4

Integer Linear Programming

A mixed-integer linear program (MIP) consists in the minimization of a linear function subject to a finite set of linear constraints, *plus* the restriction that some variables are only allowed to take integer values. In general we have:

$$\begin{aligned} &\min cx \\ &\quad a_i x \sim b_i \quad i = 1, \dots, m \\ &\quad l_j \leq x_j \leq u_j \quad j = 1, \dots, n = N \\ &\quad x_j \in \mathbb{Z} \quad \forall j \in J \subseteq N = \{1, \dots, n\} \end{aligned} \tag{1}$$

If $J = N$ we have a pure integer program, otherwise we have a mixed integer linear program. Of course, if $J = \emptyset$ we are back to linear programming.

This is deceptively similar to a standard linear program, as the only difference is the *integrality* requirement. However, the change is significant: on the one hand, the paradigm has a far wider applicability; on the other hand, this makes the problem NP-hard, so not only we don't know a polynomial algorithm for solving MIPs, but one is very unlikely to exist. We note that the integrality constraint is inherently *non-linear*, and it makes the feasible region *non-convex*, as it is not true that any convex combination of two integer solutions is also integer. However, we still treat integer programming in a convex optimization course because, as we will see, not only it is fundamental paradigm in mathematical optimization, but the state-of-the-art solution methods are based on convex optimization techniques.

It can be expressed, for example, as $\sin(\pi x_j) = 0$ *for each integer variable* x_j.

4.1 Modeling

In this section, we will give examples of constructs that frequently appear in optimization problems, and that can be modeled within the MIP framework, while they could not, in general, formulated as LPs.

4.1.1 *Yes/no decisions*

The simplest, yet most common, case is that of *yes/no* decisions: those can be naturally modeled with so-called *binary* variables, i.e., integer variables with a lower bound of 0 and an upper bound of 1, like in the knapsack example in the introduction. This is already beyond the reach of LPs, as, for a given binary variable x, the best approximation of its domain $\{0, 1\}$ a linear program can achieve is the whole interval $[0, 1]$. We will see that those binary decisions

appear very frequently in MIP models, either naturally or as an artificial tool to encode other constructs.

4.1.2 *Discrete values*

We do not assume those values to satisfy any pattern, we just need their number k to be finite.

Let us consider a variable y whose domain is a finite set of arbitrary values $v_1, \dots, v_k$. Again, the a best approximation a LP can give is the whole interval $[v_1, v_k]$, while within MIP we can interpret the choice of which value to assign to y as the k yes/no decisions "does variable y get value v_i?", that we can encode with k binary variables x_i, linked together by the fact that we need to choose exactly one value:

$$\begin{aligned} y &= \sum_{i=1}^{k} v_i x_i \\ \sum_{i=1}^{k} x_i &= 1 \\ x_i \in \{0,1\} \quad \forall i &\in \{1, \dots, k\} \end{aligned}$$

4.1.3 *Fixed costs*

Let us now consider a scenario where the objective cost a variable x includes some fixed cost, i.e., we have a unit cost a but also a fixed component b that we have to pay whenever we produce some amount, independent of its magnitude. In other words, we have the cost structure:

$$z = c(x) = \begin{cases} ax + b & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$$

For technical reasons that will become clear later, we need to assume that when the production level is positive, it is contained in some interval $[\varepsilon, U]$. With those assumptions, we can again use an auxiliary binary variable y to encode in the model whether we are in the first or second case, i.e.,

$$y = \begin{cases} 1 & \text{if } \varepsilon \le x \le U \\ 0 & \text{if } x = 0 \end{cases}$$

and obtain the MIP model:

$$\begin{aligned} z &= ax + by \\ \varepsilon y \le x &\le U y \\ y &\in \{0,1\} \end{aligned}$$

The correctness of the above model is easily shown, as we just need to consider the two cases $y = 1$ and $y = 0$: in the first case we obtain $\varepsilon \le x \le U$ and $z = ax + b$, while in the second $x = 0$ and $z = 0$, as required.

4.1.4 *Disjunctions*

When we write two constraints in a mathematical optimization model, we are expressing the fact that we want both of them to be satisfied in any feasible solution: in other words, we are considering their *conjunction*. There are

however cases where we need to express a so-called *disjunction* of two (or more) conditions, i.e., it is part of the solution process to figure out which constraint needs to be satisfied and which not (provided at least one is, of course). We implicitly just saw an example of this scenario in the fixed cost case, but this is far more common: for example, in scheduling applications, we need to enforce (disjunctive) conditions between pairs of jobs, stating that either one has to start after the other has completed, or the other way around. Note that we cannot make this decision ourselves before writing the model, as figuring out the order in which jobs must be executed is exactly the optimization problem we are trying to solve.

With this premise, let us consider a case of a disjunction of two arbitrary linear constraints:

$$(a_1^\top x \leq b_1) \vee (a_2^\top x \leq b_2)$$

Again, for technical reasons, let us also assume that the domains of the variables involved are finite, i.e., $l \leq x \leq u$, for some $l \in \mathbb{R}^n$ and $u \in \mathbb{R}^n$. The trick is, again, to introduce a binary variable for each term of the disjunction, expressing whether the corresponding condition should be enforced. A MIP model thus reads:

This approach of having an explicit binary variable that captures whether some condition is true in the model is sometimes called reification.

$$\begin{aligned} a_1^\top x &\leq b_1 + M(1 - y_1) \\ a_2^\top x &\leq b_2 + M(1 - y_2) \\ y_1 + y_2 &\geq 1 \\ l \leq x &\leq u \end{aligned}$$

The constants $M \gg 0$ serve the purpose of deactivating a given constraint whenever the corresponding binary variable is set to zero, and are called *big-M* coefficients. Their existence is guaranteed by our assumption that the variables x are all bounded: in this case the linear expressions $a_i^\top x$ are clearly bounded themselves, , and thus there exists a large enough value for the right-hand side that makes it redundant.

We can easily upper bound an expression $a^\top x$ by taking the upper bound u_j whenever the coefficient $a_j > 0$, and the lower bound l_j otherwise.

4.1.5 *Piecewise linear functions*

As a last example, let us consider a piecewise-linear (PWL) cost function $f(x)$. A PWL function can be described by a (finite) sequence of knots $(a_i, f(a_i))$ for $i = 1, \ldots, k$, as depicted in Figure 4.1.

Piecewise linear functions appear naturally in many applications, for example as approximations of nonlinear expressions. The intuition behind a MIP model for such a construct is that we need to choose which of the $k - 1$ segments we are in, and this is a discrete decision: once that is done, any point in the chosen segment can be expressed as convex combination of its endpoints, something that can easily be modeled with linear constraints. The MIP model then reads:

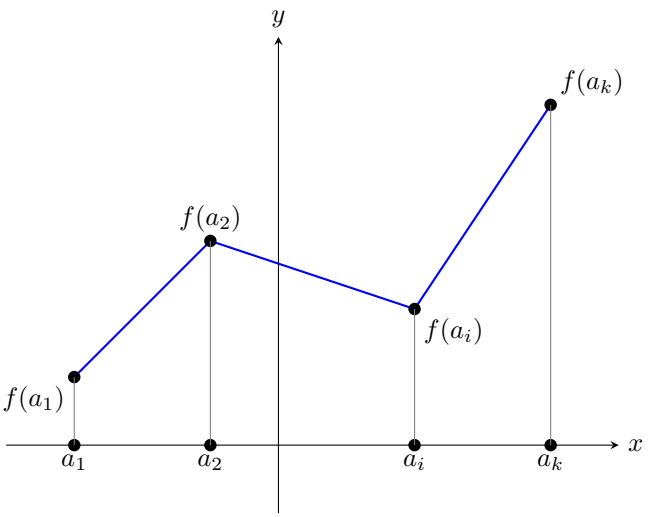

Figure 4.1: A piecewise-linear function.

$$
\begin{aligned}
x &= \sum_{i=1}^{k} a_i \lambda_i \\
y &= \sum_{i=1}^{k} f(a_i) \lambda_i \\
\sum_{i=1}^{k} \lambda_i &= 1 \\
\sum_{i=1}^{k-1} y_i &= 1 \\
\lambda_1 &\leq y_1 \\
\lambda_i &\leq y_{i-1} + y_i && 1 < i < k \\
\lambda_k &\leq y_{k-1} \\
\lambda_i &\geq 0 && 1 \leq i \leq k \\
y_i &\in \{0, 1\} && 1 \leq i < k
\end{aligned}
$$

The binary variables y_i encode the choice of the segment, while the continuous variables λ_i encode the multipliers in the convex combination. Each binary variable *activates* only the multipliers of the endpoints of the corresponding segment.

4.1.6 *MIP representability*

In the previous examples, in some cases we had to assume some technical restrictions in order to give a MIP formulation, but those were never properly justified. Now we finally address those, by answering the question: which sets can be formulated as MIPs? For LP, the answer is simple: a set is *LP-representable* if and only if it is a polyhedron. But about MIP? Let us start with what it means to be MIP-representable:

Definition 4.1. *A set $S \subseteq \mathbb{R}^n$ is* MIP-representable *if there exists a constraint set of the form*

$$\begin{aligned} Ax + Bu + Dy &\leq b \\ x \in \mathbb{R}^n, u \in \mathbb{R}^m, y_k &\in \{0,1\} \forall k \end{aligned}$$

such that the projection of its feasible set onto x is S.

In other words, we can describe S with linear inequalities and the integrality constraint, allowing the possibility to do so in a higher dimensional space by introducing additional integer and/or continuous variables.

All the examples shown so far indeed required some additional variable.

Example 4.1. *Consider the set $S \subset \mathbb{R} : [0,1] \cup [3,4]$, the solution set of the absolute value constraint $|x-2| \leq 1$ for a variable x whose domain is $[0,4]$. This set is clearly not representable with linear inequalities in the x space (it is nonconvex), but by adding a single binary variable y we can obtain the linear system:*

$$\begin{aligned} x &\geq 3y \\ x &\leq 1 + 3y \\ y &\in \{0,1\} \end{aligned}$$

and it easy to check that we obtain the interval $[0,1]$ for $y = 0$ and the interval $[3,4]$ for $y = 1$, hence the projection of the solution set onto x is exactly S.

Now that we have formally defined MIP-representability, we are ready to give a full characterization, due to Jeroslow:

Theorem 4.1. *A set $S \in \mathbb{R}^n$ is MIP-representable* if and only if *it is the union of finitely many polyhedra* with the same recession cone.

The intuition is thus that MIP generalizes LP by allowing as feasible set not just a single polyhedron, but the union of finitely many ones (and we use the integrality constraint to encode the discrete decision of selecting which of them contains the optimal solution). Note however that those polyhedra cannot be arbitrary, but need to satisfy the technical condition of all having the same recession cone. This is why, for example, we had to introduce bounds on x in the fixed cost case: without a lower bound the set corresponding to the case $x > 0$ is not closed, and without an upper bound it does not have the same recession cone as the polyhedron for $x = 0$. With the bounds in place, we have that our feasible set is the union of two polyhedra (a point and a segment), and this can indeed be modeled with MIP.

The recession cone of a polyhedron P is the set of directions r along which we can move indefinitely from any point $x \in P$ without leaving the polyhedron, i.e., such that $x + \alpha r \in P$ for any $x \in P$ and for any $\alpha \geq 0$.

A corollary of Jeroslow's theorem is that any combinatorial optimization problem is MIP representable, as its feasible set is by definition made of a finite number of points, and this somehow is yet another evidence that the MIP paradigm is far more general that it would seem at first sight. However, we must be aware of a serious limitation of the characterization above: it tells us only when a MIP formulation exists, but it gives no guarantees on its size.

Again, think about the arbitrary combinatorial problem example.

4.2 Algorithms

Consider a pure integer program: the linear constraints still define a polyhedron in $\mathbb{R}^n$, but the integrality constraint means that we are only interested in the integer points within such polyhedron. A direct consequence of this simple geometric intuition is that the fundamental property of linear programming, i.e., that there exist at least one optimal vertex in case of finite optimum, is

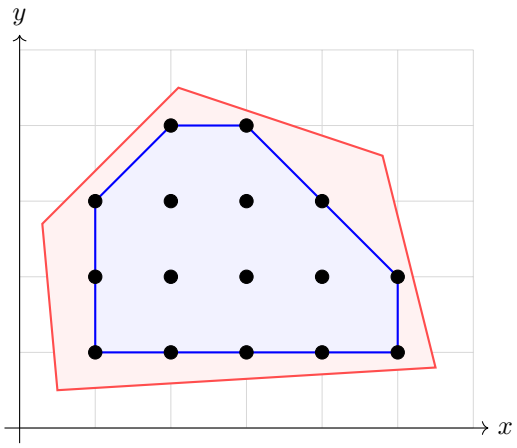

Figure 4.2: Different formulations for the same integer set.

no longer true for general MIPs, as the optimal vertex does not necessarily have integer coordinates, see for example the red polyhedron in Figure 4.2.

The same geometric intuition also shows a fundamental property of MIP: the formulation of a problem is *not* unique, as there are infinitely many different polyhedra that describe the very same set of integer points, and that are thus equivalent as long as the integrality constraint is enforced. Interestingly, those formulations can behave very different if we remove the integrality constraint and consider the underlying LP: some provide a very loose approximation, while others are much tighter. There is even an *ideal* formulation, called the *convex hull* formulation, where all vertices are integral, and thus would to solve the integer program as a *single* LP. Unfortunately, this ideal formulation is in general unknown, or very difficult to determine, and it might even contain an exponential number of inequalities, hence this insight does not (directly) yield a practical solution strategy.

The blue polyhedron in Figure 4.2.

To summarize, on the one hand, we need to design a completely new type of solution algorithm; on the the other hand, given the tight connection between LP and MIP, we would like to "recycle" our ability to solve linear programs efficiently. This leads to two general concepts in mathematical optimization, namely *search* and *relaxations*.

It is however the foundation for the theory of cutting plane *methods, a fundamental tool in MIP.*

4.2.1 *Search*

Given a problem P, performing *search* means solving a sequence of restrictions $P_1, \ldots, P_m$ of P. Each *restriction* P_k is an optimization problem on its own, obtained from P by adding constraints. The basic idea is that by adding constraints, and thus reducing the solution space, the optimization problem can become significantly easier to solve.

We talk about *exhaustive search* if the set of restrictions *covers* the solution space of P, i.e., if

$$\bigcup_i F(P_i) = F(P)$$

Thus, by exhaustive search we can actually solve the original problem P. We note however that, in the context of *heuristics*, i.e., algorithms that do not provide optimality guarantees, we are not necessarily restricted to exhaustive search: for example, in a typical *local search* scheme, we explore a dynamically

construct a sequence of restrictions (the neighborhoods of the feasible solutions visited), without covering the full solution space.

The main property of a (single) restriction is that any of its feasible solutions is also a feasible solution for the original problem, and thus gives a upper bound on the optimal value of P.

The simplest form of search is called *generate-and-test*, and consists in generating explicitly all possible solutions $x \in D$, check which of them are feasible, and keep the best objective-wise. This is equivalent to solving all the restrictions of P in which x is fixed to a particular value $v \in D$. Needless to say, this is a reasonable strategy only for the smallest problems.

A more elaborate form of search, which is the basis for most enumerative algorithms used in practice, is the so-called *tree search*, where the solution space of P is split recursively until we obtain restrictions that are "easy enough" to solve (those are the leaves of the search tree). Whether tree search is an improvement over generate-and-test depends on the definition of easy enough: the weakest possible strategy would wait until all variables are fixed before counting the restriction as solvable, but that means generating a tree where the leaf nodes are exactly all the candidate solutions needed by generate-and-test, with the additional overhead of the intermediate nodes. So, the method has chance to work only if we can do something smarter: and this brings directly to the concept of relaxations.

4.2.2 *Relaxation*

A *relaxation* of an optimization problem P is again another optimization problem R obtained by P:

1. removing some constraints and/or
2. replacing the objective $f(x)$ with a lower approximation $g(x)$.

More formally, R is a relaxation of P if

1. $F(P) \subseteq F(R)$
2. $g(x) \leq f(x) \quad \forall x \in F(P)$

A grahical example is depicted in Figure 4.3.

The reason why we solve relaxations is basically the same as for restrictions: if carefully constructed, a relaxation R might be much easier to solve than P, but still provide meaningful information about P.

In particular, we have the following properties:

1. if $F(R) = \emptyset$, then P is infeasible.
2. if x^* is an optimal solution for R, $g(x^*)$ is a valid lower bound on the optimal value of P.
3. if x^* is an optimal solution for R, $x^* \in F(P)$ and $g(x^*) = f(x^*)$, then x^* is an optimal solution for P.

Note that the properties of a relaxation are complementary with those of a restriction. Indeed, the two are often used in combination to speed up search algorithms, in particular by considering relaxations R_k of restrictions P_k in a tree search.

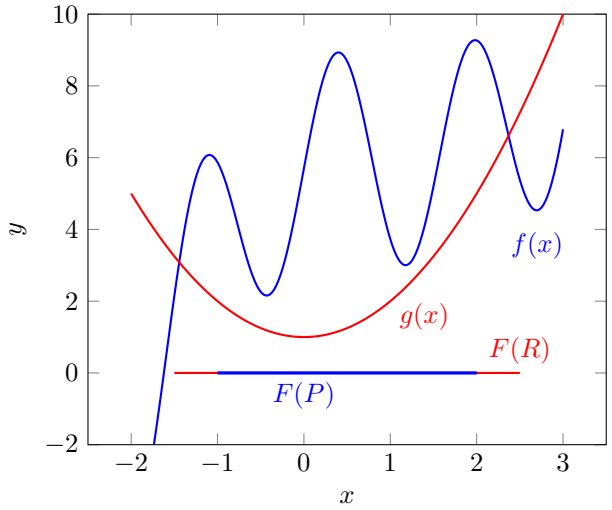

Figure 4.3: Relaxation R of an optimization problem P. In this example we both enlarged the feasible region and replaced the objective with a lower approximation.

4.3 Branch-and-bound

The branch-and-bound (B&B) algorithm is a general solution technique based on the implicit enumeration of the feasible solution space $F(P)$ of an optimization problem P. The main idea is very simple: do not explore a region of the search space if you can easily prove that it cannot contain any feasible solution better than the ones that are already known.

The B&B algorithm uses a *divide-and-conquer* strategy to partition the original solution space into subproblems. Here is a brief description of the method.

Let $F(P)$ be the set of feasible solutions of P, $c : F(P) \to \mathbb{R}$ the objective function to minimize, and $\overline{x} \in F(P)$ a known feasible solution, obtained for example by some heuristic. The value $z = c(\overline{x})$ of such solution, which is called the current *incumbent*, gives an upper bound on the optimal value of P. The algorithm proceeds as follows:

- In the bounding phase, we construct a relaxation R of the current problem. We solve this relaxation and, if the relaxation is infeasible, or its optimal solution is feasible for P, or if its objective value is greater or equal to z, then we are done.
- Otherwise, we construct a *partition* F^* of $F(P)$, i.e., a finite set F^* of subsets of $F(P)$, such that:

$$\bigcup_{F_i \in F^*} F_i = F(P)$$

  Any subset F_i is a child of $F(P)$. This phase, called *branching*, is justified by the fact that if F^* is a partition of $F(P)$, then:

$$\min\{c(x) \mid x \in F(P)\} = \min\{\min\{c(x) \mid x \in F_i\} \mid F_i \in F^*\}$$

  The children of $F(P)$ are added to a queue $\mathcal{A}$ of subproblems yet to be processed.
- We pick a subproblem P_i from the queue and solve its relaxation. There are four possible cases:

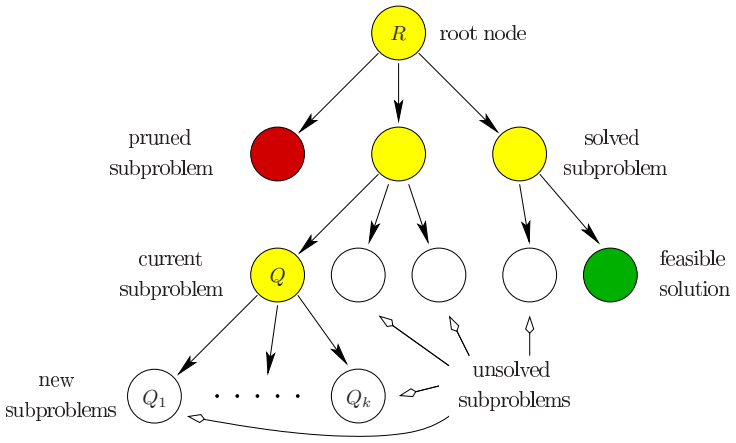

Figure 4.4: Tree search in B&B.

1. We find a new feasible solution better than $\overline{x}$. Then we update $\overline{x}$.
2. If the subproblem relaxation is infeasible, we prune the subproblem (*pruning by infeasibility*).
3. Otherwise we compare the objective value of the relaxation with z. If it is greater or equal, then we can again prune the subproblem (*pruning by optimality*).
4. Finally, if it was not possible to prune the subproblem, we need to branch again, and put the new subproblems into the queue.

- The algorithm terminates when the queue $\mathcal{A}$ becomes empty. At that point the current incumbent is the proven optimal solution.

The pseudocode for algorithm B&B is given in Figure 2, while a graphical representation is depicted in Figure 4.4.

Algorithm 2: B&B scheme

S1: $\mathcal{A} \leftarrow \{P_0\}$, $z = \infty$
S2: Pop a subproblem P_i from $\mathcal{A}$. If $\mathcal{A}$ is empty go to step S8
S3: Solve a relaxation R_i of P_i. If the solution is feasible for P_i, go to step S6
S4: If $g(R_i) \geq z$ go to step S7
S5: Decide a partition F_j^* for $F(P_j)$ and add the corresponding subproblems to $\mathcal{A}$
S6: If $f(P_i) < z$, then update $z = f(P_i)$ and the incumbent $\overline{x}$
S7: $\mathcal{A} \leftarrow \mathcal{A} \setminus \{P_i\}$. Go to step S2
S8: If $z = \infty$ the problem is infeasible, otherwise $\overline{x}$ is the optimal solution

It is worth noting that:

- The method can be started even without an incumbent. In this case the method is still correct but in general less efficient, as we can prune nodes only by feasibility and not by optimality, until the first incumbent is found.
- The choice of the relaxation is the most crucial: it must be sufficiently easy to solve, but at the same time yield a strong lower bound to prune nodes as early as possible. Needless to say, those two are conflicting goals.

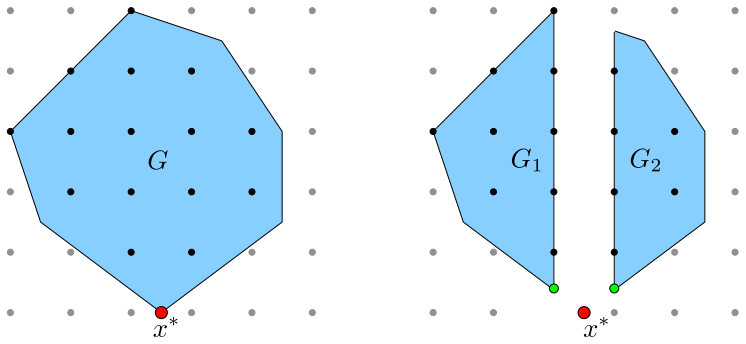

Figure 4.5: Branching on LP relaxation x^*.

The generic B&B algorithm above specializes in a straight-forward fashion for the MIP case. We solving a MIP, the most common choice is to use the LP relaxation of the subproblems. If the LP relaxation is not integer (note that the integrality requirement is the only relaxed constraint), then we can pick a variable x_j with a fractional value x_j^* in the current solution and construct the partition (or disjunction):

$$x_j \leq \lfloor x_j^* \rfloor \vee x_j \geq \lceil x_j^* \rceil$$

The process is depicted in Figure 4.5.