, we can always define another problem and its model that is strongly related with the former one. Solving one of the problem allow us to solve also the dual one.

In particular there exists two couples of dual problems and for each couple, given a problem it's always possible to mechanically obtain the dual.

9.3.1 First dual problems

The following problems are related

$$\begin{aligned} &\max cx \\ &\quad Ax \leq b \end{aligned} \qquad \begin{aligned} &\min yb \\ &\quad yA = c \\ &\quad y \geq 0 \end{aligned} \tag{9.6}$$

An important property of these couple of problems is expressed in the weak duality theorem

Theorem 1 (Weak duality). *If $\bar{x}$ and $\bar{y}$ are feasible solutions, then $c\bar{x}$ is always smaller than or equal to $\bar{y}b$.*

$$c\bar{x} \leq \bar{y}b$$

If the objective functions $c\bar{x}$ and $\bar{y}b$ are equal then the solutions $\bar{x}$ and $\bar{y}$ are optimal.

$$c\bar{x} = \bar{y}b \Rightarrow \bar{x}, \bar{y} \textit{ optimal}$$

9.3.2 Second dual problems

The following problems are related

$$\begin{aligned} &\min cx \\ &\quad Ax \geq b \\ &\quad x \geq 0 \end{aligned} \qquad \begin{aligned} &\max yb \\ &\quad yA \leq c \\ &\quad y \geq 0 \end{aligned} \tag{9.7}$$

As for the first type of dual problems, even in this case the weak duality theorem holds.

Theorem 2 (Weak duality)**.** *If $\bar{x}$ and $\bar{y}$ are feasible solutions, then $c\bar{x}$ is always bigger than or equal to $\bar{y}b$.*

$$c\bar{x} \geq \bar{y}b$$

If the objective functions $c\bar{x}$ and $\bar{y}b$ are equal then the solutions $\bar{x}$ and $\bar{y}$ are optimal.

$$c\bar{x} = \bar{y}b \Rightarrow \bar{x}, \bar{y} \textit{ optimal}$$

9.3.3 Obtaining the dual

Given a problem, to obtain the dual formulation, we can use the table 9.1.

min	max
variables	constraints
constraints	variables
cost vector c	right hand side b
right hand side b	cost vector c
$A_ix \geq b_i$	$y_i \geq 0$
$A_ix \leq b_i$	$y_i \leq 0$
$A_ix = b_i$	y_i unrestricted
$x_i \geq 0$	$yA^i \leq c_i$
$x_i \leq 0$	$yA^i \geq c_i$
x_i unrestricted	$yA^i = c_i$

Table 9.1: The table that allows to convert a problem in its dual.

For instance if we consider the following model

$$\begin{aligned} \max\ & 3x_1 + 5x_2 \\ & 1x_1 + 0x_2 \leq 4 \\ & 0x_2 + 2x_2 \leq 12 \\ & 3x_1 + 2x_2 \leq 14 \end{aligned} \qquad \begin{aligned} \max\ & \begin{bmatrix} 3 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ & \begin{bmatrix} 1 & 0 \\ 0 & 2 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} 4 \\ 12 \\ 14 \end{bmatrix} \end{aligned}$$

then the dual problem is

$$\begin{aligned} \min\ & 4y_1 + 12y_2 + 14y_3 \\ & 1y_1 + 0y_2 + 3y_3 \leq 3 \\ & 0y_1 + 2y_2 + 2y_3 \leq 5 \end{aligned} \qquad \begin{aligned} \min\ & \begin{bmatrix} 4 & 12 & 14 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \\ & \begin{bmatrix} 1 & 0 & 3 \\ 0 & 2 & 2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \leq \begin{bmatrix} 3 \\ 5 \end{bmatrix} \end{aligned}$$

Chapter 10

Solving linear programming problems

Consider the linear programming problem 10.1

$$\begin{aligned} &\max cx \\ &\quad Ax \leq b \end{aligned} \qquad\qquad \begin{aligned} &\min yb \\ &\quad yA = c \\ &\quad y \geq 0 \end{aligned} \tag{10.1}$$

The algorithm used to solve a linear programming problem is an improving algorithm, in fact it finds a feasible solution and tries to improve it until it's not possible to find another better feasible solution. In particular let $\bar{x}$ be a feasible solution for problem 10.1, we can try to improve it by moving in the solution space. In particular we can find a new solution x' moving along vector $\xi > 0$ with a module of $\lambda \in \mathbb{R}^n$.

$$x' = \bar{x} + \lambda\xi$$

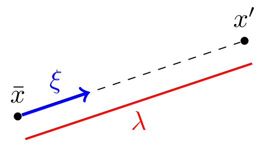


Figure 10.1: Improvement of solution $\bar{x}$.

The new solution x' is feasible if $Ax' \leq b$ (i.e. if it satisfies the constraints) and is better than the previous solution if $cx' \geq c\bar{x}$ (i.e. if the new value of the function to optimise is bigger than the previous one). Our goal is to find two values of ξ and λ so that x' doesn't violate the constraints and improves the objective function.

Algorithm Algorithmically we can define the process used to find the optimal solution x^* as follows

```
consider a feasible solution x
while exists a growing feasible direction xi:
    determine lambda
    move of lambda along xi
```

10.1 Direction

Initially we are going to focus on finding a direction ξ that satisfies the constraints and that is able to improve the objective function.

10.1.1 Improving the objective function

The direction ξ improves the objective function cx if $cx' > c\bar{x}$. The value of x' can be replaced with its definition $x' = \bar{x} + \lambda\xi$ to obtain

$$c(\bar{x} + \lambda\xi) > c\bar{x} \tag{10.2}$$
$$c\bar{x} + c\lambda\xi > c\bar{x} \tag{10.3}$$

The term $c\bar{x}$ in equation 10.3 can be simplified to obtain

$$c\lambda\xi > 0 \tag{10.4}$$

Because λ is greater than 0 by definition $c\lambda\xi$ is positive if $c\xi$ is positive, thus ξ improves $\bar{x}$ if

$$c\xi > 0 \tag{10.5}$$

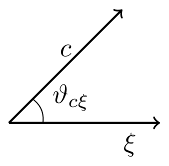
Geometric meaning

Let's focus on the geometric meaning of constraint $c\xi \geq 0$. The scalar product between c and ξ

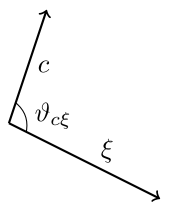
$$c \cdot \xi = |c| \cdot |\xi| \cdot \cos\vartheta_{c\xi}$$

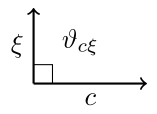
is

- Positive if $\cos\vartheta_{c\xi}$ is positive (i.e. the angle is acute).



- Negative if $\cos\vartheta_{c\xi}$ is negative (i.e. the angle is obtuse).





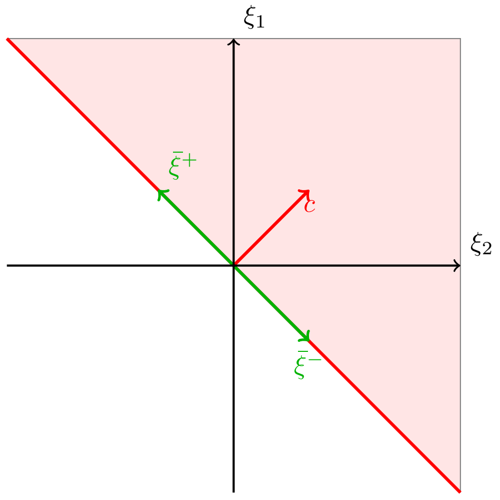


Figure 10.2: The geometric interpretation of the region of optimised $\bar{\xi}$.

- Null if $\cos\vartheta_{c\xi}$ is 0 (i.e. the angle is right).

This means that the region of solutions $\bar{\xi}$ that improve $\bar{x}$ goes from the extreme $\bar{\xi}^+$ and $\bar{\xi}^-$ (i.e. the vectors for which $c\bar{\xi} = 0$). The vectors $\bar{\xi}^+$ and $\bar{\xi}^-$ lie on the line perpendicular to c (i.e. have same direction) but opposite sign.

10.1.2 Finding a feasible solution

After finding a ξ that can improve the solution $\bar{x}$, we also have to ensure that ξ allows to obtain a feasible solution. In particular we want to ensure that the new solution $x' = \bar{x} + \lambda\xi$ satisfies the constraints $Ax' \leq b$. Geometrically we want to ensure that x' belongs to the polyhedron $Ax' \leq b$.

To find the condition for ξ to be acceptable (i.e. to generate a feasible solution) we can replace x' with its definition $x' = \bar{x} + \lambda\xi$

$$Ax' \leq b \tag{10.6}$$

$$A(\bar{x} + \lambda\xi) \leq b \tag{10.7}$$

Among all constraints we want to focus on the most restrictive ones, that is on the ones for which the equality holds. Such constraints are the most restrictive because the point can't be moved in any direction but only in the opposite direction of the gradient of the constraint (consider $A_1 x \leq b_i$, if $A_1\bar{x} = b$ then we can't move in the direction of A_1 because otherwise we will obtain a value x' so that $Ax' > b$). The constraints for which the equality holds (also called **active constraints**) are represented using the set of indices $I(\bar{x})$

$$I(\bar{x}) = \{i : A_i\bar{x} = b_i\}$$

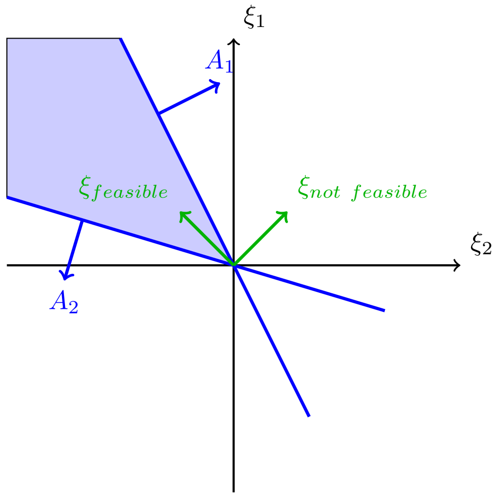


Figure 10.3: The geometric interpretation of a feasible value for ξ.

where A_i is a i-th row of matrix A and b_i is the i-th number of vector b.

Now that we have defined the set of interesting constraints (i.e. the active constraints) we can continue expanding equation 10.7 considering only the active constraints $A_i : i \in I(\bar{x})$. Let us call A_I the matrix with $A_i : i \in I(\bar{x})$ and b_I with $b_i : i \in I(\bar{x})$.

$$A_I(\bar{x} + \lambda\xi) \leq b_I \tag{10.8}$$
$$A_I\bar{x} + A_I\lambda\xi \leq b_I \tag{10.9}$$

By definition $A_i\bar{x} = b_i$ thus we can simplify A_I with b_I (because $A_I - b_I = 0$) and obtain

$$A_I\lambda\xi \leq 0 \tag{10.10}$$

Because λ is positive by definition, then $A_I\lambda\xi$ is negative if $A_I\xi$ is negative, thus the condition for ξ to generate a feasible solution is

$$A_I\xi \leq 0 \tag{10.11}$$

Geometric interpretation

The set of active constraints A_I defines a cone in the space ξ, thus any vector in the cone defined by A_I is a feasible candidate for the solution ξ. The geometric interpretation for two dimensions can be visualised in figure 10.3.

10.1.3 Finding the direction

The vector ξ has to be feasible and has to optimise the solution, thus it has to satisfy the conditions

$$A_I\xi \leq 0$$
$$c\xi > 0$$

The visual representation of these constraints in two dimensions is shown in figure 10.4, in which the green area, that is the intersection between the blue ($A_I\xi \leq 0$) and the red area ($c\xi > 0$) is the region in which we can choose ξ

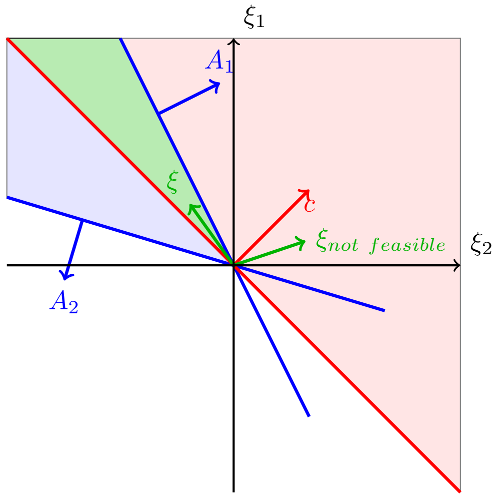


Figure 10.4: The geometric interpretation of ξ.

10.1.4 Optimal solution

When it's not possible to find a direction ξ that improves the previous solution $\bar{x}$ and that satisfies the constraints, then $\bar{x}$ is the optimal solution. To better understand this concept let us transform the constraints used to find ξ in a linear programming problem. In particular the first constraint $c\xi > 0$ can be transformed in an optimisation function to maximise.

$$P: \begin{array}{r} c\xi > 0 \\ A_I\xi \leq 0 \end{array} \qquad\qquad P': \begin{array}{l} \max\, c\xi \\ \qquad A_I\xi \leq 0 \end{array} \tag{10.12}$$

As for any other linear programming problem, we can also write the dual of 10.12

$$D: \begin{array}{r} \eta A_I = c \\ \eta \geq 0 \end{array} \qquad\qquad D': \begin{array}{l} \min\, 0\eta \\ \qquad \eta A_I = c \\ \qquad \eta \geq 0 \end{array} \tag{10.13}$$

If the system in 10.12 (on the left) has a solution, then the respective linear programming problem P' is unbounded, in fact if there exists a feasible direction ξ' that improves the solution $\bar{x}$ (i.e. a ξ' that satisfies P), then we can multiply ξ' by any positive value α and still obtain a feasible direction ξ' (if ξ' is feasible then also $\alpha\xi'$ is feasible because we are just modifying the module, not the direction of ξ', thus $\alpha\xi'$ is still a solution of the linear programming). For this reason we can say that ξ' is unbounded because we can improve ξ' by any arbitrarily big value α, thus there is no bound to the value of $\alpha c\xi'$ to maximise (i.e. if $\alpha \to \infty$ then $\alpha c\xi' \to \infty$).

If problem P' is unbounded than its dual D' in 10.13 has no solution, in fact the weak duality theorem 1 states that $c\xi$ has to be smaller or equal than ηb, but $c\xi$ is unrestricted thus it doesn't exists a value of ηb that is bigger than $c\xi$. On the other hand if there exists a solution for D' then $\xi = 0$ is a feasible and optimal solution for P'. If $\xi = 0$ is the only feasible solution then $\bar{x}$ has to be optimal because the only way to improve $\bar{x}$ is to move in the direction 0 (i.e. to stay where we are). These properties are expressed in the strong duality theorem.

Theorem 3 (Strong duality)**.** *Consider problem P, a feasible solution $\bar{x}$ of P and the problems related to the search for feasible growth directions P' and D'. If D' has a solution, then $\bar{x}$ is an optimal solution for P.*

10.2 Moving step

After defining the conditions to find the direction ξ, we have to obtain the moving step λ. In particular we want to find the maximum value of λ such that the improved solution $x' = \bar{x} + \lambda\xi$ is feasible (we are sure that x' improves $\bar{x}$ because of how we have chosen the direction ξ). In this case, to maximise λ, we want to consider the **inactive constraints**, that is the constraints for which the equality doesn't hold. To identify the inactive constraints we can use the set $\bar{I}(\bar{x})$ of indices of inactive constraints

$$\bar{I}(\bar{x}) = \{i : A_i\bar{x} < b_i\}$$

where A_i is a row of matrix A and b_i is a value of vector b.

10.2.1 Finding a feasible value

As before we have to ensure that x' satisfies the constraints $Ax' \leq b$ but in this case we want to consider only the inactive constraints, thus the condition to verify is

$$A_{\bar{I}}x' \leq b_{\bar{I}} \tag{10.14}$$

where $A_{\bar{I}}$ is the matrix made of only the rows $A_i : i \in \bar{I}(\bar{x})$ and b_I is the vector of values $b_i : i \in \bar{I}(\bar{x})$. The constraint 10.14 can be expanded replacing x' with its definition $x' = \bar{x} + \lambda\xi$

$$A_{\bar{I}}(\bar{x} + \lambda\xi) \leq b_{\bar{I}} \tag{10.15}$$
$$A_{\bar{I}}\bar{x} + A_{\bar{I}}\lambda\xi \leq b_{\bar{I}} \tag{10.16}$$

Now we can bring the term $A_{\bar{I}}\bar{x}$ to the left side to obtain the constraint

$$A_{\bar{I}}\lambda\xi \leq b_{\bar{I}} - A_{\bar{I}}\bar{x} \tag{10.17}$$

From 10.16 we notice that the right hand side is surely positive because we have considered only the inactive constraints, that is the constraints for which $A_i\bar{x} < b_i$ holds (thus $b_i - A_i\bar{x} > 0$). Furthermore we also have to remember that λ is positive by definition. Now we can have to find ourselves in two different situations

- If $A_{\bar{I}}\xi$ is negative than $A_{\bar{I}}\lambda\xi$ is surely less than $b_I - A_{\bar{I}}$ because the latter is positive and λ is positive, thus any value of λ is a solution. In other words in this case the problem is unbounded.
- If $A_{\bar{I}}$ is positive than we are approaching the border of a constraint and λ can be computed as

$$\lambda = \min\left\{\frac{b_i - A_i\bar{x}}{A_i\xi} \quad \forall i \in \bar{I}(\bar{x}), A_i\xi > 0\right\}$$

  because λ has to be smaller than or equal to $\frac{b_i - A_i\bar{x}}{A_i\xi}$ where

  - $b_i - A_i\bar{x}$ is the distance from the constraint border.
  - $A_i\xi$ is the approaching speed to the border.

10.3 Searching for a feasible solution

Until now we have considered problems in which a feasible starting solution $\bar{x}$ was given and we only had to improve it, but not in all cases such solution is available thus we have to find a way to check if a feasible solution exists and to compute it.

Let us consider problem P

$$\begin{aligned} P : \max\ & cx \\ & Ax \leq b \end{aligned}$$

If b is non negative (i.e. $b \geq 0$) than $\bar{x} = 0$ is a feasible starting point from which we can start the algorithm (if we replace $\bar{x} = 0$ in the constraints we obtain $0 \leq b$ which is true for whatever positive value of b). On the other hand if some values of b are negative, we have to define an algorithm to compute the starting $\bar{x}$. In particular we can solve an auxiliary problem AP. Before defining AP we must distinguish the non negative values of b from the others, in particular we can define two sets of indices I_- and I_+ that hold respectively the indices of negative and non negative values of b.

$$\begin{aligned} I_- &= \{i : b_i < 0\} \\ I_+ &= \{i : b_i \geq 0\} \end{aligned}$$

Given the sets I_- and I_+, we can define AP as

$$\begin{aligned} AP : \max\ & -\sum_{i \in I_-} u_i \\ & A_{I_+} x \leq b_{I_+} \\ & A_{I_-} x - u_{I_-} \leq b_{I_-} \\ & -u_{I_-} \leq 0 \end{aligned}$$

Notice that $(x^* = 0, u^*_{I_-} = -b_{I_-})$ is a feasible solution for AP, thus we can compute the optimal solution of AP (because we have a starting feasible solution $(x^*, u^*_{I_-})$ and we know the algorithm to solve a linear programming problem). Given the optimal solution $(\bar{x}, \bar{u}_{I_-})$ of AP, if $\bar{u}_{I_-}$ is 0 then $\bar{x}$ is a feasible solution for P, otherwise P has no feasible solution.

Having defined the starting solution $\bar{x}$, the direction ξ and the moving step λ we can write a more specific algorithm to find the optimal solution x^* using the primal and dual algorithm. The algorithm is shown in listing 10.1.

10.4 Complementary slackness conditions

10.4.1 Maximum problem

Let us consider problem P and its dual D in equation 10.18.

$$P : \begin{aligned} & \max cx \\ & Ax \leq b \end{aligned} \qquad\qquad D : \begin{aligned} & \min yb \\ & yA = c \\ & y \geq 0 \end{aligned} \tag{10.18}$$

Given two feasible solutions $\bar{x}$ and $\bar{y}$ the following statements are equivalent

- $\bar{x}$ and $\bar{y}$ are optimal.
- $c\bar{x} = \bar{y}b$.
- $\bar{y}(b - A\bar{x}) = 0$

The first two statements have been obtained from the weak duality theorem 1. The third statement can be obtained from the first two statements. Let us rewrite the second statement.

$$c\bar{x} = \bar{y}b \tag{10.19}$$

Since, from the first statement, $\bar{y}$ is optimal, then it is also feasible (otherwise it wouldn't be a solution). If $\bar{y}$ is feasible than it must satisfy the constraints $\bar{y}A = c$, thus we can replace $c = \bar{y}A$ in equation 10.19 to obtain

$$\bar{y}A\bar{x} = \bar{y}b \tag{10.20}$$

After bringing to the left every term and collecting the common terms we obtain equation 10.22

$$\bar{y}A\bar{x} - \bar{y}b = 0 \tag{10.21}$$
$$\bar{y}(A\bar{x} - b) = 0 \tag{10.22}$$

which is exactly the third complementary slackness condition.

Let us now examine more in detail the meaning of equation 10.22. This condition ensures that for every A_i and b_i either $\bar{y}_i$ is 0 or $A_i\bar{x} - b_i$ is 0 (or both are 0). In particular we can say that

- If $\bar{y}_i$ is not 0, then $A_i\bar{x} - b_i$ has to be 0 (i.e. $A_i\bar{x} = b_i$).
$$\bar{y}_i \neq 0 \Rightarrow A_i\bar{x} = b_i$$
- If $A_i\bar{x} - b_i$ is not 0 (i.e. $A_i\bar{x} \neq b_i$), then $\bar{y}_i$ has to be 0.
$$A_i\bar{x} \neq b_i \Rightarrow \bar{y}_i = 0$$

10.4.2 Minimum problem

The same reasoning can be applied to problem P and its dual D in equation 10.23.

$$\begin{aligned} &\min cx & &\max yb \\ P:\ &Ax \geq b & D:\ &yA \leq c \\ &x \geq 0 & &y \geq 0 \end{aligned} \tag{10.23}$$

In particular in this case, given two feasible solutions $\bar{x}$ and $\bar{y}$ the following statements are equivalent

- $\bar{x}$ and $\bar{y}$ are optimal.
- $c\bar{x} = \bar{y}b$.
- $\bar{y}(b - A\bar{x}) = 0$.
- $(c - \bar{y}A)\bar{x} = 0$.

In this case the complementary slackness conditions ensure that

- If $\bar{y}_i$ is not 0, then $A_i\bar{x} - b_i$ has to be 0 (i.e. $A_i\bar{x} = b_i$).

$$\bar{y}_i \neq 0 \Rightarrow A_i\bar{x} = b_i$$

- If $A_i\bar{x} - b_i$ is not 0 (i.e. $A_i\bar{x} \neq b_i$), then $\bar{y}_i$ has to be 0.

$$A_i\bar{x} \neq b_i \Rightarrow \bar{y}_i = 0$$

- If $\bar{x}_j$ is not 0, then $c_j - \bar{y}A^j$ has to be 0 (i.e. $c_j = \bar{y}A^j$).

$$\bar{x}_j \neq 0 \Rightarrow c_j = \bar{y}A^j$$

- If $c_j - \bar{y}A^j$ is not 0 (i.e. $c_j \neq \bar{y}A^j$), then $\bar{x}_j$ has to be 0.

$$c_j \neq \bar{y}A^j \Rightarrow \bar{x}_j = 0$$

```
PrimalDual(A, b, c, x, unbounded y):
    optimal <- false;
    unbounded <- false;
    I <- {i: A[i] * x = b[i]};
    if I not empty then
        grow_along(c, x, I, unbounded);

    while not optimal and not unbounded do
        if {eta * A[I] = c} has no solution then
            compute xi: A[I] * xi = 0 and c * xi = 1;
            grow_along(xi, x, I, unbounded);
        else if {eta * A[I] = c} has a solution and exists  h: eta[h] < 0 then
            compute xi: A[I] * xi = -u[h];
            grow_along(xi, x, I, unbounded);
        else
            optimal <- true; {the dual system has a solution: eta * A[I] = c, eta>=0}
            y[i] <- eta[i] : i in I, y[i] <- 0 otherwise.

grow_along(xi, x, lambda, unbounded):
    lambda <- min{ (b[i] - A[i] * x) / A[i] * xi : A[i] * xi > 0 and i in I, infty }
    if lambda = infty then
        unbounded <- true
    else
        x <- x + lambda * xi
    I <- {i: A[i] * x = b[i]}
```

Listing 10.1: Primal-dual algorithm to solve a linear programming problem.

Chapter 11

Basis in linear programming

11.1 Basis

Linear programming problems can be seen and solved using a different approach. To explain this new approach let us consider the following dual problems.

$$P : \max cx \qquad\qquad D : \min yb$$
$$Ax \leq b \qquad\qquad yA = c$$
$$y \geq 0$$

Such problems can be represented in matrix form as follows.

$$P : \max \begin{bmatrix} \quad c \quad \end{bmatrix} \begin{bmatrix} \\ x \\ \\ \end{bmatrix} \qquad D : \min \begin{bmatrix} \quad y \quad \end{bmatrix} \begin{bmatrix} \\ b \\ \\ \end{bmatrix}$$

$$\begin{bmatrix} & & \\ & A & \\ & & \end{bmatrix} \begin{bmatrix} \\ x \\ \\ \end{bmatrix} \leq \begin{bmatrix} \\ b \\ \\ \end{bmatrix} \qquad \begin{bmatrix} \quad y \quad \end{bmatrix} \begin{bmatrix} & & \\ & A & \\ & & \end{bmatrix} = \begin{bmatrix} \quad c \quad \end{bmatrix}$$

The constraints of P can be partitioned using a base B that contains all the linearly independent constraints for which the equality holds. In other words we are creating a set B, called base, of indices for which $A_i x = b_i$, thus the base B can be represented as

$$B = \{i : A_i x = b_i\}$$

All the indices not in B are in a set N (i.e. N contains all the constraints for which the strict inequality holds), therefore the constraints of P can be rewritten as

$$A_B x = b_B$$
$$A_N x < b_N$$

The same partitions apply to D. In this case we also have to partition y so that

$$y_B A_B = c$$

and

$$y_N A_N = 0$$

The partitions A_B and b_B (thus also A_N and b_N) are the same for P and D.

In other words in both cases B is a set of independent rows of A. In particular

- If we consider problem P then a basis B represents the constraints for which the inequality

  $$A_B x = b_B$$

  holds. In this case we can find x (pre)multiplying both sides for A_B^{-1}.

  $$x = A_B^{-1} b_B$$

- If we consider problem D then a basis B represents the components of x (i.e. the indices of x) for which

  $$y_B A_B = c$$

  and

  $$y_N = 0$$

  In this case we can find y_B (y_N is set to 0) (post)multiplying both sides for A_B^{-1}.

  $$y_B = cA_B^{-1}$$

Notice that $y_B A_B = c$ is a valid equation because y_i is a scalar (i.e. 1×1), A_i is a row (i.e. $1 \times n$), thus the product returns a $1 \times n$ row vector and c is a $1 \times n$ row.

The dual problems can be graphically rewritten as

$$P : \max \begin{bmatrix} \quad c \quad \end{bmatrix} \begin{bmatrix} \\ x \\ \\ \end{bmatrix} \qquad D : \min \begin{bmatrix} y_B & \quad y_N \end{bmatrix} \begin{bmatrix} b_B \\ \\ b_N \end{bmatrix}$$

$$\begin{bmatrix} \quad A_B \quad \\ \\ A_N \end{bmatrix} \begin{bmatrix} \\ x \\ \\ \end{bmatrix} \leq \begin{bmatrix} b_B \\ \\ b_N \end{bmatrix} \qquad \begin{bmatrix} y_B & \quad y_N \end{bmatrix} \begin{bmatrix} \quad A_B \quad \\ \\ A_N \end{bmatrix} = \begin{bmatrix} \quad c \quad \end{bmatrix}$$

11.1.1 Linear representation

Let us consider a system representation of a linear programming problem like one in Equation 11.1.

$$\begin{aligned} P : \max\ & -2x_1 + x_2 \\ & -6x_1 - 4x_2 \leq 1 \\ & 2x_1 + 6x_2 \leq 16 \\ & -2x_1 + 2x_2 \leq 7 \end{aligned} \qquad \begin{aligned} D : \min\ & y_1 + 16y_2 + 7y_3 \\ & -6y_1 + 2y_2 - 2y_3 = -2 \\ & -4y_1 + 6y_2 + 2y_3 = 1 \\ & y_i \geq 0 \end{aligned} \tag{11.1}$$

If $B = \{1, 3\}$ is a basis of P and D then we can write A_B considering

- Equations 1 and 3 of P.
- Columns 1 and 3 of D. Since D is the dual and the constraints are $yA = c$, then we have to transpose the columns 1 and 3 (intuitively the constraints are a row vector while they have been shown as a column vector, hence we have to transpose them).

In both cases we obtain

$$A_B = \begin{bmatrix} -6 & -4 \\ 2 & 6 \end{bmatrix}$$

11.2 Optimality conditions

Definition 22 (Basic solution)**.** *Given a base B we call*

$$\bar{x} = A_B^{-1} b_B \qquad \begin{aligned} \bar{y}_B &= cA_B^{-1} \\ \bar{y}_N &= 0 \end{aligned}$$

a basic solution for P and D, respectively.

Vectors $\bar{x}$ and $\bar{y}$ are also called pair of **complementary basic solutions** associated with the basis A_B.

11.2.1 Feasibility

Primal feasibility

Let us consider the basic solution $\bar{x}$ for P. Such solution is feasible if

$$A_N \bar{x} \leq b_N \tag{11.2}$$

Starting from condition 11.2 we can replace the definition of $\bar{x}$ to obtain

$$A_N A_B^{-1} b_B \leq b_N \tag{11.3}$$

Now we can rearrange the operands bringing them to the left and multiplying for -1.

$$b_N - A_N A_B^{-1} b_B \geq 0 \tag{11.4}$$

The condition in Equation 11.4 is called **primal feasibility** and tells us if a basic solution $\bar{x}$ for problem P is feasible.

If a basis A_B satisfies condition 11.4 then it is called **primal feasible basis matrix**.

Dual feasibility

Feasibility can also be checked on the basic solution $\bar{y}$. In this case we only have to impose

$$\bar{y}_B \geq 0 \tag{11.5}$$

If we replace the definition of $\bar{y}_B$ we obtain

$$\bar{c}A_B^{-1} \geq 0 \tag{11.6}$$

which is called **dual feasibility** condition and is used to check if a basic solution $\bar{y}$ is feasible.

If a basis A_B satisfies condition 11.6 then it is called **dual feasible basis matrix**.

11.2.2 Complementary slackness conditions

If we try to replace the feasible solutions $\bar{x}$ and $\bar{y}$ in the complementary slackness conditions we can verify that the equality holds. The complementary slackness conditions state that

$$\bar{y}(b - A\bar{x}) = 0 \tag{11.7}$$

If we replace $\bar{y} = [y_B, y_N]$ and $A = [A_B, A_N]^T$ we obtain

$$\begin{bmatrix} y_B & y_N \end{bmatrix} \begin{bmatrix} b_B - A_B\bar{x} \\ b_N - A_N\bar{x} \end{bmatrix} = 0 \tag{11.8}$$

After computing the product we obtain equation

$$\bar{y}_B(b_B - A_B\bar{x}) + y_N(b_N - A_N\bar{x}) = 0 \tag{11.9}$$

The second addend of the equation is trivially 0 because we imposed $y_N = 0$. To check that also the first addend is 0 we have to use the definition $\bar{x} = A_B^{-1}b_B$ of $\bar{x}$. Replacing $\bar{x}$ we obtain

$$\bar{y}_B(b_B - A_B\bar{x}) = \bar{y}_B(b_B - A_B A_B^{-1} b_B) = \tag{11.10}$$
$$= \bar{y}_B(b_B - b_B) = 0 \tag{11.11}$$

thus the first addend is also 0 and the complementary slackness conditions are satisfied.

Optimality

We have just shown that if for feasible solutions $\bar{x}$ and $\bar{y}$ conditions

$$b_N - A_N A_B^{-1} b_B \geq 0 \qquad \bar{c}A_B^{-1} \geq 0 \tag{11.12}$$

hold then also complementary slackness conditions hold. But if complementary slackness conditions hold then the solution is optimal. Thus we can conclude that when conditions 11.12 hold, the basic solutions $\bar{x}$ and $\bar{y}$ are also optimal.

Chapter 12

Integer linear programming

Integer linear programming adds integrity constrains to a linear programming problem. In other words in an integer linear programming problem all the variables have to be integer numbers. In formulas, if x is a variable then we have to specify

$$x \in \mathbb{Z}^n$$

The solution space of an integer linear programming problem isn't continuous as normal LP problems, in fact the feasible solutions are a set of points with integer coordinates. In any case, for integer linear programming problems still holds the fact that **the optimal solution is a vertex of the polyhedron formed by the constraints**.

Integer linear programming has many advantages but also disadvantages, in fact having integrity constraints forces us to modify the way the a LP is solved (because we can't freely move in a continuous space). Sometimes this isn't a problem because in some cases it's possible to drop the integrity constrains.

Solving ILP problems An integer linear programming problem can be solved, in some cases, bruteforcing all possible (i.e. feasible) solutions until the optimum is reached. In other cases it's possible to solve the problem dropping the integrity constraints, in fact if the improving algorithm seen for LP problem returns a point with integer values, then the result is optimal also if we add integrity constraints.

12.1 Advantages

Integer linear programming is very useful to solve some problems that can't be solved with linear programming.

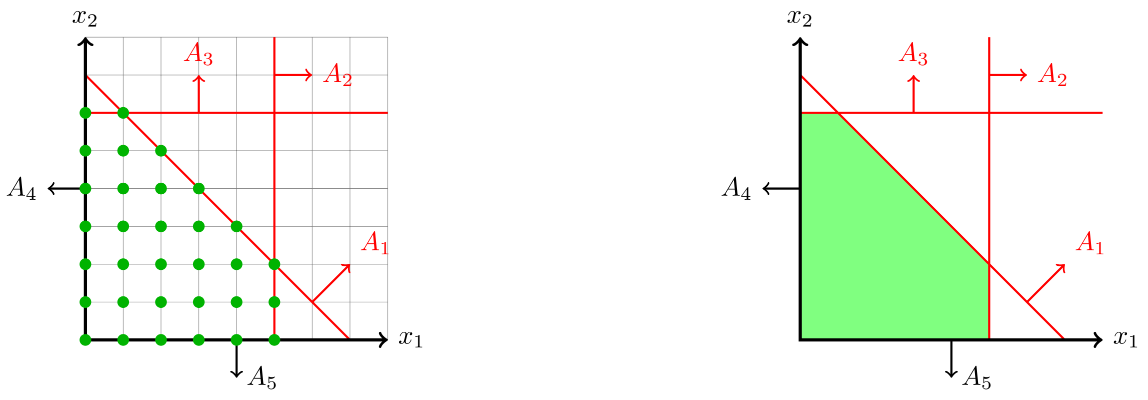


Figure 12.1: The feasible solutions of the same problem with (left) and without (right) integrity constrains.

12.1.1 Non linear objective function

Integer linear programming can be used to solve a problem in which the objective function is not linear. In particular let us consider a piecewise-defined function like the one in equation 12.1.

$$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ b_0 + 50x & \text{if } 0 < x \leq 140 \\ b_0 + 207x & \text{if } 140 < x \leq 220 \\ b_0 + 127x & \text{if } x > 220 \end{cases} \tag{12.1}$$

To solve a problem with such objective function we have can divide the space in three intervals and define three new variables z_1, z_2 and z_3, one for each interval. The variables are responsible for the intervals $(0, 140]$, $(140, 220]$, $(220, \infty]$, respectively. For this reason

- Variable z_1 can have values between 0 and 140.
- Variable z_2 can have values between 0 and 80 (because the difference between 220 and 140 is 80, in fact we consider $x = 140$ as $z_2 = 0$).
- Variable z_3 can have values between 0 and a big constant M (for the same reasons seen for z_2).

Given z_1, z_2 and z_3 we can write x as the sum of such variables, this means that we can write the following constraints

$$\begin{aligned} & x = z_1 + z_2 + z_3 \\ & 0 \leq z_1 \leq 140 \\ & 0 \leq z_2 \leq 80 \\ & 0 \leq z_3 \leq M \end{aligned} \tag{12.2}$$

The constraints 12.2 aren't enough to model the problem, in fact we must ensure that we have fully exhausted z_1 before considering z_2. To better understand why we have to consume z_1 before z_2, consider f as the price of a phone call and x the number of calls, this means that the first 140 calls have a price of 50 cent/call, the next 80 calls (from call 141 to call 220) have a cost of 207 per call, and the last ones have a cost of 127 per call. In this scenario, if we do 200 calls, we can't consume the calls with price 50 (i.e. z_1) and then the calls with price 127 (i.e. z_3) because before paying for calls after call 220 we have to pay for the calls between call 140 and 220 (actually we will consume only calls until call 200). To force such behaviour we have to add a new variable and modify a little the constraints 12.2.

Variable Let's focus on the variable to add. Let us call the new variable y_i. y_i has a value of 1 if the value of the corresponding z_i variable is positive, 0 otherwise. In other words we are using y_i to check if z_i has been consumed.

$$y_i = \begin{cases} 1 & \text{if } z_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

Updated constraints Variables y_i can be used in constraints 12.2 to model the precedence of z_i. The modified constraints are shown in 12.3.

$$\begin{aligned} & x = z_1 + z_2 + z_3 \\ & 140y_2 \le z_1 \le 140y_1 \\ & 80y_3 \le z_2 \le 80y_2 \\ & z_3 \le My_3 \end{aligned} \tag{12.3}$$

Let's try to understand why these constraints work. If we consider constraint $140y_2 \le z_1 \le 140y_1$ then

- If z_1 is 0, then both y_1 and y_2 have to be 0.
- If z_1 is less than its maximum value then y_2 has to be 0, otherwise we would obtain $140 \le z_1 \le 140$, but z_1 hasn't reached its maximum value (i.e. 140) yet, thus y_2 has to be 0. If y_2 is 0 then from the third constraint we know that $80y_2 \le z_2 \le 80y_2 = 0$, thus both z_2 and y_3 has to be 0. In this way we have shown that if z_1 hasn't reached its maximum value (i.e. hasn't been exhausted), then z_2 and z_3 have to be 0.
- If z_1 has reached it's maximum value (i.e. 140) then both y_1 and y_2 have to be 1, in fact this is the only case in which $140 \le z_1 \le 140$.

Objective function Finally we can write the objective function of this type of problem as

$$\min b_0 + 50z_1 + 207z_2 + 127z_3$$

Extension to every type of function

The same model can be applied to any type of function, in fact a non linear function can be approximated with many linear segments, thus it can be approximated as a piecewise-defined function. Given the approximation we can apply the model we have just seen to solve the problem. The model can be generalised as follows.

Function

$$f(x) = \begin{cases} 0 & \text{if } x = a_0 \\ b_0 + c_0(x - a_0) & \text{if } a_0 < x \leq a_1 \\ b_0 + c_0(a_1 - a_0) + b_1 + c_1(x - a_1) & \text{if } a_1 < x \leq a_2 \\ \dots \\ \sum_{i=1}^{k-1} \left[b_{i-1} + c_{i-1}(a_i - a_{i-1})\right] + b_{k-1} + c_{k-1}(x - a_{k-1}) & \text{if } a_{k-1} < x \leq a_k \end{cases} \tag{12.4}$$

Variable x

$$x = a_0 + \sum_{i=1}^{k-1} z_i \tag{12.5}$$

Constraints

$$\begin{aligned} &(a_1 - a_0) \cdot y_1 \leq z_0 \leq (a_1 - a_0) \cdot y_0 \\ &(a_2 - a_1) \cdot y_2 \leq z_1 \leq (a_2 - a_1) \cdot y_1 \\ &\dots \\ &0 \leq z_{k-1} \leq (a_k - a_{k-1}) \cdot y_{k-1} \end{aligned} \tag{12.6}$$

Objective function

$$\min \sum_{i=0}^{k-1} b_i y_i + c_i z_i \tag{12.7}$$

12.1.2 Concave polyhedron

In some problems the region of feasible points is a concave polyhedron, thus the algorithm we saw in linear programming problems can't be used anymore. An example of concave feasibility region is shown in Figure 12.2.

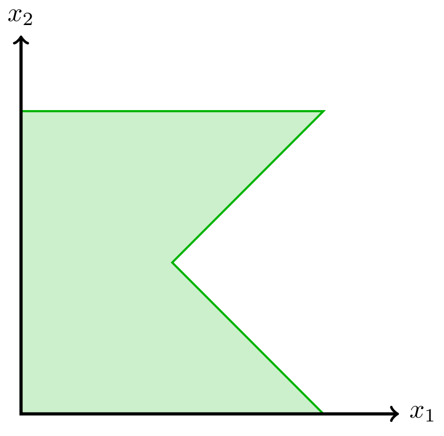


Figure 12.2: A concave feasibility region.

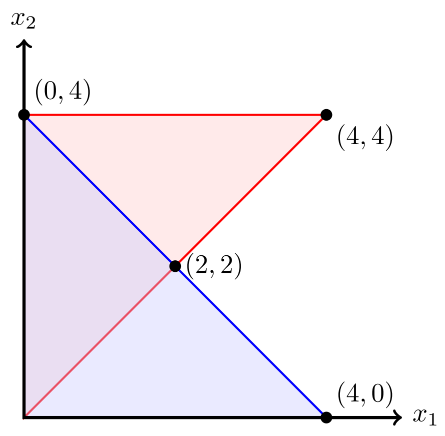


Figure 12.3: A concave feasibility region divided in two convex regions.

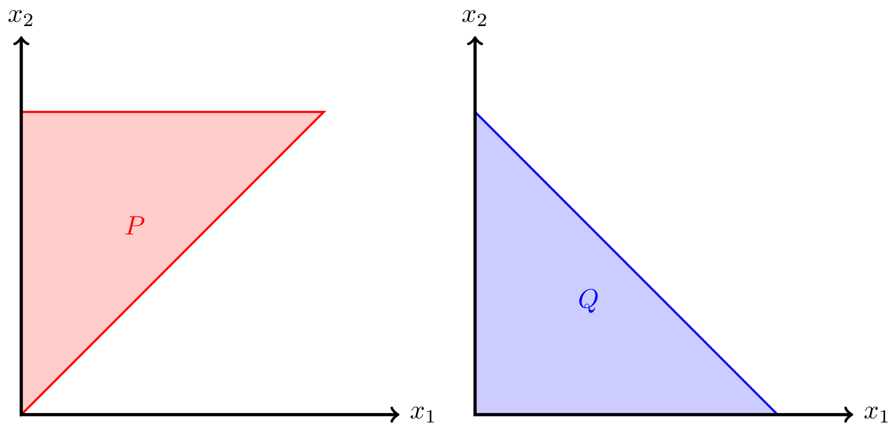


Figure 12.4: A concave feasibility region divided in two convex regions.

Fortunately we can always consider a concave polyhedron as a sum of (overlapping) convex polyhedron. For instance the region in Figure 12.2 can be divided in two triangles as shown in Figure 12.3 and Figure 12.4.

Given the two polyhedron P and Q we can write the constraints that describe them (normally we would start from the constraints and then draw the polyhedron but in this case it was more intuitive to show the concave polyhedron before the actual constraints). Such constraints are shown in 12.8 for P and 12.9 for Q.

$$\begin{aligned} x_1 &\geq 0 \\ x_2 &\leq 4 \\ x_2 &\geq 0 \\ x_2 &\geq x_1 \end{aligned} \tag{12.8}$$

$$\begin{aligned} x_1 &\geq 0 \\ x_2 &\leq 4 \\ x_2 &\geq 0 \\ x_2 &\leq -x_1 + 4 \end{aligned} \tag{12.9}$$

Given the constraints and the respective polyhedron we can notice that

- The points for which all the constraints of P and Q hold belong to the interception of P and Q.
- The points for which all the constraints of at least one of the two groups hold, belong to the union of P and Q.

Furthermore we can notice that some constraints are common to both polyhedra

$$\begin{aligned} x_1 &\geq 0 \\ x_2 &\leq 4 \\ x_2 &\geq 0 \end{aligned} \tag{12.10}$$

thus the remaining constraints are the ones that discriminate between a point that belongs to P or to Q, in particular a point belongs to P if it satisfies $x_1 - x_2 \leq 0$, otherwise it belongs to Q if it satisfies $x_2 \leq -x_1 + 4$. To model the fact that a point satisfies one constraint or (notice that until now a solution had to satisfy all the conditions, i.e. the conditions where in conjunction) the other we have to introduce a variable y and scale the constraint that shouldn't be satisfied by a big factor M. To better explain this concept consider a point $x_a = (2, 3)$ that belongs to P (but not to the interception). Such point is a feasible solution but it doesn't satisfy all the constraints in fact $3 \nleq -2 + 4$, thus to satisfy Q's constraint we have to scale it on the x_2 axis by a big factor M_Q. For instance if we consider $M_Q = 2$ we obtain

$$\begin{aligned} x_2 &\leq -x_1 + 4 + M \\ 3 &\leq -2 + 4 + 2 \end{aligned}$$

and the constraint is satisfied. Using the same reasoning if we consider a point $x_b = (2, 1)$, we have to scale P's constraint by at least a factor $M_P = 2$. The modified constraints are

$$x_1 + x_2 \leq +4 + M_Q \tag{12.11}$$

$$x_1 - x_2 \leq M_P \tag{12.12}$$

and are shown in Figure 12.5. The problem is that for some points (the ones in Q) we have to consider M_P (and ignore M_Q) while for others (the ones in P) we should consider M_Q (and ignore M_P). To solve this problem we can introduce a variable y defined as follows

$$y = \begin{cases} 1 \text{ if the point belongs to } P \\ 0 \text{ if the point belongs to } Q \end{cases}$$

Variable y can be used to write the definitive version of the constraints

$$x_1 + x_2 \leq +4 + M_Q y \tag{12.13}$$

$$x_1 - x_2 \leq M_P(1 - y) \tag{12.14}$$

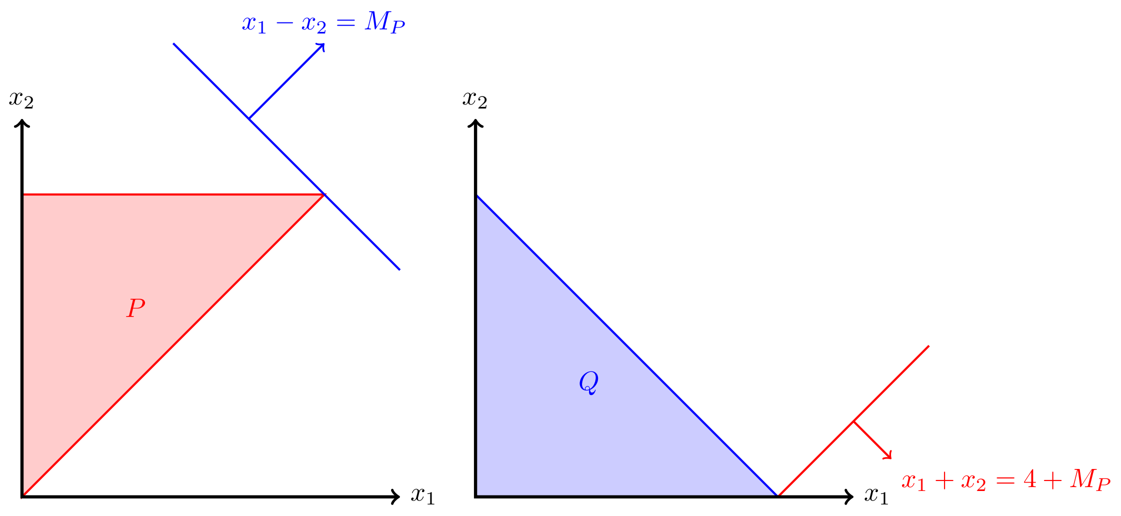


Figure 12.5: The convex regions with modified constraints.

12.2 Solving integer linear programming problems

As we have already said when introducing integer linear programming, if we solve a relaxed ILP programming problem (i.e. the problem without integrity constraints) and the solution $\bar{x}$ isn't integer (i.e. at least one of the coordinates isn't integer), it's not easy, given $\bar{x}$, to obtain the solution of the original problem. For this reason we should find an algorithm that can exploit the solving algorithm seen for linear programming problems. In particular we would like to solve the relaxed problem and always obtain an integer optimal solution so that the optimal solution of the relaxed problem corresponds to the optimal solution of the former problem.

To obtain such algorithm we have to remember that the optimal solution of a linear programming problem (either integer or not) is always on a vertex of the feasible region. The problem is that the feasible region F' of the relaxed ILP problem may not cross the points of the integer feasible region F. As we can see in Figure 12.6 the feasible region F' (in light blue) of the relaxed problem contains the feasible region F (the green points) of the former problem but it's not the minimum feasible region, in fact it's possible to shrink the feasible region F' and still it would contain F. In particular we would like to obtain a new feasible region $\bar{F}'$ that verifies all the conditions of F' and that is minimum for F (in the sense that $\bar{F}'$ is the smallest region that contains all the points of F).

12.2.1 Finding the convex hull

Convex hull

A feasible region $\bar{F}'$ is called convex hull. In a convex hull all vertices are integer, thus solving the linear programming problem considering the convex hull as feasible region (i.e. replacing the original constraints with the ones defined by the convex hull) we obtain an integer solution $\bar{x}$ that is also solution of the original integer linear programming problem.

Before understanding how to find the convex hull, we have to define some properties of a matrix A.

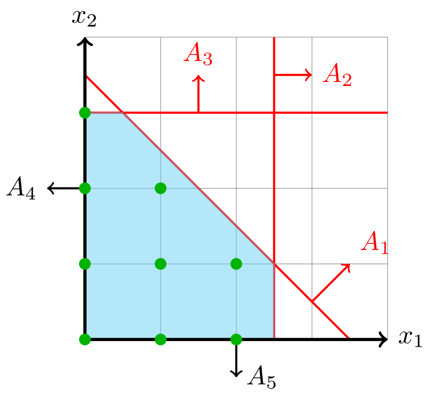


Figure 12.6: Integer linear feasible region.

Definition 23 (Unimodularity)**.** *A matrix A is unimodular if any non singular full rank submatrix has determinant 1 or -1.*

If the constraints A of an integer linear programming problem P are unimodular then they are the hull of P.

Definition 24 (Total unimodularity)**.** *A matrix A is totally unimodular if any non singular submatrix is unimodular.*

Proximity convex hull

Initially we said that we should find the convex hull but actually we can only consider the convex hull near the optimal solution. In other words we can consider the constraints near interesting points. Let us consider ILP problem P and its relaxation P' in 12.15.

$$P: \quad \begin{aligned} &\max cx \\ &\quad Ax \leq b \\ &\quad x \geq 0 \\ &\quad x \in \mathbb{Z}^n \end{aligned} \qquad\qquad P': \quad \begin{aligned} &\max cx \\ &\quad Ax \leq b \\ &\quad x \geq 0 \end{aligned} \tag{12.15}$$

Let us also call F and F' the feasibility region of P and P', respectively. Say x' is a feasible solution P'.

Given the feasible region F we can define a valid inequality as follows.

Definition 25 (Valid inequality)**.** *An inequality*

$$gx \leq \gamma$$

is valid if it holds for every point x in the feasible region on P (i.e. it holds $\forall x \in F$).

In other words a valid inequality is a new constraint that doesn't cut off any integer point of the feasible region F' (i.e. if an integer point was feasible in F', then it is also feasible when adding the new constraint).

Given a valid inequality

$$gx \leq \gamma \quad \forall x \in F$$

we can define a cut as

Definition 26 (Cut)**.** *A cut is a valid inequality that eliminates the optimal solution x' of P' from the feasible region F'. Formally a cut is a valid inequality*

$$gx' > \gamma$$

12.2.2 Solving algorithm

Thanks to the concept of cut we can keep shrinking the feasible region F' by eliminating the optimal solution x' until the optimal solution x' is integer. Algorithmically we can write this process as follows (call P' the relaxation of a ILP problem P)

1. Solve P'.
2. if the solution x' of P' is integer, return x'.
3. Add a cut to P'.
4. Repeat 1 to 4.

The same algorithm can be written using pseudocode as represented in Listing 12.1.

```
ILP_solver(P', x')
    optimal <- false
    while not optimal
        solve(P', x')
        if x' integer then
            optimal <- true
        else
            generate_cut(g, gamma)
            add(g, P')
```

Listing 12.1: The pseudocode of the algorithm to solve an integer linear programming problem.

12.2.3 Finding a cut

Finding a cut is the focal point of the algorithm for solving an ILP problem, thus it's important to describe how to find a cut, given a relaxed problem P'.

Chvátal valid inequalities

The Chvátal algorithm allows us to obtain a valid set of inequalities and it works as follows.

1. Given a set of valid inequalities (initially the constraints A of P'), the linear combination of such inequalities is a valid inequality. The newly generated inequality is valid but not a cut because it doesn't eliminate the optimal solution x'.
2. Given a valid inequality $gx \leq \gamma$ (obtained at the previous step) we can weaken it changing its coefficients g_i. In particular we can apply the floor function to each coefficient g_i.

$$\sum g_i x_i \leq \gamma \rightarrow \sum \lfloor g_i \rfloor x_i \leq \gamma$$

   If $\sum g_i x_i \leq \gamma$ is valid then also $\sum \lfloor g_i \rfloor x_i \leq \gamma$ is valid.

3. If we assume that $\sum \lfloor g_i \rfloor x_i \leq \gamma$ is valid, then each coefficient $\lfloor g_i \rfloor$ is integer, thus we can also replace γ with its floor to obtain

$$\sum \lfloor g_i \rfloor x_i \leq \gamma \rightarrow \sum \lfloor g_i \rfloor x_i \leq \lfloor \gamma \rfloor$$

The new inequality $\sum \lfloor g_i \rfloor x_i \leq \lfloor \gamma \rfloor$ is also valid.

Gomory cuts

The Chvátal algorithm allowed us to generate a valid inequality but, in order to solve an integer liner programming problem we need to generate cuts (until the solution of the relaxed problem is integer). Gomory's algorithm solves the problem of generating cuts.

Consider the problem 12.16 (where the min function can be replaced with the max function).

$$\begin{aligned} P : \min\ & cy \\ & Ay = b \\ & y \geq 0 \\ & y \in \mathbb{Z}^n \end{aligned} \tag{12.16}$$

Say $\bar{y}$ is a solution of P's relaxation (i.e. the problem without integrity constraints), then $\bar{y}$ can be non integer. Let us now consider a basis B for problem P so that

$$\bar{x}_B \geq 0 \qquad\qquad \bar{x}_N = 0$$

Let us remember that a basis B is the set of indices of variables so that $A_B x_B = b$ and N is the set of indices not in B (i.e. the indices for which $x_N = 0$). Also remember that if B is a basis we can find the optimal solution $\bar{x} = [\bar{x}_B\ \bar{x}_N]^T$ as $\bar{x}_B = A_B^{-1} b$, $x_N = 0$.

Now that we have a basis B, the constraints of P can be written as

$$A_B x_B + A_N x_N = b \tag{12.17}$$

Now we can isolate x_B on the left and multiply both members by A_B^{-1} to obtain

$$A_B x_B = b - A_N x_N \tag{12.18}$$

$$A_B^{-1} A_B x_B = A_B^{-1} b - A_B^{-1} A_N x_N \tag{12.19}$$

$$x_B = A_B^{-1} b - A_B^{-1} A_N x_N \tag{12.20}$$

Let us now call

- $\bar{b} = A_B^{-1} b$. Notice that A_B^{-1} is a $n_B \times n$ matrix and b is a $n \times 1$ column vector, thus $\bar{b}$ is a $n_B \times 1$ column vector (where n_B is the cardinality of the basis B).
- $\bar{A} = A_B^{-1} A_N$.

Variable x_B can be written as

$$x_B = \bar{b} - \bar{A} x_N \tag{12.21}$$

Now we are ready to generate Gomory cuts with the following three-steps process

1. Consider component h of x for which the optimal value is fractional (i.e. $x_h \in \mathbb{R}$). Let us associate component h to constraint t (say we associate the optimal value $\bar{x}_1 = 0.7$ of component x_1 to constraint $t = 1$). Given constraint t and component x_h we can build a valid constraint as the linear combination of constraints 12.21 (we have moved $\bar{A}x_N$ on the left side and taken only the index h in B and the row t of $\bar{A}$).

$$x_h + \sum_{i \in N} \bar{A}_{ti} x_i = \bar{b}_t = A_t^{-1} b = \bar{x}_h$$

Notice that this constraint is valid because it is a linear combination of the problem constraints (in particular $x_k : k \in B \wedge k \neq h$ has a weight of 0 and all the constraints not on row t have all weights to 0) and as we have seen in the first point of the Chávatal inequality generation algorithm, the linear combination of valid constraints is valid.

2. Since the constraint obtained at the previous point is valid we can restrict the constraint replacing $\bar{A}_{ti}$ with its floor value.

$$x_h + \sum_{i \in N} \lfloor \bar{A}_{ti} \rfloor x_i \leq \bar{b}_t$$

The new constraint is valid. Notice that we have transformed the equality in inequality because we have reduced the coefficients A_{ti}.

3. Since the constraint obtained at the previous point is valid we can further restrict it replacing $\bar{b}_i$ with its floor value.

$$x_h + \sum_{i \in N} \lfloor \bar{A}_{ti} \rfloor x_i \leq \lfloor \bar{b}_t \rfloor$$

The new constraint is valid and is a cut.

The process can be repeated for each constraint t (i.e. for each non-integer component x_h).

To prove that $x_h + \sum_{i \in N} \lfloor A_{ti} \rfloor x_i \leq \lfloor \bar{t}_i \rfloor$ is a cut we can consider a solution $[x_B = A_B^{-1} b, x_N = 0]$. In this case the inequality is reduced to

$$\bar{x}_h \leq \lfloor \bar{b}_t \rfloor$$

Since $\bar{x}_h$ is the optimal basic solution then we can write

$$\bar{x}_h = \bar{b}_t = A_{bt}^{-1} b_t$$

Since $\bar{x}_h$ is also fractional it is surely bigger than the floor of itself ($\bar{b}_t = \bar{h}_x$), thus the inequality $\bar{x}_h \leq \lfloor \bar{b}_t \rfloor$ doesn't hold and the factional optimal solution $\bar{x}_h$ has been excluded by the new constraint which for this reason is a cut.

12.3 Branch and bound

Since integer linear programming problems deal with a finite number of feasible solutions we can try to enumerate them and check which one maximise or minimise the objective function.

12.3.1 Naïve implementation

One efficient method to enumerate all possible solutions is the branch and bound algorithm. This algorithm builds a n-ary tree (where n is the number of possible values that x can take) and explores it using depth-first search. For simplicity let us consider binary trees, namely problems in which $x_i \in \{0, 1\}$. Practically the algorithm picks a variable x_i and tries to assign it a value (say 0), then picks another variable and assigns to it a value. The process is repeated until all variables have been assigned (i.e. when we have reached a leaf of the tree). Finally the value of the leaf is evaluated and replaced to the current best value if it can improve the current maximum (or minimum) value. The algorithm backtracks to the last assigned variable for which we haven't already tried all values and assigns another value.

To better understand the algorithm let us see it in action with a binary tree on two variables $x = \{x_1, x_2\}$ and the following problem.

$$\begin{aligned} \max\ & 2x_1 + x_2 \\ & 4x_1 + 2x_2 \leq 3 \end{aligned}$$

The binary tree is shown in Figure 12.7. The algorithm

1. Assigns $x_1 = 0$ and goes to n_1.
2. Assigns $x_2 = 0$ and goes to n_3.
3. Considers $\{0, 0\}$ the best solution $(3 \cdot 0 + 2 \cdot 0 \leq 3)$.
4. Backtracks to n_1 and assigns $x_2 = 1$ and goes to n_4.
5. Considers the solution $\{0, 1\}$ the new best solution because it's feasible $(3 \cdot 0 + 2 \cdot 1 \leq 3)$ and it improves the old best solution $(2 \cdot 0 + 1 \geq 2 \cdot 0 + 0)$.
6. Backtrack to n_0 because in n_1 all possible values have been assigned.
7. Assigns $x_1 = 1$ and goes to n_2.
8. Assigns $x_2 = 0$ and goes to n_5.
9. Considers $\{1, 0\}$ but doesn't set it as the new best solution because it breaks the constrain (i.e. it's not feasible).
10. Backtracks to n_3 and assigns $x_2 = 1$ and goes to n_6.
11. Considers $\{1, 1\}$ but doesn't set it as the new best solution because it breaks the constrain (i.e. it's not feasible).
12. Stops its execution since all leaves have been visited.

This algorithm doesn't take advantage of previous results and constrains. For instance in the example, when assigning $x_1 = 1$ we could have realised that it would have lead to unfeasible solutions (independently of the value of x_2) and stop the execution immediately.

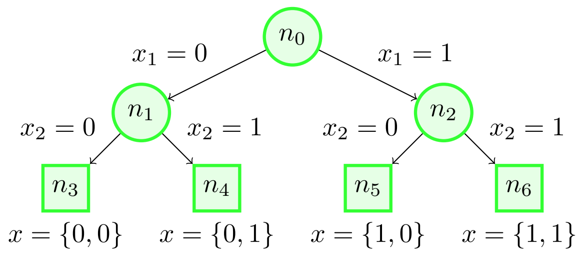


Figure 12.7: A binary search tree used in branch and bound

12.3.2 Removing unfeasible solutions

The first step for improving the branch and bound algorithm is to consider only feasible solutions, this means that when the algorithm assigns a value to a variable it checks the constraints and

- If the partial solution (i.e. the solution that considers only the assigned variables) already breaks the constraints (i.e. is not feasible) the algorithm has to backtrack without exploring the branch any further.
- Otherwise the algorithm continues normally.

In our example this method would have halted the algorithm immediately after assigning $x_1 = 1$ and saved ourselves 6 steps.

12.3.3 Upper bound

The algorithm ca be further improved adding an upper bound and pruning the results that are surely worst that the upper bound (in this case we have considered a max objective function, the same reasoning can be done in case of a min objective function but considering a lower bound).

The goal is to find (and update) an upper bound. To generate an upper bound we can consider a problem $P = \max\{c(x) : x \in F\}$ where F is the feasible region of the problem. P can be generated using an heuristic, in fact it doesn't have to be optimal.

Now we can consider a problem P'

$$P' : \max\{c'(x) : x \in F'\}$$

If

- The feasible region F is a subset of F'
$$F \subseteq F'$$
- The objective function $c'(x)$ is bigger or equal to $c(x)$ for each point x in the feasible region F
$$c'(x) \geq c(x) \quad \forall x \in F$$

then we call P' a relaxation of P and the following property holds.

$$\max\{c'(x) : x \in F'\} \geq \max\{c(x) : x \in F'\}$$

Let us now consider an optimal solution $\bar{x}$ of the relaxed problem P'. If the optimal solution $\bar{x}$ belongs to the feasible region F and $c'(\bar{x}) = c(\bar{x})$ then $\bar{x}$ is also optimal for P.

The problem now becomes how we generate the relaxation P'. In general we want to get rid of the constraints that complicate the problem so that the problem becomes efficiently solvable. Practically we want to build a feasible region F' that is as close as possible to F so that if $\bar{x}$ is optimal for F' then it's likely that $\bar{x}$ also belongs to F (thus being optimal for P).

There exist different methods to relax a problem, some examples are

- **LP relaxation** in which we relax integrity constraints.
- **Omission relaxation** in which some constraints are omitted.
- **Surrogate relaxation** in which a linear combination of constraints is used.
- **Lagrangean relaxation** in which some constraints in the objective function are penalised.

Another important thing to notice is that if we consider integer linear programming problems in which the variables can take any integer value (i.e. not only 0 or 1), we might need to branch multiple times on the same variable whilst if we consider binary integer linear programming problems (i.e. all variables can be only 0 or 1) we have to branch at most once for every variable.

Pseudocode The pseudocode of the full branch and bound algorithm is shown in Listing 12.2.

```
Procedure Branch&Bound(P, v)
    b <- upper_bound(P, x, unfeasible)
    if ( x is feasible for P ) then
        return(b)
    else if ( unfeasible ) then
        return(-INFINITY)
    else if ( b > v ) then
        Branch(P, x, P1, P2, ..., Pk)
        for i=1...k do
            t <- Branch&Bound(Pi,v)
            if ( t > v ) then
                v <- t
```

Listing 12.2: Branch and bound algorithm.

LP branch and bound

Let us now go more into the details of how branch and bound works when using an linear programming relaxation. Firstly we have to select a variable to branch. To choose the branching variable we can solve the linear programming relaxation until a non integer value is assigned to a variable x_i. The variable $x_i = f$ (where f is a non integer value) will be the branching variable and in particular

- The left branch will consider $x_i \leq \lfloor f \rfloor$.
- The right branch will consider $x_i \geq \lceil f \rceil$.

A graphical representation of branching for variable x_i is shown in figure 12.8.

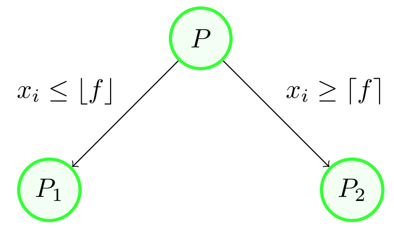


Figure 12.8: The part of a tree that represents the branching on variable x_i.

Example

To better understand how a problem is relaxed consider the following problem.

$$\begin{aligned} \max\ & 8x_1 + 10x_2 + 2x_3 + 13x_4 + 4x_5 - x_6 + 2x_7 \\ & 5x_1 + 7x_2 + 2x_3 + 9x_4 + 3x_5 + 3x_6 - 1x_7 \leq 10 \\ & x_i = \{0, 1\} \end{aligned} \tag{12.22}$$

This problem can be interpreted as revenue/cost optimisation. In particular x_i says if we have to produce a product i, the objective function tries to obtain the maximum revenue and the constraint imposes a maximum cost (the sum of the cost of each product has to be smaller than the total budget).

Variable x_7 has a negative contribute in the constraint, thus it reduces the cost. This means that we should always produce product 7 because it reduces the cost. In other words any solution with $x_7 = 1$ has a smaller cost than a solution with $x_7 = 0$. The problem can be updated as follows.

$$\begin{aligned} \max\ & 8x_1 + 10x_2 + 2x_3 + 13x_4 + 4x_5 - x_6 \\ & 5x_1 + 7x_2 + 2x_3 + 9x_4 + 3x_5 + 3x_6 - 1 \leq 10 \\ & x_i = \{0, 1\} \end{aligned} \tag{12.23}$$

For the opposite reason variable x_6 should be 0, in fact it reduces the revenue because it is negative in the objective function. The problem can be rewritten as follows.

$$\begin{aligned} \max\ & 8x_1 + 10x_2 + 2x_3 + 13x_4 + 4x_5 \\ & 5x_1 + 7x_2 + 2x_3 + 9x_4 + 3x_5 \leq 11 \\ & x_i = \{0, 1\} \end{aligned} \tag{12.24}$$

Now that we have consider all trivial assignments we have to compute a solution $\max\{c(x) : x \in F\}$ using an heuristic. In this case we can consider the ratio between the value b and the cost c and order x_i in increasing order of ratio. Then we should consider the products in order of ratio and assign $x_i = 1$ until the constraints are not satisfied. The sorted products are ($\xi_i = b_i/c_i$)

1. $\xi_1 = \frac{8}{5} = 1.6$
2. $\xi_4 = \frac{13}{9} = 1.44$
3. $\xi_2 = \frac{10}{7} = 1.42$
4. $\xi_5 = \frac{4}{3} = 1.33$

5. $\xi_3 = \frac{2}{2} = 1$

Given this order we have to assign 1 to x_1, x_5 and x_3 (because x_5 and x_3 are the only variables that can still be taken after x_1 without breaking the constraint). This heuristic has returned a feasible solution (let us call it **incumbent solution**).

$$x = [1, 0, 1, 0, 1]$$

The cost $c(x)$ if this solution is

$$c(x) = 8 + 2 + 4 = 14$$

This means that the algorithm should find feasible solutions bigger than 14 (otherwise it would be useless to have a solution smaller than 14). Knowing that the upper bound $c(x)$ is 14 we can start the branch and bound algorithm.

Root To find on which variable to branch in the root we can use the values ξ_i obtained when searching an incumbent solution and assign values in that order. In particular

1. We assign $x_1 = 1$. The new budget after the assignment is 6 (i.e. the old budget minus the cost of product 1).
2. We try to assign $x_4 = 1$ but we realise that this solution is not feasible, thus we have to assign the biggest fractional value possible $x_4 = \frac{6}{9}$.

Since x_4 is fractional we should branch on x_4.

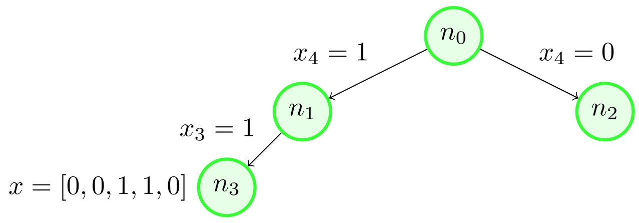


Figure 12.9: The partial binary search tree for branch and bound.

First level From the root node we branch on variable x_4. If we assign $x_4 = 1$ we obtain problem P_1

$$\begin{aligned} 13 + \max\ & 8x_1 + 10x_2 + 2x_3 + 4x_5 \\ & 5x_1 + 7x_2 + 2x_3 + 3x_5 \leq 2 \end{aligned} \tag{12.25}$$

After $x_4 = 1$ the only assignment that generates a feasible solution (and increases the total value) is $x_3 = 1$. Since the value $c(x) = 15$ of the solution $x = [0, 0, 1, 1, 0]$ is greater of the incumbent solution, then $c(x) = 15$ is the new incumbent solution.

If we assign $x_4 = 0$ we obtain problem P_2

$$\begin{aligned} \max\ & 8x_1 + 10x_2 + 2x_3 + 4x_5 \\ & 5x_1 + 7x_2 + 2x_3 + 3x_5 \leq 11 \end{aligned} \tag{12.26}$$

In this case we can assign multiple variables thus we have to branch. To decide which variable to branch we can use the heuristic used to find the upper bound (i.e. $\xi_i = b_i/c_i$) so we

1. Assign $x_1 = 1$. The new budget is 6 because we remove 5 (i.e. the cost of product 1) from the budget.
2. Try to assign $x_2 = 1$ (because x_4 is already assigned) but this assignment breaks the constraint, thus we have to assign $x_2 = \frac{6}{7}$ (i.e. the smallest value that doesn't breaks the constraints).

Since x_2 is non integer we have to branch on x_2.

Second level In the second level we branch on x_2. If we assign $x_2 = 1$ we obtain problem P_3

$$\begin{aligned} 10 + \max\ & 8x_1 + 2x_3 + 4x_5 \\ & 5x_1 + 2x_3 + 3x_5 \leq 4 \end{aligned} \tag{12.27}$$

After assigning $x_2 = 1$ we can either assign $x_3 = 1$ or $x_5 = 1$. In both cases the total value is less than the upper bound (i.e. 15) thus we should stop here because there is no way to improve our best result yet.

The algorithm continues as shown before until no feasible improving solution can be generated.

Part V

Heuristic approaches

Chapter 13

Heuristics

13.1 Introduction

Until now we have considered algorithms that allowed us to solve a problem finding the optimal solution. In practical situation such algorithms may not be fast enough to be executed in a reasonable time (at least in a reasonable time for a real life application). To solve this problem we can apply heuristics that can speed up execution and still return a good result that may not be the optimal one (but that is pretty close to the optimal result).

13.1.1 A general structure

An important property of the heuristics we are going to describe is that they should have a fixed structure that can be applied to all types of problems simply changing some parameters or some functions. In this way the algorithm has to be written only once and the developers have to define only some variables or function to adapt the general framework to a specific problem. In particular there exists two classes of general frameworks

- **Constructive algorithms**. Constructive algorithms start from an empty solution and try to add some new elements that keep the solution feasible until no more element can be added. The elements to add are chosen via some criteria defined by the developer. Greedy algorithms are constructive algorithms.
- **Improvement algorithms**. Improving algorithms start from a feasible (possibly not optimal) solution and try to improve it until it's not possible to obtain a better feasible solution. Local search is part of improving algorithms. An example of improving algorithm is the one used to solve linear programming problems.

13.1.2 Easy problems

A problem can be easy or difficult. These concept are very abstract, thus we should find a more formal definition. In particular we say that

- A problem is **easy** if there exist an algorithm that can solve it in **polynomial time**.
- A problem is **hard** if the best algorithm that can solve it need **non polynomial time**.

This definition still has some flaws, in fact if we consider a problem for which we know only non-polynomial algorithms, we can't be sure that such algorithm is the best one (maybe a polynomial algorithm hasn't been discovered yet). To solve this problem we have to rely on computational theory, in particular we can try to reduce the problem to another problem for which it's well known that the best algorithm is non-polynomial.

13.1.3 Measuring approximation

Another interesting aspect of heuristics is measuring how far the approximation is from the optimal solution. Let us consider a problem

$$P = \min\{c(x) : x \in F\}$$

To measure the distance between the approximated solution z_A obtained with an heuristic A and the optimal solution z_O we can use

- The **absolute error** that measures the difference between the approximated value and the optimal value.
$$\varepsilon_{absolute} = z_A - z_O$$
- The **relative error** that scales the absolute error by the optimal value
$$\varepsilon_{rel} = \frac{z_A - z_O}{z_O} = \frac{\varepsilon_{absolute}}{z_O}$$

One might argue that in order to compute the error we need to know the optimal result, thus it's useless to compute the approximation because we already have the optimal value. To solve this issue we can replace z_O with

- The **lower bound** of the optimal solution.
- The **optimal solution of the relaxation**.

13.2 Constructive algorithms

Constructive algorithms offer a fairly simple framework that requires the developer to define a two sets and two functions. In particular the general algorithm

1. Needs the building blocks of the solution.
2. Begins with an empty solution.
3. Selects the best building block to add to the solution and removes it from the pool of building blocks.
4. Adds the block selected at the previous point to the solution if the resulting solution is feasible.
5. Repeats 3-5 until no block can be selected.

As we can see from this description, the general algorithm is independent from the problem and the only things that depend from the specific problem are

- The solution's **building blocks** E. That is the atomic elements of which the solution is made of.
- The family F of **feasible subsets** of E.
- A `Best(S)` function that given the set of building blocks S returns the best block.
- An `Ind(X, e)` function that given a partial solution $X \subseteq F$ and a selected building block $e \in E$ returns true if $X \cup \{e\}$ (i.e. the new solution) is feasible.

Tie breaking If there are multiple nodes with the same best value then we should define a tie breaker or we can choose at random one of the best values. Selecting a value at random has many advantages, in fact we can repeat the algorithm many times (since the heuristic is fast) and obtain different values. After many iteration we can select the best value.

13.2.1 Pseudocode

The algorithm can be written in pseudocode as in Listing 13.1.

```
Constructive_heuristic(E, F):
    X <- emptyset # the partial solution, initially empty
    S <- E;       # the pool of available building blocks
    repeat
        e <- Best(S)
        S <- S - {e}
        if Ind(X, e)
            then X <- X + {e}
    until X is empty
```

Listing 13.1: The general constructive heuristic algorithm

13.2.2 Example

We have already seen some problems can be solved using this approach, for instance the knapsack problem or the investing problem. To better understand how does the constructive heuristic work we can try to solve the minimum independent set problem. Let us consider a undirected graph $G = (N, A)$. We want to find a subset of nodes $X \subseteq N$ so that each couple of nodes i, j in X is not connected by an arc. If no couple of arcs in X is connected then X is said to be **independent**. Formally $X \subseteq N$ is independent if

$$\forall i, j \in X, (i, j) \notin A$$

Let us try to solve the minimum independent set problem using graph 13.1. To solve this problem using the constructive algorithm we have to define E, F, `Best` and `Ind`, in particular

- E is the set of nodes N.
- F is the subset of nodes so that for each couple of nodes in F it doesn't exist a connecting arc. More formally
$$F = \{R \subseteq N : \forall i, j \in R, \nexists (i, j) \in A\}$$
- The function `Best` returns the node that has the minimum number of incident arcs (i.e. outgoing or incoming arcs). Formally

```
Best(S) = argmin{$|S_i| : i \in S$}
```

- The function `Ind(X, e)` returns true if the new element e isn't connected to any arc in X. Formally

```
Ind(X, e) = if $(e, j) \notin A \quad \forall j \in X$
```

Using this algorithm we can obtain a solution as follows

1. Node f is selected since it has the minimum number of incident arcs (the tie with c is broken considering nodes in reverse alphabetical order).
   (a) The new candidate solution $X \cup \{f\}$ is feasible, thus f is added to X ($X = \{f\}$).
   (b) Node f is removed from the pool of nodes ($S = S - \{f\}$).
2. The incident arcs are computed again (because f has been removed from S).
3. After computing the new labels, c is selected because it has the minimum number of incident arcs.
   (a) The new candidate solution $X \cup \{c\}$ is feasible because f and c aren't connected, thus c is added to X ($X = \{f, c\}$).
   (b) Node c is removed from the pool of nodes ($S = S - \{c\}$).
4. The incident arcs are computed again (because c has been removed from S).
5. After computing the new labels, a is selected because it has the minimum number of incident arcs.
   (a) The new candidate solution $X \cup \{a\}$ is feasible because it isn't connected neither with f nor with c, thus a is added to X ($X = \{f, c, a\}$).
   (b) Node a is removed from the pool of nodes ($S = S - \{a\}$).
6. The algorithm is repeated but every node makes the solution unfeasible, thus $X = \{f, c, a\}$ is the approximate solution obtained with the constructive algorithm.

13.3 Improvement algorithms

Improvement algorithms (also local search algorithms) start from a feasible solution and try to improve it. A feasible solution can be obtained using whatever method. In other words we want to explore the solution space by making small changes (called **moves**) to the initial solution until no improvement can be found. When no improvement is possible we have found a solution called **local optimum**.

Given the informal description of the algorithm we can define the building blocks needed to define the improvement algorithm:

- A feasible solution x.

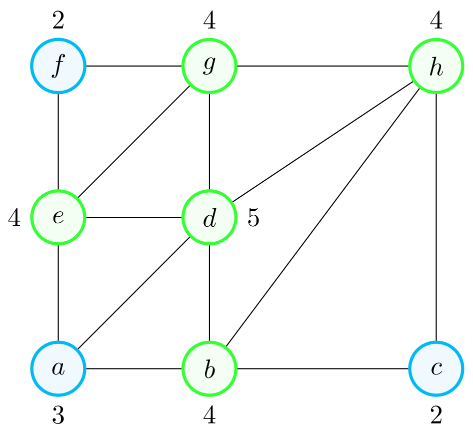


Figure 13.1: The graph used for the minimum independent set problem.

- A function $N(x)$ (neighbourhood of x) that given a feasible solution x returns the subset of solutions (feasible or not) that are close to x.
- A function $\sigma(x)$ that returns the best feasible solution (i.e. a close solution that improves x) of the neighbourhood of x (this means that σ uses N). If no feasible solution is found, $\sigma(x)$ returns x. The σ function can also be defined so that it returns the first improvement found in the neighbourhood (even if it's not the best of the neighbourhood) or a random solution among the improvements of the neighbourhood.

13.3.1 Algorithm

Given the building blocks defined before we can define the general improving algorithm as follows

1. Start from a feasible solution x.
2. Improve solution x using $\sigma(x)$.
3. If the improved solution is still x, end the algorithm and return x as the local optimum.
4. Otherwise replace x with $\sigma(x)$.
5. Repeat from step 2.

The pseudocode for the improving algorithm is shown in Listing 13.2.

```
Local_Search(x, N):
    while not (σ(x) = x):
        x <- σ(x)

    return x
```

Listing 13.2: Improving algorithm.

13.3.2 Tradeoff

This family of algorithm offers an interesting tradeoff, in fact we can define

- A big neighbourhood that allows to make big steps and that reduces the number of steps. The downside is that the function σ takes a lot of time to compute because the search space is bigger.
- A small neighbourhood that allows to compute σ swiftly but that forces the algorithm to make small steps thus increasing the number of iteration.

13.3.3 Neighbourhood function

To allow the algorithm to stop N should satisfy the following conditions

- x must belong to $N(x)$, otherwise the condition $x = \sigma(x)$ is never true and the while loop keeps running forever.
- The search space must be connected. This means that is must always be possible to find a sequence of neighbourhoods that connects any two feasible solutions x and y. In some cases this property can't be satisfied (because the constraints are too tight), thus we have relax the problem (and penalise the constraints in the objective function) so that this property is satisfied.

Definitions