Université catholique de Louvain

LINMA2450 Combinatorial Optimization

Partial work-in-progress notes

Thibault Sottiaux, Yu-Lan Scholliers, Raphaël Peschi, Thomas Walkiers, Adrien Thonet, Cassandra de Cock, Thomas Gillis, François-Xavier Arnotte, Maxime Raskin, Damien Scieur, Mathieu Descamps, Patrick Lambert, Arnaud Marenne, Quentin Vanderlinden, Jacques Cartuyvels, Jean-Charles Delvenne and Julien Hendrickx

December 10, 2020

Contents

Chapter 1

Introduction to Integer programming

This course deals with optimisation problems over discrete, often finite but very large sets. The unifying thread to all or most topics of this course will be the formulation of those problems as *integer programmes* (IP). Let us first look at some common problems.

1.1 Examples

The knapsack problem Given n objects of volume $v_1, \dots, v_n$ and value $a_1, \dots, a_n$ and a bag of total volume V, one wants to pack as much value as possible into the bag, which we formulate as

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} a_i x_i \\ & \sum_i v_i x_i \leq V \\ & x_i \in \{0,1\}, \end{aligned}$$

where x_i encodes the decisions: $x_i = 1$ if we take object i into the bag, $x_i = 0$ otherwise.

The assignment problem, or maximum weight matching We look for a matching between people and jobs resulting in the highest overall gain:

- m jobs, n people
- gain of person i on job j is c_{ij}

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij} \\ & \sum_i x_{ij} \leq 1 \text{ (at most 1 person on a job)} \\ & \sum_j x_{ij} \leq 1 \text{ (at most 1 job per person)} \\ & x_{ij} \in \{0,1\} \end{aligned}$$

Here $x_{ij} = 1$ iff we assign person i to job j.

TSP: Travelling salesperson problem A salesperson has to visit a list of cities once and only once during the day, and wants to minimise the total travelled length.

In graph-theoretic terms, we need to find a minimum weight Hamiltonian cycle on a weighted graph. Let us make a first attempt:

- Let $V = 1, \dots, n$ denote the set of n cities.
- Let $x_{ij} = 1$ if the travelling salesman uses edge (i, j) and 0 otherwise.

- Let c_{ij} be the cost (e.g. travelling time, distance, ...) to go from city i to j.

$$\begin{aligned} \min \quad & \sum_{i,j=1}^{n} c_{ij}x_{ij} \\ & \sum_{i;i\neq j} x_{ij} = 1 \quad \forall j = 1,\dots,n \\ & \sum_{j;j\neq i} x_{ij} = 1 \quad \forall i = 1,\dots,n \\ & x_{ij} \in \{0,1\} \end{aligned}$$

These expresses that in the subgraph of selected edges (edges ij s.t. $x_{ij} = 1$), every city has outdegree 1 and indegree 1. These constraints are not enough however, because we could end up with a union of disconnected cycles. We need to add new constraints to require that all cities are connected by a single cycle. For this we have two equivalent possibilites:

- The *connectedness constraints* expresses that for every possible group of cities S, there is always at least an edge leaving it.

  $$\forall \emptyset \subsetneq S \subsetneq V : \quad \textstyle\sum_{i\in S}\sum_{j\notin S} x_{ij} \geq 1 \quad (1 \leq |S| \leq n-1)$$

  This creates $2^n - 2$, i.e. exponentially many constraints, which is of course impractical to write down explicitly.

- Instead of the connectedness constraint, we can use the *subtour elimination constraint*, forbidding a cycle of size less than n:

  $$\forall \emptyset \subsetneq S \subsetneq V : \quad \textstyle\sum_{i,j\in S} x_{ij} \leq |S| - 1 \quad (2 \leq |S| \leq n-1)$$

  We also end up with $2^n - 2$ constraints.

A **general form** of an integer programme (or IP) is

$$\begin{aligned} \max \quad & cx \\ & Ax \leq b \\ & x \in \mathbb{Z}^n \end{aligned}$$

When some variables are reals and others are integers, we call it a **mixed integer program** (MIP).

1.2 Algorithmics and good formulations

A naive approach to solve an IP is **brute force**, which consists in evaluating the objective function for all the **feasible points**, i.e., elements in the **feasible set** $\{x : Ax \leq b, x \in \mathbb{Z}^n\}$ (usually a finite set), before picking the best value. What is the time complexity of this approach for the problems above?

- Knapsack problem : 2^n cases must be examined in the worst case. Remember that 2^n is the number of subsets of n-element set: to constitute a subset one has to make n binary decisions.

- Assignment problem: (if $m = n$, same number of jobs and people) $n!$, which is the number of bijections an n-element set to another n-element set, or equivalently the number of permutations of an n-element set. Indeed, we need to choose a job among n jobs for the first worker, then among the $(n-1)$ remaining jobs for the second worker, etc.

- TSP: $(n-1)!$ possible orders of visit for the cities: the first city is arbitrary, then $n-1$ choices for the second one, then $n-2$ choices, etc.

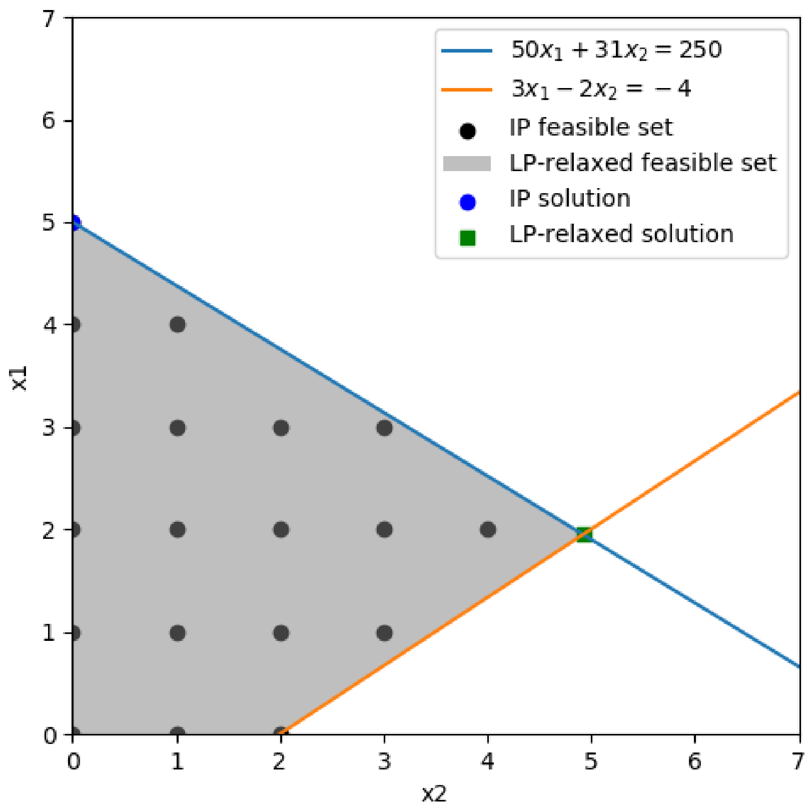


Figure 1.1: An example where the IP and LP relaxation optimal solutions are far away.

This approach takes an exponential time with respect to n, or even more, and isn't practical except for small instances. For example if evaluating one feasible point takes one nanosecond, brute force on a 50-object knapsack problem will be solved in ten days, and 60 objects will require a thousand times more. In the same condition a 20-city TSP will require four years of calculation.

Another idea that springs to mind to solve an IP is to round off the solution offered by the LP relaxation. We replace $(x \in \mathbb{Z}^n)$ by $(x \in \mathbb{R}^n)$ to obtain a linear programme (**LP relaxation**), which we can solve with multiple well-known algorithms such as the simplex algorithm, or interior points methods. Afterwards, we still need to round up/down the obtained solution to make it a feasible one under the initial constraints.

But as we could expect, this approach doesn't work in general:

(a) If $x \in \{0, 1\}$ is replaced by $0 \leq x \leq 1$ and the LP-relaxed optimal solution is $x_i = 0.5$, what should one do?

(b) Consider the following IP:

$$\begin{aligned} \max \quad & 100x_1 + 64x_2 \\ & 50x_1 + 31x_2 \leq 250 \\ & 3x_1 - 2x_2 \geq -4 \\ \\ & x_1, x_2 \geq 0 \\ & x \in \mathbb{Z}^2 \end{aligned}$$

When solving the IP and LP-relaxed problems (which we can do graphically given the two-dimensional nature of the problem), we get optimal solutions that lie at different, distant of the feasible set, see Fig **??**.

How to improve the relaxation? The LP relaxation is nevertheless useful as it provides a bound on the IP optimal value: $\max\{cx : Ax \le b, x \in \mathbb{Z}^n\} \le \max\{cx : Ax \le b, x \in \mathbb{R}^n\}$. This bound however can be more or less tight.

Let us take a look at the following equivalent formulations of the same problem. We assume that $P_3 \subset P_2 \subset P_1$ are three polyhedra (thus defined by linear inequalities in the reals) such that $P_3 \cap \mathbb{Z}^n = P_2 \cap \mathbb{Z}^n = P_1 \cap \mathbb{Z}^n$.

$$\begin{array}{lclcl} \max \; cx & = & \max \; cx & = & \max \; cx \\ \quad x \in P_1 & & \quad x \in P_2 & & \quad x \in P_3 \\ \quad x \in \mathbb{Z}^n & & \quad x \in \mathbb{Z}^n & & \quad x \in \mathbb{Z}^n \end{array}$$

Despite the fact that they are three formulations of the same IP problem (in particular with the same feasible set in the integers), when LP-relaxed, we obtain 3 different linear programmes with potentially different solutions.

$$\begin{array}{lclclcl} \max \; cx & \ge & \max \; cx & \ge & \max \; cx & \ge & \max \; cx \\ \quad x \in P_1 & & \quad x \in P_2 & & \quad x \in P_3 & & \quad x \in P_3 \cap \mathbb{Z}^n \end{array}$$

We say that P_2 is a **better formulation** than P_1, and P_3 is the best of the three.

Moreover, if it so happens that P_3 is the convex hull of the feasible set $P_3 \cap \mathbb{Z}^n$ (thus the vertices of P_3 have integral coordinates) then is is the **ideal** formulation, and the LP relaxation in this case is exact: there exists an optimal solution of the relaxed LP that has integral coordinates, thus is also an optimal solution for the IP.

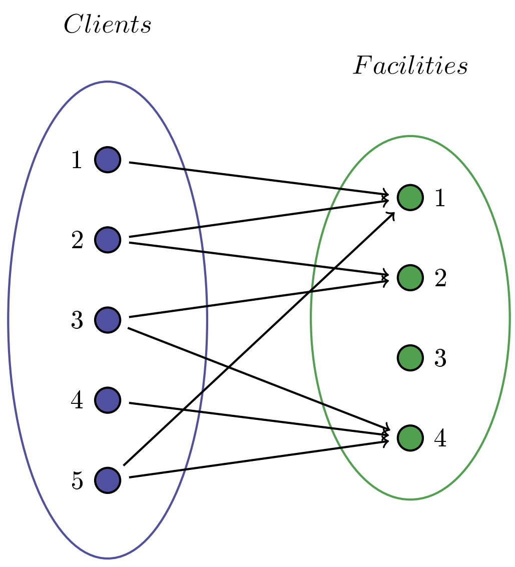
Conclusion Adding more constraints, while keeping the IP feasible set unchanged, can only improve the formulation. If we are able to find an explicit description of the convex hull of the IP feasible set then we can write the ideal formulation: LP relaxation gives the exact optimal solution.

No algorithm is known that solves all IPs in polynomial time. In this course we will focus on:

- finding a good formulation;
- finding upper/lower bounds on the value of the objective function;
- identifying some classes of problems that are easy to solve;
- describing methods and heuristics that work 'sometimes' well.

1.3 Examples of improved formulations

Uncapacitated facility location problem



- m clients and n facilities;
- x_{ij} fraction of demand of client i to facility j;
- c_j fixed cost of facility j, if open;
- v_{ij} variable cost of demand fraction x_{ij};
- $y_j = 1$ when the facility is open, 0 otherwise.

The problem is to decide whether or not to open a facility and how to dispatch the client's demands between the open facilities, so as to minimize the total cost. Here is a first possible formulation:

$$\min \quad \sum_j c_j y_j + \sum_{i,j} x_{ij} v_{ij} \tag{1.1}$$

$$\sum_j x_{ij} = 1 \quad (\forall i) \tag{1.2}$$

$$\sum_i x_{ij} \leq M y_j \quad (\forall i, \text{ for M large}) \tag{1.3}$$

$$x_{ij} \geq 0 \quad (\forall i, j) \tag{1.4}$$

$$0 \leq y_j \leq 1 \quad (\forall j) \tag{1.5}$$

$$y_j \in \mathbb{Z} \tag{1.6}$$

The constraint (1.3) encodes the fact that whenever is facility is closed ($y_j = 0$), then the total demand to that facility ($\sum_i x_{ij}$) is zero. All large enough values of M lead to the same formulation, ie same IP feasible set. Among those possible values, a smaller value corresponds to a smaller polyhedron for the LP feasible set, i.e. a better formulation. A good and simple choice for M is therefore the number of clients.

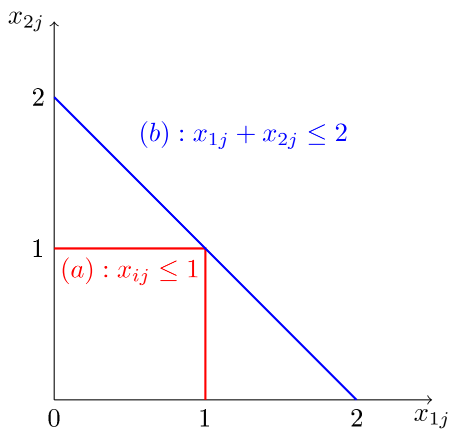
However we can improve further the formulation if we replace the single constraints (1.3) with the set of constraints

$$x_{ij} \leq y_i \quad \forall i, j \tag{1.7}$$

This is an equivalent formulation, as it conveys the fact that whenever a facility is closed, every individual demand x_{ij} is zero: the IP feasible set is not modified.

To show that (1.7) makes a strictly better formulation, one has to observe that if a point in the reals satisfies (1.1,1.2,1.4,1.5,1.7) then it also satisfies (1.3), the converse not being true.

It is easy to see graphically that (1.7) leads to a better formulation. Here is an example with two facilities $y_1 = 1$ and $y_2 = 1$:



Lot sizing

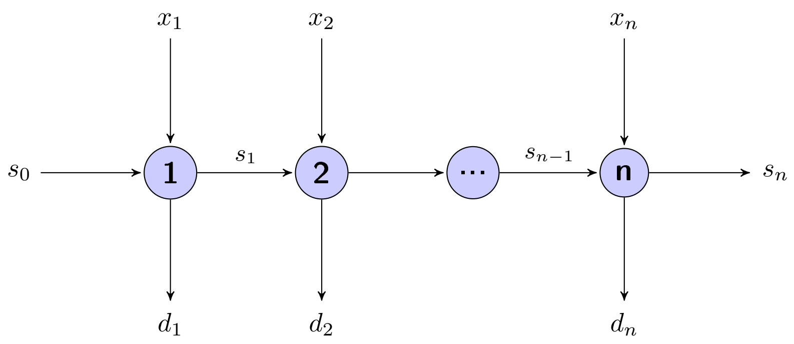


Figure 1.2: A state diagram per day of the lot sizing problem, with s_t the storage and x_t the production at time t.

- n : number of days of production
- f_t : fixed cost of production at time t.
- v_t : variable cost of one unit produced at time t.
- h_t : cost of storing one unit at time t.
- d_t : demand at time t.

The problem is to decide whether to produce at time t (binary variable y_t) and how much (real variable x_t) for a horizon of n days. Let us start with the most straightforward formulation:

First formulation

$$\begin{aligned}
\min \quad & \sum_t f_t y_t + v_t x_t + h_t s_t \\
& s_{t-1} + x_t = d_t + s_t \quad \text{(flow)} \\
& x_t \leq M y_t \quad \text{(M large enough, e.g: } M = \sum_t d_t) \\
& s_t, x_t \geq 0 \\
& y_t = \{0, 1\}
\end{aligned}$$

Note that this is a mixed integer programme (MIP), as some of the variables are allowed to be continuous, and some are requested to take discrete values.

An extended formulation We can improve the formulation with more constraints *and* more variables. Indeed let us add the variables

- x_{rt} : production at time r to satisfy demand at time t, defined for all $t \geq r$.

which we introduce into the constraints in the following way:

$$\begin{aligned}
\min \quad & \sum_t f_t y_t + v_t x_t + h_t s_t \\
& s_{t-1} + x_t = d_t + s_t \quad \text{(flow)} \\
& x_r = \sum_{t \geq r} x_{rt} \quad \forall r \\
& \sum_{r \leq t} x_{rt} = d_t \forall t \\
& x_t \leq M y_t \forall t (M = \sum_t d_t) \\
& s_t, x_t \geq 0 \\
& 0 \leq y_t \leq 1 \\
& y_t \in \mathbb{Z}
\end{aligned}$$

This formulation is called an *extended* formulation, because it introduces new variables, which are not present in the objective function, but only in the constraints. It means that the polyhedron P_{ext} defined by the linear constraints sits in a higher dimension space than P, the polyhedron defined by the linear constraints of the first formulation. Given that they do not live in the same space, how can we compare P_{ext} and P? Given a vector in P_{ext}, one may simply 'erase' the values of the new variables x_{rt} in the vector. We observe that the resulting low-dimensional vector trivially satisfies all constraints of the first formulation. This is expressed geometrically as

$$\text{Proj}_{(x_{rt})} P_{\text{ext}} \subseteq P$$

where $\text{Proj}_{(x_{rt})}$ is the projection operator that erases the (x_{rt}) values from the vector. In fact it is easy to see that each point in P can be lifted to a point of P_{ext}, by choosing (in a non-unique way) values of x_{rt} compatible with the (x_t) and the constraints of the extended formulation. This means geometrically that

$$\text{Proj}_{(x_{rt})} P_{\text{ext}} = P$$

Thus it seems this extended formulation is not better than the first formulation, as clearly the LP relaxations of these formulations will bring identical optimal values. Yet, we are now in position to improve this extended formulation.

An improved extended formulation The fact that we cannot produce at time t if the factory is closed at that time has been encoded with $x_t \leq M y_t$. We can also express this with the new variables as $x_{rt} \leq M_{rt} y_t$ for all r, with M_{rt} large enough, e.g. $M_{rt} = d_t$.

$$\begin{aligned}
\min \quad & \sum_t f_t y_t + v_t x_t + h_t s_t \\
& s_{t-1} + x_t = d_t + s_t \quad \text{(flow)} \\
& x_r = \sum_{t \geq r} x_{rt} \quad \forall r \\
& \sum_{r \leq t} x_{rt} = d_t \forall t \\
& x_{rt} \leq M_{rt} y_t \\
& s_t, x_t \geq 0 \\
& 0 \leq y_t \leq 1 \\
& y_t \in \mathbb{Z}
\end{aligned}$$

Clearly, this defines the same MIP feasible set as the extended formulation above. This also improves formulation as if a point satisfies $x_{rt} \leq d_t y_t$ for all r then it satisfies $x_t \leq M y_t$ (by summation). Thus the polyhedron $P_{\text{ext,impr}}$ defined by the LP relaxation of this formulation is included in P_{ext}.

Consequently, the projection sends this polyhedron into P:

$$\text{Proj}_{(x_{rt})} P_{\text{ext,impr}} \subseteq P$$

It so happens that this improved formulation is in fact ideal, in that the vertices of $P_{\text{ext,impr}}$ assign binary values to each y_t, which we accept here. One can thus solve the LP relaxation and deduce the optimal solution to the lot-sizing problem. Let us mention that other even more efficient algorithms exist for the lot-sizing problem.

Remark The version of the lot-sizing problem we see in this section is called the uncapacitated lot-sizing problem, because in each day one may produce as much as needed for the demand. One may also introduce capacity constraints for each day, with $x_t \leq C_t y_t$, where ffor some capacities C_t. One may also introduce multi-item versions, where several productions take place to satisfy demand for several products. For these more complex versions, the ideas above do not in general lead to an ideal formulation, but to an improved formulation.

Chapter 2

Upper and lower bounds

Consider a problem $z = \max\{cx : x \in P \cap \mathbb{Z}^n\}$ (IP) where we cannot compute z exactly. One strategy it to try and compute bounds on z: $z_L \leq z \leq z_U$. It would be even better if we can find a sequence of increasingly tighter bounds: $z_{L_1} \leq z_{L_2} \leq \ldots \leq z \leq \ldots \leq z_{U_2} \leq z_{U_1}$, with

$$\begin{cases} z_{L_i} \to z \\ z_{U_i} \to z \end{cases}$$

The ideal scenario is to find $z_{L_i} = z_{U_i}$ for the same i.

2.1 How to find primal bounds ?

Lower bounds for a maximisation problem, or upper bounds for a minimisation problems, are called **primal bounds**. The only known way to find a primal bound is to find a feasible point x, leading to $z_L = cx \leq z$. We therefore need to find 'good' feasible points, for which many different heuristics exist. Let us see two of them.

2.1.1 Greedy algorithms

Greedy algorithms build up a solution by taking a series of decisions that gives the best immediate increase of the objective function's value. But of course, this could lead to non-optimal solutions.

Example: *Knapsack problem*

$$\begin{aligned} \max \quad & 12x_1 + 8x_2 + 17x_3 + 11x_4 + 6x_5 + 2x_6 + 2x_7 \\ & 4x_1 + 3x_2 + 7x_3 + 5x_4 + 3x_5 + 2x_6 + 3x_7 \leq 9 \\ & x_i \in \{0, 1\} \end{aligned}$$

Take the object with highest ratio $= \frac{\text{value}}{\text{volume}}$ among those that still fits in the bag.

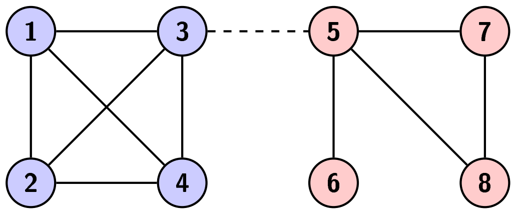
$$\begin{cases} x_1 = 1 \\ x_2 = 1 \\ x_6 = 1 \end{cases} \quad \to \text{Total value: } 22$$

2.1.2 Local search algorithms

1. Start with an initial feasible point
2. Look at (some or all) feasible points in the 'neighbourhood', given some 'neighbourhood'-relationship.
3. Pick the best 'neighbour'
4. Start again until no more improvement is possible

One may see it as the discrete analog of gradient descent. It can analogously get trapped into a local optimum (but there are many strategies to try and get around this).

Example: *Graph equipartition problem*



Find the minimum cut (number of edges) between two sets composed of $\frac{n}{2}$ nodes.
e.g.: sharing tasks between processors.

1. Start from a partition $\frac{n}{2} + \frac{n}{2}$.
2. 'Neighbours' obtained by exchanging two nodes. $\rightarrow \frac{n^2}{4}$ neighbours (polynomial, affordable to look at all of them).
3. Select the pair of nodes that reduces the value of the objective function the most.

2.2 How to find dual bounds ?

Upper bounds for a maximisation problems, or lower bounds for a minimisation problem, are called **dual** bounds. Several techniques for dual bounds will be covered in this course:

- Relaxation
- Lagrangian relaxation
- Dual formulation of the problem

We only cover relaxation and duality in this chapter, as Lagrangian relaxation is the object of a later chapter.

2.2.1 Relaxation

Relaxation consists in increasing the feasible set, usually by removing a constraint:

$$\begin{aligned} \max\{cx : x \in S\} &\leq \max\{cx : x \in T\} \quad \text{if } S \subseteq T \\ z &\leq z_U \end{aligned}$$

More specifically, we have seen LP relaxation, obtained in relaxing the integrality constraints:

$$\begin{aligned} \max\{cx : x \in P \cap \mathbb{Z}^n\} &\leq \max\{cx : x \in P\} \quad \text{P polyhedron} \\ z &\leq z_{LP} \end{aligned}$$

Remark As we have seen, better formulations lead to better LP relaxation bounds, and for the ideal formulation $z = z_{LP}$.

Example: *Knapsack problem*

$$\begin{aligned} \max \quad & 12x_1 + 8x_2 + 17x_3 + 11x_4 + 6x_5 + 2x_6 + 2x_7 \\ & 4x_1 + 3x_2 + 7x_3 + 5x_4 + 3x_5 + 2x_6 + 3x_7 \leq 9 \\ & x_i \in \{0,1\} \end{aligned}$$

LP relaxation: $x_i \in \{0,1\} \Rightarrow 0 \leq x_i \leq 1$.

The x_i are in decreasing order based on the value/volume ratio.

$$\begin{cases} x_1 = 1 \\ x_2 = 1 \\ x_3 = \frac{2}{7} \end{cases} \rightarrow \text{Total value:}24.86$$

And this is how we find the upper bound: $z \leq 24.86$. In this case since the object values are also integers, we deduce that $z \in \mathbb{Z}$, thus $z \leq 24$. Combining this result with the primal bound found earlier, we obtain:

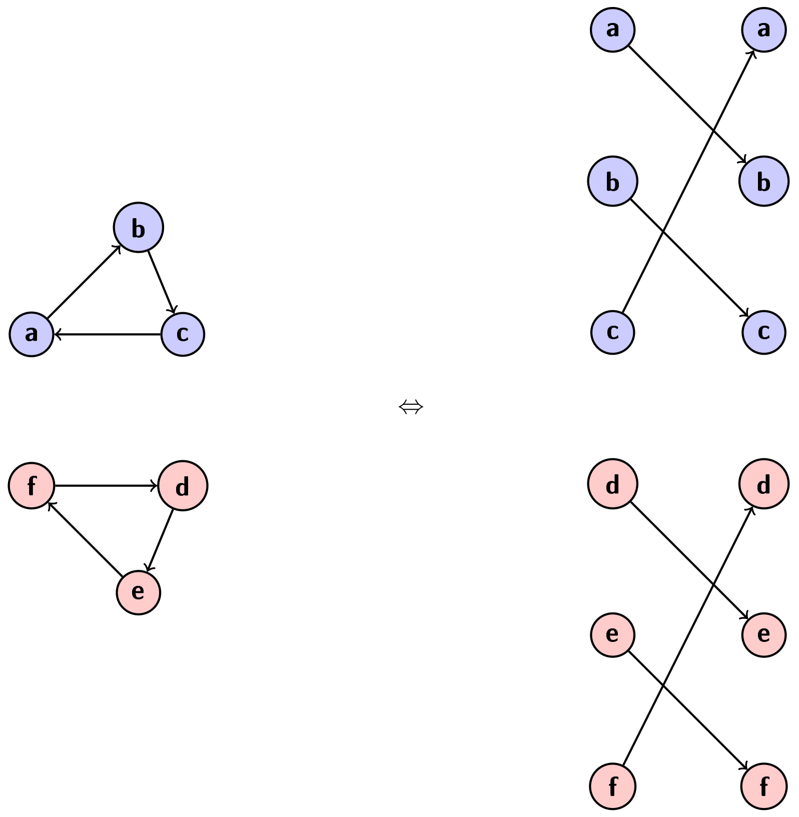
$$22 \leq z \leq 24$$

2.2.2 Relaxing a combinatorial problem to another combinatorial problem

Example: *TSP*

Remove the subtour elimination constraint.

$$z^{TSP} = \min \; \{\sum_{i,j \in C} c_{ij} : C \text{ is Hamiltonian cycle}\}$$



We can formulate it as a minimum weighted matching problem, where we assign every city to the next city being visited.

$$z^{TSP} \geq z^{ASSIGN} = \min\{\sum_{i,j \in A} c_{ij}; A \text{ is a perfect matching from cities to cities.}\}$$

NB: To avoid the trivial solution where each city connects to itself, we assign $c_{ii} = +\infty$.
We will see later in this course that the minimum weight matching problem can be solved efficiently.

Example: *Knapsack problem*

$$\begin{aligned} & z = \max\ \{cx : \sum_i a_i x_i \le b;\ x \in \{0,1\}\} \\ \le \quad & z' = \max\ \{cx : \sum_i \lfloor a_i \rfloor x_i \le \lfloor b \rfloor;\ x \in \{0,1\}\} \end{aligned}$$

where $\lfloor . \rfloor$ denotes the 'floor' (round down) operator.

Why is it a relaxation? Because

$$\begin{aligned} & \sum a_i x_i \le b \\ \Rightarrow \quad & \sum \lfloor a_i \rfloor x_i \le \sum a_i x_i \le b \\ \Rightarrow \quad & \sum \lfloor a_i \rfloor x_i \le \lfloor b \rfloor \end{aligned}$$

As a consequence, a feasible solution for the first problem will always be feasible also for the second one. Hence we have that $\{x : \sum_i a_i x_i \le b; x \in \{0,1\}\} \subseteq \{x : \sum_i \lfloor a_i \rfloor x_i \le \lfloor b \rfloor; x \in \{0,1\}\}$.

2.2.3 Duality

Definition The problems $z = \max\{c(x) : x \in S\}$ and $w = \min\{\omega(u) : u \in T\}$ are *weakly dual* to each other if $c(x) \le \omega(u) \quad \forall x \in S, u \in T$. When in addition, we have that $z = w$, the problems are called *strongly dual*. One problem is called the primal, and the other the dual.

We are interested in this generalisation of the LP duality theory to the IP theory for at least the following two reasons:

1. Every feasible dual point makes an upper bound for the primal.
2. If we find a primal and a dual feasible points x and u respectively such that $c(x) = \omega(u)$, then we know that both are optimal.

2.2.4 LP duality

Proposition 2.2.1. *The following two problems* $z = \max\{cx : Ax \le b, x \ge 0\}$ *and* $w = \min\{ub : A^T u = uA \ge c, u \ge 0\}$ *are strongly dual.*

Proof. (weak duality) For a feasible x, u: $Ax \le b \Rightarrow uAx \le ub \Rightarrow cx \le uAx \le ub$. □

A proof of strong duality comes e.g. from Farkas's Lemma.

Proposition 2.2.2. *The problems,*

$$\text{(IP) } z = \max\{cx : Ax \le b, x \ge 0, x \in \mathbb{Z}^n\},$$
$$\text{(LP) } w^{LP} = \min\{ub : uA \ge c, u \ge 0, \}$$

form a weak dual pair.

Proof. If we LP-relax the (IP) we obtain $z \le z^{LP} = w^{LP}$ □

2.2.5 Weak duality in matching and covering problems

Definition In a graph the problem of finding a set of disjoint edges of maximum cardinality is defined as the *unweighted maximum matching* problem. For a bipartite graph, this is called the *unweighted assignment* problem.

Definition A cover in a graph is a set of nodes incident to all edges.

Definition The minimum cover in a graph is the cover of minimal cardinality.

Example: *Optimal placement of road police stations*

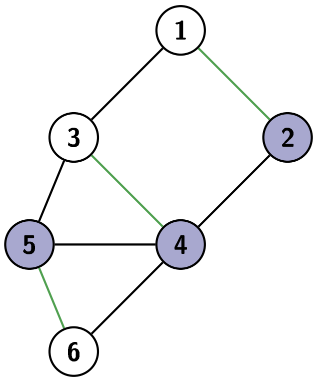
Here is an example of motivation for the cover problem. Let nodes represent cities, and edges represent highways. The problem is to place a police station such that every road has a police station at one of its extremities to intervene in case of accident, etc.

Proposition 2.2.3. *The maximum unweighted matching problem and the minimum cover problem form a weak dual pair.*

We first see a combinatorial/graph-theoretic proof.

Proof. For any matching M on a graph, a cover C on the same graph must have at least one node from every pair of the matching, otherwise the edge linking a specific pair wouldn't be covered. Therefore we can create an application $f : M \to C$ that associates one such node of the cover to every edge of the matching. As a consequence of the matching property, f is injective, otherwise two edges of the matching would have a common extremity. Therefore |matching| $\leq$ |cover|, which is the statement of weak duality. ☐

Example: *Maximum unweighted matching - min cover duality*



In this example |blue cover| = |green matching|, which implies that both are optimal. In general strong duality holds for bipartite graphs, as we shall see.

Let us now see a proof from an IP formulation and LP relaxations.

Proof. Let $M^{n\times m}$ be the node-edge incidence matrix of a graph $G = (V, E)$, thus $M_{ij} = 1$ if node i is incident to edge j and 0 otherwise. This matrix has two non-zero entries per column.

Formulation of maximum matching

$$\begin{aligned} u = \max \sum_{\text{edges e}} x_e &= \mathbb{1}\mathbf{x} \\ Mx &\leq \mathbb{1} \\ x_e &= \{0, 1\} \end{aligned}$$

Formulation of minimum cover

$$\begin{aligned} w = \min \sum_{\text{node i}} u_i &= u\mathbb{1} \\ uM &\geq \mathbb{1} \\ u_i &= \{0, 1\} \end{aligned}$$

The weak duality can be seen by looking at the LP-relaxation of both problems, which turn out to be dual to each other: $z \leq z^{LP} = w^{LP} \leq w$. ☐

Chapter 3

Easy problems

'Easy' problems are problems for which there exists a polynomial-time algorithm in $\mathcal{O}(m^d)$, where m is the size of the instance, i.e. number of bits to describe the instance. This would be essentially the number of nodes and edges for a graph, or the number of objects for a knapsack problem (if weights and values have a constant number of bits), etc.

3.1 Empirical observation

For a class of combinatorial problems $\max\{cx : x \in X \cap \mathbb{Z}^n\}$ given a polyhedron X the followings tend to occur together

(a) There exists a polynomial-time algorithm for finding the maximum.

(b) There exists a strongly dual family $\min\{ub : u \in U \cap \mathbb{Z}^n\}$ with an effective description.

(c) There is an explicit formulation of the convex hull of $X \cap \mathbb{Z}^n$ ($\text{conv}(X \cap \mathbb{Z}^n)$), i.e. an explicit ideal formulation.

(d) There exists an easy separation problem for the set $\text{conv}(X \cap \mathbb{Z}^n)$:

Input : x
Output :

$$x \in \text{conv}(X \cap \mathbb{Z}^n) \Rightarrow \text{YES}$$

$$x \notin \text{conv}(X \cap \mathbb{Z}^n) \Rightarrow d, d_0 \text{ s.t. } \begin{cases} dy \leq d_0 & \forall y \in conv(X \cap \mathbb{Z}^n) \\ dx > d_0 \end{cases}$$

3.2 Integer Programmes with Totally Unimodular Matrices

Suppose we want to solve

$$z = \max\{cx : Ax \leq b, x \geq 0, x \in \mathbb{Z}^n\}.$$

Let us consider the LP relaxation :

$$\bar{z} = \max\{cx : Ax \leq b, x \geq 0\, x \in \mathbb{R}^n\}.$$

The feasible set is a polyhedron, and the optimal solution x^* is clearly a vertex of the polyhedron. In case of several optimal solutions, at least one is a vertex. This vertex is completely defined by n tight constraints, thus satisfied with equality. One can therefore write an invertible system:

$$\begin{cases} Bx^*_{(1)} = b_{(1)} \\ x^*_{(2)} = 0 \end{cases}$$

where B is an invertible sub-matrix of A, $x^* = [x^*_{(1)}, x^*_{(2)}]^T$ and $b = [b_{(1)}, b_{(2)}]^T$. The optimal solution is thus given by

$$\begin{cases} x^*_{(1)} = B^{-1}b_{(1)} \\ x^*_{(2)} = 0 \end{cases}$$

In general the LP relaxation provides an upper bound on a maximisation IP, but we would like to know when the optimal solution x^* of the LP relaxation (3.2) is also an optimal solution of the initial IP (5.3), i.e. when $x^* \in \mathbb{Z}^n$.

3.2.1 Total Unimodularity

Definition A matrix $B \in \mathbb{Z}^{k \times k}$ is unimodular if and only if it is invertible and $B^{-1} \in \mathbb{Z}^{k \times k}$.

Proposition 3.2.1. *B is unimodular if and only if* $\det(B) = \pm 1$.

Proof. ($\Rightarrow$) Suppose B is unimodular :

$$BB^{-1} = I \Rightarrow \det(BB^{-1}) = \det(B)\det(B^{-1}) = 1$$

Because $B, B^{-1} \in \mathbb{Z}^n$, we have $\det(B), \det(B^{-1}) \in \mathbb{Z}$. So we have $\det(B) = \det(B^{-1}) = \pm 1$.
($\Leftarrow$) Suppose now that the invertible matrix B is such that $\det(B) = \pm 1$. By Cramer's formula for the inverse of a matrix,

$$B^{-1} = \underbrace{\frac{1}{\det(B)}}_{=\pm 1} \underbrace{(B^{co})^T}_{\in \mathbb{Z}} \in \mathbb{Z}$$

Indeed, B^{co} is the cofactor matrix of B : $(B^{co})_{ij} = \pm \det(B_{IJ})$, where B_{IJ} is obtained by deleting the line i and the column j of B. □

Definition The matrix $A \in \mathbb{Z}^{k \times k}$ is totally unimodular (TU) if and only if every invertible (square) sub-matrix of A is unimodular

Proposition 3.2.2. *The matrix A is TU if every square sub-matrix has* $\det = 0, \pm 1$.

Proof. Trivial by the definition above (3.2.2). □

Proposition 3.2.3. *If A is totally unimodular then A^T is TU, and so are* $\begin{bmatrix} A \\ I \end{bmatrix}$ *and* $\begin{bmatrix} A & I \end{bmatrix}$.

Proof. Trivial definition. □

3.2.2 Properties of IP and TU

Consider the two problems

$$\begin{cases} z = \max\{cx : Ax \leq b, x \geq 0, x \in \mathbb{Z}^n\} & \text{IP} \\ \bar{z} = \max\{cx : Ax \leq b, x \geq 0\}. & \text{LP relax of the IP} \end{cases}$$

If

$$\begin{cases} b \in \mathbb{Z}^\ell \\ A \in \mathbb{Z}^{\ell \times n} \text{ is TU} \end{cases}$$

Then

- $\bar{z} = z$
- There is an effective strong dual: $z = \min\{ub : uA \geq c, u \geq 0\}$
- z can be found in polynomial time (e.g. through interior-point methods for linear programming)
- The convex hull of all feasible points is effectively described : $C_{hull} = \{x : Ax \leq b, x \geq 0\}$
- There is an algorithm that finds out in a polynomial time if $x \in \mathbb{R}^n$ is in C_{hull}, which consists in checking the linear constraints defining the polyhedron C_{hull}. If one constraint is violated, this constraint is the separating hyperplane.

3.2.3 A criterion for total unimodularity

The following theorem gives a sufficient condition for total unimodularity.

Theorem 3.2.4. *If the three following conditions are satisfied, then A is TU.*

1. *All entries of A are 0 or ± 1.*
2. *Every column contains at most 2 non-zero entries.*
3. *$\exists$ a partition M_1, M_2 of the rows s.t. $\forall$ column j with exactly 2 non-zero entries,*

$$\sum_{i \in M_1} A_{ij} = \sum_{i \in M_2} A_{ij}$$

Proof. Let us show that if A is not TU, then the conditions (3.2.4) are not satisfied.

1. Take B the smallest dimension square sub-matrix of A with $\det(B) \notin \{0, \pm 1\}$. ($B$ exists as A is not TU)
2. Then in every column of B, there are 2 non zero entries. Indeed, if there were one column with all 0, then we would have $\det(B) = 0$. And if there were one column with only one 0, we compute the determinant with respect to this column, then $\det(B) = \pm \det(\text{submatrix of } B)$ and thus B would not be of minimal dimension.
3. For the last point, we have a contradiction. By the third hypothesis, we have

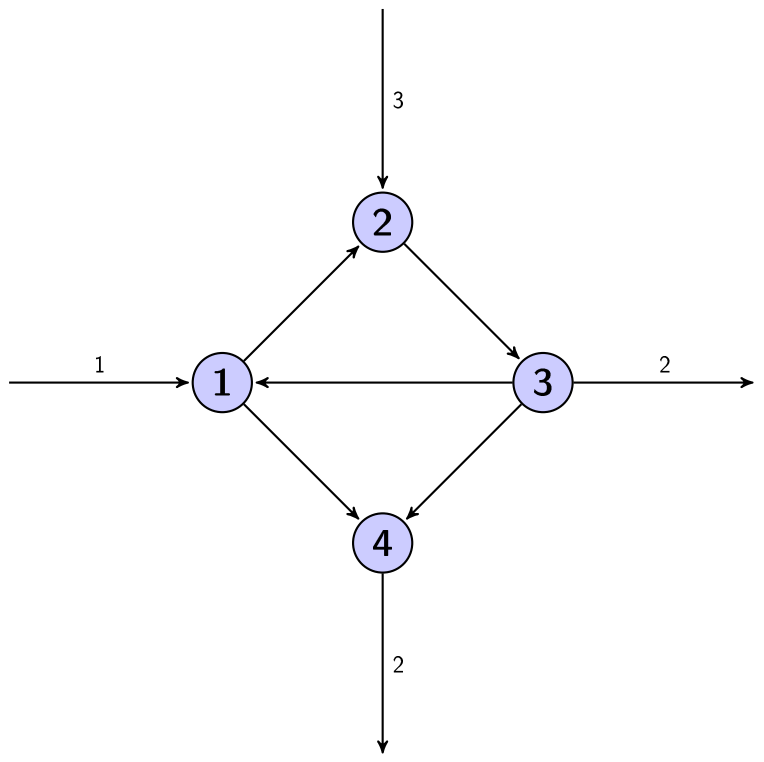
$$\sum_{i \in M_1} B_{ij} = \sum_{i \in M_2} B_{ij} \quad \forall j$$

   There is thus a linear combination of the rows which is equal to zero, which means that B is not invertible. This is a contradiction.

We have proved that if A is not TU then the conditions (3.2.4) are not satisfied. This is equivalent to say that if these conditions are satisfied, then A is TU. □

3.3 Application: flow problems

Example: *Network flows*



Let us start with an example: we want to find a minimum cost of flow in the network that distributes the supply to the demand as in the figure. We also suppose that the flows must be integers, e.g. if we transport indivisible objects.
Let us define x_{ij} = flow $i \to j$, h_{ij} = capacity edge $i \to j$ and c_{ij} = cost edge $i \to j$. Then the minimum cost flow is given by:

$$\min \sum_{i,j} c_{ij} x_{ij}$$

$$\begin{array}{rrrrrrrr}
\text{s.t. } x_{31} & -x_{12} & -x_{14} & & & = & -1 \\
& x_{12} & & -x_{23} & & = & -3 \\
-x_{31} & & & +x_{23} & -x_{34} & = & 2 \\
& & x_{14} & & +x_{34} & = & 2 \\
& & & & x_{ij} & \leq & h_{ij} \\
& & & & x_{ij} & \geq & 0 \\
& & & & x_{ij} & \in & \mathbb{Z}
\end{array}$$

The more general minimum cost problem is clearly:

$$\begin{array}{rcl}
\min cx & & \\
\text{s.t. } Mx & = & d \\
x & \leq & h \\
x & \geq & 0 \\
x & \in & \mathbb{Z}^n,
\end{array}$$

where M is the (directed) incidence matrix of the graph, including for every edge $i \to j$ a column with -1 at row i and $+1$ at row j, while d is the vector of demands (or negative supply) at each node.

We now claim that this formulation is totally unimodular. The matrix of constraints takes here the form $A = \begin{bmatrix} M \\ I \end{bmatrix}$. The question is : is the signed incidence matrix M totally unimodular?
Using criterion (3.2.4), we check the three conditions:

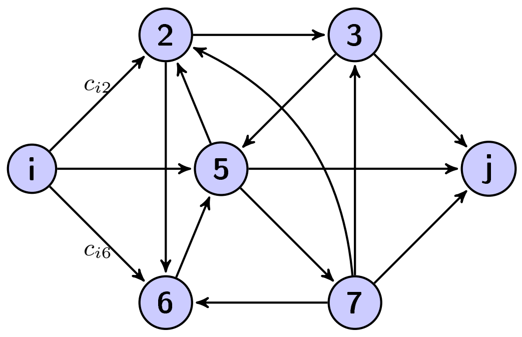
- Every entry $\in \{0, \pm 1\}$;
- there are max 2 non-zero entries per columns,
- $\exists$ a separator so that $M_1 = M, M_2 = \emptyset$

Therefore the claim is proved: the flow problem is TU. This implies that if $h, d \in \mathbb{Z}$, then there is an integral optimal flow for the LP relaxation. Therefore we can solve the problem in the reals even if we look for a solution in the integers.

3.4 A T.U. formulation for the shortest path problem

There is a collection of problems that can be cast as flow problems. Shortest path, max flow - min cut or some unweighted matching problems are part of this class. We consider here the shortest path problem.

Example: *Shortest path*



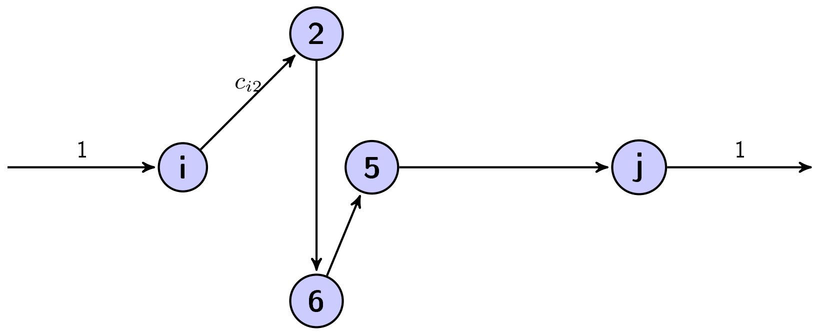
The shortest path problem from node i to node j on a graph $D = (V, A)$ given a cost matrix C can be formulated as follows:

$$\min \sum_{kl} x_{kl} c_{kl} \tag{3.1}$$

$$\text{s.t.} \quad \sum_{\ell} (x_{k\ell} - x_{\ell k}) = d_k = \begin{cases} 1 \text{ if } k = i \\ 0 \text{ if } k \neq i, j \\ -1 \text{ if } k = j \end{cases} \tag{3.2}$$

with the binary variable $x_{k\ell}$ being 1 if the edge $k\ell$ is in the path from i to j and 0 otherwise.

Example: *Shortest path - We can set the path as follows:*



This path can be seen as a flow problem from i to j. The constraint (3.2) can be rewritten as follows

$$Mx = d \tag{3.3}$$

with M of size $n \times p$ (with n and p being respectively the number of nodes $v \in V$ and of edges $e \in A$ in the graph) and d a vector of length p. The matrix M being totally unimodular, we can relax the constraint x such that $x \in \mathbb{Z}$. The shortest path problem allows then a strongly dual problem of the form (3.4) (see next section for a reminder on how to find the dual of an LP).

$$w = \max\{\pi_j : \pi_k - \pi_l \leq c_{lk} \text{ for } e = (k,l) \in A, \pi \in \mathbb{R}^n_+, \pi_s = \alpha\} \tag{3.4}$$

We can shift $\pi \to \pi + \alpha$ so we can choose $\pi_i = 0$ without loss of generality. One optimal solution is obtained by setting π_k to the distance (shortest path length) from i to k. Optimality of this can be checked by noticing that it is dual feasible and reaches the same cost as the primal. We restate this:

Theorem 3.4.1. *A path $i \to j$ of length z is the shortest path iff*

$$\exists (\pi_k)_k \ s.t. \ \begin{cases} \pi_l - \pi_k \leq c_{kl} \\ \pi_i = 0 \\ \pi_j = z \end{cases} \tag{3.5}$$

(In this case one choice is $\pi_k = d(i,k)$)

3.5 Reminder on LP duality

How to write down the dual of an LP is summarised below.

$$\text{Primal} \quad \begin{array}{r} \max cx \\ Ax \leq b \\ x \geq 0 \end{array} \quad \Leftrightarrow \quad \text{Dual} \quad \begin{array}{r} \min ub \\ uA \geq c \\ u \geq 0 \end{array}$$

In general :

max		min	
Constraint i is	$\leq$	Variable i is	≥ 0
Constraint i is	$=$	Variable i is	free
Constraint i is	$\geq$	Variable i is	≤ 0
Variable j is	≥ 0	Constraint j is	$\geq$
Variable j is	≤ 0	Constraint j is	$\leq$
Variable j is	free	Constraint j is	$=$

It can be rederived easily from the intepretation of the dual variable u_i associated to a constraint with resource (i.e. independent term) b_i. If we move the resource to $b_i + \epsilon$ then the objective increases by $u_i\epsilon$, for small enough ϵ.

Let us now recall complementary constraints.

Theorem 3.5.1. *For all feasible x, u for the primal-dual problems $z = \max\{cx|\ Ax \leq b,\ x \geq 0\} = w = \min\{ub|\ uA \geq c|\ u \geq 0\}$,*

$$x, u \text{ is optimal} \quad \Leftrightarrow \begin{cases} (uA - c)x = 0 \\ u(b - Ax) = 0 \end{cases} \tag{3.6}$$

Proof. Let us assume that x, u are optimal solution to the problem z and w. Because x and u are feasible solution to the problem,

$$\begin{cases} (uA - c)x = uAx - cx \geq 0 \\ u(b - Ax) = ub - uAx \geq 0 \end{cases} \quad \text{because} \quad \begin{cases} uA \geq c \\ Ax \leq b \end{cases} \tag{3.7}$$

From there we know that

$$cx \leq uAx \leq ub \tag{3.8}$$

but $cx = ub$ because x and u are optimal. Then,

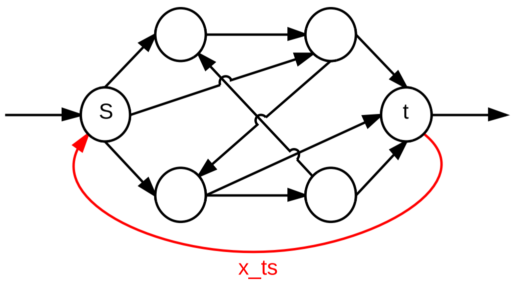
$$\begin{cases} (uA - c)x = 0 \\ u(b - Ax) = 0 \end{cases} \tag{3.9}$$

□

Equation (3.6) means that for a set of primal and dual variables (x, u) to be optimal, for each constraint of both the primal and dual problem either the constraint is tight, either the corresponding variable (dual variable for a constraint in the primal problem and primal variable for a constraint in the dual problem) is equal to zero.

3.5.1 Min cut - Max flow

Let us now try to push as much flow as possible through a graph $G = (V, E)$, or a network, from a source s to a target t, given capacity constraints h_{ij} on each edge ij. We can make the flow circular by adding an edge from t to s, with unlimited capacity. This is formulated as in (3.10).



$$\begin{aligned} z = \quad & \max x_{ts} && (3.10a)\\ \text{s.t.} \quad & \sum_j x_{ji} - \sum_k x_{ik} = 0, \quad \forall i && (3.10b)\\ & 0 \le x_{ij} \le h_{ij}, \quad \forall (ij) \neq (ts) && (3.10c)\\ & 0 \le x_{ts} && (3.10d) \end{aligned}$$

This problem is an LP problem. Its dual can be characterised as (3.11).

$$\begin{aligned} w = \quad & \min \sum_{ij \neq ts} h_{ij} w_{ij} && (3.11a)\\ (s.t.) \quad & u_i - u_j + w_{ij} \ge 0, \quad \forall i, j && (3.11b)\\ & u_t - u_s \ge 1 && (3.11c)\\ & w_{ij} \ge 0 && (3.11d) \end{aligned}$$

with

- u_i : dual variable relative to the constraint of flow on every node i
- w_{ij} : dual variable relative to the constraint on capacity on every edge $ij \neq ts$

Observe that the constraint matrix is TU and the dual resources are integers, so adding the constraints $u_i \in \mathbb{Z}$, $w_{ij} \in \mathbb{Z}$ doesn't change the dual solution. Moreover shifting the $u_i \to u_i + \alpha$ does not change the problem, so we can set $u_s = 0$. Since $u_i \in \mathbb{Z}$, it is possible to define two sets of nodes:

$$S = \{i \in V : u_i \le 0\}$$
$$T = \{i \in V : u_i \ge 1\} = V \setminus S$$

From there we can characterise the value $\underline{w}$ as follows:

$$\underline{w} = \sum_{(i,j) \in E} h_{ij} w_{ij} \ge \sum_{j \in T, i \in S} h_{ij} w_{ij} \ge \sum_{j \in T, i \in S} h_{ij}$$

(the first inequality holds because we drop positive terms from the sum, the second holds because $w_{ij} \ge u_j - u_i \ge 1$ for edges between S and T). We see then that $\underline{w}$ is a lower bound on the cut size between the sets S and T. The cut size between two sets A and B is the sum of capacities of the edges bridging A to B.

The solution $u_i = 0$ for $i \in S$ and $u_i = 1$ for $i \in T$ and $w_{ij} = 1$ for $i \in S, j \in T$ and zero otherwise is feasible and produces that lower bound. We notice that $s \in S$ and $t \in T$. We also notice that the edges (i, j) such that $w_{ij} = 1$ form the $s - t$ cut.

Theorem 3.5.2. *The minimum $s - t$ cut is a strong dual to the maximum $s - t$ flow problem.*

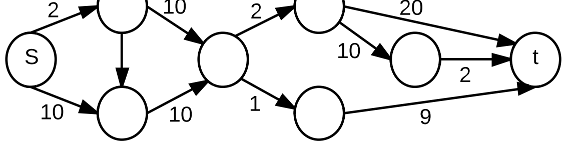
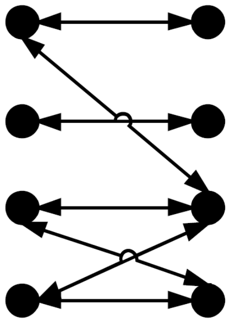


Figure 3.1: Example maximum flow and minimum cut

In figure 3.1, the min cut is 3 (cut in the middle of the graph). We can interpret that as the tightest bottleneck of the graph between source and target.

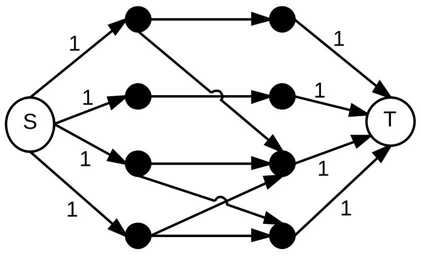
3.6 Maximum cardinality matching on bipartite graph

A matching is a set of non-adjacent edges in a bipartite graph.



3.6.1 Maximum matchings as maximum flows

An unweighted matching on a bipartite graph can be transformed into a flow problem this way by transforming the graph as follows.



For the maximum flow problem all added edges (from source or to target) have unit capacity.

- From TU, the flow between each node is in the integer, $x_{ij} \in \mathbb{Z}$.
- From capacities, $0 \leq x_{ij} \leq 1 \Rightarrow$ we can pick a 0-1 flow.
- From flow preservation constraint, there is at most one edge (i, j) incident to every node with $x_{ij} = 1$.

Thus, the flow encode a matching. Therefore the maximum flow is equivalent to matching of maximum cardinality here. This makes a T.U. formulation for the maximum matching problem.

3.6.2 Maximum matching and minimum cover strong duality in bipartite graphs

One can also write down a direct IP formulation for the matching problem, as in in Prop. (2.2.3), and observe that it is T.U. in the case of bipartite graph.

Proposition 3.6.1. *The undirected incidence matrix of a bipartite graph is totally unimodular.*

Proof. The undirected incidence matrix has 2 non zero entries in each column, indicating the two extremities of each edge. The incidence matrix M naturally splits into two blocks of rows, M_1 and M_2, corresponding to the two classes of nodes of the bipartition. In each column, there is a 1 entry in M_1 and a 1 entry in M_2. Thus the sufficient condition for T.U. is satisfied, and the matrix is T.U. indeed. □

Thus Prop. (2.2.3) establishes a strong duality between maximum matching and minimum cover for bipartite graphs, and an ideal formulation for both.

3.6.3 Alternating paths, augmenting paths and maximum matchings

The maximum matching problem can also be approached from the point of view of graph theory.

Definition In an unweighted graph G with matching M, an M-alternating path is a path that alternates between edges $\in M$ and edges $\notin M$

Definition An M-augmenting path is an M-alternating path whose extremities are exposed (i.e. not incident to any edge of M).

Theorem 3.6.2 (Berge). *A matching M is a maximum iff there exists no M-augmenting paths.*

Proof. ($\Rightarrow$): if there is an M-augmenting path P, then we can improve the matching by exchanging M and non-M membership of edges along the path. Formally: $M' = M \triangle P$ is a matching with $|M'| = |M|+1$.($\triangle$ being the symmetric difference on sets: $A \triangle B = (A \backslash B) \cup (B \backslash A)$).
($\Leftarrow$): Let us assume M is not maximum, and let M^* be maximum. Then consider $M \triangle M^*$: one can check that it is a subgraph of maximum degree two (or less), thus composed of isolated points, paths and cycles. The paths and cycles necessarily alternate between M and M^*. Since $|M^*| > |M|$, there must be a path of odd length, starting and ending with an edge in M^*. Such a path is readily verified to be M-augmenting. □

Although this theorem holds for any matching in any undirected graph, we are interested in its application for discovering maximum matching in bipartite graphs.

The proof is constructive and suggests the following way to construct a maximum matching :

1. Start from a initial matching M
2. Find an augmenting path P
3. Improve the matching $M := M \Delta P$
4. Repeat step 2 and 3 until there is no more augmenting path

How to find an augmenting path is detailed in the next section. Note that an augmenting path in a bipartite graph joins exposed nodes on opposite sides of the bipartition.

Altogether, this augmenting path algorithm is a special case of the general Hungarian method for the weighted case, explained later in this chapter.

3.6.4 The augmenting path algorithm

Let us see how to find an augmenting path in a bipartite graph $G = (U, V, E)$ with $|U| \geq |V|$. Given an initial matching M, the general idea is to explore the graph from the exposed nodes in U, alternating edges $\notin M$ and edges $\in M$, until we find an exposed node in V (thus an M-augmenting path) or we conclude there is no M-augmenting path.

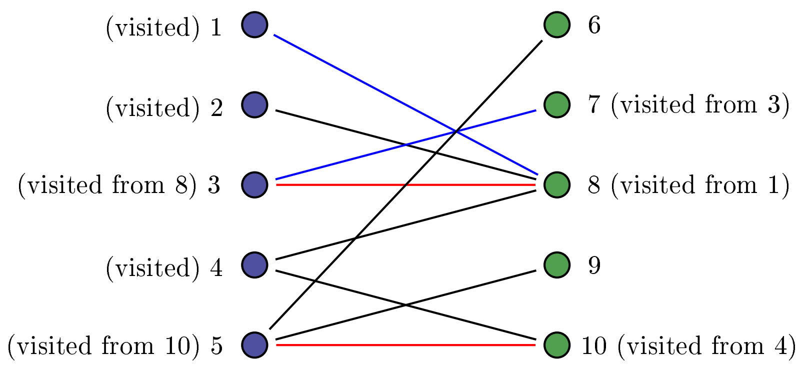
More precisely, the algorithm can be laid out as follows:

1. Build U_0, the set of exposed nodes (nodes not incident to the matching M) in U and label them as 'visited'.
2. For each node $u \in U_0$, look for neighbours of u through edges in $E \setminus M$ which are still unvisited and label them as 'visited (from u)'. Call V_1 the set of these newly visited nodes.
3. For each node $v \in V_1$, look for neighbours of v through edges in M which are still unvisited and label them as 'visited (from v)'. Call U_2 the set of these newly visited nodes.
4. Repeat steps 2 and 3, so as to build $U_0, V_1, U_2, V_3, U_4, \ldots$ until either we find an exposed node in V or we find no more nodes to visit.
5. If we find an exposed node in V, we have an M-augmenting path P joining this node to a node in U_0. We can find this path easily by backtraking the source nodes recorded in the labels. We improve the matching accordingly, $M \leftarrow M \triangle P$, and start from step 1 again.

6. If we exhaust unvisited nodes before finding any exposed node in V then there is no augmenting path, and M is the maximum matching. Stop.

Example of one iteration the augmenting path algorithm

In this example $M = \{(3,8),(5,10)\}$ is the initial matching and P= 1837 is the augmenting path.



Here is the list of operation done by the algorithm.

(a) Label $U_0 = \{1,2,4\}$ as 'visited'.

(b) From the node $1 \in U_0$, visit node 8 label it as 'visited (from 1)'.

(c) From the node $2 \in U_0$, no unvisited neighbour.

(d) From the node $4 \in U_0$, visit node 10 and label it with (4).

(e) We now have built $V_1 = \{8, 10\}$.

(f) From node $8 \in V_1$, visit node 3 (through an edge in M) and label it with 'visited from 8'.

(g) From node $10 \in V_1$, visit node 5 (through an edge in M) and label it with 'visited from 10'.

(h) We now have built $U_2 = \{3, 5\}$.

(i) From the node $3 \in U_2$, visit node 7 and label it as 'visited from 3'.

(j) We find that node 7 is exposed.

(k) Backtracking the labels from 7, we find the augmenting path 1837.

(l) Augment the matching M and remove all labels.

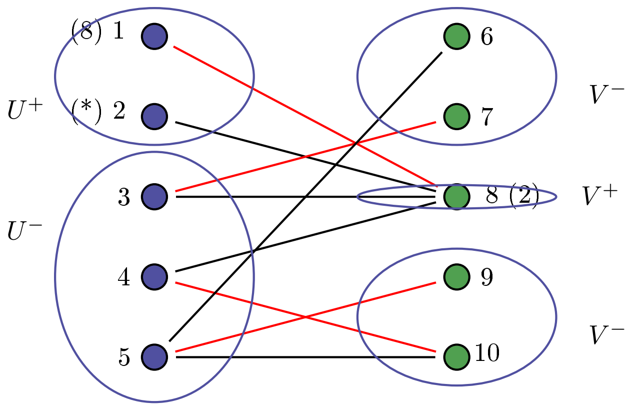
This algorithm can be repeated until no more augmenting path can be found.

When there is no more augmenting path (thus the matching is maximum), let

- $U^+ = U_0 \cup U_2 \cup \ldots$ and $V^+ = V_1 \cup V_3 \cup \ldots$ be the labeled nodes of U and V and
- $U^- = U \setminus U^+$ and $V^- = V \setminus V^+$ the unlabeled nodes of U and V.

Example of the last iteration of the augmenting path algorithm

In this example $M = \{(1,8),(3,7),(4,10),(5,9)\}$ is the final, maximum matching. Labels on visited nodes indicate the previously visited node from which it was reached, or $(*)$ for the initial exposed nodes.



From there we can make a series of observations.

1. No node $v \in V^-$ is adjacent to a node of $u \in U^+$. If they were, they would have been labeled. Indeed, if u and v are neighbours through M then u could only have been visited from v. If u and v are neighbours through $E \setminus M$ then v would have been visited from u.

2. Every node $v \in V^+$ is incident to an edge e of M. Otherwise, v would be exposed, and there would be another augmenting path. The endpoint of that edge is necessarily in U^+.

3. Every node $v \in U^-$ is incident to an edge e of M. If they were not, they would have been visited at the first step of the algorithm. The endpoint of that edge is necessarily in V^-, otherwise v would have been visited eventually.

Theorem 3.6.3. *Given U^+, U^-, V^+ and V^- obtained at the end of the augmenting path algorithm,*

1. *$R = U^- \cup V^+$ is a vertex cover of G,*

2. *$|M| = |R|$, R is a minimal vertex cover and M is a maximum matching.*

3. *$|V^+| = |U^+| - |U_0|$.*

Proof. Part 1: if R is not a vertex cover, then there is an edge between U^+ and V^-, which is impossible given Observation 1. Part 2: any edge in M must be incident to a node of V^+ or a node of U^- (from Part 1, or Observation 1), but not both (from Observations 2 and 3), and conversely, every node of $U^- \cup V^+$ is incident to exactly one edge of M. Thus $|R| = |U^- \cup V^+| = |M|$. From weak duality of the vertex cover and the matching problems, we deduce that R is minimum and M is maximum (which we already knew from the absence of augmenting paths). Part 3: the matching creates a bijection between V^+ and $U^+ \setminus U_0$, from Observation 2. □

This confirms that the vertex cover problem and the maximum matching problems are strong duals in the case of bipartite graphs.

3.6.5 Weighted assignment problem

The natural extension to the maximum cardinality matching is the weighted assignment problem. The objective here is to find a matching of maximum weight in a weighted graph.

Remark Without loss of generality, we can add nodes on one side of the graph so that $|U| = |V| = n$. This weighted graph can be made complete bipartite by adding zero weighted edges. If all initial weights are nonnegative, it does not change the value of the maximum weight matching. We can therefore reduce the problem to finding a *perfect* matching of maximum weight in a complete bipartite graph $K_{n,n}$.

The maximum weight matching in a complete bipartite graph $G = (U, V, E)$, with $V = U \cup V$, given a weight matrix C can then be formulated as follows.

$$\begin{aligned} \max \quad & \sum_{i \in U, j \in V} c_{ij} x_{ij} \\ (s.t) \quad & \sum_{j \in V} x_{ij} = 1 \quad \forall i \in U \\ & \sum_{i \in V_1} x_{ij} = 1 \quad \forall j \in V \\ & x_{ij} \in \mathbb{Z}_+, \quad \forall i \in U, j \in V \end{aligned}$$

Note that this previous formulation is equivalent to the following LP problem because the problem is *TU*.

$$\begin{aligned} \max \quad & \sum_{i \in U, j \in V} c_{ij} x_{ij} \\ (s.t.) \quad & Mx = 1 \\ & x \in \mathbb{R}_+^{2n} \end{aligned}$$

Here M is the (undirected) incidence matrix. Its totaly unimodularity can be proven from the sufficient criterion using partition M_1 and M_2 corresponding to the bipartition. The strict equality $(=)$ is imposed instead of a laxed inequality $\geq$ because we are looking for a perfect matching.

Given the equivalent linear formulation, it is possible to generate the dual of the weighted assignment problem as follows.

$$\begin{aligned} \min \quad & \sum_{i \in U} u_i + \sum_{j \in V} v_j \\ (s.t.) \quad & u_i + v_j \geq c_{ij} \end{aligned} \tag{3.12}$$

The dual problem is a minimum labelling problem where you assign a value u_i and v_j for the nodes $i \in V_1$ and $j \in V_2$ respecting the cost constraint (3.12).

The dual-primal problem here could be solved either by the simplex algorithm or by the interior-point method, but there is a better algorithm that exploits the structure of the problem: the ***primal-dual algorithm***.

Primal-dual algorithm: The primal-dual algorithm take advantage of the property of primal and dual LP (Theorem 3.5.1) that states that the two following propositions are equivalent: (i) x is primal-optimal and u is dual optimal and (ii) x is primal-feasible and u is dual-feasible and x, u verify the complementary constraints.

General scheme of the algorithm can be stated as follows:

1. Start from initial x, u
2. Modify x (**primal step**) and u (**dual step**) alternatively until they satisfy
   (a) primal constraints (x),
   (b) dual constraints (u),
   (c) complementary constraints.

For example :

- u is always feasible,
- x, u always satisfy the complementarity constraints,
- x satisfies more and more primal constraints until all the constraints are satisfied.

The details are problem dependent. The assignment problem is covered in detail here, but primal-dual algorithms exist for other problems e.g. for min-cut-max-flow, shortest path, etc.

Theorem 3.6.4. *For a weighted complete bipartite graph K_{nn} with feasible labelling (u, v) on the node, if the equality graph (i.e. the set of edges where the labelling satisfies the dual constraints with equality: $u_i + v_j = c_{ij}$) contains a perfect matching M, then M is a maximum weight matching.*

Proof. The feasible labelling is a feasible dual point. The perfect matching encodes a feasible primal point. And since the perfect matching is included in the equality graph the complementary constraints are checked (if the primal variable x_{ij} is non-zero, the corresponding dual constraint is tight ($u_i + v_j = c_{ij}$) because $x_{ij}(u_i + v_j - c_{ij}) = 0$). □

3.6.6 The Hungarian Method (Kuhn-Munkres 1955)

The Hungarian algorithm is used to find the maximum weight matching to a complete bipartite graph $G = (U, V, E)$ with weight matrix C.

Input $n \times n$ weight matrix C.

Output Maximum weight assignment.

Initialisation Let u, v be an initial labelling such that $u_i + v_j \geq c_{ij}$ for all $i \in U, j \in V$. For example, $v_j = 0 \quad \forall j \in V$ and $u_i = \{\max c_{ij} : j \in V\} \quad \forall i \in U$.

Primal step Find a maximum cardinality matching M in the equality graph using the augmenting path algorithm.

1. If $|M| = n$, then M is optimal by Theorem 3.6.4.
2. Else, save the matching M and the visited nodes U^+ and V^+.

Dual step Update the dual variables as follows:

$$\begin{aligned} \delta = \min \left\{u_i + v_j - c_{ij} : i \in U^+, j \in V^-\right\} \\ u_i \leftarrow u_i - \delta, \quad \forall i \in U^+ \\ v_j \leftarrow v_j + \delta, \quad \forall j \in V^+ \end{aligned}$$

Correction and complexity of the algorithm

1. U^+ et V^- are saved from the primal step, as they are intermediate results of the maximum matching computation.

2. The new labelling in the dual step will remain feasible. $u_i + v_j$ will be updated as follows:

$$u_i + v_j \leftarrow \begin{cases} u_i - \delta + v_j + \delta = u_i + v_j \text{ if } i \in U^+, j \in V^+ \\ u_i + v_j + \delta \text{ if } i \in U^-, j \in V^+ \\ u_i + v_j \text{ if } i \in U^-, j \in V^- \\ u_i - \delta + v_j \text{ if } i \in U^+, j \in V^- \end{cases}$$

   δ is nonzero since there is no edge of the equality graph between U^+ and $V-$. In fact δ is chosen so that at least one edge enters equality graph whilst keeping the feasibility.

3. The dual objective $\sum u_i + \sum v_j$ will decrease by at least δ, because $|U^+| > |V^+|$, see statement 3 in Theorem 3.6.3.

4. The previous matching is still included in the new equality graph (as the sum $u_i + v_j$ is unchanged on the edges of the matching).

5. The new equality graph coincides with the old equality graph between U^+ and V^+.

6. The new equality graph has at least one new edge $e : U^+ \rightarrow V^-$

7. If M remains a maximum matching for new equality graph, then the exploration performed by the augmenting path algorithm will visit new nodes:

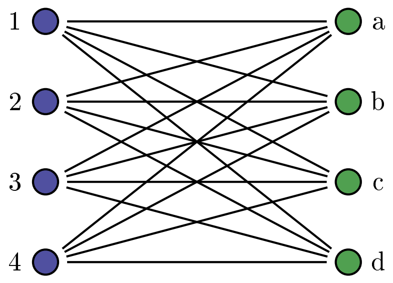
- New $V^+ \not\supseteq$ Old V^+ (by using a new edge e)
- New $U^+ \not\supseteq$ Old U^+ (because an M-alternating path following e will be followed by an M-edge putting a new node into U^+, otherwise we would have found an augmenting path)

8. As long as the matching remains maximal, we need not recompute U^+ and V^+ from scratch, only to extend them with new nodes discovered through the new edges in the equality graph such as e. The sets U^+, V^+ can only increase n times at most: although M is always in the new equality graph, at some point it is not maximum anymore: if we find an augmenting path, then we can find a matching of the equality graph with one more edge than M.
Thus in at most n^2 primal-dual steps, we find a perfect matching in the equality graph
One primal-dual step costs $O\left(n^2\right)$ (compute U^+, V^+ and find δ). So the Hungarian method as described by Kuhn and Munkres (1955, 1957) is in $O(n^4)$, a polynomial time. Later improvements by Edmonds, Karp, Tomizawa brought down this cost to $O(n^3)$.

The Hungarian method: a numerical example

Example: *Assignment problem*

$$c_{ij} = \begin{pmatrix} 27 & 17 & 7 & 8 \\ 14 & 2 & 10 & 2 \\ 12 & 19 & 4 & 4 \\ 8 & 6 & 12 & 6 \end{pmatrix}$$



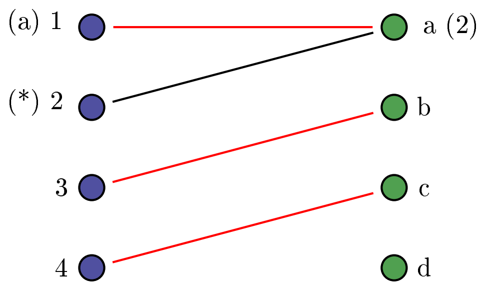
Initial step

$$u^0 = \begin{pmatrix} 27 \\ 14 \\ 19 \\ 12 \end{pmatrix}, v^0 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, (u_i^0 + v_j^0 - c_{ij}) = \begin{pmatrix} 0 & 10 & 20 & 19 \\ 0 & 12 & 4 & 12 \\ 7 & 0 & 15 & 15 \\ 4 & 6 & 0 & 6 \end{pmatrix}$$

The initial dual objective is

$$\bar{z}^0 = \sum_{i \in V_1} u_i^0 + \sum_{j \in V_2} v_j^0 = 72$$

Primal Step 1: Equality graph:



The maximum matching $M^1 = \{(1,a),(3,b),(4,c)\}$ was obtained with the augmenting path algorithm and so were $V_1^+ = \{1,2\}, V_2^+ = \{a\}$ and $V_2^- = \{b,c,d\}$.

Dual step 1: We need to find δ and to update u and v.

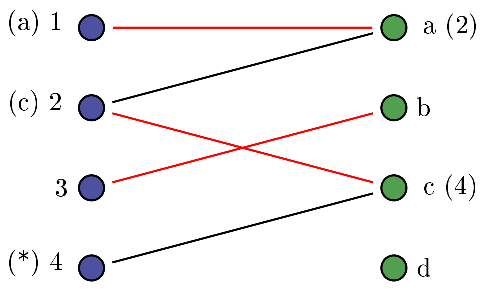
$$\delta^1 = \min_{i \in V_1^+, j \in V_2^-} u_i^0 + v_j^0 - c_{ij} = 4$$

$$u^1 = \begin{pmatrix} 23 \\ 10 \\ 19 \\ 12 \end{pmatrix}, v^1 = \begin{pmatrix} 4 \\ 0 \\ 0 \\ 0 \end{pmatrix}, u_i^1 + v_i^1 - c_{ij} = \begin{pmatrix} 0 & 6 & 16 & 15 \\ 0 & 8 & 0 & 8 \\ 11 & 0 & 15 & 15 \\ 8 & 6 & 0 & 6 \end{pmatrix}$$

New dual objective:

$$\bar{z}^1 = \sum_{i \in V_1} u_i^1 + \sum_{j \in V_2} v_j^2 = 68$$

Primal step 2: New equality graph:



There is a new maximum matching $M^2 = \{(1,a),(2,c),(3,b)\}$ obtained with the augmenting path algorithm. We can update $V_1^+ = \{1,2,4\}, V_2^+ = \{a,c\}$ and $V_2^- = \{b,d\}$.

Dual Step 2:

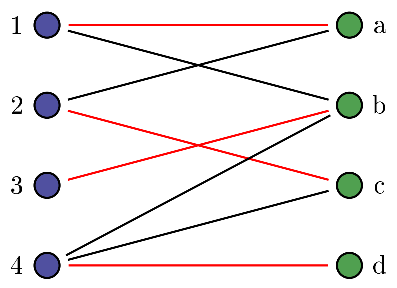
$$\delta^2 = \min_{i \in V_1^+, j \in V_2^-} u_i^1 + v_j^1 - c_{ij} = 6$$

$$u^2 = \begin{pmatrix} 17 \\ 4 \\ 19 \\ 6 \end{pmatrix}, v^2 = \begin{pmatrix} 10 \\ 0 \\ 6 \\ 0 \end{pmatrix}, u_i^2 + v_i^2 - c_{ij} = \begin{pmatrix} 0 & 0 & 16 & 9 \\ 0 & 2 & 0 & 2 \\ 17 & 0 & 21 & 15 \\ 8 & 0 & 0 & 0 \end{pmatrix}$$

New dual objective:

$$\bar{z}^2 = \sum_{i \in V_1} u_i^1 + \sum_{j \in V_2} v_j^2 = 62$$

Primal step 3: New equality graph:



The new matching $M^3 = \{(1,a),(2,c),(3,b),(4,d)\}$ is a perfect matching so we know that the dual objective that produced that matching is the optimal solution to the weight assignment problem. To convince ourselves that it is an optimal solution we see that the profit associated with that matching (so the primal objective) is also 62.

3.7 Matroids and submodular functions

3.7.1 The greedy algorithm on matroids

We see here another family of problems with a special structure that makes it 'easy', with efficient greedy algorithms, and an ideal IP formulation.

A well known problem in graph theory is to find a maximum/minimum weight spanning tree (i.e. a connected, acyclic graph incident to all nodes) in a weighted connected undirected graph. This useful for example to find a minimum cost connected sub-network between points of interest.

The solution is a greedy algorithm: Kruskal's algorithm.

Kruskal's algorithm:

Input Weighted graph $G = (V, E)$ with n nodes and m edges.

Output maximum weight spanning tree

Algorithm
```
1. Sort: $w_1 \geq w_2 \geq ... \geq w_m$
2. $E = \emptyset$
3. for $i = 1$ to $m$: if $E \cup \{i\}$ has no cycle then $E = E \cup \{i\}$. If $|E| = n - 1$ then STOP, output $= E$
```

Note that spanning trees can be defined as maximal acyclic subgraphs, or acyclic graphs with exactly $n - 1$ edges.

In this section we generalise this algorithm and the properties of trees that it relies upon to a general structure for which greedy strategies are naturally suited.

Definition Let E be a set with $\mathcal{F}$, family set of subsets of E called the "independent sets" of E, such that:

- $\emptyset \in \mathcal{F}$
- $B \in \mathcal{F}$ and $A \subseteq B \rightarrow A \in \mathcal{F}$
- $\forall S \subseteq E$ all independent subsets of S that are maximal for inclusion have the same cardinality (called the rank of S). Then the structure $(E, \mathcal{F})$ is called a *matroid.*

Some simple examples of matroids:

1. E = vector space; independent sets = linearly independent set. Rank (S) = dim span(S)
2. E = columns of a matrix, independent set = linear independent subset of columns. Rank = rank of the submatrix composed of those columns.
3. E = edge of a graph, independent set = acyclic subgraph = forest. Rank(S) = number of edges - number of connected components.

One can prove the following properties

Proposition 3.7.1. *For any matroid $(E, \mathcal{F})$,*

- *$rank(A) \leq rank(B)$ for all $A \subseteq B$ (monotonicity)*
- *for all $A, B \subseteq E$: $rank(A) + rank(B) \geq rank(A \cap B) + rank(A \cup B)$ (submodularity)*

Matroids are the natural framework to express a *generalisation of Kruskal's greedy algorithm*:

Greedy algorithm for maximal independent set of maximum weight in a matroid:

Input a matroid $(E, \mathcal{F})$ of n elements with a weight function $w : E \rightarrow \mathbb{R}$.

Output maximum weight maximal independent set of E

Algorithm
```
1. Sort: $w_1 \geq w_2 \geq ... \geq w_n$
2. $S = \emptyset$
```

3. for $i = 1$ to n: if $S \cup \{i\}$ is independent then $S = S \cup \{i\}$. If $|S| = rank(E)$ then STOP, output $= S$

Theorem 3.7.2. *At the end of this greedy algorithm, S is the maximum weight maximum independent subset of E (maximum subset for inclusion or for cardinality, equivalently).*

Proof. We assume for simplicity $w_1 > w_2 > w_3 > ... \geq 0$.

$$\begin{cases} \max w(S) \\ S \text{ is max independent set} \end{cases} \Leftrightarrow \begin{cases} \max w(S) \\ S \text{ is independent} \end{cases}$$

This can be formulated as an IP, with $x_i = 1$ for elements $i \in S$ and $x_i = 0$ otherwise:

$$\begin{cases} \max \sum_i w_i x_i \\ \forall T \subseteq E : \sum_{i \in T} x_i \leq rank(T) \\ x_i \in \{0,1\} \end{cases}$$

This is equivalent because:

$$\begin{aligned} S \text{ independent } &\Leftrightarrow \forall T \subseteq S : |T| = rank(T) \\ &\Leftrightarrow \forall T \in E : |T \cap S| \leq rank(T) \end{aligned}$$

Of course this is a highly redundant formulation as only the constraint for $T = S$ would be enough. But as we know, redundant IP formulations tend to be better. In this case, it is ideal, and we check it by writing the LP relaxation:

$$\begin{cases} \max \sum_i w_i x_i \\ \forall T \subseteq E : \sum_{i \in T} w_i \leq rank(T) \\ x_i \geq 0 \end{cases}$$

The dual of the LP relaxation is:

$$\begin{cases} \min \quad \sum\limits_{T \subseteq E} rank(T) y_T \\ \qquad \sum\limits_{i \in T} y_T \geq w_i \\ \qquad y_T \geq 0 \end{cases}$$

To prove the greedy algorithm is correct, we prove that its solution is primal feasible we find a dual feasible point that checks the complementarity constraints
Primal solution:

$$\begin{aligned} x_i &= rank\,\{1,2,...,i\} - rank\,\{1,2,...,i-1\} \\ &= \begin{cases} 1 & \text{if } e_i \text{ increase the rank of } \{1,2,...,i-1\} \\ 0 \end{cases} \end{aligned}$$

This is indeed the output of the greedy algorithm (as easily seen).
It is feasible because

$$\forall T : \sum_{i \in T} x_i = \sum_{i \in T} rank\,\{1,...,i\} - rank\,\{1,...,i-1\}$$

and from Prop. 3.7.1 with $A = \{e_1, \dots, e_i\} \cap T$, and $B = \{e_1, \dots, e_{i-1}\}$:

$$\begin{aligned} &\leq \sum_{i \in T} rank\,(T \cap \{1,...,i\}) - rank\,(T \cap \{1,...,i-1\}) \\ &\leq \sum_{i \in E} rank\,(T \cap \{1,...,i\}) - rank\,(T \cap \{1,...,i-1\}) \\ &= rank(T) - rank(\emptyset) \\ &= rank(T) \end{aligned}$$

Let us look at the following dual solution:

$$\begin{aligned} y_{\{1,\dots,i\}} &= w_i - w_{i+1} \\ y_E &= w_n \\ y_T &= 0 \text{ otherwise} \end{aligned}$$

It is dual-feasible because:

$$\sum_{T \ni i} y_T = \sum_{\substack{\{1,\dots,j\} \\ j \geq i}} w_j - w_{j+1} = w_i$$

Complementary constraints are satisfied:

$$\begin{cases} x_i > 0 \Rightarrow \sum_{T \ni i} y_T = w_i & \text{OK!} \\ y_T > 0 \Rightarrow \sum_{i \in T} x_i = rank\,(T \cap \{1, ..., i\}) \end{cases}$$

we could also check that primal and dual objective has the same value. □

As a by-product of the proof, we see that the matroid problem can be expressed as an LP. Example: The maximum weight spanning tree problem (n is the number of nodes) on a graph $G = (V, E)$:

$$\begin{cases} \max \sum_i w_i x_i \\ \forall T \subseteq E : \ \sum_{i \in T} x_i \leq n - \text{number of connected components of subgraph } (V, T) \\ x_i \geq 0 \end{cases}$$

3.7.2 Application: scheduling with deadlines

Suppose I can choose to perform a set of tasks $a_1, \dots, a_n$ of unit duration. Each task a_i must be performed before a deadline t_i, in order to earn a gain w_i. Tasks are performed sequentially, i.e. cannot overlap in time. Which tasks should I plan to maximise the total gain?

We define a matroid on the set of tasks as follows: an independent set of tasks is one that is feasible, i.e. that can be scheduled in order to respect all deadlines.

One must prove that this is a matroid indeed. One must show in particular that for any set S of tasks, any feasible subset of tasks T that is maximal for inclusion (i.e. adding any other task of S to T would make it infeasible) has the same cardinality. Assume we find two feasible subsets T, T' of tasks which are maximal for inclusion. Then they can be both scheduled in the interval $[0, |T|]$ and $[0, |T|]$ (remember that the tasks are of unit duration). Now if $|T| < |T'|$, then I can add the last task of T' to the schedule of T, and keep feasibility: contradiction. The other matroid properties are trivial.

From this we can apply the greedy algorithm: starting from an empty schedule, try to add all tasks from the most profitable to the least, skipping the tasks that would make the schedule infeasible. This provides the optimal task scheduling.

Note that a direct, elementary proof of the optimality of this algorithm is rather involved: this is a good example of the power of matroid theory.

3.7.3 The matroid intersection problem

One can see the assignment problem as an optimisation problem involving two matroids.

For the nodes $U \cup V$ of a bipartite graph, with edge set E, we can define

- $(E, \mathcal{F}_1)$, the matroid in which independent sets of edges are those for which no two edges intersect in U;
- $(E, \mathcal{F}_2)$, the matroid in which independent sets of edges are those for which no two edges intersect in V

A matching is a set that is independant in both matroids. In other words, the set of matchings is $\mathcal{F}_1 \cap \mathcal{F}_2$. Finding a set of minimum, or maximum, weight in an intersection of two matroids is called the *matroid intersection* problem. There are efficient algorithms solving that problem, which generalise ideas from the Hungarian method.

3.7.4 Submodular functions

A submodular set function f over a set E is a function that assigns a real value $f(A)$ to each subset A of E, with the following property:

$$\forall A, B \subseteq E : f(A) + f(B) \geq f(A \cap B) + f(A \cup B).$$

Evidently, it is a generalisation of the rank function from matroids. Other examples include cardinality, or weight, etc. It can be shown (as was used for the rank in the proof of optimality of the greedy algorithm for matroids) to be equivalent to the following property:

$$\forall A \subseteq B \subseteq E, \forall x \in E \setminus B : f(A \cup \{x\}) - f(A) \geq f(B \cup \{x\}) - f(B).$$

This property of 'diminishing returns' is evocative of concavity in the field of continuous optimisation.

As a matter of fact, finding the set which *minimizes* a submodular function can be done in polynomial time. Finding the *maximum* value however is hard in general.

If however the function is monotone ($f(A) \leq f(B)$ for all $A \subseteq B$) and nonnegative ($f(A) \geq 0$ for all A), it can be leveraged to produce interesting greedy algorithms. One the most representative is the following, by Nemhauser, Wolsey and Fisher (1978).

Suppose we want to find a set of given cardinality k with maximum value. Starting from an empty set, we grow a solution S adding at each step the element $x \in E$ that maximises the immediate marginal profit $f(A \cup \{x\}) - f(A)$, until we have k elements. This simple algorithm is not optimal in general, but guarantees to be within a factor $1 - 1/e \approx 63\%$ of the actual maximal value!

An example of application is choosing where to place k antennas in a definite set of possible locations that together capture the maximum area. Here the value of a set of sensors is the size of the total area covered by the set of antennas (which is the union of areas covered individually).

Another is a hard version of the uncapacitated facility location problem: Suppose that for a set of clients and locations, c_{ij} represents the profit of serving client i at facility j. One must choose a set of k facilities and assignment of clients' demands to facilities to maximise the total profit. This problem is NP-hard in general, but the total profit is a submodular function of the set of facilities. Thus the greedy heuristic above applies.

3.7.5 A bonus: the Steiner tree problem

Let us mention a general, NP-hard problem that includes some polynomial-time problems as a particular case.

Fix a set of nodes N and find a tree that connects all nodes of N with maximum or minimum weight. If N is the set of all nodes, this is a maximum weight spanning tree. if N is a 2-node set, this is the shortest path problem. however no known efficient algorithm in general, since the problem is NP-hard.

Chapter 4

Dynamic Programming

Dynamic programming is a methodology to solve certain classes of combinatorial problems that can be related to “slightly simpler" problems. We start with an introductory example in Section 4.1, present the general methodology in Section 4.2, and then visit other examples.

4.1 Introductory Example: shortest path in a graph

Given a directed graph $D = (V, A)$, nonnegative arc distances c_e for $e \in A$, and an initial node $s \in V$, the problem is to find the shortest path from s to every other node $v \in V \backslash \{s\}$. See Figure 4.1, in which a shortest path from s to t is shown, as well as one intermediate node p on the path.

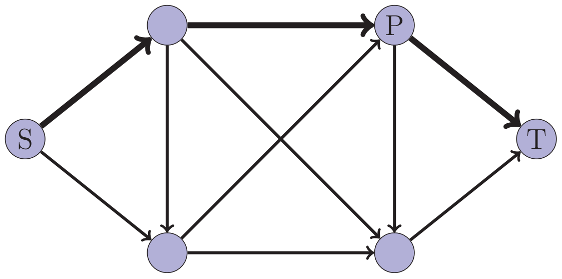


Figure 4.1: Shortest s-t path

Observation 1. *If the shortest path from s to t passes by node p, the subpath (s, p) and (p, t) are shortest paths from s to p, and p to t respectively.*

Observation 2. *Let $d(v)$ denote the length of a shortest path from s to v and $V^-(v)$ the set of neighbours of v. Then*

$$d(v) = \min_{i \in V^-(v)} \{d(i) + c_{iv}\} \tag{4.1}$$

In other words, if we know the lengths of the shortest paths from s to every neighbour (predecessor) of v, then we can find the length of the shortest path from s to v.

Observation 3. *Given an acyclic digraph $D = (V, A)$ with $n = |V|$, $m = |A|$, where the nodes are ordered so that $i < j$ for all arcs $(i, j) \in A$, then for the problem of finding shortest paths from mode 1 to all other nodes, the recurrence (4.1) for $v = 2, \cdots, n$ leads to an $O(m)$ algorithm.*

For arbitrary directed graphs with nonnegative weights $c \in \mathbb{R}_+^{|A|}$, we need to somehow impose an ordering. One way to do this is to define a more general function $D_k(i)$ as the length of a shortest path from s to i containing at most k arcs. Then we have the recurrence:

$$D_k(j) = \min\{D_{k-1}(j), \min_{i \in V-(j)} [D_{k-1}(i) + c_{ij}]\}. \tag{4.2}$$

Now by increasing k from 1 to $n - 1$, and each time calculating $D_k(j)$ for all $j \in V$ by the recursion, we end up with an $O(mn)$ algorithm and $d(j) = D_{n-1}(j)$.

4.2 Formal Description

Following Section 4.1, we propose here a general abstract formulation of the dynamic programming approach.

Let p^* be the problem instance that we want to solve. Dynamic programming supposes that we can find a finite set $\mathbb{P}$ of instances p with optimal solutions S_p (of corresponding costs $C(S_p)$), such that

- $p^* \in \mathbb{P}$: the relevant problem belongs to the set.
- There is an order relation[1] on $\mathbb{P}$: $q < p$ is q if “more basic" than p. The order can be partial, i.e. not necessarily defined for every p, q. In addition, there each problem p has a (possibly empty) set of immediate predecessors $R_p \subseteq \{q : q < p\}$ among the more basic problem.
- If the problem p has no predecessor ($R_p = \varnothing$), then its optimal solution S_p can “easily" be computed.
- If the problem has predecessors ($R_p \neq \varnothing$), then the optimal solution S_p and its cost $C(S_p)$ satisfy

$$C(S_p) = \min\{\min_{q \in R_p} C(f_{p,q}(S_q)), C(S_p^0)\}, \tag{4.3}$$

  (or a similar relation) where

  (i) $f_{p,q}(S_q)$ is a way of building a solution of p by extending the optimal solution of q. It is supposed to be easy to compute.

  (ii) S_p^0 represents is a “trivial" solution for p, that is also supposed to be easy to compute. This solution is not always present.

In the case of the shortest paths from the source s to node v, the set $\mathbb{P}$ of problems consist of computing the shortest paths of length at most k ($k = 0, \dots, n$) from the source s to every node i, and the problem p^* can be seen as the shortest path of length at most $n-1$ to the specific node v, so that $p^* \in \mathbb{P}$. The order relation can be defined by $D_k(i) < D_{k'}(i')$ if $k < k'$ i.e. the maximal length is shorter. The set of immediate predecessor R_p of $p = D_k(i)$ is the set of $D_{k-1}(j)$ for all j from which i can be reached. The relation (4.2) is then an instantiation of the generic dynamic programming relation (4.3), with no S_p^0, and with $f_{D_k(i), D_{k-1}(j)}$ consisting in adding the edge (j, i) to the solution. The problems $D_0(i)$ have no immediate predecessors, and their optimal solution is indeed trivial to compute, the cost being 0 if i is the source and infinite otherwise.

The dynamic programming methodology consist then summarized in Algorithm 1. Its efficiency is then driven by the complexity of the “easy" operations, the size of $\mathbb{P}$, and possibly by the number of immediate predecessors problems have and/or the difficulty of finding them.

```
Solve all problems $p$ for which $R_p = \varnothing$;
**while** $p^*$ *not solved* **do**
    Pick $p$ such that all problems of $R_p$ solved;
    Solve $p$ using (4.3)
**end**
```

Algorithm 1: Dynamic programming general algorithm

We stress that implementing a dynamic programming algorithm requires many choices. In particular, one must chose the problem set, the order and predecessor relation, and the way the problem p is selected in the loop of Algorithm 1. This may be nontrivial, and there may be multiple solutions possibly leading to algorithms of very different efficiency. The key issue will usually be the discovrery of a "recurrence" relation of the form (4.3). Finally, we will see in Section 4.6 that one may sometimes use variations of (4.3).

Moreover, an efficient implementation will often require taking advantage of specific features of the problem at stake, allowing for example not solving a large class of the subproblems, or to detect that a problem already solved actually provides the solution to p^*. In the case of the shortest path from s to v, one can easily find sufficient conditions for the algorithm to stop without going to $k = n - 1$.

[1]A real order relation should be of the form $\leq$, but equality is not useful is this context

4.3 0-1 Knapsack

We consider the $0-1$ knapsack problem:

$$
\begin{aligned}
z \quad &= \max \sum_{j=1}^{n} c_j x_j \\
\text{s.t.} \quad & \sum_{j=1}^{n} a_j x_j \le b \\
& x \in \mathbb{B}^n
\end{aligned}
\tag{4.4}
$$

where the coefficients $a_1, \cdots, a_n$ and b are positive integers.

As set $\mathbb{P}$, we will define problems with smaller costs and smaller sets of objects: Let λ take values from $0, 1, \cdots, b$ as state, and the subset of variables $x_1, \cdots, x_r$ represented by r as stage, leads us to define the problem $P_r(\lambda)$ and optimal value function $f_r(\lambda)$ as follows,

$$
\begin{aligned}
f_r(\lambda) \quad &= \max \sum_{j=1}^{r} c_j x_j \\
\text{s.t.} \quad & \sum_{j=1}^{r} a_j x_j \le \lambda \\
& x \in \mathbb{B}^n
\end{aligned}
\tag{4.5}
$$

If $x_r = 0$, we have $f_r(\lambda) = f_{r-1}(\lambda)$; if $x_r = 1$, then $f_r(\lambda) = c_r + f_{r-1}(\lambda - a_r)$. Thus we have the following instantiation of (4.3)

$$
f_r(\lambda) = \max\{f_{r-1}(\lambda), c_r + f_{r-1}(\lambda - a_r)\}, \tag{4.6}
$$

for a set of two predecessors of $f_r(\lambda)$ defined by $\{f_{r-1}(\lambda - a_r)\}$ and $f_{r-1}(\lambda)$ and the operation $f_{q,p}$ consisting in either adding the object r in the first case and keeping the solution $f_r(\lambda)$ as it is in the second one.

Example : Consider the 0-1 knapsack instance:

$$
\begin{aligned}
z = \max\ & 10x_1 + 7x_2 + 25x_3 + 24x_4 \\
& 2x_1 + 1x_2 + 6x_3 + 5x_4 \le 7 \\
& x \in \mathbb{B}^4
\end{aligned}
\tag{4.7}
$$

The values of $f_r(\lambda)$ are shown in Table 4.1. The values of $f_1(\lambda)$ are calculated by formula described above. The next column is then calculated from top to bottom using the recursion. For example, $f_2(7) = \max\{f_1(7), 7 + f_1(7-1)\} = \max\{10, 7+10\} = 17$, and as the second term of maximization gives the value of $f_2(7)$, we set $p_2(7) = 1$. The optimal value $z = f_4(7) = 34$ with $x^* = (1, 0, 0, 1)$. The same Table 4.1 contains the values $p_r(\lambda)$, with $p_r(\lambda) = 1$ if the item r is selected in the problem $P_r(\lambda)$, i.e. in the problems where we only consider the first r items and a maximal weight λ, and 0 else. These values allow keeping track of the optimal solution: if $p_r(\lambda) = 1$, then the optimal solution of $P_r(\lambda)$ contains $x_r = 1$ and the optimal solution of $P_{r-1}(\lambda - a_r)$. When seeing that $p_4(7) = 1$, one knows for example that the optimal solution contains the item 4 and the optimal solution of $P_3(7-5)$.

Remark The complexity of this dynamic programming algorithm is $O(b.n)$ which would appear to contradict with the NP-hardness of the Knapsack problem. Unfortunately, this does not bring a positive answer to $P = NP$. Indeed, the polynomial character of an algorithm should be understood as *polynomial in term of the size of the instance description.* In the case of the Knapsack with integer weights, describing the instance requires specifying b, and the weights a_i, which can be achieved with a number of bits $\log_2 b + \sum_{i=1}^{n} \log_2 a_i \le \log_2 b + n \max \log_2 a_i$. The apparently polynomial $b.n$ remains thus exponential with respect to this number of bits.

Table 4.1: $f_r(\lambda)$ for a 0-1 knapsack problem

λ	f_1	f_2	f_3	f_4	p_1	p_2	p_3	p_4
0	0	0	0	0	0	0	0	0
1	0	7	7	7	0	1	0	0
2	10	10	10	10	1	0	0	0
3	10	17	17	17	1	1	0	0
4	10	17	17	17	1	1	0	0
5	10	17	17	24	1	1	0	1
6	10	17	25	31	1	1	1	1
7	10	17	32	34	1	1	1	1

4.4 Integer knapsack problems

The integer knapsack problem can be formulated as:

$$\begin{aligned} z = \max \sum_{j=1}^{n} c_j x_j \\ \textstyle\sum_{j=1}^{n} a_j x_j \le b \\ x \in \mathbb{Z}_+^n \end{aligned} \tag{4.8}$$

where the coefficients $a_1, \cdots, a_n$ and b are positive integers. We define the problem $P_r(\lambda)$ and optimal value function $g_r(\lambda)$ as follows,

$$\begin{aligned} g_r(\lambda) = \max \sum_{j=1}^{r} c_j x_j \\ \textstyle\sum_{j=1}^{r} a_j x_j \le \lambda \\ x \in \mathbb{Z}_+^r \end{aligned} \tag{4.9}$$

If x^* is an optimal solution to $P_r(\lambda)$ giving value $g_r(\lambda)$, then we consider the value of x_r^*. If $x_r^* = t$, then $g_r(\lambda) = c_r t + g_{r-1}(\lambda - t a_r)$ for $t = 0, 1, \cdots, \lfloor \frac{\lambda}{a_r} \rfloor$, and we obtain the recursion:

$$g_r(\lambda) = \max_{t=0,1,\cdots,\lfloor \frac{\lambda}{a_r} \rfloor} \{c_r t + g_{r-1}(\lambda - t a_r)\}. \tag{4.10}$$

The reasoning is thus the same as in the 0-1 Knapsack problem, except that the set of predecessors of $g_r(\lambda)$ is now the set $\{g_{r-1}(\lambda - t a_r) : 0 \le t \le \lambda/r\}$., and the function $f_{p,q}$ consist in adding t copies of the object r.

Example Consider the knapsack instance:

$$\begin{aligned} z = \max 7x_1 + 9x_2 + 2x_3 + 15x_4 \\ 3x_1 + 4x_2 + 1x_3 + 7x_4 \le 10 \\ x \in \mathbb{Z}_+^4 \end{aligned} \tag{4.11}$$

The values of $g_r(\lambda)$ are shown in Table 4.2. With $c_1 > 0$, the values of $g_1(\lambda)$ are easily calculated to be $c_1 \lfloor \frac{\lambda}{a_r} \rfloor$. The next column is then calculated from top to bottom using the recursion. For example, $g_2(8) = \max\{g_1(8), 9 + g_2(8-4)\}$=$\max\{14, 9+9\}$.

Working back, we see that $p_4(10) = p_3(10) = 0$ and thus $x_4^* = x_3^* = 0$. $p_2(10) = 1$ and $p_2(6) = 0$ and so $x_2^* = 1$. $p_1(6) = p_1(3) = 1$ and thus $x_1^* = 2$. Hence we obtain an optimal solution $x^* = (2, 1, 0, 0)$.

An alternative approach[2] leading to the same result at a smaller algorithmic cost consist in using the recursion

$$g_r(\lambda) = \max\{g_{r-1}(\lambda), c_r + g_r(\lambda - a_r)\}, \tag{4.12}$$

[2]We thank Victorien Cornet for pointing this.

Table 4.2: $f_r(\lambda)$ for a 0-1 knapsack problem

λ	f_1	f_2	f_3	f_4	p_1	p_2	p_3	p_4
0	0	0	0	0	0	0	0	0
1	0	0	2	2	0	0	1	0
2	0	0	4	4	1	0	1	0
3	7	7	7	7	1	0	0	0
4	7	9	9	9	1	1	0	0
5	7	9	11	11	1	1	1	0
6	14	14	14	14	1	0	0	0
7	14	16	16	16	1	1	0	0
8	14	18	18	18	1	1	0	0
9	21	21	21	21	1	0	0	0
10	21	23	23	23	1	1	0	0

relying on the idea that the optimal solution to the problem for r types of objects and allowed weight λ is obtained by either discarding the possibility of selecting the object of type r and picking the optimal solution on $r-1$ types of object, or selecting one object of that type and the optimal solution to the same problem on the remaining allowed weight.

Nodes now have only two predecessors, resulting in a reduced computation cost for obtaining each $g_r(\lambda)$ (the precise comparison of the algorithmic costs linked to the two solutions presented above is left as an exercise).

4.5 Uncapacitated lot-sizing

An uncapacitated lot-sizing problem can be formulated as follows,

$$\begin{aligned} \min \sum_{t=1}^{n} p_t x_t + \sum_{t=1}^{n} h_t s_t + \sum_{t=1}^{n} f_t y_t & \\ s_{t-1} + x_t = d_t + s_t \ for\ t = 1, \ldots, n & \\ x_t \leq M y_t \ for\ t = 1, \ldots, n & \\ s_0 = s_t = 0,\ s \in \mathbb{R}_+^{n+1},\ x \in \mathbb{R}_+^n, y \in \mathbb{B}^n & \end{aligned} \tag{4.13}$$

in which $\{d_t\}_{t=1}^n$ are all the nonnegative demands, $\{p_t\}_{t=1}^n$, $\{h_t\}_{t=1}^n$ and $\{f_t\}_{t=1}^n$ are the production, storage and set-up cost respectively. Moreover, x_t denotes the production in period t, s_t the stock at the end of period t and y_t the action of activating the production process in period t.

The following proposition states that in an optimal solution, the stock is always empty when one starts producing. The intuition behind it is the following:

Proposition 4.5.1. *(i)There exists an optimal solution with $s_{t-1}x_t = 0$ for all t. (Production takes place only when the stock is zero.)*
(ii)There exits an optimal solution such that if $x_t > 0$, $x_t = \sum_{i=t}^{k} d_t$ for some $k \geq t$. (If production takes place in t, the amount produced exactly satisfies demand for periods t to k.)

Proof. Consider a solution. Let τ be the last time t at which s_{t-1} and x_t are both positive, i.e. at which there is a production, while the stock is not empty yet, and τ' the last time before τ at which the production was positive (such a time must exist since there is initially nothing stored). We consider the two modifications of the solution without changing the production times are fixed, i.e. we will not change the values of the y_t even if the production drops to 0 at t, and we only add production at times at which there already is a positive production. The two modifications are (i) increasing the production at time τ' (and the storage until τ') by an amount sufficient to replace all the production at time τ, and (ii) decreasing the production at τ' (and the storage until t') until either $s_{\tau-1} = 0$ (the stock is empty at time τ), or nothing is produced at τ'; $x_{tau'} = 0$. Observe that once the y_t are fixed, the problem is a linear program, and that the two modifications (i) and (ii) are feasible, and exactly in opposite direction. Hence at least one of them will not increase the cost (In most case, one of them should increase the cost and

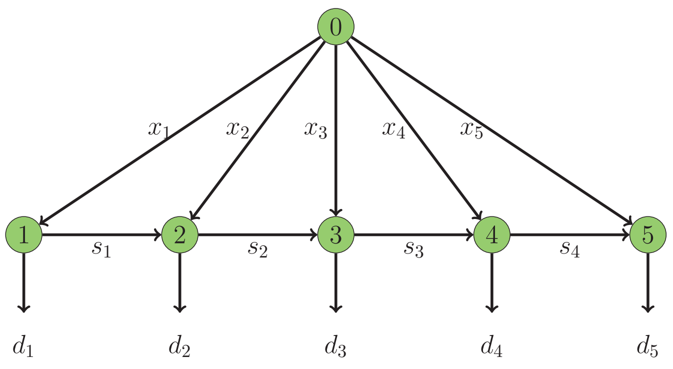


Figure 4.2: Network for lot-sizing problem

the other one decrease it). Further setting y_τ or $y_{\tau'}$ to 0 will then not increase the cost either. Hence we can build a solution with cost no larger than the initial one, and where $x_\tau = 0$, or $s_{\tau-1} = 0$, or $x_{\tau'} = 0$. Doing this reccursively (by selecting a smaller τ in the first two cases, or a smaller τ' in the latter one) eventually leads to a solution with $s_{t-1}x_t = 0$ for all t. The second statement follows immediately. □

As a consequence, building a solution can be seen as selecting the right production time, and producing then at each such time exactly what is needed to satisfy the demand until the next production time. This observation crucially relies on the absence of capacity constraints on storage and production.

Observation 4. *The problem formulation can be simplified: Let $d_{tk} := \sum_{i=t}^{k-1} d_i$. It follows from $s_{t-1} + x_t = d_t + s_t$ and $s_0 = 0$ that $s_t = \sum_{i=1}^{t} x_i - d_{1t}$. Hence the stock variables can be eliminated from the objective function giving $\sum_{t=1}^n p_t x_t + \sum_{t=1}^n h_t x_t = \sum_{t=1}^n p_t x_t + \sum_{t=1}^n h_t(\sum_{i=1}^t x_i - d_{1t}) = \sum_{t=1}^n c_t x_t - \sum_{t=1}^n h_t d_{1t}$, where $c_t = p_t + \sum_{i=t}^n h_i$. This allows us to work with the modified cost function $\sum_{t=1}^n c_t x_t + 0\sum_{t=1}^n s_t + \sum_{t=1}^n f_t y_t$, and constant term $\sum_{t=1}^n h_t d_{1t}$ must be subtracted at the end of the calculations.*

We will consider as set $\mathbb{P}$ the problems of uncapacited lot-sizing starting at time $t \le n$ (with empty storage), and call $H(t)$ the optimal cost of that problem. Consider such a problem and its optimal solution, and let k be the first time after t at which the production is positive ($x_k > 0$). Since there is nothing stored at the beginning at t, and nothing remaining at time k (by Proposition 4.5.1), the production at time t must be exactly $x_t = d_{tk} := \sum_{i=t}^{k-1} d_i$, i.e. exactly what will be consumed between t and $k-1$. The corresponding cost will then be $c_t d_{tk} + f_t$ (assuming d_{tk} is positive, otherwise f_t must be removed). Moreover, the situation at time k will then be exactly the same as if the problem had started at that time, so that the remaining cost would be $H(k)$. The total cost if the next production time is k is thus

$$c_t d_{tk} + f_t + H(k).$$

The next production time is of course a priori unknown, the equality above yields the following instantiation of (4.3):

$$H(t) = \min_{k>t}\{H(k) + f_t + c_t d_{tk}\}, \tag{4.14}$$

where we take as predecessor of $H(t)$ all $H(k)$ for larger times k (the number of predecessors grows thus linearly with the size of the problem). The last problem $H(n)$ does not have a predecessor, but is easy to solve as one just need to take $x_n = d_n$. Observe that the problem is thus solved by starting from the future and gradually moving back in time.

Remark Every feasible solution corresponds to a flow in the network shown in Figure 4.2, where p_t is the flow cost on the arc $(0, t)$, h_t is the flow cost on arc $(t, t + 1)$, and the fixed costs f_t are incurred if $x_t > 0$ on arc $(0, t)$. Thus ULS is a fixed charge network flow problem in which one must choose which arcs $(0, t)$ are open (which are the production periods), and then find a minimum cost flow through the network. It is always possible to find a minimum cost flow whose arcs with positive flows form a tree. In particular, at most one positive flow arrives at every node t, which means that s_{t-1} and x_t cannot be both positive in such a solution. This provides an other proof of Proposition 4.5.1.

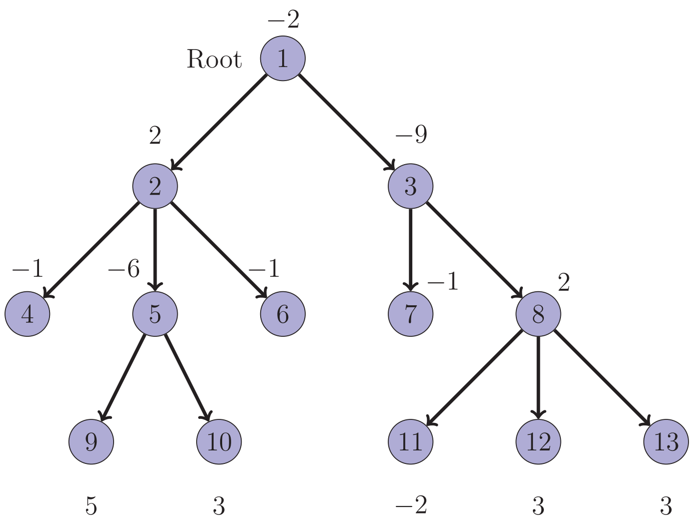


Figure 4.3: Rooted tree with node weights c_v

4.6 An optimal Subtree of a tree

The optimal subtree of a tree problem involves a tree $T = (V, E)$ with a root $r \in V$ and weights c_v for $v \in V$. Weights can also negative. The problem is to choose a subtree of T rooted at r of maximum weight, or the empty tree if there is no positive weight rooted subtree.

We will take as $\mathbb{P}$ the set of problems of finding the optimal subtree $T(v)$ rooted at node v, for every v. We define as $N^+(v)$ the successors of node v, i.e. its out-neighbors. Let us start with a tautology: $T(v)$ is either the empty tree in which case $C(T(v)) = 0$, or is nonempty. Observe that if $T(v)$ must in the latter case contain an optimal subtree $T(w)$ rooted at w for every one of its successors $w \in N^+(v)$. Otherwise, replacing the subtree of $T(v)$ in the direction of w by $T(w)$ would increase the value of $T(v)$, which is impossible. Hence there holds

$$C(T(v)) = \max\{0, c_v + \sum_{w \in N^+(v)} C(T(w))\}. \tag{4.15}$$

where C is the cost function. We can therefore define $R_{T(v)}$ by saying that computing $T(w)$ is a predecessor of the problem of computing $T(v)$ if $w \in N^+(v)$. The relation (4.15) instantiates a variation of (4.3) (with a maximization instead of a minimization, where the solution for v is computed directly based on all predecessors as opposed to by taking the minimum of over all the predecessors). The problem $T(v)$ for the leaves nodes have no predecessors, but are easy to compute: $T(v) = \{v\}$ if the benefit is positive and 0 else.

The dynamic programming solution of the problem will therefore consists in beginning to compute the solution for the leaves, and then building solutions for higher and higher nodes, eliminating every subtree $T(v)$ encountered for which $C(T(v)) = 0$. Finally note that each of the terms c_v and $T(v)$ occurs just once on the right-hand side during the recursive calculations, and so the algorithm is complexity is $O(n)$.

Example 5.1 : For the instance of the optimal subtree of a tree problem shown in Figure 4.3 with root $r = 1$, we start with the leaf nodes $T(4) = T(6) = T(7) = T(11) = \{\}$, $C(T(9)) = 5$, $C(T(10)) = 3$, $C(T(12)) = 3$, and $C(T(13)) = 3$. Working in, $C(T(5)) = \max[0, -6 + 5 + 3] = 2$ and $C(T(8)) = \max[0, 2 + 0 + 3 + 3] = 8$. Now the values of $C(T(v))$ for all successors of nodes 2 and 3 are known, and so $C(T(2)) = 4$ and $C(T(3)) = 0$ can be calculated. Finally $C(T(1)) = \max[0, -2 + 4 + 0] = 2$. Cutting off subtrees $T(3), T(4)$, and $T(6)$ leaves an optimal subtree with nodes $1, 2, 5, 9, 10$ of value $C(T(1)) = 2$.

Chapter 5

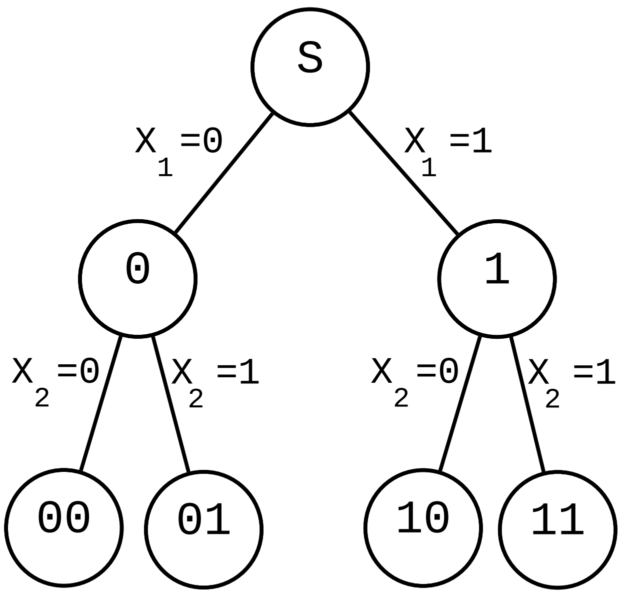
Branch and Bound

5.1 Introduction and Background

A brute force enumeration of all feasible solutions in order to pick the optimal one will typically take an exponential number of operations. A branch-and-bound algorithm can be seen a clever enumeration of all feasible cases.

Example: *Enumeration tree*

Let $x_1, x_2 \in \{0, 1\}$ be binary variables. We might construct the enumeration tree like this :



Remark : At the end, the leaves are all the feasible solutions.

A branch-and-bound algorithm consists in applying a divide-and-conquer strategy on a tree enumerating all solutions, to avoid having to explore the whole tree whenever possible. It relies on two principles, that we will expose on the following generic optimization problem, denoting z the optimal value.

$$Z = \max_{x \in S} f(x) \tag{5.1}$$

Suppose we can find primal (lower) bound $\underline{z}$ and dual (upper) bound $\bar{z}$ for the optimal value [1] z. If these bounds are equal or close enough, then we can consider the problem solved.

Otherwise, the first principle is to seek a better approximation of z by splitting the feasible set into two parts S_1, S_2 or more, and put together the information given by optimizing over those smaller feasible sets, with the hope that the upper and lower bounds on S_1, S_2 will be more accurate than those on S. (For finite problems, this is eventually always the case if one keeps splitting the set, as one eventually

[1] Be careful ! In case of minimization problem, primal bound is an upper bound and dual bound is a lower bound.

is left with sets containing single elements). The formalization of this approach relies on the following proposition:

Proposition 1. *Consider the problem*

$$z = \max_{x \in S} f(x).$$

Let $S = S_1 \cup S_2 \cup \cdots \cup S_n$ be a partition of S. Suppose moreover that

$$\underline{z}_i \leq \max_{x \in S_i} f(x) \leq \bar{z}_i.$$

Then there holds

$$\max_i \underline{z}_i \leq z \leq \max_i \bar{z}_i.$$

Proof. This trivially follows from the definition of the maximum. □

We remind the reader that our hope will be to obtain values $\underline{z}_i$ and $\bar{z}_i$ leading to tighter bounds than what could be directly obtained on S.

The second principle of the Branch and Bound method is the possibility to cut branches, that is, to discard sets from which we know we will not get any relevant information. The following proposition is also an immediate consequence of the notion of maximum.

Proposition 2.
(a) If $\bar{z}_i < \underline{z}_j$ for some $j \neq i$, then there is no optimal solution of the initial problem in S_i. ($\max_{x \in S} f(x) = \max_{x \in S \setminus S_i} f(x)$.)

(b) If $\bar{z}_i = \underline{z}_j$ for some $j \neq i$, then there exists an optimal solution of the initial problem that is not in S_i. ($\max_{x \in S} f(x) = \max_{x \in S \setminus S_i} f(x)$.)

(c) If $\bar{z}_i = \underline{z}_i$, then the feasible solution leading to $\underline{z}_i$ is an optimal solution of the sub-problem over S_i.

Whenever we are in condition (a), we can thus discard the set S_i. If we are in conditions (b), we may discard it provided that we are not simultaneously discarding j. If we are in conditions (c), we can consider the problem as solved over S_i.

Before moving to the algorithm, we note that some methods to find feasible solutions (and thus lower bounds) do not always provide a result. We will model this situation as obtaining solution of trivial cost $-\infty$.

5.2 Conceptual algorithm

We now present a conceptual generic version of the Branch and Bound approach (several equivalent descriptions are possible). For this purpose we will represent subsets S_i of S by nodes. We will maintain two sets of nodes, that are not necessarily disjoint:

Active nodes: This will be a set of nodes i such that the union of the S_i contains at least one optimum of the initial problem. (But there might be other optimum points in non-active nodes):

$$\max_{x \in \cup_{i \in A} S_i} f(x) = \max_{x \in S} f(x) \tag{5.2}$$

Closed (Solved) nodes: Those are nodes i for which the problem $\max_{x \in S_i} f(x)$ is solved, i.e. we have a feasible solution $\underline{z}_i = \bar{z}_i$ and which do not need to be explored anymore.

At the risk of being redundant, we observe these sets divide thus the nodes in four categories

- Active non-closed: The corresponding problem is not solved yet, its solution could potentially be the solution to the initial global problem. It should remain under consideration and may need to be further investigated.

- Active closed: The corresponding problem is solved, and its solution could potentially be the solution to the initial problem. It should remain under consideration, but need not be further investigated.
- Inactive non-closed: The corresponding problem is not solved, but its solution cannot be the solution to the initial problem (or, possibly, a solution at least as good as already been found). It does not need to remain under consideration. There is also no need to further investigate it, as finding its solution would be useless. The hope is to find many such nodes, which would allow to rapidly narrow the search space.
- Inactive closed: The corresponding problem is solved but its solution cannot be the solution of the initial problem (or, possibly, a solution at least as good as already been found). It does not need to remain under consideration nor to be further investigated. We observe that the problem should have been solved before we realized the node could be closed, for investing in finding the solution of an inactive problem is useless.

(0) Initialization The algorithm is initialized by computing lower and upper bound $\underline{z}$ and $\bar{z}$, and setting a node 0 corresponding to the set $S_0 = S$ in the set of active nodes. (If $\underline{z} = \bar{z}$, one stops of course there). The invariant (5.2) clearly holds, and there exists an active non-closed node.

We then proceed to the main iteration, which should be pursued until (i) no active non-closed node is found (which will see implies $\underline{z} = \bar{z}$) or (ii) one has found a feasible solution whose cost $\underline{z}_i$ is sufficiently close to the upper bound $\bar{z}$. (Or in some cases, until the computation time exceeds a pre-determined time budget.)

(1) Branching: Select an active non-closed node i and split it into $i_1, i_2, \dots i_{n_i}$ by partitioning S_i in non-intersecting sets $S_i = S_{i_1} \cup S_{i_2} \cup \dots \cup S_{i_{n_i}}$. Add these i_k to the set of active nodes A, and remove i from it. Since $S_i = \cup S_{i_k}$, the invariant (5.2) remains valid.

(2) Local Bounding: Compute upper and lower bounds $\bar{z}_{i_k}$ and $\underline{z}_{i_k}$ for all nodes i_k of the partition. Add any node i_k for which $\bar{z}_{i_k} = \underline{z}_{i_k}$ to the set of closed nodes. This does not affect (5.2).

(3) Global bounds updates: Using Proposition 1, we update the general bounds by

$$\bar{z} = \max_{j\in A} \bar{z}_j \text{ and } \underline{z} = \max_{j\in A} \underline{z}_j,$$

(4) Pruning/Cutting: We remove from the set A every node i for which $\bar{z}_i \leq \underline{z}$, i.e. every node for which we are sure that the corresponding optimum is smaller than a feasible solution already obtained. The invariant (5.2) remains valid, since for every such node i there holds

$$\max_{x\in S_i} f(x) \leq \bar{z}_i < \underline{z} \leq \max_{x\in S} f(x).$$

We *may* also remove from A all nodes i with $\bar{z}_i = \underline{z}$ provided that we keep in A at least one node j for which $\underline{z}_j = \underline{z}$. The invariant (5.2) remains again valid. Such nodes might indeed contain an optimal solution, but this solution would in that case be equivalent to a feasible solution already found. If the goal is just to find one optimal solution, they should be removed. But one may sometimes be interested in finding several optimal solutions, in which case they can be kept. Note that this operation does not affect $\max_{i\in A} \bar{z}_i$ or $\max_{i\in A} \underline{z}_i$.

Main Iteration: Observe that in case every active node is closed, i.e. if $\underline{z}_i = \bar{z}_i$ for every $i \in A$, then

$$\underline{z} = \max i \in A \underline{z}_i = \max i \in A \bar{z}_i = \bar{z},$$

which means that we have found a feasible solution whose value reaches the upper bound on the optimal value, and is therefore optimal. In that case, the algorithm should stop. Otherwise there exists an active node that is not closed, and we can iterate. (From a practical point of view, we should check if $\underline{z} = \bar{z}$ at step 3).

Observe that the approach above requires making several choice. In particular,

- **How to pick the active node i in step (1)?** Several strategies are possible, and their selection may depend on the specific problem considered and of the practical context. In some cases, one will for example prefer to first select nodes that have just been created in order to get very small sets on which good feasible solutions can be obtained. Other possible strategies involve e.g. favoring sets with the highest upper bounds.

- **How to compute the bounds $\underline{z_i} \leq z \leq \bar{z}_i$? With which precision to compute them?** A classical solution to obtain a lower bound is to find an arbitrary feasible solution, if possible a good one. Not that $-\infty$ is always a trivial lower bound. Hence it is possible to try finding a lower bound by a heuristic method, and use $-\infty$ if it fails. For the upper bound, one can solve a relaxation of the problem that is easier than the actual problem.

- **How to split the set S_i into subsets?** If a relaxed problem was use for the upper bound, a good approach consists in ensuring that the subsets all exclude the optimal solution of that relaxed problem, this avoids re-finding the same upper-bound after the new step. Note that the partition $S_i = S_{i_1} \cup S_{i_2} \cup \dots$ need not be a real partition in the sense that overlaps are allowed. However, it is usually better to avoid them as they may lead to consider a same solution several times.

We also note that there are many practical features that can be added to this conceptual algorithm to make it more efficient. The computation of the local bound may for example be stopped as soon as one realize that $\bar{z}_{i_k}$ is lower than some feasible solution, so that the node will be certainly be pruned in step (4). And one can in that case actually immediately prune the node. Similarly, one can update the global bounds (3) after each computation of a local bound, or stop the whole algorithm immediately in the fortunate case where a feasible solution with value $\bar{z}$ has been found. Also, one does not necessarily need to compute the bounds for all nodes simultaneously in step (2) or may begin by computing certain lower bounds or certain upper bounds, provided that the computation of $\bar{z}$ and $\underline{z}$ are adapted.

The branch and bound process is conveniently represented by a planted tree, whose root is the node corresponding to the set S, whose leaves are the actives nodes, and where the children nodes of any non-leaf nodes represent a partition of its set. The operation of branching consist then in adding children nodes. This allows for a sometimes more efficient update of the global bounds: every node i, whether active or non-active, maintains bounds $\bar{z}_i, \underline{z_i}$. When one computes the bounds for a leaf node j, one then updates the bounds of its parent node i by

$$\bar{z}_i = \max_{k:i\to k} \bar{z}_k \text{ and } \underline{z_i} = \max_{k:i\to k} \underline{z_j}.$$

If this modifies one of the bound, one then iterates the process for the parent of that parent node, etc. until reaching the root. The pruning process also has a nice interpretation: if a nodes i, whether a leaf or not, satisfies $\bar{z}_i \leq \underline{z}$, it is removed from the three together with all its children and all the nodes below them. Finally, the branching of a node often consist in fixing variables and/or adding constraints. This is conveniently represented on the edges of the tree, as for example in Figure 5.3.

5.3 Specific branch-and-bound for IP

Suppose we have the following general IP

$$z = \max \left\{ c^T x \mid x \in S \right\} \qquad \text{with } S = P \cap \mathbb{Z}^n \text{ and } P \text{ a polyhedra.} \tag{5.3}$$

We can get a specific branch-and-bound algorithm by doing the following :

Upper bound: We solve the linear relaxation of the problem, removing the integrity constraint $x \in \mathbb{Z}^n$. (One can of course solve a less relaxed problem if some integrity constraints can easily be incorporated).

Lower bound: The idea to find a feasible point $\underline{z} \in \{c^T x : x \in S\}$. This may be problem dependent. In the absence of any alternative method, one can take the solution of the linear relaxation if it happens to be integer, and $-\infty$ otherwise.

Branching: Let $X^* = \{x^* \in P : c^T x^* = \bar{z}\}$ be the set of optimal points for the linear relaxed problem[2]. If we can find x^* an integer vector that satisfy X^*, then we are lucky because x^* is also a solution of the IP. If not, we split S following

$$\begin{cases} S_1 = \{x : x \in S,\ x_i \leq \lfloor x_i^* \rfloor\} = P_1 \cap \mathbb{Z} \\ S_2 = \{x : x \in S,\ x_i \geq \lceil x_i^* \rceil\} = P_2 \cap \mathbb{Z} \end{cases} \quad \text{for only one } i \text{ such that } x_i^* \notin \mathbb{Z} \tag{5.4}$$

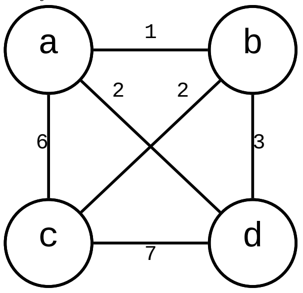
It is obvious that $S_1 \cup S_2 = S$ and that x^* is not in S_1 and S_2. Moreover, the optimal value z_1 and z_2 are strictly below $\bar{z} = c^T x^*$.

Note that Although x^* is not feasible yet for P_1 and P_2, the associated dual variable u^*, optimal in P^{dual}, is still feasible but no more necessarily optimal for P_1^{dual} and P_2^{dual}. So we can use it to have a good starting point for dual simplex algorithm to find an upper bound for respectively z_1 and z_2.

5.4 Complete example : TSP with four cities

Consider the following complete weighted graph $G(U, V)$ described by matrix A.

$$A = \begin{bmatrix} - & 1 & 6 & 2 \\ 1 & - & 2 & 3 \\ 6 & 2 & - & 7 \\ 2 & 3 & 7 & - \end{bmatrix}$$



We have to find a minimum weight Hamiltonian cycle by using the branch-and-bound algorithm. Thus we must define how to compute a primal (upper) bound and a dual (lower) bound.

Branching method: In order to split the solution space, we will force certain edges to belong to the solution and/or certain other edges not to belong to the solution. This will be done in a way that excludes the optimal solution to the relaxation used (see below). Moreover, we will take into account the fact that as soon as we force two edges incident on a same node, we can remove all other edges, since the TSP path can only contain two edges incident on a same node.

Primal upper bound We can simply choose a “good" Hamiltonian cycle. Hamiltonian cycle is NP-hard in general, but (i) it is usually much simpler than TSP, and (ii) as long as the graph is “sufficiently connected" a Hamiltonian cycle can easily be found. Now, due to the branching method chosen, we need to be able to find a Hamiltonian cycle that does not contain certain forbidden edges and that must contain certain forced edges. Forbidden edges can just be removed from the graph. Forced edges are more challenging, but it will in most case be easy to find such a Hamiltonian cycles.

Dual lower bound We will take the optimal solution of a relaxation that is simple to solve. Let's consider a specific node u^*. Observe that the optimal traveling salesman tour can be seen as a tree on $V \subset \{u^*\}$ (actually a very specific “path" tree) together with two edges from u to the tree. Hence, the optimal tour is a feasible solution of the following problem: “*Find the minimal weight object consisting of a tree on $V \subset \{u^*\}$ and two edges connecting u^* to the tree*", which is thus a relaxation of the TSP. Moreover, the optimal solution of this problem is actually easy to compute: one just needs to take the optimal spanning tree on $V \subset \{u^*\}$ and the two cheapest edges leaving u^*. Due to the branching method, we must again be able to enforce the presence/absence of certain edges, but the optimal spanning tree algorithm can easily be adapted to such constraints.
Note that this relaxation is not the unique possible one. One could for example also find the minimum cost subgraph of the form “spanning tree + 1 edge".

Step 1 : set S

[2]Also called LP-relax : original problem without integer constraint $x \in \mathbb{Z}^n$

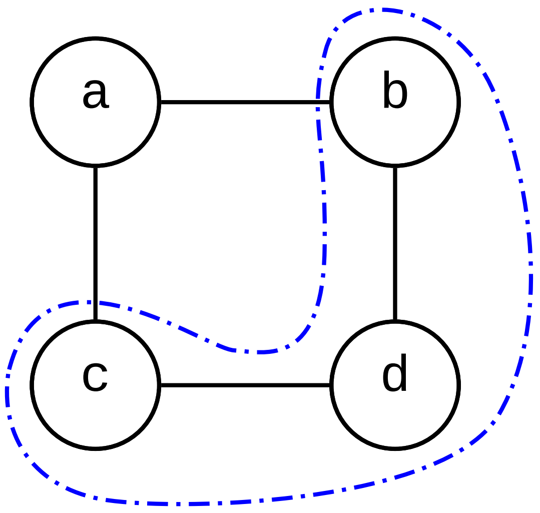


Figure 5.1: If we remove node a in a specific Hamiltonian path, we have a particular kind of tree.

Upper bound We choose for example the Hamiltonian cycle $[abdc]$. So, we have the upper bound $\bar{z} = 1 + 3 + 7 + 6 = 17$.

Lower bound We take node a as node to be removed (i.e. as u^*) and compute the minimum spanning tree on the remaining graph. The total cost of this tree is $2 + 3 = 5$. We add two minimum weighted edges from a to the tree : (ad) and (ab). Thus, the lower bound $\underline{z}$ is equal to $5 + 2 + 1 = 8$. We can see that the relaxed solution is not a TSP solution because degree $b = 3$. So it provides a lower bound on the cost, but not an actual feasible solution (in which case we would have been done).

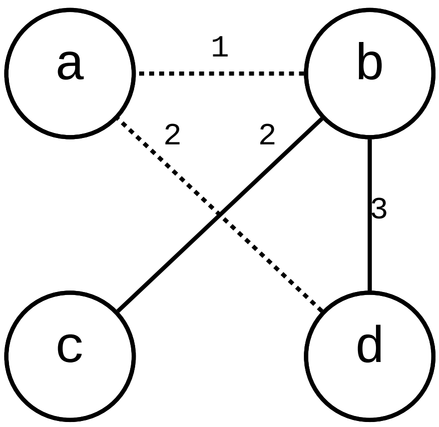


Figure 5.2: Lower bound for the step 1 : minimum spanning tree in full line, and the two added vertex are dotted.

Branching: We now split S into three different subsets S_1, S_2, S_3. The subset S_1 is the set S without edge (ab). In the subset S_2 we force the presence of (ab) and exclude (bc). Finally, in S_3 we force $(ab), (bc)$ and this allows excluding (bd) (and any other edge that would have been incident on b) since b is already incident on two edges and all nodes in the optimal tour have a degree 2. Observe these choices indeed guarantee that the optimal solution of our relaxed problem is excluded from the three sets, making a strict improvement of the upper bound $\bar{z}$ very likely. Note also that $S_1 \cap S_2$, $S_1 \cap S_3$ and $S_2 \cap S_3$ are empty by construction.

Step 2 : subset S_1

Upper bound The previous Hamiltonian cycle $[abdc]$ cannot be taken here, because edge (ab) is excluded. We have to take another cycle, for example $[adbc]$ which is feasible in S_1. The total weight of

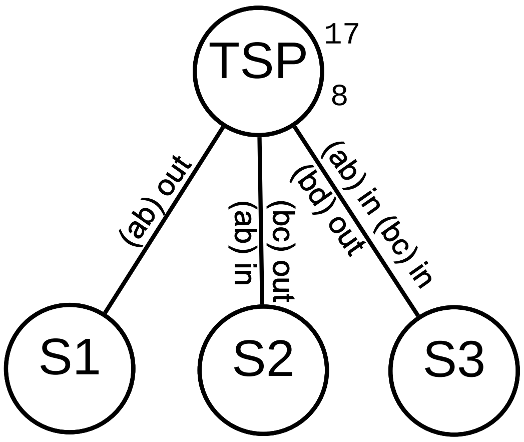


Figure 5.3: Branching of first set S into tree subsets without optimal solutions of the relaxed problem.

this cycle is equal to $2+3+2+6=13$ and we have $\bar{z}_1 = 13$.

Lower bound The cost of minimum spanning tree without a is equal to $2+3=5$. The two minimum edges from a to the tree are (ac) and (ad) because (ab) is out. In conclusion, the total weight is equal to $2+6+5=13$ and the lower bound $\underline{z}_1$ is also equal to 13. We note that $\bar{z}_1 = \underline{z}_1$, so the optimal value z_1 is equal to 13.

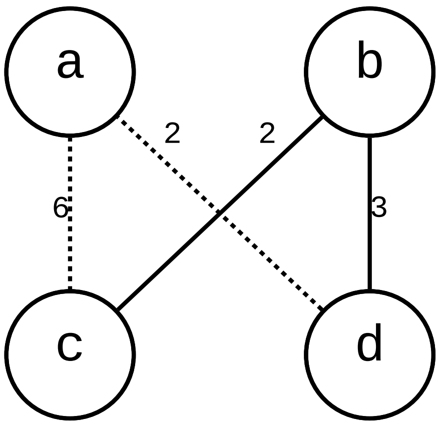


Figure 5.4: Lower bound for the step 2 : minimum spanning tree in full line, and the two added edges are dotted.

Conclusion: The node is now closed as we have a solution matching the lower bound for problem S_1

Step 3 : subset S_2

Upper bound The previous Hamiltonian cycle $[abdc]$ remains feasible : $\bar{z}_2 = 13$.

Lower bound The cost of minimum spanning tree without a is equal to $3+7=10$, by (bd) and (cd) because (bc) is out. For the edges leaving a, w must take (ab), and the cheapest is to add (ad), leading to a weight 3. Hence the total coast is $1+2+10=13$. Observe this cost is equal to that of the feasible solution, but the solution used for the lower bound is not a Hamiltonian path.

Conclusion We have a feasible solution whose cost equals a lower bound. The node is thus closed. (We may also make it inactive since we already have a solution with the same cost).

Step 4 : subset S_3

Upper bound: We cannot take the Hamiltonian cycle $[abdc]$ because (bd) is excluded. Also we must take (ab) and (bc). Hence we can take $[abcd]$ (this is actually the only solution). The weight of this cycle and the upper bound $\bar{z}_3$ are equal to $1+2+7+2=12$.

Lower bound: The spanning tree on b, c, d must contain (bc) and cannot contain (bd), hence the only solution is $(bc), (cd)$, with a cost $2+7=9$. For the edges leaving a, we must take (ab), and the cheapest solution is to add (ad), leading to a cost 3. Hence the lower bound $\underline{z}_3$ is $3+9=12$.

Conclusion: We have a feasible solution whose cost matches the lower bound. The node is closed.

Branch Cutting

We can update the global upper bound to 12 and the lower bound to 12. Hence the optimal solution has been found as the solution of S_3: $z = z_3 = 12$, and $x^* = [abcd]$ achieves z. Note that if for some reason the lower bound $\underline{z}_3$ for S_3 had been smaller than 12, then we would have updated the global upper bound $\bar{z}$ to $12 = \min(\bar{z}_1, \bar{z}_2, \bar{z}_3)$, and the lower bound $\underline{z}$ to $\underline{z}_3 = \min(\underline{z}_1, \underline{z}_2, \underline{z}_3)$. This would have allowed cutting S_1 and S_2 since their lower bound (best cost one could hope to achieve) would have been larger than the upper bound (cost of a solution already found) on S_3.

Chapter 6

Valid inequalities and Cutting planes

6.1 Introduction: The different polytopes

Consider a standard linear integer program

$$\begin{aligned} &\max c^T x \\ \text{s.t. } &x \in \mathbb{X} \subset \mathbb{Z}_+^n \end{aligned} \tag{6.1}$$

We typically represent $\mathbb{X}$ by the intersection between a polytope $\{x : Ax \leq b\}$ and the nonnegative integers $\mathbb{Z}_+^n$ (From a practical point of view, we don't have access to $\mathbb{X}$, and the modeling step will in most case give such a polytope). As represented in Figure 6.1 and already discussed in Sections 1.2 and 1.3, a set $\mathbb{X}$ may be represented by different polytopes, which can have very different complexities, i.e. number of constraints describing them.

Definition The *convex hull* of $\mathbb{X}$, conv($\mathbb{X}$) can be equivalently defined as (i) the smallest polytope whose intersection with Z_+^n defines $\mathbb{X}$, (ii) the intersection of all such polytopes, or (iii) explicitly as the set of all convex combinations[1] of the elements of $\mathbb{X}$ that we enumerate by $x^{(1)}, x^{(2)}, \ldots, x^{(|\mathbb{X}|)}$ (assuming here that there is a finite number of them):

$$\text{conv}(\mathbb{X}) := \{x : \sum_{t=1}^{|\mathbb{X}|} \lambda_t x^{(t)} : \lambda_t \geq 0, \sum_{t=1}^{|\mathbb{X}|} \lambda_t = 1\}.$$

The proof of equivalence between the three definition is left as an (not necessarily trivial) exercise for the interested readers. One can also show that the vertices of conv($\mathbb{X}$) are all elements of $\mathbb{X}$. As a consequence, the solution of the “relaxed" linear program

$$\max c^T x \text{ s.t. } x \in \text{conv}(\mathbb{X}),$$

[1]i.e. weighted averages

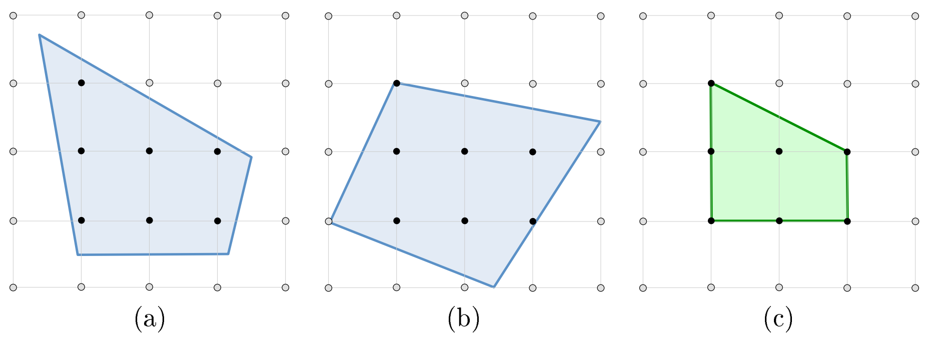


Figure 6.1: Example of three polytopes whose intersections with the integers yields the same set $\mathbb{X}$. The polytope (c) is the convex hull of $\mathbb{X}$.

is always an element of $\mathbb{X}$, which implies that it is an actual optimal solution to (6.1). The convex hull is thus the best polytope in that sense, as it allows solving the integer program simply by solving the linear relaxation. Unfortunately, obtaining conv($\mathbb{X}$) can be computationally very costly. Furthermore, simply describing conv($\mathbb{X}$) may require an exponential number of linear constraints.

One should thus in general give up hope to simply obtain conv($\mathbb{X}$). Nevertheless, not all other polytopes are equal, and using certain polytopes will lead to significant improvement of the upper bounds obtained by linear relaxation and/or the the Branch and Bound algorithm.

In this chapter we will see methods to improve the quality of polytopes by reducing them. This will be achieved by adding constraints that are satisfied at every integer point of the polytope.

We stress in particular that polytopes with concise descriptions, which are aesthetically more appealing, typically lead to worse polytopes. A higher number of constraints gives more flexibility to “stick" to $\mathbb{X}$. Thus, there will be a trade-off between two objectives, (i) obtaining polytopes as close as possible to conv($\mathbb{X}$) to improve the quality of the linear relaxations and (ii) keeping the complexity of those polytopes low as to control the computational burden of solving the LP and of manipulating these polytopes.

6.2 Valid inequalities

We begin by defining certain relations between inequalities and the sets $\mathbb{X}$. Below, inequalities have the general form $\pi x \leq \pi_0$, where π is a row vector.

Definition An inequality $\pi x \leq \pi_0$ is **valid** for a set $\mathbb{X}$ if it holds for every $x \in \mathbb{X}$

Definition An inequality $\pi x \leq \pi_0$ **dominates** an inequality $\mu x \leq \mu_0$ if

$$\{x \in \mathbb{R}^n_+ : \pi x \leq \pi_0\} \subseteq \{x \in \mathbb{R}^n_+ : \mu x \leq \mu_0\}$$

i.e. every nonnegative x satisfying the former also satisfies the latter (without consideration for integrity).

Definition Given a set of inequalities $\pi^j x \leq \pi^j_0$, $j = 1, \dots, m$, we say that $\pi x \leq \pi_0$ is **redundant** if there exists positive scalar $u^1, u^2, \dots, u^m$ such that $\pi x \leq \pi_0$ is dominated by

$$\left(\sum_{j=1}^{m} u^j \pi^j\right) x \leq \sum_{j=1}^{m} u^j \pi^j_0,$$

i.e. if it is dominated by a positive combination of the other constraints considered.

Our goal will be to create new valid inequalities.

6.3 Use of the valid inequalities

A few techniques to improve the description of polytopes are presented here.

1. **Static:** One approach is to first improve the polytope description, and then to run a standard Branch-and-Bound algorithm. The treatment of the polytope could thus be seen as part of a pre-processing step. This allows the use of a standard branch and bound algorithms. However, we run the risk of adding many constraints that serve no real purpose for the specific optimization problem at stake and burdening the computation of the linear relaxation. This would particularly the case if we spend time and effort adding constraints in a zone of the polytope where the objective is much lower than the optimal values (for a maximization problem) or far from the zone where the optimums of the linear relaxations are.

2. **Dynamic, cutting plane:** The idea here is to solve an IP by solving LP relaxation that are improved at each iteration. This technique is done by iterating over the following steps.

```
Solve the LP relaxation;
    If the LP solution $x^*$ is integer, stop;
    Else add a valid inequality cutting $x^*$, i.e. s.t. $\pi x^* > \pi_0$. (but with $\pi x \leq \pi_0$ for all $x \in \mathbb{X}$
    since the inequality is valid).
```

The advantage of this method is that it will generate inequalities specifically in the zone of the optimum of the linear relaxation. We will see that we can always find a constraint of the form given as example although that constraint may be relatively complex.

3. **Dynamic, Branch and Cut:** The idea here is to combine the cutting plane idea with the Branch and Bound, i.e. instead of just solving an LP when considering a node of the Branch and Bound, we perform a few iteration of the cutting plane algorithm.

Practical implementation of these ideas require balancing several trade-offs and needs to be parametrized (i.e. how many cutting plane steps has to be done in the branch and cut). These choices may be problem specific and often rely on heuristic ideas. We will also see that we may have to solve auxiliary optimization problems in the process.

6.4 Chvátal-Gomory procedure

We begin with an example. Suppose that $\mathbb{X}$ is defined by

$$\begin{aligned} 7x_1 - 2x_2 &\leq 14 \\ x_2 &\leq 3 \\ x_1, x_2 &\in \mathbb{Z}_+ \end{aligned}$$

Any linear combination with positive coefficient of these two inequalities is valid. In particular, multiplying the first one by $\frac{1}{7}$ and the second one by $\frac{37}{126}$ leads to

$$x_1 + \frac{1}{126}x_2 \leq \frac{121}{42}.$$

Since $x_1, x_2 \geq 0$, decreasing the coefficients leads to a weaker inequality, which is also valid. In particular, we can round them to the nearest integer below them, which yields

$$x_1 + 0x_2 \leq \frac{121}{42}.$$

Observe now that x_1 is also an integer. Hence if it is smaller than or equal to $\frac{121}{42} \simeq 2.88$, it must be no larger then 2, hence

$$x_1 \leq 2.$$

The reader can easily verify that adding this inequality restricts the polytope.

Let us know describe the general procedure. Suppose that a polytope P is described by $\{x \in \mathbb{R}_n^+ : Ax \leq b\}$ and that $\mathbb{X} = P \cap \mathbb{Z}_n^+$. Denote the n columns of A by $a_1, a_2, \dots, a_n$, so that P can also be described as

$$P := \{x \in \mathbb{R}_n^+ : \sum_{j=1}^{n} a_j x_j \leq b\}, \tag{6.2}$$

where the inequality has to be interpreted component by component and we remind that each a_j is a column vector. The Chvátal Gomory procedure consists in selecting a nonnegative vector $u \in \mathbb{R}_+^n$ and applying the next three steps:

1. *Positive combination of constraints:* It follows from (6.2) and the nonnegativity of u that the positive combination of constraints $u^T Ax \leq u^T b$ is valid for P and hence for $\mathbb{X}$. Observe that it can be rewritten as

$$\sum_{j=1}^{n}(u^T a_j)x_j \leq u^T b \tag{6.3}$$

where we remark that $u^T a_j$ and $u^T b$ are scalar.

2. *Rounding of coefficients:* Since all x_j are nonnegative, decreasing the value of the coefficient can only weaken inequality (6.3). In particular, the following inequality

$$\sum_{j=1}^{n}\lfloor u^T a_j \rfloor x_j \leq u^T b, \tag{6.4}$$

obtained by rounding all multiplicative coefficients to the largest integer below, is weaker than (6.3), and is thus valid for P and $\mathbb{X}$.

3. *Rounding of independent terms:* The previous steps generated an inequality that is valid for P. This step is valid for $\mathbb{X}$ (and not P), as it exploits the integrity of the x_j. Observe that since coefficients $\lfloor u^T a_j \rfloor$ are integer and the x_j in $\mathbb{X}$ are integer, the left hand of (6.4) is integer for every $x \in \mathbb{X}$. Hence it will be smaller than or equal to $\lfloor u^T b \rfloor$, the largest integer that is no greater than the right-hand side of (6.4). The following inequality is thus valid for $\mathbb{X}$

$$\sum_{j=1}^{n} \lfloor u^T a_j \rfloor x_j \leq \lfloor u^T b \rfloor. \tag{6.5}$$

This is a simple way of adding new constraints valid for $\mathbb{X}$. The quality of the results obtained strongly depends on the choice of u. In particular, a poor choice of u may lead to constraints that do not decrease P. Observe that step (2) decreases the quality of the bound, while step (3) increases it. Hence one should look for vectors u for which the decrease of coefficients in (2) is as small as possible and the decrease of independent terms in (3) as large as possible.

Theorem 6.4.1. *Every inequality valid for $\mathbb{X}$ can be obtained by applying the procedure (1)-(3) finitely many times from P. In particular, conv($\mathbb{X}$) can be obtained by applying this procedure finitely many times.*

This result shows that one could in principle solve the problem by generating sufficiently many valid inequalities until obtaining conv($\mathbb{X}$) and then solve the corresponding LP. However, this procedure is very slow and may introduce many redundant constraints. This make this technique unusable in practice.

6.5 Gomory cutting plane

We now move to a cutting-plane algorithm. The idea is to first solve the LP relaxation, and then to find a valid inequality that excludes the optimal solution x^* of that relaxation. This method will thus create inequalities in “relevant" parts of the set $\mathbb{X}$.

By an appropriate change of variables or introduction of slack variables, one can always write a linear program as

$$\begin{aligned} \max \quad & c^T x \\ & Ax = b \\ & x \geq 0 \end{aligned} \tag{6.6}$$

The equality constraint $Ax = b$ consists of m constraints (that we will suppose here linearly independent), while the constraint $x \geq 0$ consists of n constraints. The optimum will make (at least) n constraints tight, out of which

- the m constraints from $Ax = b$, which have to be tight by definition,
- $n - m$ remaining constraints from $x_i = 0$.

This means that $n - m$ elements of x^* are zero, and (at most) m are nonzero. Those m (potentially) nonzero elements are called the basis. Separating the variables according to the basis (B) and the “non-basis" (NB), we can rewrite the equality constraints of the LP as:

$$\begin{pmatrix} A_B & A_{NB} \end{pmatrix} \begin{pmatrix} x_B \\ x_{NB} \end{pmatrix} = b,$$

or equivalently

$$A_B x_B + A_{NB} x_{NB} = b.$$

Assuming A_B is invertible (this is the case without loss of generality, although establishing this requires treating some technicalities), we can further rewrite this as

$$x_B + \underbrace{A_B^{-1} A_{NB}}_{\widetilde{A}_{NB}} x_{NB} = \underbrace{A_B b}_{\widetilde{b}},$$

So that the LP is equivalent to

$$\boxed{\begin{array}{ll} \max & c^T x \\ s.t. & x_i \geq 0 \\ & b_i \geq 0 \\ & x_B + \widetilde{A}_{NB} x_{NB} = \widetilde{b} \end{array} \qquad (6.7)}$$

Remember now that variables not in the basis are by definition zero at the optimum. Hence the following holds.

$$\begin{array}{l} x^*_{NB} = 0 \\ x^*_B + 0 = \widetilde{b} \\ x^*_B \geq 0 \end{array}$$

We now consider two cases.

1. Suppose first that x^*_B is an integer vector. Then so is $x^* = \begin{pmatrix} x^*_B \\ x^*_{NB} \end{pmatrix}$, which belongs to $\mathbb{X}$. x^* is then a solution to the initial IP problem.

2. On the other hand, if x^*_B is not integer, then there exist at least one index i such that $\tilde{b}_i$ is not integer. The corresponding constraint in $x_B + \widetilde{A}_{NB} x_{NB} = \widetilde{b}$ is

$$x_i + \sum_{j \notin B} \tilde{A}_{ij} x_j = \tilde{b}_i, \qquad (6.8)$$

and it also implies that the left-hand side is lower than or equal to the right-hand side. Applying step (2) and (3) of the technique of the previous section shows then that

$$x_i + \sum_{j \notin B} \lfloor \tilde{A}_{ij} \rfloor x_j \leq \lfloor \tilde{b}_i \rfloor \qquad (6.9)$$

is a valid inequality (note that we have started from an equality, but the same reasoning applies). It is possible to show that this inequality effectively cuts x^*. Introducing the x^* in (6.9) and remembering that $x^*_{NB} = 0$ and $x^*_B = \tilde{b}$, we obtain.

$$\tilde{b}_i + \sum_{j \notin B} \lfloor \tilde{A}_{ij} \rfloor 0 \leq \lfloor \tilde{b}_i \rfloor,$$

which is clearly not satisfied since $\tilde{b}_i > \lfloor \tilde{b}_i \rfloor$ for non integer b_i. The new polytope does thus not contain x^*. Inequality (6.9) does not directly fit in the format of the LP (6.6). One should therefore modify it by adding a slack variable.

One can then re-iterate the procedure with this new (smaller) polytope. We also note that there is some freedom in the choice of the index i on which the constraint will be built, and one can also simultaneously add several valid constraints, each cutting x^*, by running this procedure in parallel on several indices.

Example

Consider the problem

$$\begin{array}{ll} \max & 10x_1 + x_2 \\ \text{s.t.} & x \in \mathbb{Z}^n_+ \\ & 2x_1 + x_2 \leq 3 \end{array}$$

The optimal point x^* of the LP relaxation is $x^*_1 = 1.5, x^*_2 = 0$ (this can for example be seen graphically), with x_1 being the only basis variable. Besides, the inequality constraint can be rewritten as an equality constraint (as in (6.6)) by introducing the slack variable $s_1 \geq 0$:

$$2x_1 + x_2 + s_1 = 3$$

Rewriting this constraint in the form (6.8), yields

$$x_1 + \frac{1}{2}x_2 + \frac{1}{2}s_1 = \frac{3}{2},$$

which leads to the new constraint (of the form (6.9))

$$x_1 \leq 1.$$

One can verify that it indeed cuts $x^* = (1.5, 0)$. The new formulation of this problem, using a smaller polytope, is now

$$\begin{aligned}
&2x_1 + x_2 + s_1 = 3 \\
&\Rightarrow x_1 + \frac{1}{2}x_2 + \frac{1}{2}s_1 = \frac{3}{2} \\
&\overbrace{\Rightarrow}^{integer} \;\; x_1 \leq 1 \\
&\Rightarrow x_1 + s_2 = 1
\end{aligned}$$

The new linear programming problem is now:

$$\begin{array}{ll}
\max & 10x_1 + x_2 \\
\text{s.t.} & x \in \mathbb{Z}^n_+ \\
& 2x_1 + x_2 \leq 3 \\
& x_1 \leq 1.
\end{array}$$

The optimal solution of the LP relaxation is $(1, 0)$, and is thus also the optimal solution of the initial integer program.

Note that rewriting this problem with reduced polytope in the form (6.6), requires two slack variables:

$$\begin{array}{ll}
\max & 10x_1 + x_2 \\
\text{s.t.} & 2x_1 + x_2 + s_1 = 3 \\
& x_1 + s_2 = 1 \\
& x \geq 0
\end{array}$$

We finish by stating the following theorem.

Theorem 6.5.1. *Gomory's cutting plane algorithm described above converges to the optimal solution of the IP after finitely many iterations.*

6.6 Knapsack based inequalities

6.6.1 Basic Idea

In this section we show how we can add valid inequalities based on a single "knapsack" constraint

$$\sum_{j=1}^{n} a_j x_j \leq b, \tag{6.10}$$

with $b \geq 0, a_j > 0$ and $x_j \in \{0, 1\}$. Such constraints naturally occur in the Knapsack problem, but also as part of many other problems (in fact, any binary problem can be reformulated as such with reformulation). Moreover, if some a_j is not positive, then one can always make it positive (for the purpose of this section) by a change of variable $\bar{x}_j = 1 - x_j$. Note also that we can discard the variables for which $a_j = 0$ by appropriately relabeling them and taking n to be the the number of those with nonzero coefficients.

Consider for example the constraint

$$11x_1 + 6x_2 + 6x_3 + 5x_4 + 5x_5 + 4x_6 + x_7 \leq 19, \tag{6.11}$$

for binary variables. It is clearly impossible to have the 7 variables equal to 1. Hence, since they can be either 0 or 1, their sum is at most 6, and we have the valid inequality

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 \leq 6.$$

Observe that it is also impossible to have $x_1 = x_2 = x_3 = x_4 = 1$, so the inequality

$$x_1 + x_2 + x_3 + x_4 \leq 3$$

is also valid. One can see that a large number of such inequalities can be generated, so there is a need to create those new constraints in an ordered and optimized way.

6.6.2 Covers and extensions

Definition A *cover* C for a constraint of the form (6.10) is a subset $C \subseteq \{1, \ldots, n\}$ such that $\sum_{j \in C} a_j \geq b$

The following proposition is immediate

Proposition 3. *If C is a cover, then $\sum_{j \in C} x_j \leq |C| - 1$ is a valid inequality, with $|C|$ the cardinality of C.*

For example, the set $\{1, 2, 3, 4\}$ is a cover, but $\{1, 2\}$ is not.

Definition A cover is minimal if for every $i \in C$, the smaller set $C \setminus \{i\}$ is not a cover.

For example, $\{1, 2, 3\}$, $\{1, 2\}$ and $\{3, 4, 5, 6\}$ are minimal covers.

Observe the small cover will lead to a small independent term in the constraint of Proposition 3, which is desirable. On the other hand a large cover will lead to a large number of variables in the left-hand side of the constraint. To take advantage of the two aspects, we will now see how to start from a minimal cover to get a small independent term $|C| - 1$, and then add other variables on the left-hand side. The next proposition states that one can always add variables with sufficiently large coefficients.

Proposition 4. *Let C be a cover, and let the extension of C*

$$E(C) = \{k : a_k \geq a_j, \forall j \in C\},$$

be the set of the indices of the coefficients larger than or equal to all those of the cover. Then

$$\sum_{j \in C \cup E(C)} x_j \leq |C| - 1 \tag{6.12}$$

is a valid inequality

Proof. Suppose we set $x_i = 1$ for all indices i in a set $C' \subset C \cup E(C)$ of cardinality $|C'| = |C|$ Let $\bar{a} = \max_{j \in C} a_j$. Observe that by definition of $E(C)$, there holds

$$\begin{aligned}
\sum_{j \in C'} a_j &= \sum_{j \in C' \cap C} a_j + \sum_{j \in C' \cap (E(C) \setminus C)} a_j \\
&\geq \sum_{j \in C' \cap C} a_j + \sum_{j \in C' \cap (E(C) \setminus C)} \bar{a}. \\
&\geq \sum_{j \in C} a_j > b
\end{aligned}$$

where the penultimate inequality follows from $|C' \cap C| + |C' \cap (E(C) \setminus C)| = |C'| = |C|$ and $\bar{a} = \max_j a_j$, and the last inequality from the fact that C is a cover. Hence, it is impossible to set $|C|$ elements of $C \cup E(C)$ to 1 without violating the knapsack constraints, which implies (6.12). □

6.6.3 Constraint lifting

We now see a technique to strengthen constraints in an optimized way. Suppose we have obtained from (6.10) an initial valid inequality

$$\sum_{j \in D} \alpha_j x_j \leq \beta \tag{6.13}$$

for some $\alpha_j > 0$ and $D \subset \{1, \ldots, n\}$ and binary variables x. This inequality could for example come from a cover inequality, in which case the α_j are 1, $D = C$ and $\beta = |C| - 1$. We want to add x_t to the left-hand side of this constraint with the largest possible coefficient, for some $t \notin D$. In other words we want to solve the problem

$$\max \alpha_t \text{ s.t. } \sum_{j \in D} \alpha_j x_j + \alpha_t x_t \leq \beta \text{ is valid for any binary } x \text{ satisfying (6.10)}$$

Observe that the new inequality holds if and only if it holds for all x satisfying (6.10) with $x_t = 0$ *and* for all those with $x_t = 1$. Consider first a x for which $x_t = 0$. In that case

$$\sum_{j\in D} \alpha_j x_j + \alpha_t x_t = \sum_{j\in D} \alpha_j x_j,$$

which is never larger than β, whatever the value of α_t, since we know (6.13) is valid. Hence we can pick α_t based solely on the case $x_t = 1$, in which case the our inequality becomes

$$\sum_{j\in D} \alpha_j x_j + \alpha_t.1 \leq \beta,$$

or

$$\sum_{j\in D} \alpha_j x_j \leq \beta - \alpha_t. \tag{6.14}$$

We must thus find the largest α_t, or equivalently the smallest $\beta - \alpha_t$, such that (6.14) holds for every x satisfying (6.10) and $x_t = 1$, that is, for every binary x satisfying

$$\sum_{j\neq t} a_j x_j \leq b - a_t. \tag{6.15}$$

The smallest value $\beta - \alpha_t$ for which (6.14) holds is precisely the maximum value of $\sum_{j\in D} \alpha_j x_j$ under the same constraints. Hence we should take α_t such that

$$\begin{aligned} \beta - \alpha_t = \max & \sum_{j\in D} \alpha_j x_j \\ \text{s.t. } & x_j \in \{0,1\}, \forall j \in D \\ & \sum_{j\in D, j\neq t} a_j x_j \leq b - a_t, \end{aligned} \tag{6.16}$$

which is a knapsack problem. Observe that we only consider variables in D, as all the others can be set to 0 since they do not contribute to the objective. Solving it is also NP-hard, but may be comparatively much simpler than the initial problem, especially if D is small, so that it would make sense to solve it if it allows getting a tighter polytope. We remind indeed that this technique can be applied as soon as the polytope description contains a Knapsack-like inequality, and not only in Knapsack problems. Besides, we note that overestimating the solution of (6.16) also leads to a valid inequality, albeit with a suboptimal a_t, so one could solve relaxations of (6.16) in case solving it exactly is considered too costly. One can then replace (6.13) by the constraint obtained and iterate the process.

Example

Suppose the following inequality

$$5x_1 + 6x_2 + 5x_3 + 11x_4 + 7x_5 \leq 15$$

occurs in the description of a polytope with binary variable. We suppose we start from the valid inequality

$$x_1 + x_2 + x_3 \leq 2,$$

derived from the cover $(1, 2, 3)$, and we want to add the variable x_4 with a coefficient α_4. Following the discussion above, we should pick α_4 such that

$$\begin{aligned} 2 - \alpha_4 = \max \quad & x_1 + x_2 + x_3 \\ \text{s.t. } & 5x_1 + 6x_2 + 5x_3 \leq 15 - 11 = 4. \end{aligned}$$

One can see that optimal cost of this Knapsack problem is 0, leading to $\alpha_4 = 2$. So the following inequality is valid.

$$x_1 + x_2 + x_3 + 2x_4 \leq 2.$$

We could then also add x_5 with coefficient α_5, obtained by solving

$$\begin{aligned} 2 - \alpha_5 = \max & x_1 + x_2 + x_3 + 2x_4 \\ \text{s.t. } & 5x_1 + 6x_2 + 5x_3 + 11x_4 \leq 15 - 7 = 8. \end{aligned}$$

The optimal cost is now 1, leading to $\alpha_5 = 2 - 1 = 1$ an an inequality

$$x_1 + x_2 + x_3 + 2x_4 + x_5 \leq 2.$$

Observe that this inequality is stronger that what would have been obtained by considering the extension $(1, 2, 3, 4, 5)$ of the cover $(1, 2, 3)$ as in Section 6.6.2.

Chapter 7

Lagrangian Duality

7.1 Introduction

Lagrangian Duality can be used to generate parametric upper bounds for linear (integer) problems, i.e. upper bounds that depend on arbitrary parameters. Note that not all parameters lead to bounds of the same quality. Hence we will see how we can then improve the quality of the upper bounds by applying an optimization algorithm on the parameters, until possibly converging to the "optimal upper bounds".

The idea will be to dualize certain constraints. It means that we will replace them by a penalty in the objective functions. The choice of the constraints to dualize is theoretically arbitrary. But since the upper bounds will be obtained by solving an auxiliary problem in which those constraints are removed, one should of course prefer dualizing constraints that make the problem hard to solve, so that the auxiliary problem becomes simple. This will typically be the case of coupling constraints etc.

7.2 Lagrangian Dual

Consider the following IP problem

$$\begin{aligned} z = & \max c^T x \\ \text{s.t. } & x \in \mathbb{Z}_+^n \\ & Dx \le d \\ & Ax \le b, \end{aligned}$$

and suppose the constraints $Dx \le d$ is "*hard*", in the sense that the problem would be easy to solve if it was removed. The Lagrangian dual for an arbitrary vector $u \in \mathbb{R}_+^m$ (where m is the dimension of d or the number of line in $Dx \le d$) is

$$\begin{aligned} z(u) = & \max c^T x + u^T (d - Dx) \\ \text{s.t. } & x \in \mathbb{Z}_+^n \\ & Ax \le b. \end{aligned}$$

Observe that if $u_i > 0$, then violating the i^{th} constraint of $Dx \le d$: $(Dx)_i > d_i$ results in $u_i(d_i - D_i x) < 0$, which decreases the objective. Such a violation is therefore discouraged because costly, but not forbidden. On the other hand, satisfying "strongly" the constraints $(Dx)_i < d_i$ results in a reward, which is actually not very relevant for the initial problem. The "cost" u_i of this constraint is called the *Lagrange multiplier.* A part of the technique developed below will consist in tuning them to penalize constraints that are violated without offering reward for constraints that are violated. The theorem 7.2.1 shows that the optimal solution of the Lagrange relaxation is always an upper bound on the solution of the initial problem.

Theorem 7.2.1. *If $u \ge 0$, then $z(u) \ge z$.*

Proof. Let S be the feasible set of the initial problem:

$$S = \{x \in \mathbb{Z}^n : Ax \le b, Dx \le d, x \ge 0\}.$$

Since $u \geq 0$, for any point $x \in S$ we have

$$c^T x + \underbrace{u^T}_{\geq 0} \underbrace{(d - Dx)}_{\geq 0} \geq c^T x,$$

which implies

$$\max_{x \in S} c^T x + u^T(d - Dx) \geq z = \max_{x \in S} c^T x$$

But $z(u)$ is the max of $c^T x + u^T(d - Dx)$ over a larger set than S, so $z(u) \geq \max_{x \in S} c^T x + u^T(d - Dx) \geq \max_{x \in S} c^T x = z$.

$\square$

The Lagrangian relaxation provides thus indeed an upper bound on z, for every u In particular, $u = 0$ leads to the “usual" relaxation of the constraints $Dx \leq d$.

Example:

Let us consider a simple example:

$$\begin{aligned} \max \quad & 2x_1 + x_2 \\ \text{s.t.} \quad & x_1, x_2 \in \mathbb{Z}_+ \\ (B1): \quad & x_1 \leq 3 \\ (B2): \quad & x_2 \leq 2 \\ (C): \quad & 2x_1 + 2x_2 \leq 7 \end{aligned} \tag{7.1}$$

Observe that the problem would be trivial if it were not for the coupling constraint (C), which we will consider here as the difficult variable. for a given parameter u, the Lagrangian dual will be

$$\begin{aligned} z(u) = & \max 2x_1 + x_2 + u(7 - 2x_1 - 2x_2) \\ \text{s.t.} \quad & x_1, x_2 \in \mathbb{Z}_+ \\ (B1): \quad & x_1 \leq 3 \\ (B2): \quad & x_2 \leq 2) \end{aligned}$$

In particular, suppose $u = \frac{3}{4}$. Then the objective is $(\frac{1}{2})x_1 - (\frac{1}{2})x_2 + \frac{3}{4}7$. The maximum of this decoupled problem is obtained by taking x_1 as large as possible and x_2 as small as possible: $x = (3, 0)$. As a result, $z(\frac{3}{4}) = 1.5 + 5.75 = 7.25$, which means that $z \leq 7$ since z has to be an integer.

For such a simple example we can solve the problem for a generic parameter u, consider the different cases appearing , and select later the parameter u leading to the best possible bound. The objective for a generic u is indeed

$$7u + (2 - 2u)x_1 + (1 - 2u)x_2.$$

Observe that x_1 should be as large as possible if $u < 1$ and as small as possible if $u > 1$, and x_2 should be as large as possible is $u < \frac{1}{2}$ and as small as possible if $u > \frac{1}{2}$. The different cases are thus

u	x_1	x_2	$z(u)$
> 1	0	0	$7u$
$= 1$	$*$	0	7
$(\frac{1}{2}, 1)$	3	0	$6 + u$
$= \frac{1}{2}$	3	$*$	6.5
$< \frac{1}{2}$	3	2	$8 - 3u$

In this case, $u = \frac{1}{2}$ leads to the best bound $z \leq \lfloor z(\frac{1}{2}) \rfloor = \lfloor 6.5 \rfloor = 6$. Such an exhaustive exploration is unfortunately impossible for larger number of constraints. Hence we will see later an algorithm that optimizes over the u.

Before that, we present a proposition stating sufficient conditions under which the optimal solution of the Lagrangian relaxation is in fact the optimal solution of the initial problem.

Proposition 5. *The optimal solution $x(u)$ of the Lagrangian relaxation for a given u is also the optimal for the initial integer program if*

i. i) $Dx(u) \leq d$, and

ii. ii) $(Dx(u))_i = d_i$ for all i for which $u_i > 0$ (complementarity condition)

Condition (ii) means that the penalty/rewards associated to the constraint do not apply, and implies $u^t(d - Dx) = 0$

Proof. Condition (i) implies that $x(u)$ is admissible for the initial IP and hence that $c^T x(u) \leq z$. Due to condition (ii), we have then

$$z(u) = c^T x(u) + u^T(d - D(x)u) = c^T x(u) + 0 \leq z.$$

On the other hand we have seen in Theorem 7.2.1 that $z(u) \geq z$, so that $z(u) = z$, which means that $x(u)$ is an optimal solution. □

7.3 Optimal upper bound

The value of the upper bound $z(u)$ can be seen as a function of u. We let

$$z_{LD} = \min_{u \in \mathbb{R}^m_+} z(u)$$

be the smallest of the upper bounds. Unfortunately, it is in general not true that $z_{LD} = z$, even if this holds in certain cases.

Theorem 7.3.1. *Let $\mathbb{X} = \mathbb{Z}^{\ltimes}_+ \cap \{x : Ax \leq b\}$. The best Lagrangian upper bound satisfies*

$$\begin{aligned} z_{LD} = &\max c^T x \\ s.t.\ & x \in conv(\mathbb{X}) \\ & Dx \leq d \end{aligned}$$

Proof. Let us assume for the convenience of exposition that $\mathbb{X}$ is a finite set (a slightly more complex version of the present argument applies otherwise), and let us enumerate its element $x^1, x^2, x^3, \ldots, x^N$. By definition of the z_{LD}, there holds

$$\begin{aligned} z_{LD} = \min_u z(u) &= \min_u \max_{x \in \mathbb{X}} c^T x + u^T(d - Dx) \\ &= \min_u \max_{t=1,\ldots,N} c^T x^t + u^T(d - Dx^t). \end{aligned}$$

For a given u, the equation above involves a maximization of the expression $c^T x^t + u^T(d - Dx^t)$ over the finitely many possible values of t. This is equivalent to taking the smallest value η that is above $c^T x^t + u^T(d - Dx^t)$ for all these t, so that the problem can be rewritten as

$$\begin{aligned} z_{LD} = &\min_{u,\eta} \eta \\ \text{s.t. } & \eta \geq c^T x^t + u^T(d - Dx^t) \qquad \forall t = 1, \ldots, N \end{aligned}$$

which is a linear program. The variables are u and η and there is a potentially very large number N of constraints. Note that the x^t are constant in this setting. The dual of this linear program is 7.2.

$$\begin{aligned} z_{LD} = &\max_{\mu \in \mathbb{R}^N_+} \sum_{t=1}^N (c^T x^t)\mu^t \\ \text{s.t. } & \sum_t \mu_t = 1 \\ & \sum_t (Dx^t - d)\mu_t \leq 0, \end{aligned} \tag{7.2}$$

where we remind the reader that the first constraint ($\sum_t \mu_t = 1$) is an equality because η iis not constrained to be non-negative, while the other constraints are inequalities because the vector u is non-negative in the primal. This dual can be rewritten as

$$\begin{aligned} z_{LD} = \max_{\mu \in \mathbb{R}^N_+} & \; c^T \sum_{t=1}^N \mu^t x^t \\ \text{s.t.} & \; \sum_t \mu_t = 1 \\ & D \sum_t \mu_t x^t \le d, \end{aligned}$$

where we have used $\sum_t \mu_t = 1$ to obtain the concise form of the last line. Let then $x := \sum_t \mu_t x^t$. The constraints that μ_t are nonnegative and sum to 1 translates into x being a convex combination of the x^t, or equivalently, belonging to conv($\mathbb{X}$) since the x^t are all the elements of $\mathbb{X}$. Hence the dual is equivalent to

$$\begin{aligned} z_{LD} &= \max c^T x \\ \text{s.t. } & x \in \mathbb{X} \\ &= Dx \le d, \end{aligned}$$

which is the desired result. □

One could therefore wonder about the interest of using the Lagrangian Relaxation if one can at best obtain a result equivalent to the optimization of the function over conv($\mathbb{X}$) $\cup \{x : Dx \le d\}$. The answer is twofold: (i) First we have assumed that we can easily optimize over $\mathbb{X} = Z^n_+ \cap \{x : Ax \le b\}$, but this does not imply that we have access to conv($\mathbb{X}$), nor even that it has a tractable description. (ii) Second, even if we have access to conv($\mathbb{X}$), it might be much more efficient to compute Lagrangian bound, especially when one does not need to have *the best bound*.

Coming back to our example (7.1), remember that $z_{LD} = 6.5$ is reached for $u = \frac{1}{2}$. On the other hand, the convex hull of $\mathbb{X}$ is $[0, 3] \times [0, 2]$, so that the optimization over conv($\mathbb{X}$) under the constraint $Dx \le d$ is

$$\begin{aligned} &\max 2x_1 + x_2 \\ &\text{s.t.} 0 \le x_1 \le 3 \\ &\quad 0 \le x_2 \le 2 \\ &\quad 2x_1 + 2x_2 \le 7, \end{aligned}$$

the optimal solution of which is $(3, \frac{1}{2})$ and has a cost 6.5.

7.4 Optimization of the upper bound

We now move to the problem of finding z_{LD}. We assume again that for the convenience of exposition that $\mathbb{X} = \mathbb{Z}^n_+ \cup \{x : Ax \le b\}$ is finite and enumerate its elements $x^1, x^2, \ldots, x^N$. Remember that

$$\begin{aligned} z(u) &= \max_{x \in \mathbb{X}} c^T x + u^T (d - Dx) \\ &= \max_{t=1,\ldots,N} c^T x^t + u^T (d - Dx^t). \end{aligned} \tag{7.3}$$

The function $z(u)$ is thus the maximum of N linear functions and is thus convex, as represented in Figure 7.1. This is good news, as optimizing over convex functions is generally a tractable problem. Observe though that the maximum of a finite number of linear function is a continuous function but almost never one that is differentiable everywhere; it will almost always have angular points or subspaces, as also seen in Figure 7.1. This implies that a classical gradient descent will in general not converge to the optimal solution: First the gradient is not defined everywhere, and second, even if one never encounters a point at which the gradient is not defined, one can verify on a simple example such as $f(x) = |x|$ that a fixed-step

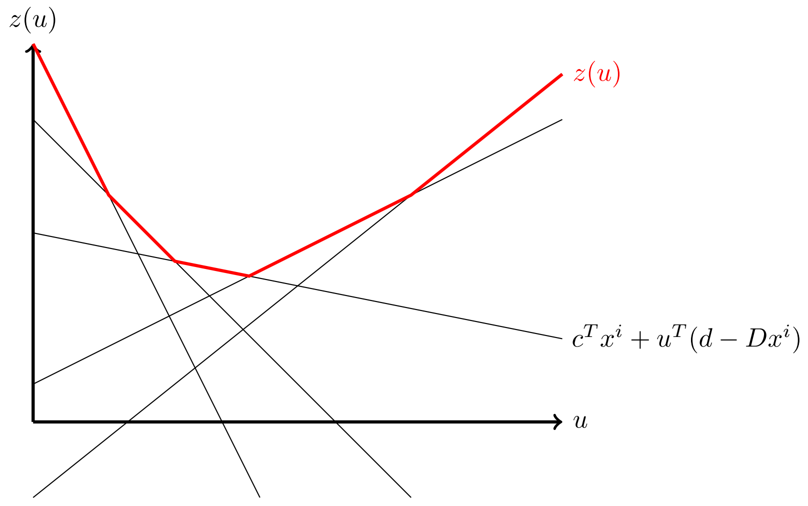


Figure 7.1: Graph illustrating the convex function $z(u)$, according to the expression in (7.3)

gradient descent of the form $x' = x - h\nabla f(x)$ can oscillate around the optimal solution.

Optimization of a continuous but not necessarily differentiable convex function can be achieved using a *subgradient algorithm.*

Definition The vector γ is a subgradient of the convex function f at point x if there holds

$$f(y) \geq f(x) + \gamma^T(y - x),$$

for every y. This is denoted by $\gamma \in \partial f(x)$.

Note that $\partial f(x)$ is a (convex) set of vectors. At those points where f is differentiable, it only contains the gradient $\nabla f(x)$. But at those points where f is not differentiable, it typically contains an infinity of vectors. See the illustration in Figure 7.2. As another example, consider $f(x) = |x|$ and observe that $\partial f(0) = [-1, 1]$, i.e. contains every "vector" of $\mathbb{R}$ between -1 and 1. It turns out that subgradients of $z(u)$ can easily be computed:

Proposition 6. *$(d - Dx(u))$ is a subgradient of $z(u)$, where $x(u) \in \mathbb{X}$ is the value of the optimum x for the relaxation with vectors u.*

Proof. Let us consider a vector u and a vector v. There holds

$$z(v) = c^T x(v) + v^T(d - Dx(v)) \geq c^T x + v^T(d - Dx), \forall x \in \mathbb{X},$$

because $z(v)$ is by definition the optimal value on $\mathbb{X}$ for the Lagrangian relaxation with vector v. In particular

$$\begin{aligned} z(v) &\geq c^T x(u) + v^T(d - Dx(u)) \\ &= c^T x(u) + u^T(d - Dx(u)) + (v - u)^T(d - Dx(u)) \\ &= z(u) + (v - u)^T(d - Dx(u)), \end{aligned}$$

which shows that $d - Dx(u)$ is a subgradient of $z(u)$ at u. □

There remains to see how subgradients can be used in optimization. The subgradient algorithm works essentially like a gradient descent, replacing the gradient by a subgradient, but its convergence to a minimum is only guaranteed if the step-size satisfy certain conditions: Specifically, the iteration

$$x^{k+1} = x^k - h_k\gamma_k, \text{ with } h_k \in \partial f(x_k)$$

converges if $\sum_k h_k \to \infty$ and $\sum_k h_k^2 < \infty$ (alternative sufficient conditions exist). An example of such steps is the class $h_k = \frac{a}{b+k}$. The guarantees would hold for any $a, b > 0$, but the choice of a, b will strongly affect the speed of convergence. If the steps remain large for too long, the methods may oscillate for a

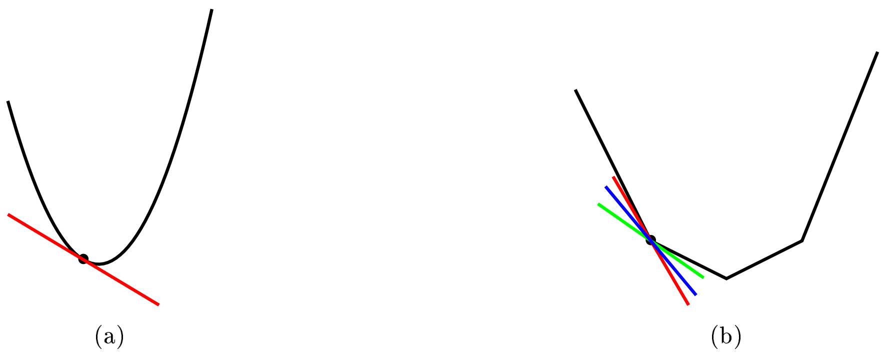


Figure 7.2: Illustration of the gradient (a) and subgradients (b).

long time. On the other hand, if they decrease too fast, the method will take a long time to approach the optimum. One can also show that $h_k = h_0\rho^k$ for some $\rho < 1$ leads to convergence provided that ρ is sufficiently large (with the meaning of sufficiently large possibly depending on the problem), and is from a practical point of view often faster than $h_k = \frac{a}{b+k}$.

The last complication comes from the fact that u should only contain positive values. For this, we can use the projected subgradient algorithm, which consist in performing a subgradient step and reprojecting the result on the admissible set, here $\Re^m_+$. Taking into account Proposition 6, this leads to the algorithm

Subgradient algorithm

```
*Choose initial* $u^0$;
**for** $k = 0, 1, 2, ...$ **do**
    $u^{k+1} = \max\left(u^k - h_k\left(d - Dx(u^k)\right), 0\right)$ ;
**end**
```

where $x(u)$ is the optimal solution of IP(u) and γ_k is the stepsize.

7.5 Application: Generalized assignment problem

Consider the problem (7.4).

$$\begin{aligned} z = \max & \sum_{i,j} c_{ij}x_{ij} && (7.4)\\ \text{s.t.} & \sum_j x_{ij} = 1 \quad \forall i \\ & \sum_i x_{ij} = 1 \quad \forall j \\ & \sum_{ij} t_{ij}x_{ij} \leq t && (7.5)\\ & x_{ij} \in \mathbb{Z} \\ & x_{ij} \geq 0 \end{aligned}$$

Notice that constraint (7.5) makes the problem NP hard.

First Lagrangian relaxation We relax the *hard* constraint (7.5), and obtain

$$\begin{aligned}
z(u) = \max & \sum_{ij} c_{ij}x_{ij} + u(t - \sum_{ij} t_{ij}x_{ij}) \\
\text{s.t.} & \sum_{j} x_{ij} = 1 \quad \forall i \\
& \sum_{i} x_{ij} = 1 \quad \forall j \\
& x_{ij} \in \mathbb{Z} \\
& x_{ij} \geq 0
\end{aligned}$$

The Lagrangian dual value z_{LD} is exactly the LP relax bound, because the assignment problem is totally unimodular. Algorithmically, it may provide a good algorithm to solve LP relaxation, as there is only one variable u to optimize.

$$w_{LD} = \min_u z(u)$$

This is "easy" to compute for a given u with the Hungarian method.

Second Lagrangian relaxation : relax the assignment constraints.

$$\begin{aligned}
z(u,v) = \max & \quad \sum_{ij} c_{ij}x_{ij} + \sum_{j} u_j(1 - \sum_{i} x_{ij}) + \sum_{i} v_i(1 - \sum_{j} x_{ij}) \\
\text{s.t.} \ \sum_{ij} t_{ij}x_{ij} \leq t & \\
x_{ij} \in \mathbb{Z} & \\
x \geq 0 &
\end{aligned}$$

This lagrangian relaxation can be assimilated to a Knapsack problem.

$$w_{LD} = \underbrace{\min_{u \in \mathbb{R}^n, v \in \mathbb{R}^n}}_{\text{free}} z(u,v)$$

We can expect a **strictly better bound** than the LP relaxation, however computing $z(u,v)$ is a harder problem. Even for given (u,v), a knapsack problem needs to be solved.

This example shows that the choice of constraint to dualize is not neutral. In particular, one should balance

- the tightness of the bounds obtained with the optimal u,
- the ease of computing $z(u)$ and the subgradient, and
- the ease of solving the minimization of $z(u)$ over u. This is typically hard to evaluate, but the dimension of u already provides some information.

An ideal situation is to dualize a small number of constraints that were responsible for coupling subproblems, making the whole problem hard.

Chapter 8

Column Generation

Column generation is a method relevant when one has a huge-scale linear optimization problem with the structure of the problem is such that the base (in the simplex algorithm) is of limited size. This approach allows us to apply the simplex algorithm on such problems without ever having to write their actual full formulation, which would be prohibitively expensive. We will first begin by reminding the principles of the simplex algorithm. We will then see an application where the problem naturally leads to huge number of variables. Finally, we will see how column generation approach can be used to solve certain linear relaxations of integer problems.

8.1 Reminder on simplex algorithm and linear optimization

There are several canonical forms for linear program, which can be obtained one from each other by change of variables and/or introduction of slack variables. The one we consider for the simplex algorithm is the following,

$$\begin{aligned} \max \quad & c^T x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0, \end{aligned} \tag{8.1}$$

where $x \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ and $A \in \mathbb{E}^{m \times n}$. For simplicity's sake, we will assume here that the rows of A are linearly independent, i.e. the there is no redundancy or incompatibility between the equality constraints.

One can show that if an optimum solution exists, then it is always possible to find one that is a vertex of the polytope defined by $\{x \geq 0 : Ax = b\}$, that is, a point at which at least n constraints are tight, i.e. satisfied with equality. The simplex algorithm consists in starting from such a vertex, and moving from vertex to neighboring vertex until reaching the optimum.

Observe now that the m constraints of $Ax = b$ are necessarily tight since they are equality constraints. Hence for any vertex x^* there remains (at least) $n - m$ nonnegativity constraints that are tight, and thus at least $n - m$ variables that are 0 at any vertex. For a given vertex, we call *basis* the $m = n - (n - m)$ remaining variables that are potentially nonzero and denote the corresponding set of indices by B. We denote the complementary set by NB. We have thus $x^*_{NB} = 0$, and x^*_B potentially different from 0. For a given basis B and a general x we can rewrite $Ax = b$ as

$$A_B x_B + A_{NB} x_{NB} = b,$$

which together with $x^*_{NB} = 0$ implies

$$A_B x^*_B = b,$$

and thus,

$$x^*_B = A_B^{-1} b,$$

(assuming A_B is invertible). Therefore *the set of nonzero variables B entirely determines x^*_B and thus x^*, the vertex.*

The simplex iteration consists in adding a variable to the basis and removing another one, in such a way that the benefit of the new basis is larger than the previous one, i.e. $c^T \tilde{x}$ for the new vertex $\tilde{x}$ (Many additional details have to be treated, such as the possibility of A_B not being invertible, or the possibility of moving to a vertex with equal benefit, but they are out of the scope of this class).

The main question in the simplex iteration is the selection of the new variable that will enter the basis. An idea is of course to take the variable x_k that we think will lead to the largest improvement. It is actually computationally easier to pick the variable x_k leading to the largest improvement *relative to the variation of* x_k, or in some cases to a sufficiently large improvement relative to the variation of x_k.

Suppose indeed we pick the variable k that we will add to the basis, and imagine that we first only increase it by a small amount Δx_k. The constraints $Ax = b$ are no longer satisfied, so we must also adapt x_B^* to $x_B + \Delta x_B$ (remember that x_k is by hypothesis not part of x_B). Specifically, since x_k is set to Δx_k, we need to guarantee

$$A_B(x_B^* + \Delta x_B) + a_k \Delta x_k = b,$$

with a_k being the row of A corresponding to the variable k. Remembering that $Ax_B^* = b$, we obtain

$$\Delta x_B = -(A_B)^{-1} a_k \Delta x_k$$

Let us now analyze the corresponding variation of cost:

$$\begin{aligned}\Delta\text{cost} &= c_k \Delta x_k + c_B^T \Delta x_B \\ &= c_k \Delta x_k - c_B^T (A_B)^{-1} a_k \Delta x_k \\ &= (c_k - c_B^T (A_B)^{-1} a_k) \Delta x_k \qquad =: \tilde{c}_k \Delta x_k.\end{aligned}$$

For this reason, the coefficient $\tilde{c}_k = c_k - c_B^T (A_B)^{-1} a_k$ is called the *reduced cost.*

When $\tilde{c}_k < 0$, one should certainly not increase x_k nor put k in the basis, as it would reduce the benefit. On the other hand, if $\tilde{c}_k > 0$ and k as been selected, the simplex algorithm consists in taking Δx_k as large as possible, with the constraint being that $x_B + \Delta x_B$ should remain non-negative. We will thus increase Δx_k until one variable of the basis becomes 0, and this variable will then leave the basis. From a practical point of view, this leads to $x_k = \Delta x_k = \min_{i \in B} -\frac{(\Delta x_B)_i}{(x_B)_i}$. From there we will iterate.

There remains to determine which k to select. In the classical simplex method, one will select the variable with the highest reduced cost $\tilde{c}_k$. At each iteration, one has then to solve

$$k = \arg\max_{j \in NB} \tilde{c}_j = \arg\max_{j \in NB} c_j - c_B^T (A_B)^{-1} a_j.$$

Note that

$$c_B^T (A_B)^{-1} =: d_B^T \tag{8.2}$$

in the expression above only has to be computed once since it is independent of j. Moreover, one should not invert the matrix A_B, but solve the linear system $A_B^T d_B = c_B$, which is computationally much cheaper. The problem of selecting k can thus finally be written as

$$k = \arg\max_{j \in NB} c_j - d_B^T a_j. \tag{8.3}$$

Summary: To apply an iteration of the simplex algorithm on the problem (8.1) with n variables and m equality constraint, one just need to be able to

1. Compute $d_B = c_B^T (A_B)^{-1}$, i.e. solve a $m \times m$ linear problem.
2. Solve the optimization problem (8.3) to select the optimal k, which does not necessarily requires computing $c_j - d_B^T a_j$ for every possible j.
3. Compute the ration $\frac{\Delta x_B}{\Delta x_k} = -(A_B)^{-1} a_k$ and select $x_k = \min_{i \in B} \frac{(\Delta x_B / \Delta x_k)_i}{\Delta x_k} (x_B)_i$, deduce Δx_B and compute the new x_B. The main computational cost comes again from solving a linear $m \times m$ system.

So, independently of the number n of variables, if the number m of constraints is small (even if they involve many variables), one can perform simplex iterations at a cost that depends on m and the complexity of solving (8.3). Moreover, one does actually not necessarily always need to solve that problem exactly: Introducing any variable k with positive reduced cost $\tilde{c}_k$ improves the objective, and one can thus stop solving (8.3) once a variable with "good enough" reduced cost has been found. There is of course a trade-off here between the quality of the step and the cost of obtaining a better k.

This opens the possibility of applying the simplex algorithm to huge-scale problems provided that he number of equality constraints remains low. We will present applications in the next sections. The number of simplex iterations needed may, however, grow with n, and will depend on the structure of the polytope.

8.2 Beam cutting

Suppose you have beams of length L, and a customer orders d_1 beams of length l_1, d_2 beams of length l_2, ... and d_m beams of length l_m, with all these lengths being smaller than L. Your goal is to produce these beams by cutting the smallest possible number of beams of length L.

For this purpose, we define a pattern π as one way of cutting a beam of length L in smaller beams. Specifically, a pattern $\pi^T = (\pi_1, \ldots, \pi_m)$ is a vector of integer, with π_k representing the number of sub-beams of length l_k cut into the large beam. Clearly, a pattern is valid (or feasible) if and only if

$$\sum_k \pi_k l_k \leq L.$$

The number of valid pattern is potentially huge, but finite. Hence we can enumerate the set of valid patterns using an index p: $\Pi = \{\pi^1, \pi^2, \ldots, \pi^{|P|}\}$.

Letting x_p be the number of times the pattern p is used, the problem can be written as

$$\begin{aligned} \min \sum_{p=1}^{|\Pi|} x_p, & & (8.4) \\ \text{s.t. } x_p \in \mathbb{Z}_+, & \qquad \forall p = 1, \ldots, |P| \\ \sum_{p=1}^{|\Pi|} x_p \pi_k^p = d_k, & \qquad \forall k = 1, \ldots, m \end{aligned}$$

Note that we could have used an inequality for the last constraint. The equality is preferred because it will allow a simpler formulation of the simplex algorithm. It does not represent a loss of generality, as using patterns that "waste" part of the beams always allows decreasing the number of beams produced to exactly d_k.

We will present a way of solving the linear relaxation of this problem, allowing x_p to be real instead of integer:

1. Build some valid initial solution, possibly to the relaxed problem instead of the IP, for example using a greedy approach,
2. Apply simplex iteration to (8.4),
3. when a sufficiently low cost is obtained, and build a valid solution to the IP by heuristic means.

Steps (1) and (3) are straightforward. For step (2), observe that problem (8.4) contains a huge number of variables, but a much smaller number of constraints. We have seen in Section 8.1 that simplex iterations can be performed provided that the selection of the next basis variable can be performed, i.e. provided one can (approximately) solve

$$\max_j c_j - d_B^T a_j,$$

for a problem under the classical form $\max\{c^T x : x \geq 0, Ax = b\}$, and where d_B was defined in (8.2) and a_j is the j^{th} column of A. Let us apply this to the (linear relaxation) of (8.4), we see that $c_j = -1$ and $a_j = \pi^p$, so the reduced cost is $1 - d_B^T \pi^p$, and the selection of the next basis variable x_p becomes (taking into account that we face a minimizaton problem as opposed to a maximization in Section 8.1),

$$\min_p 1 - d_B^T \pi^p = -1 + \max_p \sum_k d_{B,k} \pi_k^p.$$

Optimizing over all indices p is the same as optimizing over all valid patterns, i.e. over all patterns π such as $\sum_k \pi_k l_k \leq L$ holds. Hence the variable selection problem can be rewritten as

$$\begin{aligned} \max \sum_k d_{B,k} \pi_k \quad & s.t. \\ \pi_k \in \mathbb{Z}^+ & \\ \sum_k \pi_k l_k \leq L, & \end{aligned}$$

which is an integer Knapsack problem. This problem is NP-hard, but the number of variable here is proportionally small, and it can be efficiently solved in many cases. Moreover, even though the classical simplex algorithm require solving the problem exactly to find the best variable to add to the basis, finding a "sufficiently good" solution will also work provided that the reduced cost $1 - d_B^T \pi^p$ remains negative.

We finish this section by commenting on the results of the linear relaxation or of the intermediate steps in the optimization. From a valid solution to the linear relaxation, we can always build a valid solution to the initial integer problem by rounding all the x_p^* to $\lceil x_p^* \rceil$, at a cost that is bounded by the number of non-integer variables, and thus by the number of nonzero variables. Now at any step of the simplex, the number of potentially nonzero variable is the size of the basis, i.e. the number of equality constraints in (8.4). That number is actually m, the number of different lengths of the sub-beams. So provided that the number of different sub-beams length is small with respect to the number of beams involved in the problem, which is expected, the extra-cost of going from x_p^* to $\lceil x_p^* \rceil$ is limited. In addition, one can always design heuristic method to try finding a slightly better integer solution than $\lceil x_p^* \rceil$.

8.3 Dantzig-Wolfe

The problem we considered in Section 8.2 naturally lead to a huge scale linear program. We will see in this section how, in view of the column generation approach, it may make sense to relax a reasonably sized IP into a huge size problem having few constraints, provided that the pivot selection (i.e. selection of the k that will join the basis) is "easy".

The method we present below leads to bound similar to the Lagrangian relaxation. It has the reputation of being efficient in terms of time needed to generate good bound. However, implementing it may be more complex than for the comparatively simpler Lagrangian relaxation. Lagrangian relaxation allows indeed using linear solver in black-box, while the approach below typically requires coding certain sub-procedures of the simplex, and possibly the way the solutions are stored.

8.3.1 Unique set

Consider the problem

$$\begin{aligned} \max c^T x \quad & s.t. \\ x &\in X \\ Dx &= d, \end{aligned}$$

where X is a finite set. We suppose moreover that X is simple, in the sense that optimizing over X is easy, and that the problem complexity comes thus from the constraints $Dx = d$. Typically X would describe simple local conditions, while $Dx = d$ would be coupling constraints.

In order to obtain an upper bound on the solution (for example in the context of a Branch and Bound algorithm), we may want to solve the relaxation

$$\begin{aligned} \max c^T x \quad & s.t. \\ x &\in conv(X) \\ Dx &= d, \end{aligned} \tag{8.5}$$

which is actually what we would obtain with the Lagrangian dual.

Taking x in the convex hull of X means that x is a convex combination of elements of X. Since X is finite, we can theoretically enumerate its elements

$$x = \{x^1, x^2, \ldots, x^{|X|}\}.$$

Hence $x \in conv(X)$ is equivalent to

$$x = \sum_{i=1}^{|X|} \lambda_i x^i,$$

for some $\lambda_i \geq 0$ such that $\sum_{i=1}^{X} \lambda_i = 1$. We have then $c^T x = \sum_i (c^T x^i)\lambda_i$ and $Dx = \sum_i (Dx^i)\lambda_i$, so that problem (8.5) becomes

$$\begin{aligned} \max \sum_{i=1}^{|X|} (c^T x^i)\lambda_i \quad s.t. \\ \sum_{i=1}^{|X|} \lambda_i = 1 \\ \lambda_i \geq 0 \\ \sum_{i=1}^{|X|} (Dx^i)\lambda_i = d, \end{aligned} \tag{8.6}$$

The equality constraints above can be summarized as

$$\begin{pmatrix} 1 & 1 & \dots & 1 \\ Dx^1 & Dx^2 & \dots & Dx^{|X|} \end{pmatrix} \lambda = \begin{pmatrix} 1 \\ d \end{pmatrix}$$

The number of variables is of course potentially enormous, but the number of constraints is restricted. We have seen in Section 8.1 that simplex iterations can be performed provided one can (approximately) solve

$$\max_j c_j - \tilde{d}_B^T a_j,$$

(where $\tilde{d}_B$ is used to avoid confusion with d used in the problem considered in this section), which is this case means

$$\max_j c^T x^j - \tilde{d}_B^T (1, (Dx^j)^T)^T,$$

or

$$d_{B,1} + \max_j (c^T - \tilde{d}_B^T D) x^j.$$

Remember now that j indexes all the elements of X. Hence optimizing over j an expression that depends on x_j (and some terms independent of j) is equivalent to optimizing over $x \in X$:

$$d_{B,1} + \max_{x \in X} (c^T - \tilde{d}_B^T D) x,$$

and this problem is supposed to be easy to solve. Hence we can perform simplex iterations.

It should be noted that the optimal solution of (8.5) gives an upper bound on the optimal solution of the original IP, but no such guarantee exists for the intermediate solutions produced by the simplex during the process of optimizing the relaxation.

8.3.2 Decomposition

The cardinality of X in Section 8.3.1 has no influence on the cost of computing one gradient step, but it may have a strong influence on the time needed to converge to the optimum, or on the speed at which progress are made. We now show how the number of variables to consider can be reduced when X consist of the product of independent sets : $X = X^1 \times X^2 \times \cdots \times X^n$, that is

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix},$$

with $x_k \in X_k$ for every k, for sets X_k on which it is simple to optimize. This structure will again very often occur when the problem consist of a sequence of local subproblems linked by a small number of coupling constraints.

We consider thus IP problems of the form

$$\begin{aligned} \max \sum_k c_k^T x_k \quad s.t. \\ x_k \in X_k \\ \sum_k D x_k = d, \end{aligned}$$

with X^k finite sets, and their relaxation

$$\begin{aligned}
\max \sum_k c_k^T x_k \quad s.t. \\
x_k \in conv(X_k) \\
\sum_k D x_k = d.
\end{aligned}$$

Following exactly the same reasoning as in Section 8.3.1, we enumerate all elements of every set $X_k = \{x_k^1, x_k^2, \dots, x_k^{|X_k|}\}$ and $x_k \in conv(X_k)$ is equivalent to $x_k = \sum \lambda_{k,i} x_k^i$, where $\sum_i \lambda_{k,i} = 1$ and $\lambda_{k,i} \geq 0$. Hence our relaxed problem can be written as

$$\begin{aligned}
\max \sum_k \sum_i (c_k^T x_k^i)\lambda_{k,i} \quad s.t. \\
\lambda_{k,i} \geq 0, \qquad \forall i, k \\
\sum_i \lambda_{k,i} = 1, \qquad \forall k \\
\sum_k \sum_i (D_k x_k^i)\lambda_{k,i} = d
\end{aligned} \tag{8.7}$$

The equality constraints can be summarized as

$$\left(\begin{array}{cccc|cccc|c|cccc}
1 & 1 & \dots & 1 & & & & & & & & & \\
 & & & & 1 & 1 & \dots & 1 & & & & & \\
 & & & & & & & & \ddots & & & & \\
 & & & & & & & & & 1 & 1 & \dots & 1 \\
\hline
D_1x_1^1 & D_1x_1^2 & \dots & D_1x_1^{|X_1|} & D_2x_2^1 & D_2x_2^2 & \dots & D_2x_2^{|X_2|} & \dots & D_nx_n^1 & D_nx_n^2 & \dots & D_nx_n^{|X_n|}
\end{array}\right)\lambda = \left(\begin{array}{c} 1 \\ 1 \\ \vdots \\ 1 \\ \hline d \end{array}\right)$$

Using again Section 8.1, the reduced cost corresponding to a variable $\lambda_{k,i}$ is

$$c_k x_k^i - \tilde{d}_{B,k} - \tilde{d}_{B,n+1:n+m} D_k x_k^i = \tilde{d}_{B,k} + (c_k - \tilde{d}_{B,n+1:n+m} D_k) x_k^i, \tag{8.8}$$

where m is the number of rows in the matrices D_k, $\tilde{d}_B$ is defined in (8.2) (using $\tilde{d}$ instead of d to avoid confusion here), $\tilde{d}_{B,k}$ represents the k^{th} element of $\tilde{d}_B$ and $d_{B,n+1:n+m}$ the elements $n+1$ to $n+m$.

For a given k, finding the variable $\lambda_{i,k}$ with the best reduced cost consists thus in solving

$$\tilde{d}_{B,k} + \max_{x \in X_k} (c_k - \tilde{d}_{B,n+1:n+m} D_k) x_k,$$

which is supposed to be an "easy" problem. Once can then either solve this problem for every k and pick the variable with the best reduced cost (which correspond to the best reduced cost among all variables), or decide to only look at one k at the time and introduce in the basis the variable with the best reduced cost (if positive) among those corresponding to set X_k.

One could of course have ignored the possible decomposition of X in $X_1 \times X_2 \times \dots$ and directly applied the approach of Section 8.3.1. But this would have lead to an enormously larger number of variables. Suppose indeed for simplicity that all X_k have the same cardinality c. The number of variables in (8.7) is $n.c$. By contrast, if we had considered a simple set $X = X_1 \times X_2 \times \dots \times X_n$, its cardinality would have been c^n, resulting in the same number of variables in problem (8.5).

Chapter 9

Heuristic methods

9.1 Introduction

Heuristic methods are methods developed to find acceptable solutions (sometimes relatively rapidly) or to improve solutions. They are typically not designed to find *the optimal solution* and come with no such guarantee.

They can for example be used when classical optimization approaches prove too costly or impractical. In many situations, one wants to reduce an actual cost (e.g. in euros), but is not necessarily interested in finding the optimal solution provided that the solution is good enough. Investing 5 days of computations to save 5 more euros is not interesting if the first minutes have allowed saving 100.000 euros. Heuristics methods would be very relevant in such situations. They can be used in conjunction with classical optimization techniques, for example

i. to make final improvements to the best solution found by an algorithm,

ii. to obtain primal bound and feasible solutions.

We'll present in this chapter different heuristics. Some will be generic methodologies that can be applied to almost any problem. Others are tailored for specific problems. We will also show how accuracy guarantees can be derived in certain cases, e.g. the cost of the solution found is at most $x\%$ more than that of the optimal solution. This allows for example knowing what the maximal possible improvement would be. In some cases (not presented here), one can actually chose the x, with smaller x resulting of course in higher computational cost.

9.2 Local explorations

This class of methods is designed to provide solutions to problems with finite (or discrete) solution sets by giving a topology to the solution set and exploring it.

Note that mixed problems can sometimes be turned into finite problems. Suppose indeed we want to find

$$\min_{x \in \mathbb{X}, y \in \Re^n} g(x, y), \tag{9.1}$$

where $\mathbb{X}$ is a finite or discrete set. We can define the function

$$f(x) = \min_{y \in \Re^n} g(x, y) \tag{9.2}$$

representing for every x the benefit if the best y (given that x) is selected. Problem 9.1 is then equivalent to the discrete problem

$$\min_{x \in \mathbb{X}} f(x). \tag{9.3}$$

Computing $f(x)$ consists then in solving an optimization problem, and one should keep its complexity in mind when evaluating the computational cost of solving (9.3).

9.2.1 Graph representation

For this section, we need to represent the set of solution as a graph, whose nodes are the solutions. A solution v is then a neighbor of w (i.e. there is an edge (w, v)) if v can be obtained from w by a basic transformation, where the meaning of basic transformation is problem-specific, and is actually part of the design of the algorithm.

In a Knapsack problem for example, it could consist in replacing an object by another one, or removing an object. In the traveling salesman problem, it could consists in switching two edges in the paths, etc. Note that the graph will often be symmetric, but this is not a requirement. The problem becomes then one of finding the optimum of a function $f(v)$ defined on a graph, and the algorithms we will see in the following subsections typically pursue this objective by some form of local exploration of the graph, which will require building an initial solution.

The choice of the operations that defines the neighborhood relation is far from neutral, and may have dramatic implications on the efficiency of the algorithms we will consider. To obtain efficient algorithms, one should for example have a reasonably short path from any initial solution to the optimal one, or to "good" solutions. Operations leading to graphs with long diameter should therefore be avoided.

We also insist on the fact that one should never explicitly build that graph, the size of which is the number of possible solutions. Rather, one will maintain a solution (or several) that will be gradually transformed in the process of graph exploration.

9.2.2 Local search

Local search is the most basic approach, and is parallel to a gradient descent. At each iteration, one will consider all neighbors w of the current node v_i and pick as v_{i+1} that for which $f(w)$ is the smallest. Alternatively, one can select a random neighbor w and select it as v_{i+1} if $f(w) \leq f(v_i)$, or randomly select k neighbors and pick that with the smallest value.

This method is simple, and allows improving the initial solutions. Its main problem is the possibility of being stuck in a local minimum, i.e. at a node (or group of nodes) where f takes values smaller than all other nodes around, but larger than in some other parts of the graph. The algorithm will indeed always push the exploration process towards the local minimum (or push it with a high probability, depending on the variation used).

One solution is to run several instance of the algorithm in parallel, starting from different initial nodes. Another one is to detect when the process appears to have converged to a local minimum, and to then restart it from another point. This requires of course the ability to generate sufficiently different starting points. The methods described in the next subsections offer alternative solutions to the local minimum issue.

9.2.3 Tabu search

Tabu search consists in maintaining a list of nodes that have already been visited and preventing the algorithm from returning to these nodes. The typically has a bounded size. In the simplest version, it simply consists of the last k nodes visited. The rationale is that this will force the algorithm to get away from the local minima. In such case, a typical value for the length of the tabu list is 7. However, we cannot insist too much on the need to use common sense when using such algorithms, and to adapt its parameters to the problem at stake and its structure.

More complex versions bias the search towards part of the network that appear to yield good values, or towards unexplored parts. See e.g. [**?**].

9.2.4 Simulated Annealing

Simulated annealing tackles the problem of local minima in a probabilistic way: it allows, with a certain probability, moving towards solution with higher cost, in such a way that there is always a positive probability of escaping any local minimum basin.

A basic version of simulated annealing work in the following way. Starting from the current node v_i, select a candidate neighbor w. If $f(w) < f(v_i)$, then move to w, i.e. set $v_{i+1} = w$. If $f(w) > f(v_i)$, the algorithm sets $v_{i+1} = w$ with a probability $e^{\frac{-(f(w)-f(v_i))}{T}}$, where T is the *temperature*. The name of Simulated Annealing and the choice of T comes from an analogy with metallurgy. When T is large, the system has a lot of "energy" and tends to explore a lot of different neighborhoods, even though it will always be less likely to explore zone leading to very high increase of the cost function. Over time, when T gets lower, the heuristic will tend to stay in the most promising neighborhoods. In order to converge to a good neighborhood, T should be chosen to slowly decrease over time. A possibility is to decrease T every I iterations:

$$T := T_0 \frac{a}{a+k}$$

for some well-chosen constants T_0, I, a and k.

If the temperature decreases sufficiently slowly, simulated annealing has been proven to converge to a *global* optimum. This result is a nice justification of simulated annealing, but it is also often useless in practice because "sufficiently slowly" is far too slow for most applications. In particular, it does not explain the success of simulated annealing in numerous applications.

9.2.5 A final word on local exploration techniques

In all the techniques above, it goes without saying that the algorithm should return the best solution found during the exploration, and not necessarily the last one.

9.3 Greedy Algorithms

Greedy algorithms are useful to rapidly building a solution that is supposed to be of "acceptable quality". They consist in building a solution step by step by adding at each step the elements that appears to minimize the cost (or maximize the benefit).

We do not resist casting this method in our graph theoretical representation, we can consider a directed graph where nodes are "subsets of solutions" (e.g. a directed path in the TSP problem, or a set of objects in the Knapsack problem). There is then a directed edge between v and w if w is "larger" than v and can be obtained from w by a simple "additive" operation, where the meaning of larger and additive has to be specified depending on the problem, e.g. adding one edge to the path, or one object in the Knapsack. Greedy algorithms consist then in starting from a node corresponding to an empty or trivial solution, and exploring the directed network by keeping moving the out-neighbor of the current node with the lowest cost (resp. highest benefit), until reaching a sink node, i.e. a solution that cannot be increased.

For the Knapsack problem, a greedy heuristic consists in adding at each step the object with the highest benefit/weight ratio. We will see that this lead to a solution with at least 50% of the optimal benefit. In the TSP case, a greedy heuristic consist in building a cycle by always moving towards the closest unvisited node. Unlike for the Knapsack, it can be shown that this greedy method can lead to particularly bad solutions for the TSP.

9.4 Genetic algorithms

Genetic Algorithms mimic process of natural selection. The solutions are viewed as individuals that have a given fitness and that can procreate to produce offsprings. At each iteration

- Certain individuals are selected to procreate. This consist in mixing two or more solutions to produce new ones. The selection of pro-creating individuals may be made dependent of their fitness, the quality of the solutions.

- Random perturbations (mutations) are also introduced in some of the newly created individuals, to allow for more diversification. This step is important to ensure the exploration of the solution space.

- Each solution has a probability of being removed (killed), depending on their fitness, i.e. the quality of the solution. This allows keeping the number of "living" solutions small, and hence to control the computational cost of the algorithm. (Keeping all solutions alive would lead to an exponential growth of the number of solutions to be considered).

Implementing of Genetic Algorithms require specifying several concepts in addition to choosing parameters such as the death/reproduction rate, etc. In particular, one needs

A fitness function: It take into account not only the objective value of a solution but also how long this solution has "lived" and maybe more parameters.

A way to merge solutions: This is not trivial for many problems, *e.g.* how should we merge two TSP solutions?

A mutation function

Since there are many parameters to design and choose before running the algorithm, Genetic Algorithms are often tried on problems for which we do not know any other specific way to find good solutions. For example, they could be used on problems where we cannot define any sensible neighborhood which makes Local Search methods impractical. They are very flexible, but a common complain is that they can be slow.

9.5 Accuracy Guarantees: the Knapsack example

We now show how accuracy guarantees can be obtained for certain heuristic methods, beginning with a guarantee on the result of a greedy approach for the Knapsack. Remember that the integer knapsack problem can be formalized as

$$z = \max\{\sum_{j=1}^{n} c_j x_j \ : \ \sum_{j=1}^{n} a_j x_j \le b,\ x \in \mathbb{Z}_+^n\} \tag{9.4}$$

where $\{a_j\}_{j=1}^n$, $b \in \mathbb{Z}_+$. Without loss of generality we assume that $a_j \le b$ for $j \in N$ (otherwise we can immediately discard the object), and $\frac{c_j}{a_j} \ge \frac{c_{j+1}}{a_{j+1}}$ for $j \in N\backslash\{n\}$ where $N = \{1,\dots,n\}$. The greedy heuristic solution to this problem is then $x^H = (\lfloor \frac{b}{a_1} \rfloor, \lfloor \frac{b-a_1x_1}{a_2} \rfloor, \dots)$ with value $z^H = cx^H$. We now show that

$$z^H \ge \frac{z}{2}. \tag{9.5}$$

The solution to the LP-relaxed problem is $x^{LP} = (\frac{b}{a_1}, 0, \dots, 0)$ with value $z^{LP} = c_1 \frac{b}{a_1}$. The heuristic solution's cost satisfies $z^H \ge c_1 \lfloor \frac{b}{a_1} \rfloor$. Moreover, using our assumption $a_1 \le b$, we note that $(\lfloor \frac{b}{a_1} \rfloor + 1) \le 2\lfloor \frac{b}{a_1} \rfloor$. Eventually we have

$$z \le z^{LP} = c_1 \frac{b}{a_1} \le c_1(\lfloor \frac{b}{a_1} \rfloor + 1) \le 2c_1 \lfloor \frac{b}{a_1} \rfloor \le 2z^H,$$

establishing that the value of the heuristic solution is at least 50% of the optimal one.

9.6 Accuracy guarantees: Traveling Salesman Problem

There are several variations of the Traveling Salesman Problem (TSP). All are NP-hard to solve exactly, but some are even NP-hard to approximate.

A key question to determine whether approximations are tractable is the following: Suppose there exists a path p_{ij} of cost $w_{p_{ij}}$ from node i to node j (one can even consider its the minimal cost path). Suppose also that we have built a partial solution (i.e. a directed path) to the TSP that ends at i and does not contain j, and we want to extend it to go to j. Is it always possible to achieve this extension at a cost no greater than $w_{p_{ij}}$?

The answer will be yes in the two following cases:

(a) The rules of the TSP allow going several times through the same node. In that case, extending the directed path arriving at i to include j can always be achieved at a cost $w_{p_{ij}}$ by just adding the path p_{ij}.

(b) The graph satisfies the triangular inequality (and is thus complete): the cost w_{ij} of the edge (i,j) is smaller than or equal to the cost $w_{p_{ij}}$ for any path from i to j. Hence one can always extend a directed path arriving at i to j at a cost at most $w_{p_{ij}}$ by just adding the edge (i,j), node j having by assumption not been visited yet.

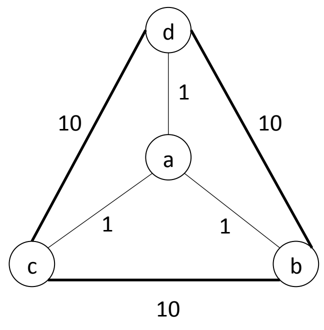


Figure 9.1: Example of graph where the shortest TSP has a cost significantly large with respect to the sum of the costs of the shortest paths between the nodes involved if one is not allowed to pass more than once through nodes.

If we are neither in situation (a) nor (b), then some a priori counter-intuitive phenomena can occur. Consider for example the graph represented in Figure 9.1. The cost of the shortest path between any of the four nodes is at most 2, hence one would think that the TSP would be achievable at a cost at most 8. However, if one is not allowed to pass more than once at a node, then one can verify that the optimal TSP cost is 22, and is achieved for example by (a, d, b, c, a). It can actually then be shown that even approximating the TSP with any fixed guarantee is NP-hard.

In the sequel we suppose that we are in situation (b), i.e. the triangular inequality holds. Our results will also apply to (a), which can be re-cast into situation (b) by working on a modified graph where every pair of nodes (i, j) is connected by an edge of weight w'_{ij} equal to the cost of the shortest path between i and j in the initial graph. The final tour is then obtained by replacing each edge of the tour obtained on this modified graph by that path to which it corresponds.

Before describing the heuristics and proving their guarantees, we need the following Lemma that will support the intuition behind these. We remind that a walk is a sequence of edges such that the starting node of an edge is always the end-node of the previous one (i.e. can be seen as a path that may pass more than once at nodes). We denote by $C(W)$ the cost of a walk W, i.e. the sum of the cost w_{ij} over all edge edges along the walk.

Lemma 9.6.1. *Consider an instance of the TSP satisfying the triangular inequality. If there exists a walk W passing through every node and finishing at its starting point, then one can construct from W a salesman tour S with cost $C(S) \leq C(W)$, at a cost proportional to the length of W.*

Proof. The idea of the proof is to build S by following W and taking a direct shortcut to the next unvisited nodes whenever W passes through nodes already visited.

More formally, take the first node v_0 of W as current node and first node of T, and mark the node. The enumerate the nodes of W one by one. At each step, if the node is not marked yet, add it to S, connecting it to the last node of T by its direct edge, and mark the node. At the end connect the last node added to the first node of S by its direct edge.

Observe that the tour S obtained passes through every node. Moreover, it can be obtained from W by replacing certain paths by direct edges. Since the weights satisfy the triangular inequality, this implies its weight is smaller than that of W. □

The rationale behind the approximation methods we describe below is then to build a walk W passing though every node and for which we can have a guarantee on the cost as compared to that of the optimal salesman tour, and to derive a tour using Lemma 9.6.1.

9.6.1 Spanning tree algorithm

The guarantee here is based on the following lemma.

Lemma 1. *The weight of the optimal salesman tour S^* exceeds that of the minimum weight spanning tree T^*, i.e. $C(T^*) \leq C(S^*)$.*

Proof. Observe that removing one edge to the optimal salesman tour yields a tree (where all nodes have degree 2 except two of them), whose weight is of course smaller than or equal to that of the tour. Besides, the weight of that tree is by definition at least as large as that of the minimum weight tree. □

The interest of this Lemma is that one can easily build the minimum weight spanning tree, using e.g. Krusal's or Prim's algorithms. Moreover, by following a depth-first search of the tree, one can pass through every node, and twice by every edge. Alternatively, observe that if we double all the edges, then all nodes have an even degree, and there exist an Eulerian tour by classical result of graph theory. In both cases, we have a tour/walk W that passes through every node and uses twice every edges, and whose cost satisfies

$$C(W) = 2C(T^*),$$

and from which, using using Lemma 9.6.1, we can generate a salesman tour S with cost

$$C(S) \leq C(W) = 2C(T^*).$$

It remains to apply Lemma 1 to conclude that $C(S) \leq 2C(S^*)$.

9.6.2 Christofidès algorithm

This more evolved heuristic yields better guarantee, and combine the bound derived from the spanning tree with the following one on perfect/near-perfect matching, and the triangular inequality. We remind the reader that a matching is a set of edges so that no nodes is incident to more than one edge of the set. A matching is perfect if all nodes are incident to one edge (this require the number of nodes to be even). Fully connected graphs, which is the case of those satisfying the triangular inequality, always admit a perfect matching if the number of nodes is even.

We begin by revisiting the spanning tree method of Section 9.6.1 in a different manner. This method can be seen as

(i) building the optimal spanning tree,

(ii) doubling all the edges so as to obtain a “double-tree" subgraph (with repetition) where all nodes have an even degree,

(iii) taking an Eulerian tour of the double-tree sub graph, i.e. a tour that uses every edges of $T^* \cup M^*$ exactly once and coming back to its initial point. The existence of this Eulerian tour is guaranteed by the fact that all degree are even. And the cost of this tour is twice the cost of the spanning tree,

(iv) if possible, reducing the cost by taking shortcut, as in Lemma 9.6.1.

The idea of Christofidès algorithm is to replace step (ii) by an alternative step that would still guarantee that all degrees become even for step (iii), while potentially introducing less edges. Rather then doubling each edge, we will just increase every odd degree by one, so that each node would have an even degree. This requires adding a number of edges equal to exactly half of the number of nodes with odd degrees, in such a way that each of these nodes is incident to exactly one such edge. The added edges are thus a perfect matching for the subgraph induced by the vertices with odd degrees in the spanning tree. To minimize the total cost, we will chose the cheapest perfect matching, which can be easily computed thanks to the Hungarian algorithm. There remains to quantify the cost of the solution obtained. We already know from Lemma 1 that the cost of the optimal spanning tree is lower than the cost of the optimal TSP. The next lemma implies that the cost of this optimal matching is less than half that of the optimal TSP.

Lemma 2.
(a) Let S^* be the optimal salesman tour on a graph with an even number of nodes and M^* the minimal weight perfect matching. Then $C(S^*) \geq 2C(M^*)$.
(b) On an arbitrary graph, the cost $C(S^*)$ of the optimal TSP is at least twice the cost $C(M^*_V)$ of the minimal perfect matching M^*_V on the subgraph induced by any subset V^* of an even number of vertices.

Proof. We first prove (a). Since the number of nodes is even, so is the number of edges in S^*. Take an arbitrary edge of S^* and assign it to a set M_A, take then the next edge along S^* and assign it to M_B, and continue this process of assigning subsequent edges alternatively to M_A and M_B until exhausting W. Since the number of edges is S^* is even, the last edges of S^* will be assigned to M_B. Hence no two subsequent edges will be assigned to the same set, and every node will be incident to exactly one edge of M_A and one of M_B. Both M_A and M_B are thus perfect matchings, which implies that their cost is larger than that of the minimal cost perfect M^*. On the other hand, we clearly have $C(S^*) = C(M_A) + C(M_B)$ since every edge of S^* has been assigned to one of the set. Hence there holds

$$C(S^*) = C(M_A) + C(M_B) \geq 2C(M^*),$$

which proves (a). Part (b) follows then from the application of (a) to the subgraph induced by V, and the observation that the optimal TSP on that subgraph is cheaper than the optimal TSP on the whole graph. □

We now formally describe the algorithm: Start from a minimum weight spanning tree T^*, and let V_o be the set of nodes having an odd degree in T^*. This set must contain an even number of nodes, and the restriction of the graph to V_o contains thus a perfect matching. Let then M_o be the minimum weight perfect matching on V_o. It follows from Lemma 2 that the optimal TSP satisfies

$$C(S^*) \geq C(M_o). \tag{9.6}$$

There remains to create a tour on which we can use that bound. Remember that every node in $T^* \cup M^*$ has an even degree, because M^* increase the degree by one precisely at those nodes with odd degrees. Hence classical results in graph theory allow building an Eulerian walk W on $T^* \cup M_o$, i.e. a tour that uses every edges of $T^* \cup M_o$ exactly once (and coming back to its initial point). Using Lemma 9.6.1, we can then obtain a tour S satisfying

$$C(S) \leq C(W) = C(T^* \cup M_o) = C(T^*) + C(M_o) \leq C(S^*) + \frac{1}{2}C(S^*) = \frac{3}{2}C(S^*),$$

where we have used (9.6) and Lemma 1 for the last inequality.

The algorithm produces thus a tour whose cost is at most 50% larger than the optimal cost. From a practical point of view, 50% remains large for a worst-case guarantee. However, the algorithm often perform much better, and can for example also be used to generate a reasonable initial solution that can then be improved using other algorithms.

9.7 Heuristic for MILP Branch and Bounds

In this last section we present heuristic methods to attack MILP problems.

9.7.1 Dive and fix

The idea of this method is to find a feasible by rounding part of optimal solutions to the LP relaxation. Algorithm 2 presents a possible pseudocode for this strategy.

```
Data: An Integer Program $\mathcal{P}$
while $\exists i$ s.t. $x_i \notin \mathbb{Z}$ do
    $x^* = \text{Solve}(LP(\mathcal{P}))$ ;
    if $LP(\mathcal{P})$ was infeasible then
        return Failure;
    else if $x^*$ is a feasible solution of $\mathcal{P}$ then
        return $x^*$;
    else
        Set $x_i = [x_i^*]$ for some chosen $i$;
    end
end
if $x$ is a feasible solution of $\mathcal{P}$ then
    return $x$;
else
    return failure;
end
```

Algorithm 2: Dive and Fix algorithm

Although it is often very efficient, this method provides absolutely no guarantee:

- It may not find any feasible solution
- If it finds one, the solution may be very poor

We also need to be able to choose which variable x_i^* to round off. This choice can be based on coefficients of A, b and c, or we can simply try rounding off the x_i^* whose value is closest to an integer.

9.7.2 Relax and Fix

Suppose that the variables x of the MILP problem can be decomposed in (x^1, x^2), where (i) x^1 represent the important long term or strategic variables, such as the decision to open a factory, etc., and (ii) x^2 represent some less important variables such as the specific amount to be produced.

The relax and fix approach consists in first relaxing the less important x^2 to real numbers and solving the simpler MILP problem obtained where only x^1 is subject to integrity constraints. One then fix x^1 to the optimal x^{1*} of the relaxed problem, and solve for x^2, having re-introduced the integrity constraint. The idea is thus to first make the important decisions based on a non integer approximation for the less important variables, and then to adapt the less important variables once those important decisions have been made. This technique can also be appropriate if x^2 can be decomposed in several subsets of variable that are uncouple once x^1 has been fixed.

In some cases one can also alternatively fix and relax x^1 and x^2: one would first relax x^2 and solve for x^1. Then fix x^{1*} and solve for x^2, then fix the x^{2*} obtained an re-solve for x^1, etc.