Università di Pisa

Dipartimento di Informatica

Technical Report

Optimization Methods: an Applications-Oriented Primer

Antonio Frangioni
Dipartimento di Informatica, Università di Pisa
Largo B.Pontecorvo 3, 56127 Pisa, Italia, frangio@di.unipi.it

Laura Galli
Dipartimento di Informatica, Università di Pisa
Largo B.Pontecorvo 3, 56127 Pisa, Italia, laura.galli@unipi.it

ADDRESS: Largo B. Pontecorvo 3, 56127 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

Optimization Methods: an Applications-Oriented Primer

Antonio Frangioni
Dipartimento di Informatica, Università di Pisa
Largo B.Pontecorvo 3, 56127 Pisa, Italia, frangio@di.unipi.it

Laura Galli
Dipartimento di Informatica, Università di Pisa
Largo B.Pontecorvo 3, 56127 Pisa, Italia, laura.galli@unipi.it

Abstract

Effectively sharing resources requires solving complex decision problems. This requires constructing a mathematical model of the underlying system, and then applying appropriate mathematical methods to find an optimal solution of the model, which is ultimately translated into actual decisions. The development of mathematical tools for solving optimization problems dates back to Newton and Leibniz, but it has tremendously accelerated since the advent of digital computers. Today, optimization is an inter-disciplinary subject, lying at the interface between management science, computer science, mathematics and engineering. This chapter offers an introduction to the main theoretical and software tools that are nowadays available to practitioners to solve the kind of optimization problems that are more likely to be encountered in the context of this book. Using, as a case study, a simplified version of the *bike sharing problem*, we guide the reader through the discussion of modelling and algorithmic issues, concentrating on methods for solving optimization problems to proven optimality.

Keywords: *Optimization methods, combinatorial optimization, integer programming, bike sharing*

1 Introduction

Effectively sharing resources is a complex task. A benefit of having dedicated resources (say, a personal car) for a given task is that there is no need to take complex decisions about their use, as they are always available. The obvious drawback is that they can be heavily under-utilized, thereby substantially decreasing their value while consuming other valuable resources (say, scarce parking space). Allowing a resource to be shared may dramatically increase its social and economic value, but it also opens up a number of complex issues regarding its fair and efficient management that need to be solved for the system to deliver its potential benefits. Besides the many technological aspects (say, self-driving capabilities, accurate navigation, reliable communication, ...) that are crucial to make sharing possible, the ability of taking optimal decisions about highly complex systems (say, a large fleet of self-driving cars) is also a fundamental pre-requisite for the sharing economy to thrive.

Many, although not necessarily all, of the decision procedures can be cast under the form of *optimization problems*, where one *objective function* depending on the *decision variables* has to be optimized (minimize the effort or maximize the benefit) along all possible states of the system, typically represented via *constraints*. Problems of this kind are pervasive and can be found e.g. in design of structures and trajectories, production planning, resource allocation, scheduling, control, and many others. The development of mathematical and software tools for the solution of these problems is a highly inter-disciplinary branch of applied mathematics lying at the interface with diverse other subjects, including (but not limited to) management science, computer science, and engineering, and is often generally referred to as *Operations Research* (OR). At its heart, OR is based on mathematical techniques that can be traced

back to Newton and Leibniz, and that have seen a tremendous development since the second half of last century together with the meteoric rise of availability of computational power. This is indeed relevant, as optimization problems are generally "difficult". Without going into the mathematical details, suffices here to say that the worst-case computational cost of the best known algorithms for solving most optimization problems grows extremely rapidly (exponentially) with the size of the problem, and not just in theory: the increase in complexity often shows off in practice. This motivated an enormous effort to develop efficient (as much as possible) algorithms for different classes of problems, exploiting all available *structure* that each one has. The result is a complex panorama of many different *classes* of optimization problems: for instance, it is customary to distinguish between *continuous optimization* where decision variables can take, say, real values, *combinatorial optimization* where solutions belong to some combinatorial space (sets, paths, trees, ...) (a.k.a. *discrete optimization* since usually decision variables are restricted to only attain discrete values), and *functional optimization* (a.k.a. *optimal control*) where solutions are functions. Each of these classes typically requires different mathematical tools to be addressed (say, analysis for continuous and functional optimization and combinatorics for combinatorial optimization), although in practice the boundaries are blurred, and efficient optimization methods typically borrow from several different areas of mathematics and computer science.

Using mathematical optimization in practice is therefore a complex task that requires navigating nontrivial trade-offs. The process typically starts by constructing a *mathematical model* trying to capture the "essence" of one's practical system to be managed. However, the model should attain the right compromise between correctly representing the reality at hand, and admitting "sufficiently efficient" algorithms that can provide the desired solutions in a time compatible with the constraints that the actual use case imposes (which can be rather tight). Therefore, a crucial decision while developing a mathematical optimization model is what *class* of optimization problems one selects to express the model into, since this dictates what (mathematical, hence algorithmic and ultimately software) tools one has at disposal to actually solve it, and therefore the chances to obtain a practical system. Indeed, it is one of the most important cultural legacies of the last century—from Gödel's incompleteness theorems to the discovery of non-computable functions down to complexity theory—that just being able to write a mathematical model of one's system does not imply that solutions can actually be found. Thus, using optimization methods requires in principle some understanding of the several "core techniques" that have been developed to tackle different kind of problems, such as linear, non-linear, combinatorial and stochastic optimization, calculus of variations, optimal control, game theory, and several others.

However, for problems in contexts like the one of this book, the choices are typically limited. In fact, the problems are usually large-scale (thousands to hundreds of thousands of decision variables), with rather complex constraints encompassing both discrete and continuous decisions. In order to have any chance of solving problems of this type, in most cases one has to restrict to the class of *Mixed Integer Linear Programs* (MILP), which only allows linear relationships between decision variables, thereby forcing to use somehow un-natural constructs for modelling certain relationships. However, the class is expressive enough to allow to represent very many systems, and the vast majority of optimization models in the applications of interest for this book (logistics, transportation, telecommunications, ...) are customarily written as MILP. Furthermore, many software tools are currently available for solving this class of problems. Which does not mean that MILP are not "difficult" problems: in general, the complexity of algorithms for their (exact) solution is exponential. However, due to more than 70 years of continuous research by a large community, the current status of solution algorithms for MILP is such that for several practical problems, just writing the MILP model and passing it to a solver may be enough to obtain solutions with a "reasonable" computational effort. Although there is no guarantee that this will happen for any specific application, the effort for testing this approach is low enough that most often this is the first step that should be attempted (unless perhaps if the problem is extremely-large scale, or needs to be solved in extremely short time, or has many complex nonlinear relationships); with any luck, it may also be the only step that is needed.

In this chapter, after an introduction about general optimization models in Section 2 we will therefore concentrate on MILP. The concepts will be illustrated by means of a case study inspired by a bike sharing system, described in Section 3. In Section 4 we will concentrate on a crucial and nontrivial aspect, which is that there are many possible ways to *formulate* the same system as a MILP model, and that this choice

is actually crucial for the effectiveness of the solution algorithms. Finally, in Section 5 we will provide pointers for further reading on the subject.

2 Optimization problems

A quite general (although not the most general) optimization problem is

$$\text{(P)} \quad \min\{\, f(x) \,:\, G(x) \leq 0 \;\;,\;\; x_i \in \mathbb{Z} \quad i \in I \,\} \;\;,$$

where $x = [\, x_1 \,, \ldots \,, x_n \,]$ is the (*finite*) vector of *decision variables*. When not otherwise specified, each x_i can attain real values, i.e., $x \in \mathbb{R}^n$; this already restricts (P) to a finite-dimensional space, thereby excluding problems where x_i may be, say, itself a function. The *objective function* $f(x)$ and the (*finite number of explicit*) *constraints* $G(x) = [\, g_j(x) \,]_{j=1,\ldots,m}$ are usually assumed to be real-valued functions, i.e., $f : \mathbb{R}^n \to \mathbb{R}$ and $G : \mathbb{R}^n \to \mathbb{R}^m$. For f, this implies that the decision-maker only has one objective, which is often not true in practice as there could be any number of—possibly, contrasting—measures to be optimized. Yet, this is necessary to univocally define the concept of *optimal solution*; with a vector-valued $f(x) = [\, f_h(x) \,]_{h=1,\ldots,k} : \mathbb{R}^n \to \mathbb{R}^k$, the lack of a complete order in $\mathbb{R}^k$ (for $k > 1$) means that one has rather to resort to the concept of *nondominated* (a.k.a. *Pareto-optimal*) solution, but this would mean that (P) does not identify *one* solution that can be automatically taken as the answer without human oversight, as it is often necessary to do. Thus, in presence of multiple objective functions it is customary to either form a weighted sum ($f(x) = \sum_{h=1,\ldots,k} \alpha_h f_h(x)$, a.k.a. *weighting*) or define thresholds on the least acceptable value for all objectives save one (*budgeting*), and fall back to the single-objective case. Even so, f and G cannot clearly be just "any" function, as there are functions that simply cannot be computed; in general, the assumption is that they are "easy" to compute, most often algebraic functions. Yet, even restricting (P) to algebraic functions is not enough to tame its complexity: even with simple polynomials, the problem may be undecidable (provably, for some instances there cannot be any solution algorithm). This justifies why it is customary to work with *classes of optimization problems* corresponding to restricting the shape of f and G. Indeed, by far the most common class of optimization problems for the applications of interest here is that of *Mixed-Integer Linear Programs* (MILP), where both f and G are *linear*; that is,

$$\text{(MILP)} \quad \min\{\, cx \,:\, Ax \leq b \;\;,\;\; x_i \in \mathbb{Z} \quad i \in I \,\} \;\;,$$

where $c \in \mathbb{R}^n$ is the *cost* (row) *vector*, $A \in \mathbb{R}^{m \times n}$ is the *constraint matrix*, $b \in \mathbb{R}^m$ is the (column) vector of *right-hand sides*. This finally justifies why, besides the explicit constraints, (P) also has *integrality constraints* on a (possibly empty) subset $I \subseteq N = \{\, 1 \,, \ldots \,, n \,\}$ of the variables; although these could be expressed in an algebraic way, doing so would require "complex" functions.

Even with such a dramatic restriction to the set of available functions, MILP is a "hard" problem. However, at least it is now "clear where the hardness comes from": with no integrality constraints ($I = \emptyset$) the problem becomes a *Linear Program* (LP), which is instead "easy". Formally, this means that there are solution algorithms whose complexity grows at most as a polynomial (although, not a very low degree one) in the size of the problem. In practice, this means that LPs with up to hundreds of thousands of variables and constraints can routinely be solved with standard approaches on standard PCs, and that LPs with up to billions of variables can be approached with appropriate techniques on HPC systems. As we shall see in Section 4, this is a crucial property upon which all the available solution approaches to MILP rely. Indeed, the fact that LP admit "efficient" algorithms is the main reason why MILP is the most widely used class of optimization problems: although MILPs are in general "hard", formulating one's problem in this shape allows to exploit the huge amount of research, and the many available actual software products, dedicated to their solution. The advances in those approaches over the last decades have been such that many practical problems, despite being theoretically "hard", can nowadays be routinely solved by just applying off-the-shelf technologies once they have been *properly* formulated as MILPs.

Yet, doing so is not straightforward, for two reasons. The first is that while MILP is a quite "expressive" class, in the sense that an enormous number of different relevant problems can be written as such, the severe restriction of being only able to use linear functions requires getting familiar with a number of *modelling tricks*, whereby the limitations of the framework are sidestepped by clever contraptions. The

second, and perhaps more important, is that there are usually very *many different MILP formulations* of a given problem, which are typically *not* equivalent in terms of computational cost. Quite on the contrary, writing the "right" MILP formulation of the problem can easily be the deciding factor in being able to solve it, with orders-of-magnitude differences between apparently similar formulations being not uncommon. All in all, there is no systematic way to formulate an optimization problem as a MILP, and devising a good model is often more of an art than of a science. However, some general guidelines can be provided, which is what we will attempt in the rest of this chapter. In particular, the next section, using one specific case study, will focus on the process of writing a MILP formulation out of the "informal" description of one's situation. Then, in Section 4 we will very briefly summarize the main concept underlying the most effective (least ineffective) solution methods for MILP, as doing so allows to pinpoint the characteristics that a "good formulation" should ideally possess.

However, to conclude this section it is important to remark that MILP is not the most general class of optimization problems for which (relatively) efficient solution tools are available. Indeed, it is possible to "slightly" enlarge the set of functions available to express f and G while keeping a solution cost comparable with that of a MILP of the same size. For instance, this is possible with Mixed-Integer Quadratic Programs (MIQP) that have *convex quadratic functions* of the form $f(x) = x^T Qx + qx$, with Q a positive semidefinite matrix. More in general, *Second-Order Cone constraints* of the form $x_n \geq \sqrt{\sum_{i=1}^{n-1} x_i^2}$ can be used as well; a surprising number of different nonlinear functions can be formulated as such, and the best current software tools can usually handle Mixed-Integer Second-Order Cone Programs (MISOCP) efficiently. Even Mixed-Integer Nonlinear Programs (MINLP) with general *convex functions* can be tackled with similar approaches, although with a somehow lower efficiency. Finally, there are tools capable of solving (or attempting to) MINLP where f and G are general algebraic or transcendental functions; however, their efficiency in practice can easily be orders of magnitude lower than those for MILP problems of comparable size. All in all, the best course of action when devising a mathematical model is most often to start with a MILP one, unless perhaps if the practical problem have either complex nonlinear relationships that cannot reasonably be simplified, or simple nonlinear relationships that can be expressed within the classes of functions that make MINLPs "not too much more difficult" than MILPs.

3 Case Study: the bike sharing problem

Bike Sharing (BS) is an urban mobility service that makes public bicycles available for shared use. The bicycles are located at some stations distributed across the urban area. Customers can take a bicycle from one of the stations, use it for a journey, then leave it at a possibly different station, paying according to the time of usage. This is an important service in *green logistics*, in that on one hand it helps to decrease traffic and CO_2 emissions, and on the other hand it offers a partial solution to the so-called "last mile problem" related to proximity travels.

The cost of operating a BS system has several components: the setup costs (i.e., buying and installing stations and bikes), the cost of the back-end system to operate the equipment, and daily operating costs like maintenance, insurance, and the cost of redistributing bikes among the stations. Ideally, this being a "system" one should optimize all its aspects symulteneously; however, in practice this is usually impossible. For once, different costs are incurred at different times: setup costs are paid once, before the system can start operating, and the consequences of the corresponding *strategic decisions* affect the day-to-day operations for a long period. Instead, *operating decisions* such as how many bikes should be moved from one station to another can only be taken after the particular situation of a given day is known. Taking all these decisions in one step is impossible, due both to lack of data about the future and to the fact that that the corresponding optimization problem would be so huge as to being intractable. Therefore, it is crucial to devise the right trade-off between system model tractability and accuracy.

In practice, optimization of a complex system is usually decomposed into independent sub-problems, where independence comes either from considering different time horizons (strategic vs. operational decisions), or from purposely ignoring present but sufficiently "weak" dependencies, say by approximating the effect of the choices in one subproblem on the others' feasibility and cost. Besides making the problems easier to solve, this usually leads to the discovery that each of the sub-problems has strong similarities

with other known problems, coming from similar or even completely different settings. This is why the OR literature often focuses on classes of "abstract" models (say, Vehicle Routing models, Location models, ...) that represent common mathematical structures that appear in many different real-world contexts, possibly as sub-problems or with variants. Once one's problem has been identified, it is therefore usually possible to find similar cases in the literature, which can greatly help in devising a good (MILP) model.

Our case study gives rise to several optimization sub-problems such as: bike station location and fleet dimensioning, allocation of bikes to stations, re-balancing incentives setting, and bike repositioning. We will concentrate on the latter, that represents one of the main daily operating costs, with a consistent impact on the budget. In fact, even if in the early morning all the bike stations meet the desired level of occupation, due to the users' travel behaviour in the evening some stations are typically full and others are empty. Repositioning is crucial in order to offer a good service all day long, and is usually done by means of capacitated vehicles, based at a central depot, that pick up some bicycles from the stations where the level of occupation is too high and deliver them to those stations where the level is too low. The depot also keeps a buffer of bicycles to allow a more flexible redistribution. Driving the vehicles around the urban area is expensive, thus one needs to decide how to route the vehicles to perform the redistribution at minimum cost. This optimization problem is known as the *Bike sharing Re-balancing Problem* (BRP), and has recently received considerable attention from both the OR community and practitioners in the area. We will only consider the *static version* in which the occupancy of each station is considered fixed, as this corresponds to the heavier re-balancing operations performed at night when the system is closed or demand is very low. A *dynamic version* exists where real-time usage information is taken into account to update the redistribution plan even as the vehicles are performing it, but this requires more complex modelling techniques to represent uncertainty of future events that are out of the scope of this simple treatment.

3.1 BRP: system model

The first step to develop a mathematical model is to define the so-called *system model*, that describes the involved entities, their relations and the corresponding parameters. A system model is usually defined already as much as possible in terms of mathematical objects, and significant decisions are made as to which entities and relations are significant, which ones are ignored, and which relations are simplified.

In the BRP case, the system consists of the depot, the vehicles, the stations, and the urban road network that connects them. This can be described using a complete directed graph $G = (N, A)$, where the set of nodes $N = \{0, 1, \ldots, n\}$ contains the depot, node 0, and the stations, nodes $N' = \{1, 2, ..., n\}$. An arc $(i, j) \in A$ represents the shortest path in the actual city road network connecting the locations corresponding to the two nodes i and j, with the associated traveling cost c_{ij}. Since the BS system is located in an urban area, where one-way streets often strongly affect the feasible paths, arcs (i, j) and (j, i) may have different costs. Each station node i has a request q_i, which can be either positive or negative: if $q_i > 0$ then i is a pickup node, where q_i bikes must be removed, while if $q_i < 0$ then i is a delivery node, where $-q_i$ bikes must be supplied. The requests are computed as the difference between the number of bikes present at station i when performing the redistribution and the desired number of bikes in the station in the final configuration. This is a typical example of a link between two decision phases: the desired number of bikes is established during the planning phase, where re-distribution costs are estimated without taking into account detailed routing choices (e.g., by some average), and then taken as a constraint in the operational phase. A fleet of m identical vehicles, each with capacity Q (bikes), is located at the depot. The bikes removed from pickup nodes can either go to a delivery node or back to the depot, and similarly for those supplied to delivery nodes.

The problem is to decide how to route at most m vehicles through the graph, with the *objective* of minimizing the total traveling cost while satisfying the following *constraints*:

1. each vehicle performs a route that starts and ends at the depot;
2. each station is visited exactly once, which implies that its request is completely fulfilled by the one vehicle visiting it;

3. each vehicle starts from the depot with some initial load, comprised between 0 (empty) and Q (full), and the vehicle load at each step of the route, corresponding to the the sum of requests of the visited stations plus the initial load, is never negative or greater than Q.

In setting the system model, some important decisions have already been taken. For instance, it might be possible to serve a station with more than one vehicle, each satisfying a fraction of the demand. This would enlarge the space of feasible BRP solutions, allowing more flexibility in the redistribution plan at the cost of complicating the operational procedures; we assume that the operator has decided against it, but a somehow different model could be developed to check the economical impact of the choice and perform what-if analysis. Also, we allow flow of bikes on the depot to be either positive or a negative, which is useful to model cases in which some bikes have to be brought back to the depot for maintenance, and then put in operations again. Finally, it is assumed that there is enough time available to perform all the operations (although some routes can be "long") and that stations can be serviced at any time; this allows to completely ignore the exact moment at which each operation is performed, that may instead be a relevant information in other variants of the problem (e.g., the dynamic one).

3.2 BRP: MILP formulation

The second step is developing a MILP formulation. As it often happens, the model is closely related to a widely studied class of (difficult) combinatorial problems known in the literature as *Capacitated Vehicle Routing Problem* (CVRP) [2, 3]. In particular, BRP is a *Pickup and Delivery Vehicle Routing Problem* (PDVRP). Thus, to define a MILP formulation one can use as a guide those that have already been proposed in the literature.

A MILP model of BRP uses three types of *decision variables*:

- *binary arc* variables $x_{ij} \in \{0, 1\}$ taking value 1 if arc (i, j) is used by a vehicle (irrespectively to which one), and 0 otherwise;
- non-negative *continuous arc flow* variables f_{ij} representing the current load of the vehicle traveling along arc (i, j), if any;
- *integer* arc variables y_{ij}, representing the position of arc (i, j), if used, in the corresponding route.

The MILP model is the following:

$$\min \sum_{(ij)\in A} c_{ij} x_{ij} \tag{1}$$

$$\sum_{(ij)\in A} x_{ij} = 1 \qquad i \in N' \tag{2}$$

$$\sum_{(ji)\in A} x_{ij} = 1 \qquad i \in N' \tag{3}$$

$$\sum_{(0j)\in A} x_{0j} \leq m \tag{4}$$

$$\sum_{(0j)\in A} x_{0j} - \sum_{(j0)\in A} x_{j0} = 0 \tag{5}$$

$$\sum_{(ij)\in A} f_{ij} - \sum_{(ji)\in A} f_{ji} = q_i \qquad i \in N' \tag{6}$$

$$\underline{f}_{ij} x_{ij} \leq f_{ij} \leq \bar{f}_{ij} x_{ij} \qquad (i, j) \in A \tag{7}$$

$$\sum_{(ji)\in A} y_{ji} - \sum_{(ij)\in A} y_{ij} = 1 \qquad i \in N' \tag{8}$$

$$x_{ij} \leq y_{ij} \leq (n+1) x_{ij} \qquad (i, j) \in A \tag{9}$$

$$x_{ij} \in \{0, 1\} \ , \ \ y_{ij} \in \mathbb{N} \qquad (i, j) \in A \tag{10}$$

The objective function (1) minimizes the traveling cost. Constraints (2)–(3) ensure that each station node has exactly one incoming and one outgoing arc, while (4) and (5) ensure, respectively, that no more than m routes (= vehicles) leave the depot, and that each leaving vehicle eventually returns. Next, *flow conservation* constraints (6) ensure that f_{ij} properly account for the number of bikes on the (single)

vehicle traversing arc (i, j). Then, (7) serve a double purpose. On one hand, when $x_{ij} = 1$, i.e., a vehicle is traversing arc (i, j), they guarantee that the load on a vehicle is feasible by imposing appropriate upper and lower bounds on it (for instance $\underline{f}_{ij} = \max\{\, 0\,,\, q_i\,,\, -q_j\,\}$ and $\bar{f}_{ij} = \min\{\, Q\,,\, Q + q_i\,,\, Q - q_j\,\}$, whose validity can be easily verified with some reflection). On the other hand, when $x_{ij} = 0$ they guarantee that $f_{ij} = 0$; hence, bikes can only "flow" on the graph following vehicles, as it is logically required. One may believe that these constraints (plus (10) dictating the binary nature of x_{ij}) be enough to correctly model the problem, but this is not so because they do not forbid *sub-tours*, i.e., closed oriented loops that *do not include the depot*. To picture this, consider two nodes i and j having $q_i = -q_j > 0$: it would be therefore possible to set $x_{ij} = x_{ji} = 1$, $f_{ij} = q_i$ and $f_{ji} = 0$, i.e., have a vehicle "appearing" at i, carrying all required bikes to j, getting back to i and "disappearing" there. Constraints (8)–(9), which are a version of the so-called *Miller-Tucker-Zemlin (MTZ) subtour elimination constraints* originally devised for the *Traveling Salesman Problem* (TSP) [1], avoid sub-tours (of any length). The idea is to associate an ordering to the vertices of each route by assigning an integer "position" variable y_{ij} to each arc in the route. Starting from the depot, the value of the position variables increases by one as we move to the next arc along the route. Clearly, since a route is cyclic, we can impose an ordering for all the vertices in the route except from the depot: indeed, (8) is not imposed for $i = 0$. Note that the maximum number of arcs in a route is $n + 1$—a single route that visits all the vertices—yielding the upper bound on the position variables.

Admittedly, devising such a formulation is not a trivial task. A number of *formulation tricks* have been used, such as extensive use of *flow conservation constraints* ((6), (8) and in fact even (2)–(5)) and constraints imposing "logical" conditions, such as "$x_{ij} = 0 \implies y_{ij} = 0$, $x_{ij} = 1 \implies y_{ij} \in [\,1\,,\, n + 1\,]$" ((9), and similarly for (7)). Getting used to all the devious tricks necessary to devise a MILP formulation requires some experience; however, as already recalled, most problems one encounters have very likely already been modeled, possibly with some variations, and therefore help is usually available. For instance, the MTZ constraints for the PDVRP are more complex than those for the CVRP where all goods originate from the depot; actually, in that case, the flow constraints (6) are sufficient to avoid sub-tours, as one can easily see considering our counter-example above. Yet, the more complex version had already been devised far before that BRP was ever conceived.

The advantage of formulating one's problem as a MILP is that, once this is done, efficient (as much as possible) software tools are available for (attempting to) solving it.

3.3 Software tools

Due to the huge number of practical applications leading to MILP models, there is no shortage of software tools for solving these problems. Actually, even forming the coefficient matrix A and the right-hand-side vector b corresponding to a MILP is itself a nontrivial and error-prone task, especially considering that practical applications may originate even considerably more complex MILP models than (1)–(10). This is why software tools known as *algebraic modelling languages* are available to perform it, allowing to describe one's model in a way that is very similar to how equations are written in a document; these equations are then processed to automatically construct the data of the problem (A, b, c, I). Several of these tools exist, both commercial ones such as `AMPL` [9] and `GAMS` [10], or open-source ones like `Coliop` [11] and `ZIMPL` [12]. Also, *modelling systems* are available that implement similar functionalities within many different general-purpose programming languages, such as `C++` (`FlopC++` [13], `COIN-Reharse` [26]), `python` (`PuLP` [14], `Pyomo` [15]), `Julia` (`JuMP` [16]), `Matlab` (`YALMIP` [17]) and others. As an alternative, it is possible to construct the model by either writing a file with proper format (e.g., MPS [18] or LP [19]), or directly calling the in-memory API of the desired solver.

However the modelling phase is performed, the problem can then be solved using any of the several available solvers, again both commercial ones like `Cplex` [20], `GuRoBi` [21], `MOSEK` [22] and others, or open-source ones like `Cbc` [23] or `SCIP` [24]. The effectiveness of the solver should be expected to vary considerably; although all more or less based on the same ideas, quickly summarised in the next Section, implementation details may make an enormous difference on specific instances. Indeed, MILPs are "difficult", and therefore one should in principle expect the problems to take a long time to solve; a large set of sophisticated algorithmic techniques has been developed to try to improve on this, whose

implementation differs between different solvers. In general one should expect commercial solvers to be more efficient than open-source ones (which justify the high licensing fees they usually require), but exceptions are not unheard-of. Indeed, all solvers have a large set of *algorithmic parameters* controlling their behaviour, whose appropriate tuning can make a very substantial difference.

Above all, however, *choosing the right formulation* is usually the most important factor dictating how efficiently the problem will ultimately be solved. To be able to at least introduce all this aspects, a very quick recap of the algorithmic techniques that are employed is necessary.

4 Algorithmic approaches and "good formulations"

There are very many different algorithms for "hard" problems. In many practical applications, recognising the fact that efficiently finding *provably* optimal solutions is difficult, it is usual to resort to *heuristics*, i.e., algorithms that strive to efficiently provide "good" solutions, but give no guarantees on their quality. Most often heuristics are developed for a much more specific class of problems than MILP (say, PDVRP), although general frameworks for specific classes of heuristic approaches have been developed, such as *local search*, *tabu search*, *simulated annealing*, *genetic algorithms* and many others, and general-purpose solvers exist (e.g. `LocalSolver` [25]). In some cases, *approximation algorithms* are available that provide explicit *a-priori* bounds on the quality of the solution obtained within a given computational effort, but again these strongly depend on the specific problem, and are not available for general MILP. In this section we will rater concentrate on *exact* algorithms, that guarantee to find an optimal solution, albeit possibly at a computational cost that may grow exponentially fast with the size of the problem. In particular we will (briefly) discuss the *Branch-and-Bound* (B&B) approach underlying all the general-purpose solvers alluded to above, with its crucial variants known as *Branch-and-Cut* and *Branch-and-Price*.

These approaches are inherently based on the concept of *relaxation* to provide *bounds* on the optimal value of the problem. In general, a relaxation of an optimization problem (P) (assuming minimization) is another optimization problem (P$'$) such that: (i) its feasible set is larger than that of (P), and (ii) its objective function has the same or smaller value than that of (P) on the latter's feasible solutions. By denoting as $\nu(\cdot)$ the optimal value of an optimization problem, one clearly has $\nu(\text{P}') \leq \nu(\text{P})$. The question then arises of how to construct interesting relaxations. However, as anticipated, the answer is trivial for (MILP), as one can simply use its *continuous relaxation*

$$\text{(LP)} \quad \min\{\, cx \,:\, Ax \leq b \,\} \ ,$$

i.e., the LP obtained by simply dropping the integrality constraints. It is immediate to realise that $\nu(\text{LP}) \leq \nu(\text{MILP})$, which may provide *optimality conditions* for a feasible solution $\bar{x}$ of (MILP). Indeed, consider the (very fortunate) case in which the optimal solution $\bar{x}$ of (LP) satisfies the integrality constraints: it is immediate to realise that, in this case, $c\bar{x} = \nu(\text{LP}) \leq \nu(\text{MILP}) \leq c\bar{x}$. Thus, exploiting the efficient available LP algorithms we could, in principle, be able to not only find an optimal solution of the "difficult" (MILP), but also have a *certificate of optimality* proving it without any doubt. This actually hinges on being able to prove that $\bar{x}$ is optimal for (LP) in the first place, which typically relies on *duality* and ultimately *convexity* of the problem, but we are not going to delve deeper in these concepts.

Unfortunately, in general $\nu(\text{LP}) < \nu(\text{MILP})$. Thus, besides *bounding* by relaxations some other mechanism is required to ensure that the lower bound is increased and eventually reaches $\nu(\text{MILP})$. Unfortunately, the only (practical) general ways that have been devised for ensuring this are *branching*, a.k.a. (partial) *enumeration*, and *cutting*. Both are, for the best variants known so far, processes that may require an exponential number of iterations to achieve the desired results. Yet, they are also the best (least worse) general-purpose available algorithms for MILP, hence what is used in practice. We will now give a brief recount of their basic principles.

4.1 Branch-and-Bound

Branch-and-Bound (B&B) uses a "divide and conquer" approach to explore the set X of feasible solutions of (MILP), as described in Algorithm 1. The idea is to partition X into a finite collection of subsets

$X_1, \ldots, X_k$ and solve separately each one of the subproblems restricted to each X_i; one then compares the optimal solutions to the subproblems, and chooses the best one. However, since the subproblems are usually almost as difficult as the original problem, it is typically necessary to solve each of them by recursively iterating the same approach, that is, splitting them into further subproblems. This is the *branching* part of the method, which leads to a *tree of subproblems* T. Because X is typically exponentially large, the B&B uses *lower bounds* on the optimal value of each subproblem to try to avoid exploring parts of it. These are obtained by solving the continuous relaxation. Actually, the corresponding (LP) may have no solution, which immediately implies that X_i has no integer solution as well; any LP solver can efficiently detect it, customarily returning ∞ as the optimal value and therefore allowing to *fathom* the corresponding node of T *by infeasibility*. Otherwise, a continuous optimal solution $\bar{x}_i$ is found that may occasionally be integer, in which case we can possibly have found a better *upper bound* on ν(MILP) than previously known, updating the *incumbent value* $\bar{z}$ (the cost of the best solution found thus far). The essence of the algorithm lies in the following observation: if the optimal value $\underline{z}_i = c\bar{x}_i$ of the continuous relaxation of X_i satisfies $\underline{z}_i \geq \bar{z}$, then this subproblem need not be considered further, since the optimal solution of X_i is no better than the best feasible solution encountered thus far. In this case, the corresponding node of T can be *fathomed by the bound*; it is immediate to realize that this surely happens if $\bar{x}_i$ is integer, since then $c\bar{x}_i \geq \bar{z}$. Otherwise, *branching* has to occur, i.e., the current X_i is further split. Provided that the total number of subproblems that can be generated is finite, the B&B algorithm clearly terminates in a finite (albeit, very possibly, exponentially large) number of iterations having correctly identified the optimal value $\bar{z} = \nu$(MILP). The *incumbent solution* having produced the incumbent value $\bar{z}$ is usually conserved as well, and at termination it is guaranteed to be an optimal solution. Finiteness of the branching operation is usually trivial. For instance, if all variables are binary, the typical branching consists in selecting one variable in $\bar{x}_i$ that has a fractional value and creating two subproblems, in one of which the variable is fixed to 0, while in the other it is fixed to 1; a slightly more general version is easily devised for integer variables. We remark in passing that branching for nonconvex MINLPs is a considerably more sophisticated process, justifying the higher practical cost of the latter w.r.t. MILPs.

Algorithm 1: Branch-and-Bound

Input: (MILP), `subproblem-selection-rule`, `LP-relaxation`, `branching rule`
Output: *Optimal value* $\bar{z}$

```
1  Initialize T = { X }, z̄ = ∞
2  while T ≠ ∅ do
3  |  X_i ← subproblem-selection-rule(T)   // select an active subproblem
4  |  ( x̄_i , z_i ) ← LP-relaxation(X_i)          // solve continuous relaxation
5  |  if z_i = ∞ then
6  |  |  delete X_i                          // subproblem X_i infeasible
7  |  else
8  |  |  if x̄_i is feasible for (MILP) then
9  |  |  |  z̄ = min{ z̄ , cx̄_i }               // new feasible solution found
10 |  |  if z_i ≥ z̄ then
11 |  |  |  delete X_i                        // subproblem X_i fathomed by bound
12 |  |  else
13 |  |  |  T ← branching rule(X_i)
14 |  |  |  // break X_i into further subproblems and add them to T
15 |  |  end
16 |  end
17 end
```

Many aspects of the practical implementation of a B&B are potentially crucial, such as the exact choice of the branching operation, the selection of the next active subproblem, and the details of the solution of the continuous relaxation. Also, MILP solvers usually employ *general-purpose heuristics* which try

to generate feasible integer solutions, say by "cleverly" rounding the optimal continuous solutions $\bar{x}_i$. However, by far the most important factor dictating the effectiveness of a B&B is how often fathoming of a node (line 11) happens. This is clearly related to *tightness* of the lower bound, i.e., how close the optimal value $\underline{z}_i$ of the continuous relaxation is to the actual optimal value of X_i (although, of course, the quality of the upper bound $\bar{z}$ also is a factor). As a rule of thumb, MILPs for which $\underline{z}_i$ is within a very few percentage points off the real value can be solved efficiently via the B&B, whereas if the *relative gap* is, say, above 10% then a huge number of nodes will be enumerated.

Crucially, this does not depend on the B&B solver, but rather on the *quality of the formulation* that the user has provided it. It is therefore important to explicitly define what the "quality" of a formulation is.

4.2 Strong, "large" formulations

For LP, a "good" formulation is one that has a *small number* of variables and constraints, because the computational cost of its solution depends (polynomially) on these. The situation for MILP is drastically different.

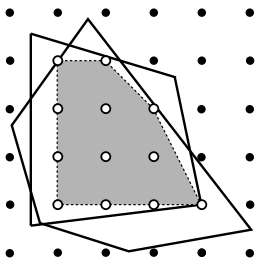

Figure 1: 2D illustration

The relevant mathematical concepts are pictorially illustrated in Figure 1, depicting a MILP in two integer variables. Three different formulations are represented, as *polyhedra*. The three formulations are *equivalent* in a MILP sense, in that they define the same feasible region (white points, intersection between the $\mathbb{Z}^2$ lattice and each polyhedron). However, the polyhedra have different "size"; intuitively, the larger the polyhedra, the smallest the value of the corresponding continuous relaxation, and therefore the worse the gap, although of course the "direction" of the objective function also has an impact. Among the three, "the best" polyhedron (shaded area with dotted edges) is depicted. This is the smallest polyhedron representing a correct formulation, a.k.a. the *convex hull* of all feasible integer solutions. Without entering into details of the definition, the convex hull providers the "perfect" formulation: all its *vertices* are integer-valued (a.k.a. *integrality property*), and it can be seen that this implies that solving the continuous relaxation solves the problem to provable optimality. Unfortunately, most MILP formulations do *not* have the integrality property. However, it is now clear that the *quality* of a MILP formulation can be generally judged by how close it is to defining the convex hull of the integer solutions.

Unfortunately, in most cases such formulations are necessarily "very large". For all "difficult" problems, the "perfect" formulation includes an *exponential* number of different constraints (say, $m \in O(2^n)$). In fact, to define "strong" formulations it is usually necessary to use families of constraints that are exponential in size.

Our BRP problem provides a fitting example of this phenomenon: to impose connectivity of the solution, an alternative to (8)–(9) are the *Subtour Elimination Constraints* (SEC)

$$\sum_{i \in S,\, j \in S} x_{ij} \leq |S| - 1 \qquad S \subset N' \;\; , \;\; 2 \leq |S| \leq n - 2 \;\; . \tag{11}$$

Clearly, a sub-tour touching a subset $S \subset N'$ of stations must contain at least $|S|$ arcs, as our example illustrated: (11) forbids this, so that all sub-tours must necessarily include the depot. If (11) are added to the formulation, the y_{ij} variables and the corresponding constraints (8)–(9) are no longer needed; this saves $|A|$ variables, but at the cost of *exponentially many* constraints. Yet, the formulation using (11) is well-known to be usually much *stronger* than that using MTZ, and therefore in principle preferable. The issue, of course, is that even for graphs with relatively few nodes writing down an exponentially large set of constraints would not be possible, and anyway it would lead to a very inefficient solution algorithm due to the enormous cost of solving the corresponding LP. Yet, formulations of this kind can be effectively used.

The idea is to only add constraints that are "needed". More precisely, one can initialize a *Restricted Master Problem* (RMP) containing only a small subset of the original constraints (say, only those corresponding to all possible subsets S of size two). The corresponding LP is solved, and the solution $\bar{x}_i$ is

checked: if it satisfies *all* the original constraints one stops, having clearly solved the continuous relaxation as if all the constraints had been there, otherwise one or more violated constraints are added to the RMP and the process is iterated. In practice this scheme is usually quite effective, since although several LPs are solved instead of only one, their size is tiny if compared to that that the complete one would have. However, for this scheme to make sense, a "smart" way is needed to verify whether or not $\bar{x}_i$ satisfies all the constraints (11), and if not find at least a violated one. Indeed, if this was done by enumerating them all, then it would have an exponential cost, which would rapidly become unbearable as the size of the graph grows. Fortunately, again, this can be done. Without going into details, the idea is to recast the question under the form of an *optimization problem*, typically that of finding the *least violated constraint*. Despite having an exponential number of solutions, the problem may be "easy": for instance, for (11) it reduces to a *maximum flow/minimum cut* problem on a properly defined digraph (depending on $\bar{x}_i$), which admits efficient solution algorithms. In other cases, these *separation problems* may be "hard" in principle, but still be solvable efficiently enough for the required size to make the approach viable. The advantage of having a much stronger formulation may counterbalance the cost of solving many LPs and many (possibly, "hard") separation problems, making these "large but strong" formulations powerful tools for the solution of problems such as the BRP.

Iteratively adding violated constraints is called the *cutting-plane method*, since we add constraints (i.e., hyperplanes), a.k.a. *cuts* or *valid inequalities*, that cut away the current (fractional) solution $\bar{x}_i$. Interestingly, it is in theory possible to *entirely solve any* MILP by only relying on a cutting-plane approach, without any branching ever occurring. This can be done e.g. by relying on the *Chvàtal-Gomory* (CG) procedure, that is a systematic way of deriving valid inequalities for the convex hull of the integer solutions. Indeed, a "classic" result in polyhedral analysis shows that every valid inequality for the convex hull can be obtained by applying the CG procedure a finite number of times. In other words, a cutting-plane approach based on CG can be shown to be an exact method to solve MILP. Despite the theoretical interest, in practice pure cutting-plane algorithms are not successful on their own, because: (*i*) a huge number of cutting planes is typically needed, (*ii*) cuts tend to get weaker and weaker as the algorithm proceeds (a.k.a. "tailing off"), (*iii*) no feasible integer solution is obtained until the very end. Yet, these ideas are useful from a practical point of view. Unlike SEC, the CG cuts do not need any specific structure, and therefore can be applied to any MILP. Similarly, other classes of "general purpose cuts" have been developed (such as clique, cover, mixed-integer rounding, implied bounds, flow path, disjunctive cuts, ...) that are now available in most MILP solvers, where they can be automatically generated for any MILP. Even though these cutting planes alone cannot usually solve a MILP, they can be very useful to considerably strengthen the given formulation. Indeed, all current most successful solution algorithms for MILP are based on a *combination* of cutting-plane techniques, typically using several different classes of cuts, with the B&B approach. This is known as *Branch-and-Cut* (B&C), which is in essence just a B&B where cutting planes are generated throughout the tree T to improve the LP bounds. A somewhat delicate balance has to be attained at each iteration, since the approach has to decide which of the two basic operations (branching or cutting) to apply at a node that is not fathomed; significant research has gone in finding rules that work efficiently in many cases, and algorithmic parameters can be set to change this behaviour for a given instance. Finally, most current solvers allow the user to add "specific" cuts for her own MILP, say SEC for BRP, which can also considerably improve the performances of the B&C.

Adding (in principle, exponentially many) cuts is not the only way to construct "tight" formulations. Another (in some sense "dual", but we cannot delve in the precise mathematical description of this concept) way is to work in a completely different space of variables, possibly having an exponential size in n. We now briefly illustrate the idea for our BRP case study.

In BRP, a solution consists of a subset of all the *feasible vehicle routes*; a feasible route $r \in \mathcal{R}$ is a directed (simple) cycle that starts from the depot, visits a subset of the nodes exactly once and returns to the depot, such that the vehicle load never exceeds the capacity Q and never becomes negative. With this definition, BRP can be recast as the problem of selecting a *feasible* subset $X \subset \mathcal{R}$ with minimum cost. Note that there are two "levels" of feasibility: that of the *routes*, and that of the *subsets of routes*. In other words, the crucial step is distinguishing the constraints that define the feasibility of the individual routes, from those that define the feasibility of the subset of routes as a whole; the former will be encapsulated in the concept of route, and therefore will not appear explicitly in the formulation. A subset $X \subset \mathcal{R}$ is

feasible if each station node is visited by exactly one route, and the number of routes does not exceed the number of available vehicles m. Hence, we can define a (*set partitioning*) formulation of BRP using $|\mathcal{R}|$ binary variables (columns) $x_r \in \{0, 1\}$, one for each $r \in \mathcal{R}$:

$$\min \sum_{r \in \mathcal{R}} c_r x_r \tag{12}$$

$$\sum_{r \in \mathcal{R} : i \in r} x_r = 1 \qquad i \in N' \tag{13}$$

$$\sum_{r \in \mathcal{R}} x_r \leq m \tag{14}$$

$$x_r \in \{0, 1\} \qquad r \in \mathcal{R} \tag{15}$$

In the objective function (12), c_r is the cost of a feasible route $r \in \mathcal{R}$, i.e., just the sum of the costs of the arcs in route r. The *set partitioning* constraints (13) guarantee that each station is served by exactly one route; they can be written in terms of the subset $\mathcal{R}(i) \subset \mathcal{R}$ of the feasible routes that visit node i. Finally, (14) are the *cardinality constraints* on the fleet of vehicles. We remark, again, that the constraints describing a feasible route do not appear in the formulation, but are "implicit" in the definition of $\mathcal{R}$; this makes (12)–(15) remarkably simpler than (1)–(10), but at the cost of using a number of variables that is typically *exponential* in the size of the graph. Simplicity is, however, not the most relevant advantage of (12)–(15): it can be seen that the formulation is usually quite "tight", i.e., it produces (much) better lower bounds than (1)–(10). Thus, (12)–(15) would be in principle a better starting point to apply a B&B approach, it it were possible to efficiently solve its continuous relaxation, which is an LP with an exponential number of variables (columns).

Fortunately, this is, again, possible. Indeed, we have presented the approach already for LPs with an exponential number of constraints (rows), and it turns out that the exactly the same idea works here just by looking at the *dual* of the LP. Again we cannot go into the mathematical details; suffices here to say that, exactly as we can define *separation sub-problems* for generating new rows (constraints), we can define *pricing sub-problem* for generating new variables (columns). Starting with a RMP with a "small" number of initial columns, the pricing problem identifies columns with *negative reduced cost* that, if added to the RMP, may decrease its optimal value; if there is none, the current optimal RMP solution is also optimal for the LP with all the columns. In the BRP application, columns correspond to routes, and the pricing problem is a *Constrained Shortest Path* problem, i.e., the problem of finding the path (route) on the graph with minimal *reduced cost* (a modified cost taking into account the current *dual* optimal solution of the RMP). This is a "hard" problem in general, but typically solvable for the size of realistic instances; besides, different variants can be used (e.g., allowing or not the path to pass multiple times through a single node) that offer a trade-off between the complexity of the pricing problem and the quality of the corresponding LP bound.

Solving LPs with a very large (say, exponential) number of variables is known as *Column Generation*; a B&B where such LPs are solved at each node is also called a *Branch-and-Price* (B&P). Often, formulations amenable to CG are quite similar to in shape to (12)–(15), and are referred to as *set partitioning* formulations. Over the last twenty years, set partitioning formulations have become very popular for many combinatorial optimization problems, and the reason for this success is twofold. First, for some problems (e.g., crew pairing), there are basically no alternative formulations. Second, in many cases, these formulations provide quite strong lower bounds and therefore can be the basis of efficient solution procedures. Unfortunately, while B&C, comprised the possibility of adding problem-specific cuts, is implemented in basically all modern MILP solvers, B&P is much less supported; among the main MILP solvers, only `SCIP` [24] does B&P natively. However, B&P and B&C are not alternative; there is nothing conceptually preventing using valid inequalities in a formulation with a large number of variables. The all-singing, all-dancing algorithmic framework is therefore the so called Branch-and-Price-and-Cut (B&P&C), which can be considered the ultimate way of developing "strong" formulation. Such an approach cannot typically be implemented by inexperienced users, whereas B&C using only "general purpose" cuts only requires familiarity with an algebraic modelling language and use of a general-purpose solver as a "black box"; hence, development of "large", sophisticated formulations is typically not the first step, and it is only justified if the run-of-the-mill approach is not effective. Yet, it is important to realise that relatively simple tools are available that allow to construct a "simple" MILP formulation of one's application and quickly test it, and that these same tools support increasingly sophisticated approaches that can be used

when strictly required, possibly with the help of OR experts.

5 Conclusions and further reading

Optimization problems arise in basically all facets of human activity, and in particular whenever the system to be managed is complex. This is almost by necessity the case in sharing economy applications, as only when the set of users is large, allowing to averaging out individuals' needs, effectively sharing resources is viable; in other words, sharing economy systems are necessarily complex. Unfortunately, most optimization problems are "difficult": the cost of their solution grows exponentially fast with the size of the problem (system). However, conceptual and software tools are nowadays available that may make it possible to solve optimization problems of the size required by applications in reasonably short times. The usual steps for using them are developing an appropriate system model, and then formulating the problem as a Mixed-Integer Linear one (although formulations using convex, maybe quadratic or conic, may also be viable). If a "strong" formulation is used, chances are that the problem is solved efficiently enough. If this does not happen, possible resorts are implementing ad-hoc heuristics, or seek assistance by OR experts to develop "larger" but "stronger" formulations. However, the first attempt can be done by relatively inexperienced users, and still have some chances of yielding satisfactory results.

Of course, optimization is a vast subject, and this primer has barely scratched the surface of the huge amount of theoretical and practical tools that have been developed for the solution of optimization problems, usually combining techniques from different areas of mathematics (linear algebra, geometry, discrete mathematics, graph theory, ...), and computer science. Several textbooks are available that present modern optimization methods in details, such as the classical [4] for B&C and [6] for CG/B&P approaches. Two other valuable recent reference works, among the many others, are [7] and [8]. While MILP is usually the go-to class for most applications, MINLPs can also be (carefully) considered; a recent useful reference is [5].

Acknowledgements

The authors acknowledge the financial support of the Italian Ministry for Education, Research and University (MIUR) under the project PRIN 2015B5F27W "Nonlinear and Combinatorial Aspects of Complex Networks".

References

[1] C. Miller, A. Tucker, R. Zemlin, *Integer programming formulations and traveling salesman problems*, J. Association Comput. Machinery, 7, 32–329, 1960.

[2] P. Toth, D. Vigo (Eds.), *The Vehicle Routing Problem*. SIAM, 2002.

[3] P. Toth, D. Vigo (Eds.), *Vehicle Routing: Problems, Methods, and Applications*. SIAM, 2014

[4] L. A. Wolsey, G. L. Nemhauser. *Integer and Combinatorial Optimization*. Wiley, 1999.

[5] J. Lee, S. Leyffer. (Eds.) *Mixed Integer Nonlinear Programming*. The IMA Volumes in Mathematics and its Applications. Springer, 2012.

[6] G. Desaulniers, J. Desrosiers, and M.M. Solomon, eds. *Column Generation*. Springer, 2005.

[7] M. Conforti, G. Cornuejols, G. Zambelli, *Integer Programming*. Springer, 2014.

[8] B. Korte, J. Vygen, *Combinatorial Optimization - Theory and Algorithms*. Springer, 2018.

[9] `AMPL. https://ampl.com/`

[10] GAMS. https://www.gams.com/

[11] Coliop. http://www.coliop.org/

[12] ZIMPL. http://zimpl.zib.de/

[13] FlopC++. https://projects.coin-or.org/FlopC++

[14] PuLP. https://pythonhosted.org/PuLP/

[15] Pyomo. http://www.pyomo.org/

[16] JuMP. https://github.com/JuliaOpt/JuMP.jl/

[17] YALMIP. https://yalmip.github.io/

[18] MPS. http://lpsolve.sourceforge.net/5.0/mps-format.htm

[19] LP. http://lpsolve.sourceforge.net/5.1/lp-format.htm

[20] Cplex. https://www.ibm.com/analytics/cplex-optimizer

[21] GuRoBi. http://www.gurobi.com/

[22] MOSEK. https://www.mosek.com/

[23] Cbc. https://projects.coin-or.org/Cbc

[24] SCIP. http://scip.zib.de/

[25] LocalSolver. http://www.localsolver.com/

[26] COIN-Reharse. https://github.com/coin-or/Rehearse