LEO LIBERTI

MATHEMATICAL PROGRAMMING

ECOLE POLYTECHNIQUE

PUBLISHED BY ECOLE POLYTECHNIQUE

TYPESET USING LaTeX ON TEMPLATE https://tufte-latex.github.io/tufte-latex/

Zeroth printing, February 2018

Contents

Introduction

These lecture notes are a companion to a revamped INF580 course at DIX, Ecole Polytechnique, academic year 2015/2016. The course title is actually "Constraint Programming" (CP), but I intend to shift the focus to *Mathematical Programming* (MP), by which I mean a summary of techniques for programming computers by supplying a mathematical description of the solution properties.

This course is about *stating what your solution is like*, and then letting some general method tackle the problem. MP is a paradigm shift from solving towards modelling. This course is different from traditional MP courses. I do not explain the simplex method, or Branch-and-Bound, or Gomory cuts. Heck, I don't even explain what the *linear programming dual* is! (There are other courses at Ecole Polytechnique for that.) I want to convey an overview about *using MP as a tool*: the many possible types of MP formulations, how to change formulations to fit a solver requirement or improve its performance, how to use MP within more complex algorithms, how to scale MP techniques to large sizes, with some comments about how MP can help you when your client tells you he has a problem and does not know what the problem is, let alone how to solve it!

A word about notation: letter symbols (such as x) usually denote *tensors*[1] of quantities. So x_i could be an element of x, and x_{ij} an element of x_i. And so on. Usually the correct dimensions are either specified, or can be understood from the context. If in doubt, ask (write to `liberti@lix.polytechnique.fr`) — I will clarify and perhaps amend the notes.

[1] I.e. multi-dimensional arrays.

Part I

An overview of Mathematical Programming

1
Imperative and declarative programming

People usually program a computer by writing a "recipe" which looks like a set of prescriptions: do this, do that if condition is true, repeat.

But there is another way of programming a computer, which consists in putting together complex functions using simpler ones, such that the end function exactly describes the output of the problem we are trying to solve.

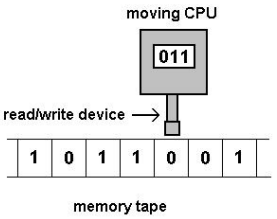


Figure 1.1: A Turing Machine (from 2009.igem.org/Turing_machines).

1.1 *Turing machines*

You might be familiar with the idea of a Turing Machine (TM): there is an infinitely long tape divided in cells, each of which can bear a symbol from some alphabet[1] A.

A mechanical head, capable of writing symbols on cells or erasing them, can move left and right along the tape, or simply rest, according to some instructions labeled by *states*[2].

For example, a state q from a finite state set Q corresponds to the following instruction:

> q: read symbol s from current cell, depending on s perform an movement $\mu \in M = \{\mathsf{right}, \mathsf{left}, \mathsf{stay}\}$, write a symbol $t \in A$ in the new cell, and switch to a new state r.

We can formally store such instructions in a dictionary[3] $\pi : Q \to A \times M \times A \times Q$ mapping states q to quadruplets (s, μ, t, r).

We assume Q must contain an initial state S_0 and a final state S_{halt}. Then the TM with program π starts in state S_0, and applies the rules in π until $r = S_{\mathsf{halt}}$.

Any TM is given the sequence of symbols written in the cells of the tape prior to starting the computation as input, and obtains as an output the sequence of symbols written in the cells of the tape after the computation ends. Note that the computation need not always end. Consider, e.g., the TM with alphabet $A = \{0, 1\}$ and the program $\pi(1) = (0, \mathsf{stay}, 0, 2)$ and $\pi(2) = (0, \mathsf{stay}, 0, 1)$ with starting tape $(0, 0, \ldots)$, $S_0 = 1$ and $S_{\mathsf{halt}} = 3$: this simply flips states from 1 to 2 and back to 1 forever. If we take the sequence $(1, 1, \ldots)$

[1] An *alphabet* is an arbitrary set of symbols, such as e.g. $\{0, 1\}$ or $\{\square, \boxplus, \boxtimes\}$ or $\{\mathrm{a}, \mathrm{b}, \ldots, \mathrm{z}\}$. We assume it is finite.

[2] The name "state" probably stems from a wish of Turing to equate the machine states with "states of mind" [Turing, 1937].

[3] A *dictionary* is a data structure which stores a map *key* $\mapsto$ *value* in the form of (*key*, *value*) pairs, and is optimized for efficiently retrieving values corresponding to a given key — see your favorite basic data structure course.

as a starting tape, we find no relevant instruction in $\pi(1)$ which starts with 1, and hence the TM simply "gets stuck" and never terminates.

1.2 Register machines

COMPUTATION MODELS which are closer to today's computers are *register machines* (RM) [Minsky, 1967]. RMs consist of a finite set $\mathcal{R}$ of registers $R_1, R_2, \ldots$ which can contain a natural number, a finite set Q of states, and some very basic instructions[4] such as:

(a) given $j \in \mathcal{R}$, let $R_j \leftarrow R_j + 1$, then switch to a new state r

(b) given $j \in \mathcal{R}$, if $R_j > 0$ then $R_j \leftarrow R_j - 1$ then switch to a new state r else switch to a new state s.

Registers are similar to bytes of memory, states are used to enumerate instructions in a program (as for TMs), and the instructions have very basic semantics on the memory similar to what you could find in a very simple CPU.

[4] The instruction may vary, according to different authors; those given below are very simple, and due to Marvin Minsky.

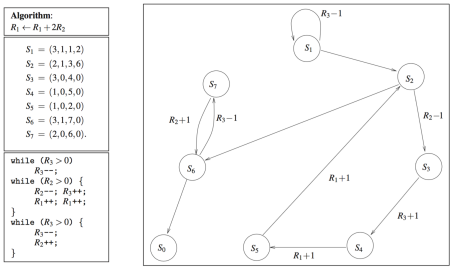


Figure 1.2: A program in Minsky's RM (from [Liberti and Marinelli, 2014]).

1.3 Universality

TMS AND RMS are flexible enough so that they can "simulate" other TMs and RMs. Specifically, we are able to design a TM U such that

$$U(\langle T, x\rangle) = T(x), \tag{1.1}$$

where T is the description of any TM, $\langle T, x\rangle$ is a pair given to U as input, and $T(x)$ denotes the output of T when fed the input x. The TM U is called a *universal TM* (UTM).[5] The same can be said for RMs.

How do we "describe a TM"? Since there are only countably many possible TM (or RM) descriptions, we can count them, and then match them to natural numbers. We can therefore speak of the i-th TM (or RM) where $i \in \mathbb{N}$. In fact we can do better: through *Gödel numbers*[6] we can compute i from the TM (or RM) description, and given an $i \in \mathbb{N}$ computed this way, we can infer the TM (or RM) which originated it.

UTMs can be used to prove several mind-boggling results, such as the existence of problems which cannot be solved by any TM.[7]

[5] In more recent terminology, the program of a UTM is sometimes described as an *interpreter*.

[6] Many Gödel number schemes are possible. Gödel actually defined them for logical propositions, but every sequence $s = (s_1, \ldots, s_k)$ of symbols of A can be encoded as an integer, and any thusly obtained integer can be decoded into the original sequence. Gödel assigned the number $\mathcal{G}(s) = 2^{s_1} 3^{s_2} 5^{s_3} \cdots p_k^{s_k}$ to the sequence s. [Gödel, 1930]

[7] Just google *halting problem*, see Fig. 1.4.

1.4 Partial recursive functions

WITHOUT ANY LOSS of generality, we consider TMs or RMs having natural numbers as inputs and output. We investigate a class of functions $\phi : \mathbb{N}^k \to \mathbb{N}$, called *partial recursive* (p.r.) [Cutland, 1980],

for all values of $k \geq 0$, built recursively using some basic functions and basic operations on functions.

The basic p.r. functions are:

- for all $c \in \mathbb{N}$, the constant functions $\mathsf{const}_c(s_1, \ldots, s_k) = c$;
- the successor function $\mathsf{succ}(s) = s + 1$ where $s \in \mathbb{N}$, defined for $k = 1$ only;
- for all k and $j \leq k$, the projection functions $\mathsf{proj}^k_j(s_1, \ldots, s_k) = s_j$.

The basic operations are:

- composition: if $f : \mathbb{N}^k \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$ are p.r., then $g \circ f : \mathbb{N}^k \to \mathbb{N}$ mapping $s \mapsto g(f(s))$ is p.r.
- primitive recursion:[8] if $f : \mathbb{N}^k \to \mathbb{N}$ and $g : \mathbb{N}^{k+2} \to \mathbb{N}$ are p.r., then the function $\phi_{f,g} : \mathbb{N}^{k+1} \to \mathbb{N}$ defined by:

$$\phi(s, 0) = f(s) \tag{1.2}$$
$$\phi(s, n+1) = g(s, n, \phi(s, n)) \tag{1.3}$$

  is p.r.
- minimalization: if $f : \mathbb{N}^{k+1} \to \mathbb{N}$, then the function $\psi : \mathbb{N}^k \to \mathbb{N}$ maps s to the least[9] natural number y such that, for each $z \leq y$, $f(s, z)$ is defined and $f(s, y) = 0$ is p.r.

Notationally, we denote minimalization by means of the *search quantifier* μ:[10]

$$\psi(s) \equiv \mu y \, (\forall z \leq y \, (f(s, z) \text{ is defined}) \wedge f(s, y) = 0). \tag{1.4}$$

Figure 1.3: Alonzo Church, grey eminence of recursive functions.

[8] For example, the factorial function $n!$ is defined by primitive recursion by setting $k = 0$, $f = \mathsf{const}_1 = 1$, $g(n) = n(n-1)!$.

[9] This is a formalization of an optimization problem in number theory: finding a minimum natural number which satisfies a given p.r. property f is p.r.

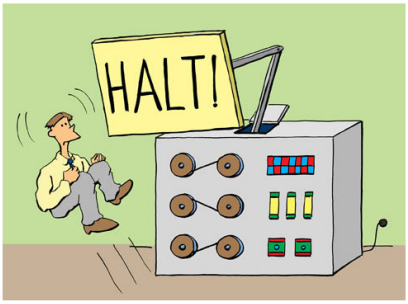
[10] The μ quantifier is typical of computability. Eq. (1.4) is equivalent to $\psi(s) = \min\{y \in \mathbb{N} \mid \forall z \leq y \, (f(s, y) \text{ is defined}) \wedge f(s, y) = 0\}$.

1.5 *The main theorem*

The main result in computability states that TMs (or RMs) and p.r. functions are both universal models of computation. More precisely, let T be the description of a TM taking an input $s \in \mathbb{N}^k$ and producing an output in $\mathbb{N}$, and let $\Phi_T : \mathbb{N}^k \to \mathbb{N}$ the associated function $s \mapsto T(s)$. Then Φ_T is p.r.; and, conversely, for any p.r. function f there is a TM T such that $\Phi_T = f$.

Here is the sketch of a proof. ($\Rightarrow$) Given a p.r. function, we must show there is a TM which computes it. It is possible, and not too difficult,[11] to show that there are TMs which compute basic p.r. functions and basic p.r. operations. Since we can combine TMs in the appropriate ways, for any p.r. function we can construct a TM which computes it.

($\Leftarrow$) Conversely, given a TM T with input/output function $\tau : \mathbb{R}^n \to \mathbb{R}$, we want to show that $\tau(x)$ is a p.r. function.[12] We assume T's alphabet is the set $\mathbb{N}$ of natural numbers, and that the output of T (if the TM terminates) is going to be stored in a given



Alan designed the perfect computer

Figure 1.4: The Halting Problem (vinodwadhawan.blogspot.com).

[11] See Ch. 3, Lemma 1.1, Thm. 3.1, 4.4, 5.2 in [Cutland, 1980]

[12] This is the most interesting direction for the purposes of these lecture notes, since it involves "modelling" a TM by means of p.r. functions.

"output" cell of the TM tape. We define two functions:

$$\omega(x,t) = \begin{cases} \text{value in output cell after } t \text{ steps of } T \text{ on input } x \\ \tau(x) \text{ if } T \text{ stopped before } t \text{ steps on input } x \end{cases}$$

$$\sigma(x,t) = \begin{cases} \text{state label of next instruction after } t \text{ steps of } T \\ 0 \text{ if } T \text{ stops after } t \text{ steps or fewer,} \end{cases}$$

where we assumed that $S_{\text{halt}} = 0$. Now there are two cases: either T terminates on x, or it does not. If it terminates, there is a number t^* of steps after which T terminates, and

$$t^* = \mu t(\sigma(x,t) = 0) \tag{1.5}$$

$$\tau(x) = \omega(x,t^*). \tag{1.6}$$

Note that Eq. (1.5) says that t^* is the least t such that $\sigma(x,t) = 0$ and $\sigma(x,u)$ is defined for each $u \leq t$.

If T does not terminate on x, then the two functions $\mu t(\sigma(x,t) = 0)$ and $\tau(x)$ are undefined on x. In either case, we have

$$\tau(x) \equiv \omega(x, \mu t(\sigma(x,t) = 0)).$$

By minimalization, the fact that $\tau(x)$ is p.r. follows if we prove that ω and σ are p.r. This is not hard to do, but it is quite long, and is beside the scope of these lectures. Informally, one starts from the Gödel number of T, decodes it into T (this decoding function is p.r.), and essentially executes T over its input until step t. That this last task is p.r. follows by decomposing it into two functions: finding the new state and tape content after an instruction has been executed, and finding the next instruction. These two functions are in fact primitive recursive,[13] and hence p.r. (see Fig. 1.5), which concludes this proof sketch.[14]

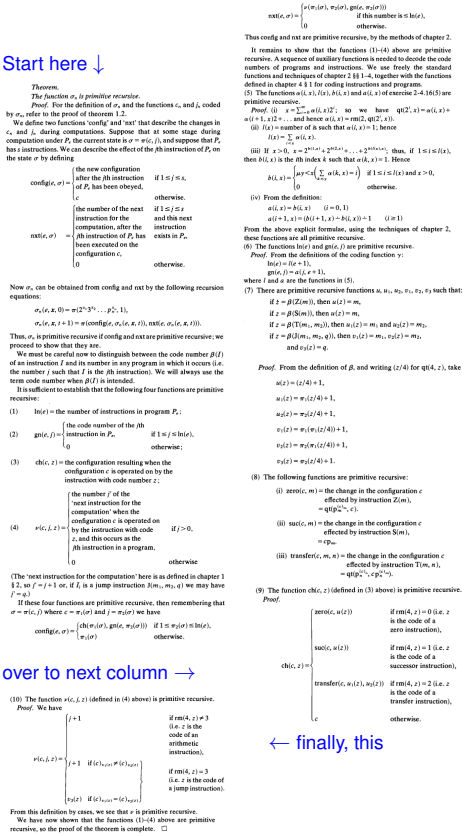


Figure 1.5: Details of the proof from [Cutland, 1980], p. 95-99 — with slightly different notation.

[13] A function is *primitive recursive* if it can be build from repeated application of composition and primitive recursion to the basic p.r. functions — i.e. minimalization is not necessary.

[14] This proof sketch was adapted from [Cutland, 1980]

1.5.1 *The Church-Turing thesis and computable functions*

NOT ONLY TMs AND P.R. FUNCTIONS can be used to express universal computation. Every "sensible" computational model ever imagined was shown to be equivalent to TMs, RMs and p.r. functions. The Church-Turing thesis states that this empirical observation is in fact a truth. Any function which can be computed by a TM is called a *computable function*.

1.6 *Imperative and declarative programming*

TODAY, MOST PEOPLE program a computer by means of imperative instructions. We might tell a computer to store a certain value in a given register, to test a condition and move control to a different instruction if the condition is verified, or unconditionally move control to another instruction. If we are using a high-level language, we use assignments, tests and loops.[15] Languages whose programs

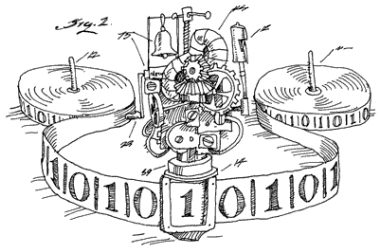


Figure 1.6: TMs are all that is the case! (rationallyspeaking.blogspot.fr).

[15] Assignments, tests and loops are the essential syntactical elements of any computer language which makes the language universal, i.e. capable of describing a universal TM, see [Harel, 1980]

consist of finite lists of imperative instructions are called *imperative languages*.[16]

The execution environment of an imperative language is a TM, or RM, or similar — in short, a "real computer". As is well known, CPUs take instructions of the imperative type, such as "move a value from a register to a location in memory" or "jump to instruction such-and-such".

We saw in previous sections that p.r. functions have the same expressive capabilities as imperative languages. And yet, we do not prescribe any action when writing a p.r. function. This begs the question: can one program using p.r. functions? This is indeed possible, as the many existing *declarative languages* testify.[17]

Declarative languages based on p.r. functions usually consist of a rather rich set of basic functions, and various kinds of operators in order to put together new functions.[18] The complicated functions obtained using recursive constructions from the basic ones are the "programs" of a declarative language. Each such function is evaluated at the input in order to obtain the output. Since each complex function is the result of various operators being applied recursively to some basic functions, its output can be obtained accordingly.

The natural execution environment of a declarative language would be a machine which can natively evaluate basic functions, and natively combine evaluations using basic operators. In practice, interpreters are used to turn declarative programs RM descriptions which can be executed on real computers.

[16] E.g. Fortran, Basic, C/C++, Java, Python.

[17] E.g. LISP, CAML.

[18] Although it would be theoretically possible to limit ourselves to the basic functions and operators above, this would yield unwieldy programs for a human to code.

Figure 1.7: Academic journals in MP.

1.7 Mathematical programming

We now come to the object of these notes. By "Mathematical Programming" we indicate a special type of declarative programming, namely one where the interpreter varies according to the mathematical properties of the functions involved in the program.

For example, if all the functions in a MP are linear in the variables, and the variables range over continuous domains,[19] one can use very efficient algorithms as interpreters, such as the *simplex*[20] or *barrier*[21] methods [Matoušek and Gärtner, 2007]. If the functions are nonlinear or the variables are integer, one must resort to enumerative methods such as *branch-and-bound*. [Liberti, 2006] Working implementations of these solution methods are collectively referred to as *solvers*. They read the MP (the program), the input of the problem, and they execute an imperative program which runs on the physical hardware.

[19] Optimizing a linear form over a polyhedron is known as *linear programming*.

[20] From any vertex of the polyhedron, the simplex method finds the adjacent vertex improving the value of the linear form, until this is no longer possible.

[21] From any point inside a polyhedron, the barrier method defines a *central path* in the interior of the polyhedron, which ends at the vertex which maximizes or minimizes a given linear form.

1.7.1 The transportation problem

MP is commonly used to formulate and solve optimization problems. A typical, classroom example is provided by the *transportation*

problem[22] [Munkres, 1957].

> Given a set P of production facilities with production capacities a_i for $i \in P$, a set Q of customer sites with demands b_j for $j \in Q$, and knowing that the unit transportation cost from facility $i \in P$ to customer $j \in Q$ is c_{ij}, find the optimal transportation plan.

If this is the first optimization problem you have ever come across, you are likely to find it meaningless. Since the only costs mentioned by the definition are transportation costs, we understand that the optimization must affect the total transportation cost. However, what exactly is meant by a "transportation plan"?

The formalization of a problem description is really an art, honed with experience and by knowing how to recognize patterns and structures within an informal text description of the problem. The first question one needs to ask when confronted with an informal description of a problem is **what are the decision variables?**, or, in other words, what is being decided? There is rarely a unique correct answer,[23] but the answer will influence the type of formulation obtained, and the efficiency of the solver. Experience will point the most likely correct direction.

In the transportation problem we are given the costs of transporting a unit of material from some entity (facility) to another (customer). This appears to imply that, in order to minimize the total cost, we shall need to decide on some quantities. Which quantities? We know that each facility cannot exceed its production capacity, and we know that each customer must receive its demanded supply. We should therefore decide quantities *out* of each production facility and *into* each customer site. One possibility[24] is to consider decision variables x_{ij} for each facility $i \in P$ and customer $j \in Q$, x_{ij} which denote the quantity of commodity shipped from i to j. The total cost is the sum of the unit costs multiplied by quantities:

$$\min \sum_{\substack{i \in P \\ j \in Q}} c_{ij} x_{ij}.$$

The capacity constraints dictate that the amount of commodity shipped out of each facility cannot exceed its capacity:

$$\forall i \in P \quad \sum_{j \in Q} x_{ij} \leq a_i.$$

The demand constraints make sure that each customer receives at least its desired demand:

$$\forall j \in Q \quad \sum_{i \in P} x_{ij} \geq b_j.$$

What else? Well, suppose you have two facilities with capacities $a_1 = a_2 = 2$ and one customer with demand $b_1 = 1$. Suppose that the transportation costs are $c_{11} = 1$ and $c_{21} = 2$ (see Fig. 1.10). Then

[22] The transportation problem is one of the fundamental problems in MP. It turns up time and again in practical settings. It is at the basis of the important business concern of *assignment*: who works on which task?

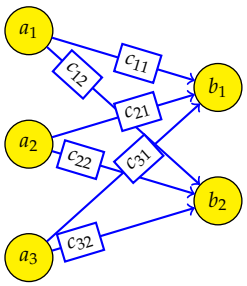


Figure 1.8: Visualize the setting of the transportation problem using a bipartite graph, facilities on the left, customers on the right; arcs are the potential transportation possibilities.

[23] Sometimes there is none; this often happens when consulting with industry — if they had a formal description of the problem, or one which is close to being formal, they would not be needing our services!

[24] MP beginners usually come up with the following set of decision variables: $\forall i \in P$, y_i is the quantity of commodity shipping from i; $\forall j \in Q$, z_j is the quantity of commodity shipping to j. Intuition fails them when trying to express the transportation costs in terms of y_i and z_j.

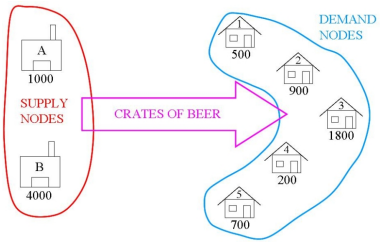


Figure 1.9: Student's view of the transportation problem (twiki.esc.auckland.ac.nz).

the problem reads:

$$\begin{aligned} \min x_{11} &+ 2x_{21} \\ x_{11} &\leq 2 \\ x_{21} &\leq 2 \\ x_{11} + x_{21} &\geq 1. \end{aligned}$$

The optimal solution is $x_{11} = 2, x_{21} = -1$: it satisfies the constraints and has zero total cost.[25] This solution, however, cannot be realistically implemented: what does it mean for a shipped quantity to be negative (Fig. 1.11)?

This example makes two points. One is that we had forgotten a constraint, namely:

$$\forall i \in P, j \in Q \quad x_{ij} \geq 0.$$

The other point is that forgetting a constraint might yield a solution which appears impossible *to those who know the problem setting*. If we had been given the problem in its abstract setting, without specifying that x_{ij} are supposed to mean a shipped quantity, we would have had no way to decide that $x_{21} = -1$ is inadmissible.

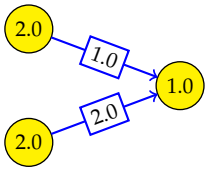


Figure 1.10: A very simple transportation instance.

[25] Every solution on the line $x_{11} + x_{21} = 1$ satisfies the constraints; in particular, because of the costs, the solution $x_{11} = 1$ and $x_{21} = 0$ has unit cost.

Figure 1.11: Negative shipments look weird (artcritique.wordpress.com).

1.7.2 *Network flow*

In the optimization of industrial processes, the *network flow* problem[26] [Ford and Fulkerson, 1956] is as important as the transportation problem.

> Consider a rail network connecting two cities s, t by way of a number of intermediate cities, where each link of the network has a number assigned to it representing its capacity. Assuming a steady state condition, find a maximal flow from city s to city t.

Note that the network flow problem is also about transportation.

- We model the rail network by a directed graph[27] $G = (V, A)$, and remark that s, t are two distinct nodes in V.
- For each arc $(u, v) \in A$ we let C_{uv} denote the capacity of the arc.

The commodity being transported over the network (people, freight or whatever else) starts from s; we assume this commodity can be split, so it is routed towards t along a variety of paths. We want to find the paths which will maximize the total flow rate.[28] We have to decide which paths to use, and what quantities of commodity to route along the paths. Since paths are sequences of arcs, and the material could be split at a node and follow different paths thereafter, we define decision variables x_{uv} indicating the flow rate along each arc $(u, v) \in A$.

A solution is feasible if the flow rates are non-negative (Fig. 1.12):

$$\forall (u, v) \in A \quad x_{uv} \geq 0,$$

if the flow rates do not exceed the arc capacities:

$$\forall (u, v) \in A \quad x_{uv} \leq C_{uv},$$

[26] Network flow is another fundamental MP problem. Its constraints can be used as part of larger formulations in order to ensure network connectivity.

[27] A *directed graph*, or *digraph*, is a pair (V, A) where V is any set and $A \subseteq V \times V$. We shall assume V is finite. Elements of V are called *nodes* and elements of A *arcs*.

[28] Assuming a constant flow, this is equivalent to maximizing the total quantity.

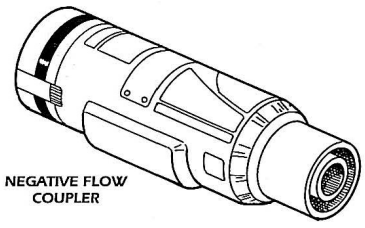


Figure 1.12: A negative flow coupler from starships of the rebellion era (starwars.wikia.com).

and if the positive flow rates along the arcs define a set of paths from s to t.

How do we express this constraint? First, since the flow originates from the source node s, we require that no flow actually arrives at s:

$$\sum_{(u,s)\in A} x_{us} = 0. \tag{1.7}$$

Next, no benefit[29] would derive from flow leaving the target node t:

$$\sum_{(t,v)\in A} x_{tv} = 0. \tag{1.8}$$

Lastly, the total flow entering intermediate nodes on paths between s and t must be equal to the total flow exiting them:[30]

$$\forall v \in V \setminus \{s,t\} \quad \sum_{(u,v)\in A} x_{uv} = \sum_{(v,u)\in A} x_{vu}. \tag{1.9}$$

The objective function aims at maximizing the amount of flow from s to t. Since flow is conserved at any node but s, t, it follows that however much flow exits s must enter t, and hence that it suffices to maximize the amount of flow exiting s:[31]

$$\max \sum_{(s,v)\in A} x_{sv}.$$

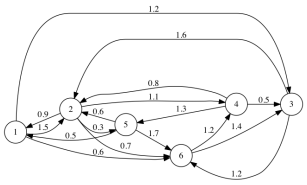


Figure 1.13: A MAX FLOW instance.

[29] Eq. (1.8) follows from (1.7) and (1.9) (prove it), and so it is redundant. If you use these constraints within a MILP, nonconvex NLP or MINLP, however, they might help certain solvers perform better.

[30] These constraints are called *flow conservation* equations or *material balance* constraints.

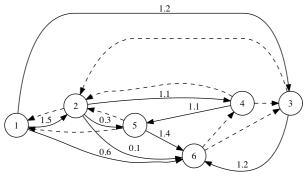


Figure 1.14: An optimal solution for the instance in Fig. 1.13.

[31] We could equivalently have maximized the amount of flow entering t (why?).

1.7.3 *Extremely short summary of complexity theory*

Formally, decision and optimization problems are defined as sets of infinitely many instances (input data) of increasing size. This is in view of the asymptotic worst-case complexity analysis of algorithms, which says that an algorithm A is $O(f(n))$ (for some function f) when there is an $n_0 \in \mathbb{N}$ such that for all $n > n_0$ the CPU time taken by A to terminate on inputs requiring n bits of storage is bounded above by $f(n)$.

If a problem had finitely many (say m) instances, we could precompute the solutions to these m instances and store them into a hash table, then solve any instance of this problem in constant time, making the problem trivial.

We say a decision problem is in **P** when there exists a polynomial-time algorithm that solves it, and in **NP** if every YES instance has a solution which can be verified to be YES in polynomial time.

Problems are **NP**-hard when they are as hard as the hardest problems in **NP**: **NP**-hardness of a problem Q is proved by reducing (i.e. transforming) a known **NP**-hard problem P to Q in polynomial time, such that each YES instance in P maps to a YES instance in Q and every NO instance in P to a NO instance in Q.

This method works since, if Q were any easier than P, we could solve P by reducing it to Q, solve Q, then map the solution of Q back to P; hence P would be "as easy as Q" (under polynomial-time reductions), contradicting the supposition that Q is easier than P.

These notions are informally applied to optimization problems by referring to their *decision version*: given an optimization problem

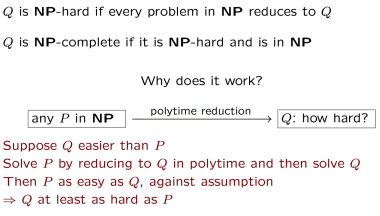


Figure 1.15: Hardness/completeness reductions.

Find the minimum of $f(x)$ subject to $g(x) \leq 0$,

its decision version is

given a number F, is it the case that there exists a solution x such that $f(x) \leq F$ and $g(x) \leq 0$?

Barring unboundedness, solving the decision version in polytime allows the (at least approximate) solution of the optimization problem in polytime too via the bisection method, which takes a logarithmic number of steps in terms of the range where f varies, at least if this range is discrete.

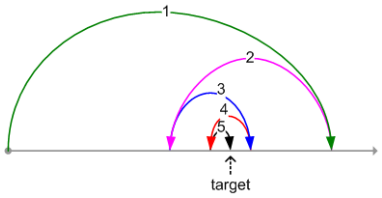


Figure 1.16: The bisection method (from codeproject.com).

1.7.4 *Definitions*

A MATHEMATICAL PROGRAM (MP) consists of parameters, decision variables, objective functions and constraints.

- The parameters encode the input (or *instance*): their values are fixed before the solver is called. The decision variables encode the output (or *solution*): their values are fixed by the solver, at termination.

- Decision variables are usually specified together with simple constraints on their domain, such as e.g. interval ranges of the form

$$x \in [x^L, x^U],$$

where x is a vector of n decision variables and $x^L, x^U \in \mathbb{R}^n$ such that $x_j^L \leq x_j^U$ for each $j \leq n$, or integrality constraints of the form

$$\forall j \in Z \quad x_j \in \mathbb{Z},$$

where Z is a subset of the decision variable indices $\{1, \ldots, n\}$.

- Objective functions have one of the following forms:

$$\min f(p, x)$$
$$\max f(p, x),$$

where p is a vector of q parameters, x is a vector of n decision variables, and f is a function mapping $\mathbb{R}^{q+n} \to \mathbb{R}$. A MP can have zero or more objective functions. Usually, it has one.

- Constraints have one of the following forms:

$$\begin{aligned} g(p, x) &\leq 0 \\ g(p, x) &= 0 \\ g(p, x) &\geq 0, \end{aligned}$$

where, again, $g : \mathbb{R}^{q+n} \to \mathbb{R}$. A MP can have zero or more constraints.

For most MP subclasses, the functions f, g in the objectives and constraints are explicitly given[32] as mathematical expressions recursively built up of parameter symbols and decision variable symbols, combined by means of the usual arithmetical operators and a few transcendental operators such as logarithms, exponentials and (very rarely) some trigonometric functions.

[32] When f, g are not given explicitly we have *Black-box optimization*.

1.7.5 The canonical MP formulation

Most *MP formulations* have the following form:

$$\left.\begin{array}{lrcl} \min_x & f(x) & & \\ \forall i \leq m & g_i(x) & \leq & 0 \\ & x & \in & [x^L, x^U] \\ \forall j \in Z & x_j & \in & \mathbb{Z}. \end{array}\right\} \tag{1.10}$$

Remarks:

1. we do not explicitly mention the parameters in the function forms;
2. most MPs in the mathematical literature have exactly one objective;[33]
3. objectives in maximization form $\max f$ can be transformed to minimization form by simply considering the problem $\min -f$ then taking the negative of the optimal objective function value, since:
$$\max f = -\min(-f);$$
4. most MPs in the mathematical literature have a finite number of constraints;[34]
5. constraints in the form $g(x) \geq 0$ can be replaced by $-g(x) \leq 0$, and those in the form $g(x) = 0$ by pairs of constraints $(g(x) \leq 0, g(x) \geq 0)$.

Let P be a MP formulation such as Eq. (1.10). We let:

$$\mathcal{F}(P) = \{x \in [x^L, x^U] \mid \forall i \leq m\ g_i(x) \leq 0 \wedge \forall j \in Z\ x_j \in \mathbb{Z}\} \tag{1.11}$$

be the *feasible set* of P. Any $x \in \mathcal{F}(P)$ is a *feasible solution*. A feasible solution x' is a *local optimum*[35] if there is a neighbourhood B of x' such that $B \subseteq \mathcal{F}(P)$ and $f(x')$ is minimal in B. A locally optimal solution x^* is a *global optimum* if $B = \mathcal{F}(P)$. We also let $\mathcal{G}(P)$ be the set of global optima of P, and $\mathcal{L}(P)$ be the set of local optima of P.

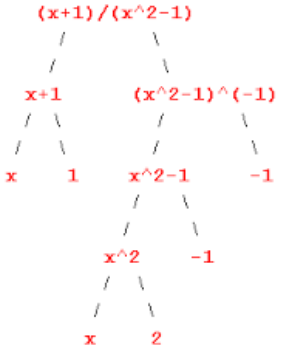


Figure 1.17: Parsing a recursively built mathematical expression yields an *expression tree* (from maplesoft.com).

[33] The study of MPs with multiple objectives is known as multi-objective programming (MOP), see Fig. 1.18 and Sect. 2.11.

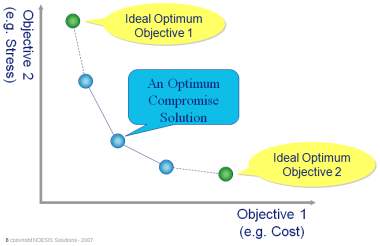


Figure 1.18: MOP yields a Pareto set of solutions (from www.noesissolutions.com).

[34] The study of MPs with infinitely many constraints is known as semi-infinite optimization, see Sect. 2.13.

[35] There are other definitions of local optimality, involving e.g. the derivatives of f at x' (but for them to hold, f must obviously be differentiable).

1.8 Modelling software

If the solvers are the interpreters for MP, the modelling software provides a mapping between optimization problems, formulated in abstract terms, and their instances, which is the actual input to the solver.

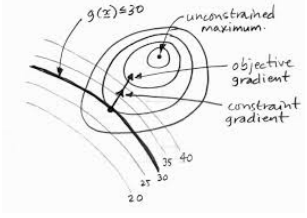


Figure 1.19: Gradient-based minimality conditions (from www.sce.carleton.ca).

1.8.1 *Structured and flat formulations*

Compare the transportation problem formulation given above:

$$\left.\begin{array}{rrcl} \min & \sum\limits_{\substack{i\in P\\ j\in Q}} c_{ij}x_{ij} & & \\ \forall i \in P & \sum\limits_{j\in Q} x_{ij} & \leq & a_i \\ \forall j \in Q & \sum\limits_{i\in P} x_{ij} & \geq & b_j \\ & x & \geq & 0, \end{array}\right\} \tag{1.12}$$

given in general terms, to its *instantiation*

$$\left.\begin{array}{rrcl} \min & x_{11} + 2x_{21} & & \\ & x_{11} & \leq & 2 \\ & x_{21} & \leq & 2 \\ & x_{11} + x_{21} & \geq & 1 \\ & x_{11}, x_{21} & \geq & 0, \end{array}\right\} \tag{1.13}$$

which refers to a given instance of the problem (see Sect. 1.7.3). The two formulations are different: the former is the generalization which describes all possible instances of the transportation problem,[36] and the latter corresponds to the specific instance where $P = \{1,2\}$, $Q = \{1\}$, $a = (2,2)$, $b = (1)$, $c = (1,2)$. The former cannot be solved by any software, unless P, Q, a, b, c are initialized to values; the latter, on the other hand, can be solved numerically by an appropriate solver.

Formulation (1.12) is as a *structured* formulation, whereas (1.13) is a *flat* formulation (see Fig. 1.20).

[36] An instance of the transportation problem is an assignment of values to the parameter symbols a, b, c indexed over sets P, Q of varying size.

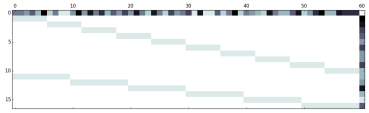


Figure 1.20: A flat linear MP can be represented by the sparsity pattern of the problem matrix (objective in top row, and RHS in rightmost column). Here is a random transportation problem instance.

1.8.2 *AMPL*

The syntax of the AMPL [Fourer and Gay, 2002] modelling software is very close to a structured formulation such as (1.12), and which interfaces with many solvers, some of them are free and/or open source. AMPL itself, however, is a commercial product. It has a free "demo" version which can deal with instances of up to 300 decision variables and 300 constraints.

The transportation problem can be formulated in AMPL as follows.[37]

```
# transportation.mod
param Pmax integer;
param Qmax integer;
set P := 1..Pmax;
set Q := 1..Qmax;
param a{P};
param b{Q};
param c{P,Q};
var x{P,Q} >= 0;
minimize cost: sum{i in P, j in Q} c[i,j]*x[i,j];
subject to production{i in P}: sum{j in Q} x[i,j] <= a[i];
subject to demand{j in Q}: sum{i in P} x[i,j] >= b[j];
```

[37] Note how close AMPL is to a structured form. On the other hand, AMPL can also take flat forms: see the encoding of Formulation (1.13) below.

```
var x11 >= 0;
var x21 >= 0;
minimize cost: x11 + x21;
subject to production1: x11 <= 2;
subject to production2: x21 <= 2;
subject to demand1: x11 + x21 >= 1;
```

The point is that AMPL can automatically transform a structured form to a flat form.

The various entities of a MP (sets, parameters, decision variables, objective functions, constraints) are introduced by a corresponding keyword. Every entity is named.[38] Every entity can be quantified over sets, including sets themselves. The cartesian product $P \times Q$ is written `{P,Q}`. AMPL formulations can be saved in a file with extension `.mod`.[39] Note that comments are introduced by the `#` symbol at the beginning of the line.

The instance is introduced in a file with extension `.dat`.[40]

```
# transportation.dat
param Pmax := 2;
param Qmax := 1;
param a :=
 1  2.0
 2  2.0
;
param b :=
 1  1.0
;
param c :=
 1 1  1.0
 2 1  2.0
;
```

Every `param` line introduces one or more parameters as functions[41] from their index sets to the values.

In order to actually solve the problem numerically, another file (bearing the extension `.run`) is used.[42]

```
# transportation.run
model transportation.mod;
data transportation.dat;
option solver cplexamp;
solve;
display x, cost;
```

The `transportation.run` file is executed in a command line environment by running[43] `ampl < transportation.run`. The output should look as follows.

```
CPLEX 12.6.2.0: optimal solution; objective 1
1 dual simplex iterations (0 in phase I)
x :=
1 1   1
2 1   0
;

cost = 1
```

which corresponds to $x_{11} = 1$, $x_{21} = 0$ having objective function value 1.

1.8.3 *Python*

Python is currently very popular with programmers because of the large amounts of available libraries: even complex tasks require very little coding.

[38] This is obvious for sets, parameters and variables, but not for objective functions and constraints.

[39] E.g. `transportation.mod`.

[40] E.g. `transportation.dat`

AMPL `.dat` files have a slightly different syntax with respect to `.mod` or `.run` files. For example, you would define a `set` using

```
set := { 1, 2 };
```

in `.mod` and `.run` files, but using

```
set := 1 2;
```

in `.dat` files. Also, there are ways to bi-dimensional arrays in tabular forms and transposed tabular forms. Multi-dimensional arrays can be written as many bi-dimensional array slices. For simplicity, I suggest you stick with the basic syntax.

[41] The parameter tensor $p_{i_1,\dots,i_k}$, where $\mathcal{P}$ is the set of tuples $(i_1,\dots,i_k)$ is encoded as the function $p : \mathcal{P} \to \mathrm{ran}(p)$. Each pair $(i_1,\dots,i_k) \mapsto p_{i_1,\dots,i_k}$ is given as a line `i1 i2 ... ik p_value`.

[42] The AMPl `.run` file functionality can be much richer than this example, and implement nontrivial algorithms. The imperative part of the AMPL language is however severely limited by the lack of a function call mechanism.

[43] AMPL actually has an Integrated Development Environment (IDE). I prefer using the command line since it allows me to pipe each command into the next. For example, in Unix (including Linux and MacOSX) you can ask for the optimal value by running

```
ampl < transportation.run
| grep optimal
```

(all on one line).

Figure 1.21: The Python language namesake.

Python is a high-level,[44] general purpose, imperative interpreted programming language. We assume at least a cursory knowledge of Python.

The transportation problem can be formulated in Python as follows. First, we import the PyOMO [Hart et al., 2012] libraries:

```
# define concrete transportation model in pyomo and solve it
from pyomo.environ import *
from pyomo.opt import SolverStatus, TerminationCondition
import pyomo.opt
```

Next, we define a function which creates an instance of the transportation problem, over the sets P, Q and the parameters a, b, c (see Eq. (1.12)).[45]

```
def create_transportation_model(P = [], Q = [], a = {}, b = {}, c = {}):
    model = ConcreteModel()
    model.x = Var(P, Q, within = NonNegativeReals)
    def objective_function(model):
        return sum(c[i,j] * model.x[i,j] for i in P for j in Q)
    model.obj = Objective(rule = objective_function)
    def production_constraint(model, i):
        return sum(model.x[i,j] for j in Q) <= a[i]
    model.production = Constraint(P, rule = production_constraint)
    def demand_constraint(model, j):
        return sum(model.x[i,j] for i in P) >= b[j]
    model.demand = Constraint(Q, rule = demand_constraint)
    return model
```

Finally, we write the "main" procedure,[46] i.e. the point of entry to the execution of the program.

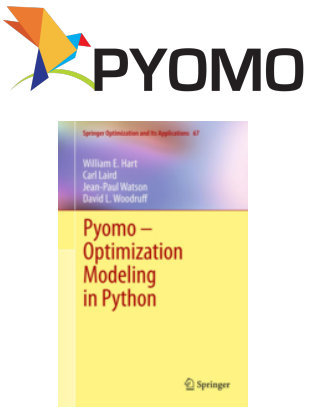
```
# main
P = [1,2]
Q = [1]
a = {1:2,2:2}
b = {1:1}
c = {(1,1):1, (2,1):2}
instance = create_transportation_model(P,Q,a,b,c)
solver = pyomo.opt.SolverFactory('cplexamp', solver_io = 'nl')
if solver is None:
    raise RuntimeError('solver not found')
solver.options['timelimit'] = 60
results = solver.solve(instance, keepfiles = False, tee = True)
if ((results.solver.status == SolverStatus.ok) and
    (results.solver.termination_condition == TerminationCondition.optimal)):
    # feasible and optimal
    instance.solutions.load_from(results)
    xstar = [instance.x[i,j].value for i in P for j in Q]
    print('solved to optimality')
    print(xstar)
else:
    print('not solved to optimality')
```

Note that these three pieces of code have to be saved, in sequence, in a file named `transportation.py`, which can be executed by running `python transportation.py`. The output is as follows.

```
timelimit=60
```

[44] A programming language is high-level when it is humanly readable (the opposite, low-level, would be said of machine code or assembly language). The opposite of "interpreted" is "compiled": in interpreted languages, the translation to machine code occurs line by line, and the corresponding machine code chunks are executed immediately after translation. In compiled languages, the program is completely translated to an executable file prior to running.

[45] PyOMO also offers an "abstract" modelling framework, through which you can actually read the instance directly from AMPL `.dat` files.



[46] What you do with the solution once you extract it from PyOMO using `solutions.load_from` is up to you (and Python). Since Python offers considerably more language flexibility and functionality than the imperative part of AMPL, the Python interface is to be preferred whenever solving MPs is just a step of a more complicated algorithm.

```
CPLEX 12.6.2.0: optimal solution; objective 1
1 dual simplex iterations (0 in phase I)
solved to optimality
[1.0, 0.0]
```

1.9 Summary

- Most people are used to imperative programming
- There is also declarative programming, and it is just as powerful
- Mathematical programming is a kind of declarative programming for optimization problems
- Using parameters, decision variables, objectives and constraints, optimization problems can be "modelled" using MP
- Efficient solution algorithms exist for different classes of MP
- Algorithms are implemented into *solvers*
- MP interpreted as "modelling, then calling an existing solver" shifts focus from *creating a solution algorithm for a given problem* (perceived as more difficult) to *modelling the problem using MP* (perceived as easier)
- MP formulations are created by humans using quantifiers, sets, and tensors/arrays of variables and constraints (*structured form*); solvers require their input to be MP formulation as explicit sequences of constants, variables, objectives, constraints (*flat form*)
- Modelling software is used to turn structured form to flat form, submit the flat form to a solver, then retrieve the solution from the solver and structure it back to the original form
- We look at two modelling environments: AMPL and Python.

Are you wondering how I produced Fig. 1.20? Take `transportation.py` and modify it as follows: (i) include

```
import numpy as np
import itertools
import np.random as npr

import matplotlib.pyplot as plt
```

at the beginning of the code. Then replace the definition of `a`, `b`, `c` after `# main` to

```
U=10; p=10; q=6
P=range(1,p+1); Q=range(1,q+1)
avals=npr.random_integers(1,U,size=p)
bvals=npr.random_integers(1,U,size=q)
cvals=npr.random_integers(1,U,size=p*q)
a=dict(zip(P,avals))
b=dict(zip(Q,bvals))
c=dict(zip(list(itertools.product(P,Q)),cvals))
M=np.zeros((1+p+q, p*q+1))
for i in P:
    for j in Q:
      M[0,q*(i-1)+(j-1)]=c[(i,j)]
for i in P:
    for j in Q:
      M[i,q*(i-1)+(j-1)]=1
    M[i,p*q] = a[i]
for j in Q:
    for i in P:
      M[p+j,p*(j-1)+(i-1)]=1
    M[p+j,p*q] = b[j]
plt.matshow(M, cmap = 'bone_r')

plt.show()
```

Essentially, I'm sampling random integer dictionaries for `a`, `b`, `c` (note the use of `zip`: look it up, it's an important function in Python), then I'm constructing the problem matrix M consisting in the objective function coefficients (top row), the constraint matrix, and the right hand side (RHS) vector in the rightmost column.

2
Systematics and solution methods

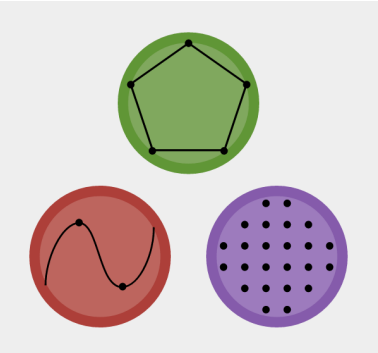


Figure 2.1: This picture, showing a polyhedron (for LP), a nonlinear function (for NLP) and an integer lattice (for MILP), is the logo of another modeling environment for MP, namely JuliaOpt (`www.juliaopt.org`). You are very welcome to try it!

MP formulations can belong to different classes. We list some of the most important in this chapter, together with a few notes about solution methodologies, software packages, and applications. Although all objective function directions have been cast to minimization, this bears no loss of generality. As already remarked in Sect. 1.7.5, $\max f$ can be replaced by $-\min(-f)$.

2.1 *Linear programming*

Linear Programming (LP) [Dantzig et al., 1955] is the minimization of a linear form on a polyhedron. The standard form is[1]

$$\left.\begin{array}{lrcl} \min & c^\top x & & \\ & Ax & = & b \\ & x & \geq & 0, \end{array}\right\} \tag{2.1}$$

where $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, A an $m \times n$ matrix are the parameters. LP is the most fundamental problem in MP.

LP is arguably the most fundamental problem in MP. It can be solved in polynomial time by the ellipsoid algorithm [Khachiyan, 1980] and by the interior point algorithm [Karmarkar, 1984], and is therefore in the computational complexity class[2] **P**. By continually improving and tuning the simplex and interior point methods, modern solver technology[3] can efficiently solve (sparse) LPs with millions of nonzeros in their constraint matrix.

LP often occurs as a subproblem of higher-level algorithms. In the Branch-and-Bound (BB) algorithm for solving MILPs, for example, every node of the BB search tree either gives rise to subnodes, or is pruned, according to the feasibility or integrality of the LP relaxation of the MILP.

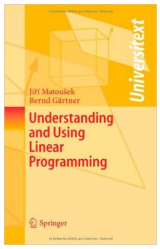
Classical applications of LP are: modelling points of equilibrium in economic behaviour, determining optimal blending recipes for composite food or chemical products, the already mentioned transportation, assignment, flow problems, planning production over time, finding sparse solutions to under-determined linear systems [Candès, 2014] and approximate solutions to over-determined linear systems, finding upper bounds to sphere packing problems, and many more. Several references exist on LP; [Vanderbei, 2001,

[1] LP can also be formulated in *canonical form* as $\min\{c^\top x \mid Ax \leq b\}$. The associated feasibility problem $Ax \leq b$ was first tackled by Fourier in 1827. Kantorovich and Koopmans won the Nobel prize in Economics in 1975 for their seminal work on formulating and solving economics problems as LPs. Hitchcock formulated the transportation problem (see Sect. 1.7.1) as an LP. Dantzig formulated many operation planning problems as LPs, and in 1947 he proposed the celebrated *simplex method* for solving LPs.

[2] Its associated decision problem turns out to be **P**-complete under log-space reductions, a useful fact in the complexity analysis of parallel algorithms.

[3] See e.g. IBM-ILOG CPLEX [IBM, 2010] and GLPK [Makhorin, 2003] for the state of the art in commercial and open-source LP solvers.

Matoušek and Gärtner, 2007] are among my favorites; the latter, in particular, contains a very interesting chapter on the application of LP to upper bounds for error correcting codes and sphere packing problems [Delsarte, 1972].



2.2 *Mixed-integer linear programming*

Mixed-Integer Linear Programming[4] (MILP) consists in the minimization of a linear form on a polyhedron, restricted to a non-negative integer lattice cartesian product the non-negative orthant. The standard form is

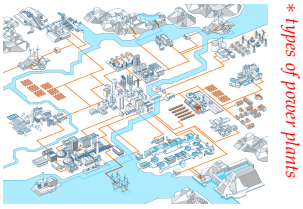
$$\left.\begin{array}{rrcl} \min & c^\top x & & \\ & Ax & = & b \\ & x & \geq & 0 \\ \forall j \in Z & x_j & \in & \mathbb{Z}_+, \end{array}\right\} \tag{2.2}$$

where c, b, A are as in the LP case, and $Z \subseteq \{1, \ldots, n\}$.

This is another fundamental problem in MP. It is **NP**-hard[5]; many problems can be cast as MILPs — its combination of continuous and integer variables give MILP a considerable expressive power as a modelling language. It includes ILP (integer variables only, see below), BLP (binary variables only, see below) and LP as special cases. MILPs can either be solved heuristically [Fischetti and Lodi, 2005, Fischetti et al., 2005] or exactly, by BB[6]

Here is an example of a problem which can be formulated as a MILP.

Power Generation Problem. There are $K_{\max}$ types * of power generation facilities, and, for each $k \leq K_{\max}$, n_k facilities of type k, each capable of producing w_k megawatts. Switching on or off a facility of type $k \leq K_{\max}$ has a fixed cost f_k and a unit time cost c_k. This set of power generation facilities has to meet demands d_t for each time period $t \leq T_{\max}$, where $T_{\max}$ is the time horizon. The unmet demand has to be purchased externally, at a cost C per megawatt. Formulate a minimum cost plan for operating the facilities.



Some of the decision variables[7] are defined as the number of facilities of a certain type used at a given period (these are integer variables). And yet other decision variables will be the quantity of power which needs to be purchased externally (these are continuous variables). Every relation between variables is linear, as well as the expression for the costs.

Another example is the following problem, which has its roots in [Euler, 1736] (see Fig. 2.2).

Traveling Salesman Problem[8] (TSP) [Flood, 1956]. Consider a traveling salesman who has to visit $n+1$ cities $0, 1, \ldots, n$ spending the minimum amount of money on car fuel. Consumption is assumed to be proportional to inter-city distances c_{ij} for each pair of cities $i < j \leq n$. Determine the optimal order of the cities to be visited.

[4] The first exact method for solving MILPs was proposed by Ralph Gomory [Gomory, 1958]. After joining IBM Research, he exploited integer programming to formulate some cutting problem for a client, and derived much of the existing theory behind cutting planes from the empirical observation of apparently periodic data in endless tabulations of computational results.

[5] A problem P is **NP**-hard when any other problem in the class **NP** can be reduced to it in polynomial time: from this follows the statement that an **NP**-hard problem must be at least as hard to solve (asymptotically, in the worst case) as the hardest problem of the class **NP** (see Sect. 1.7.3).

[6] The state of the art BB implementation for MILP is given by IBM-ILOG CPLEX [IBM, 2010]; the SCIP solver [Berthold et al., 2012], also very advanced, is free for academics. Other advanced commercial solvers are XPress-MP and GuRoBi.

[7] Try and formulate this problem as a MILP — use variables $x_{kt} \in \mathbb{Z}_+$ to mean the number of facilities of type k generating at period t, $y^1_{kt} \in \mathbb{Z}_+$ to mean the number of facilities of type k that are switched on at time t, y^0_{kt} switched off, and $z_t \in \mathbb{R}_+$ to mean the quantity of power purchased externally at period t.

[8] There is an enormous amount of literature about the TSP — it appears to be the most studied problem in combinatorial optimization! Entire books have been written about it [Applegate et al., 2007, Gutin and Punnen, 2002].

We define binary decision variables $x_{ij} \in \{0,1\}$ to mean that the order visits city i immediately before city j if and only if $x_{ij} = 1$. We also introduce continuous decision variables $u_0, \ldots, u_n \in \mathbb{R}$. Consider the following MILP formulation:

$$\min_{x,u} \sum_{i \neq j}^{n} c_{ij} x_{ij} \tag{2.3}$$

$$\forall i \leq n \quad \sum_{j=0}^{n} x_{ij} = 1 \tag{2.4}$$

$$\forall j \leq n \quad \sum_{i=0}^{n} x_{ij} = 1 \tag{2.5}$$

$$\forall i \neq j \neq 0 \quad u_i - u_j + n x_{ij} \leq n - 1 \tag{2.6}$$

$$\forall i, j \leq n \quad x_{ij} \in \{0,1\}. \tag{2.7}$$

The objective function Eq. (2.3) aims at minimizing the total cost of the selected legs of the traveling salesman tour. By Eq. (2.4)-(2.5),[9] we know that the feasible region consists of permutations of $\{0, \ldots, n\}$: if, by contradiction, there were two integers j, ℓ such that $x_{ij} = x_{i\ell} = 1$, then this would violate Eq. (2.4); and, conversely, if there were two integers i, ℓ such that $x_{ij} = x_{\ell j} = 1$, then this would violate Eq. (2.5). Therefore all ordered pairs (i, j) with $x_{ij} = 1$ define a bijection $\{0, \ldots, n\} \to \{0, \ldots, n\}$, in other words a permutation. Permutations can be decomposed in products of disjoint cycles. This, however, would not yield a tour but many subtours. We have to show that having more than one tour would violate Eq. (2.6). Suppose, to get a contradiction, that there are at least two tours. Then one cannot contain city 0 (since the tours have to be disjoint by definition of bijection): suppose this is the tour $i_1, \ldots, i_h$. Then from Eq. (2.6), by setting the x variables to one along the relevant legs, for each $\ell < h$ we obtain $u_{i_\ell} - u_{i_{\ell+1}} + n \leq n - 1$ as well as $u_{i_h} - u_{i_1} + n \leq n - 1$. Now we sum all these inequalities and observe that all of the u variables cancel out, since they all occur with changed sign in exactly two inequalities. Thus we obtain $n \leq n - 1$, a contradiction.[10] Therefore the above is a valid MILP formulation for the TSP.

Eq. (2.6) is not the only possible way to eliminate the subtours. Consider for example the exponentially large family of inequalities

$$\forall \varnothing \neq S \subsetneq \{0, \ldots, n\} \quad \sum_{\substack{i \in S \\ j \notin S}} x_{ij} \geq 1. \tag{2.8}$$

These *subtour elimination inequalities* state that for every nontrivial subset S of the cities, there must be a leg in the tour which exits S. Since it would be too time-consuming to generate the whole family of constraints (2.8), one usually proceeds iteratively: solve the formulation, and hope the resulting solution is a single tour. If not, identify a constraint which is violated by the current subtour solution, add it to the formulation, and repeat. This algorithm is called *row generation*.[11]

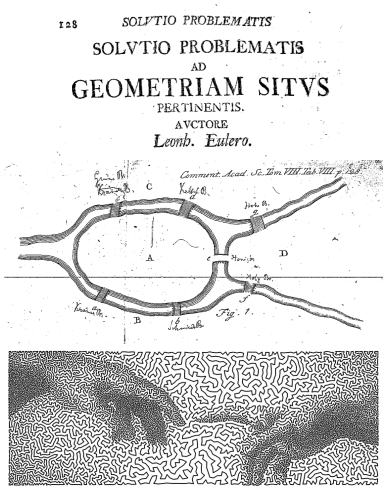


Figure 2.2: Euler, and TSP art.

[9] Constraints (2.4)-(2.5) define the *assignment constraints*, which define an incredibly important substructure in MP formulation — these crop up in scheduling, logistics, resource allocation, and many other types of problems. Assignment constraints define a bijection on the set of their indices.

[10] At least one tour (namely the one containing city 0) is safe, since we quantify Eq. (2.6) over i, j both non-zero.

[11] Note, however, that Eq. (2.8) are only defined on the binary variables x rather than the continuous ones u; so the formulation actually belongs to the BLP class (see Sect. 2.4).

2.3 *Integer linear programming*

INTEGER LINEAR PROGRAMMING (ILP) consists in the minimization of a linear form on a polyhedron, restricted to the non-negative integer lattice. The standard form is

$$\left.\begin{array}{lrcl} \min & c^\top x & & \\ & Ax & = & b \\ & x & \in & \mathbb{Z}^n_+, \end{array}\right\} \tag{2.9}$$

where c, b, A are as in the LP case. ILP is just like MILP but without continuous variables. Consider the following problem.

> SCHEDULING. There are n tasks to be executed on m processors. Task $i \leq n$ takes τ_i units of processor time, where $\tau \in [0, T]^n$ and $T \in \mathbb{N}$ is a time horizon. There is a precedence relation digraph $G = (V, A)$ where $V = \{1, \ldots, n\}$ and $(i, j) \in A$ if the execution of task i must end before the execution of task j begins. Determine the execution plan which ensures that all tasks are executed exactly once, and which minimizes the total time.

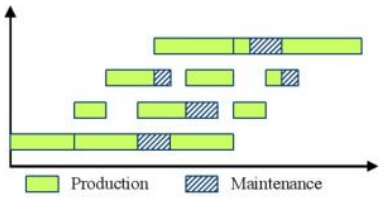


Figure 2.3: In scheduling, solutions are often represented on a plane, with processors on a discretized vertical axis, and time on the horizontal axis (from www.femto-st.fr).

Scheduling problems[12] are a vast area of combinatorial optimization, business planning and, in general, of all practical life! We can solve them using ILP by using the following decision variables: $x_i \in \mathbb{Z}_+$ indicates the starting time for task $i \leq n$, and $y_{ik} \in \{0, 1\}$ for each $i \leq n, k \leq m$ indicates that task i is executed by processor k if and only if $y_{ik} = 1$.

[12] Scheduling problems usually have two intertwined decisions: a temporal (partial) order, and an assignment. In this case the order is encoded by the $<$ relation on the values of the x_i variables and the assignment is encoded by the y_{ik} variables. The optimal value of these variables must depend on each other, since if two tasks i, j are assigned to the same processor, then necessarily one must end before the other can begin. Can you write these constraints? You might need other binary variables to determine a precedence for $(i, j) \notin A$.

2.4 *Binary linear programming*

BINARY LINEAR PROGRAMMING (BLP) consists in the minimization of a linear form on a polyhedron, restricted to the hypercube. The standard form is

$$\left.\begin{array}{lrcl} \min & c^\top x & & \\ & Ax & = & b \\ & x & \in & \{0, 1\}^n, \end{array}\right\} \tag{2.10}$$

where c, b, A are as in the LP case. BLP is usually employed for purely combinatorial problems.

2.4.1 *Linear assignment*

Find the assignment of minimum cost [Burkard et al., 2009] (Fig. 2.4).

> LINEAR ASSIGNMENT PROBLEM (LAP)[13] Given a set $V = \{1, \ldots, n\}$ and a cost matrix c_{ij} for $1 \leq i, j \leq n$, find the permutation π of V such that $\sum_{i \leq n} c_{i\pi(i)}$ is minimum.

Since we are looking for a permutation, we can exploit some of the constraints of the MILP formulation for the TSP, namely Eq. (2.4)-

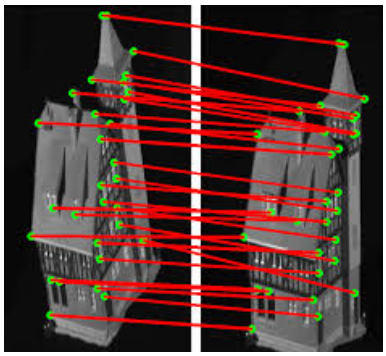


Figure 2.4: Image recognition application of linear assignment (from pages.cs.wisc.edu/~pachauri).

[13] The LAP can be solved in polynomial time: it is the same as the weighted perfect matching problem in a bipartite graph, which is known to be in **P** (also see Sect. 4.3.2 and its sidenotes).

(2.5):

$$\left.\begin{array}{rrcl} \min & \sum_{i,j\le n} c_{ij}x_{ij} & & \\ \forall i \le n & \sum_{j\le n} x_{ij} & = & 1 \\ \forall j \le n & \sum_{i\le n} x_{ij} & = & 1 \\ \forall i,j \le n & x_{ij} & \in & \{0,1\}. \end{array}\right\} \tag{2.11}$$

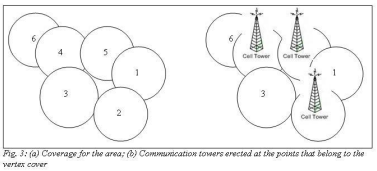


Fig. 3: (a) Coverage for the area; (b) Communication towers erected at the points that belong to the vertex cover

Figure 2.5: Vertex cover: location of communication towers (from www.bluetronix.net).

2.4.2 *Vertex cover*

Find the smallest set of vertices such that ever edge is adjacent to a vertex in the set[14] (see Fig. 2.5).

VERTEX COVER.[15] Given a simple undirected graph $G = (V, E)$, find the smallest subset $S \subseteq V$ such that, for each $\{i, j\} \in E$, either i or j is in S.

[14] In other words, the edge is *covered* by the vertex.

[15] This is an **NP**-hard problem [Karp, 1972].

Since we have to decide a subset $S \subseteq V$, we can represent S by its *indicator vector*[16] $x = (x_1, \ldots, x_n)$, where $n = |V|$, $x_i \in \{0, 1\}$ for all $i \le n$, and $x_i = 1$ if and only if $i \in S$.

[16] In combinatorial problems, subsets are very often represented by indicator vectors. This allows a natural formalization by MP.

$$\left.\begin{array}{rrcl} \min & \sum_{i\le n} x_i & & \\ \forall\{i,j\} \in E & x_i + x_j & \ge & 1 \\ \forall i \le n & x_i & \in & \{0,1\}. \end{array}\right\} \tag{2.12}$$

The generalization of VERTEX COVER to hypergraphs[17] is the

[17] In a *hypergraph*, *hyperedges* are arbitrary sets of vertices instead of just being pairs of vertices.

MINIMUM HYPERGRAPH COVER. Given a set V and a family $\mathcal{E} = \{E_1, \ldots, E_k\}$ of subsets of V, find the smallest subset $S \subseteq V$ such that every set in $\mathcal{E}$ has at least one element in S.

Generalizing from VERTEX COVER, every edge $e \in E$ has been replaced by a set E_ℓ in the family $\mathcal{E}$. We now have to generalize the cover constraint $x_i + x_j \ge 1$ to a set.

$$\left.\begin{array}{rrcl} \min & \sum_{i\le n} x_i & & \\ \forall \ell \le k & \sum_{i\in E_\ell} x_i & \ge & 1 \\ \forall i \le n & x_i & \in & \{0,1\}. \end{array}\right\} \tag{2.13}$$

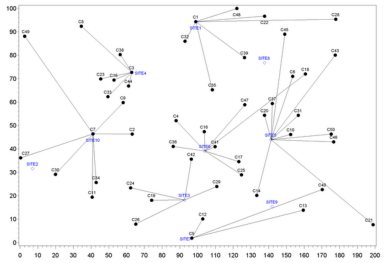


Figure 2.6: A minimum hypergraph cover is useful to choose the smallest numbers of optimally placed sites from a set (from support.sas.com).

2.4.3 *Set covering*

Given a family $\mathcal{E}$ of k subsets of V, we want to find the smallest subset of $\mathcal{E}$ the union of which contains V (see Fig. 2.7 for an application).

SET COVERING. Given a set V and a family $\mathcal{E} = \{E_1, \ldots, E_k\}$ of subsets of V, find the smallest subset $S \subseteq \{1, \ldots, k\}$ s.t. $\bigcup_{\ell\in S} E_\ell = V$.

Figure 2.7: In airline crew rostering, V are flight segments (or *legs*), $\mathcal{E}$ contains tours (or *rosters*), to which are assigned crews (since they eventually have to get back to where they started from) — the optimal crew roster is given by a minimum set cover.

The formulation is deceptively similar to Eq. (2.13):

$$\left.\begin{array}{rrcll} \min & \sum_{\ell\le k} x_\ell & & & \\ \forall v \in V & \sum_{\substack{\ell\le k \\ v\in E_\ell}} x_\ell & \ge & 1 & \textit{(covering constraint)} \\ \forall \ell \le k & x_\ell & \in & \{0,1\}. & \end{array}\right\} \tag{2.14}$$

The covering constraints state that every $v \in V$ must belong to at least one set E_ℓ. SET COVERING is **NP**-hard.

2.4.4 *Set packing*

In the same setting as SET COVERING, we now want to find the largest disjoint subset of $\mathcal{E}$.

> SET PACKING. Given a set V and a family $\mathcal{E} = \{E_1, \dots, E_k\}$ of subsets of V, find the largest subset $S \subseteq \{1, \dots, k\}$ such that $E_\ell \cap E_h = \varnothing$ for all $\ell \neq h$ in S.

We simply invert objective function direction and inequality sign:

$$\left.\begin{array}{rlcll} \max & \sum\limits_{\ell \leq k} x_\ell & & & \\ \forall v \in V & \sum\limits_{\substack{\ell \leq k \\ v \in E_\ell}} x_\ell & \leq & 1 & \textit{(packing constraint)} \\ \forall \ell \leq k & x_\ell & \in & \{0,1\}. & \end{array}\right\} \tag{2.15}$$

The packing constraints state that every $v \in V$ must belong to at most one set E_ℓ. In general, SET PACKING is **NP**-hard.

If V is the set of vertices of a graph, and $\mathcal{E}$ consists of the set of its edges, then SET PACKING is equal to the MATCHING problem, i.e. find a subset M of non-incident edges such that each vertex is adjacent to at most an edge in M. This particular version of the problem can be solved in polytime [Edmonds, 1965].

2.4.5 *Set partitioning*

Putting covering and packing together, we get partitioning.

> SET PARTITIONING. Given a set V and a family $\mathcal{E} = \{E_1, \dots, E_k\}$ of subsets of V, find a set $S \subseteq \{1, \dots, k\}$ such that (i) $\bigcup\limits_{\ell \in S} E_\ell = V$ and (ii) $\forall \ell \neq h \in S$ $(E_\ell \cap E_h = \varnothing)$.

This is a pure feasibility problem which can be formulated as follows.

$$\left.\begin{array}{rlcll} \forall v \in V & \sum\limits_{\substack{\ell \leq k \\ v \in E_\ell}} x_\ell & = & 1 & \textit{(partitioning constraint)} \\ \forall \ell \leq k & x_\ell & \in & \{0,1\}. & \end{array}\right\} \tag{2.16}$$

The partitioning constraints state that every $v \in V$ must belong to exactly one set E_ℓ. SET PARTITIONING is **NP**-hard.

Eq. (2.16) becomes an optimization problem whenever we can assign a cost c_ℓ to each set E_ℓ ($\ell \leq k$), for example we could have $c_\ell = |E_\ell|$. Other types of objective functions yield applications such as that of Fig. 2.10.

2.4.6 *Stables and cliques*

A *stable set* in a graph $G = (V, E)$ is a subset $I \subseteq V$ such that $\{u, v\} \notin E$ for any pair $u \neq v \in I$.

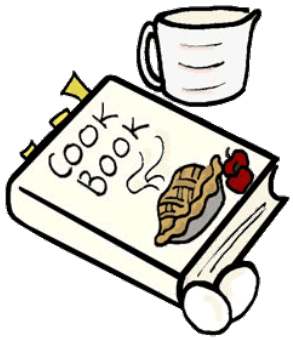


Figure 2.8: What is the largest number of most diverse ($E_\ell \cap E_h = \varnothing$) recipes (from your cookbook $\mathcal{E}$) that you can make with the ingredients V you have at home?

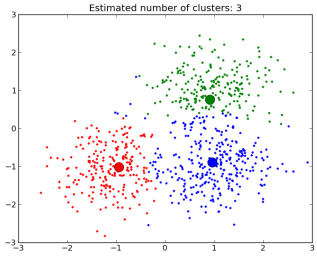


Figure 2.9: Set partitioning is sometimes useful as a modelling tool in clustering, specifically when one has an exhaustive list of all the possible clusters as part of the input (from `scikit-learn.sourceforge.net`).

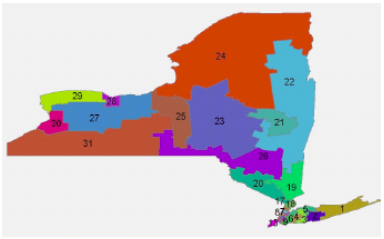


Figure 2.10: Political districting in NY state (from `andrewgelman.com`). Given a set $\mathcal{E}$ of potential (overlapping) districts, choose the partition of the state which minimizes the maximum deviation from the average.

Given a set X and a family $\mathcal{E}$ of k subsets of V, consider the graph $\mathcal{G}$ having $\mathcal{E}$ as its vertex set, and such that $\{E_\ell, E_h\}$ is an edge of $\mathcal{G}$ if and only if $E_\ell \cap E_h \neq \varnothing$. Then[18] S is a set packing of $(X, \mathcal{E})$ if and only if S is a stable set of $\mathcal{G}$.

The relevant problem is to find a stable set of maximum size:

MAX STABLE SET. Given a simple undirected graph $G = (V, E)$, find a stable set $S \subseteq V$ of having maximum cardinality.

If we now look at the *complement graph*,[19] a stable set becomes a *clique*[20] The corresponding MAX CLIQUE problem is finding the clique $S \subseteq V$ of G having maximum cardinality.[21] Both MAX STABLE SET and MAX CLIQUE are **NP**-hard.[22]

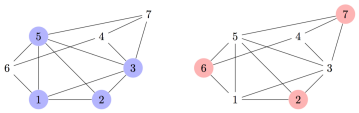


Figure 2.11: A clique and a stable set in a graph.

[19] This is the graph obtained by negating each edge; i.e., if $\{i, j\} \in E$ in the given graph, $\{i, j\} \notin E$ in the complement, and vice versa.

[20] A set of vertices pairwise linked by edges.

[21] Write the BLP formulations of MAX STABLE SET and MAX CLIQUE.

[22] Prove this.

2.4.7 *Other combinatorial optimization problems*

There cannot be an exhaustive list of BLP formulations for all combinatorial problems on graphs and hypergraphs. Those we mentioned here are supposed to give some indication of the most basic modelling techniques using BLP.

2.5 *Convex nonlinear programming*

CONVEX NONLINEAR PROGRAMMING (cNLP) consists in the minimization of a convex function on a convex set. The standard form is

$$\left.\begin{array}{rll} \min & f(x) & \\ & g(x) \leq & 0, \end{array}\right\} \tag{2.17}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^m$ are convex functions.

cNLPs can be solved efficiently using local NLP solvers such as IPOPT or SNOPT, which are designed to find a local optimum on (general, possibly nonconvex) NLPs (see Sect. 2.6 below). Since every local optimum of a cNLP is also global (by convexity of f, g), local solvers find global optima in this setting.

cNLPs can in fact be solved in polynomial time. Strictly speaking, however, we cannot state the cNLP is in **P**, since solutions of cNLPs could be irrational and even transcendental — and our computational model based on TMs does not allow us to write all[23] irrational and transcendental numbers *precisely*. So we have to speak about ε error tolerances: With this *caveat*, we can state that we can find arbitrary close approximations of cNLP optima in polynomial time.[24]

Similarly to LP, which is used within other algorithms such as BB, cNLP is also used in the nonlinear version of the BB algorithm.

Applicationwise, many problems can be cast as cNLPs (see [Boyd and Vandenberghe, 2004]). For example, given a point $p \in \mathbb{R}^n$ and a convex feasible set $\mathcal{F} = \{x \in \mathbb{R}^n \mid \forall i \leq m\ g_i(x) \leq 0\}$, finding the point $y \in \mathcal{F}$ closest to p is a cNLP:[25]

$$\left.\begin{array}{rll} \min & \|x - p\|_\ell & \\ \forall i \leq m & g_i(x) \leq & 0. \end{array}\right\} \tag{2.18}$$

Figure 2.12: Let $A_1, \ldots, A_n \in \mathbb{R}^m$ be vectors representing a small set of high-dimensional images from a live feed (e.g. a security camera), and $b \in \mathbb{R}^m$ be a vector representing a target image (e.g. a thief). Can we approximately express b as a linear combination of $A_1, \ldots, A_m$ (in other words, can we detect the thief)? Since $m > n$, there will of course be an error $\|Ax - b\|_2$, where A is the $m \times n$ matrix consisting of the columns A_i ($i \leq n$), and $x \in \mathbb{R}^n$ are the coefficients we want to find. Depending on how small we can make $\|Ax - b\|_2$, we might convict the thief or let her walk. The problem $\min \|Ax - b\|_2$ is a cNLP.

[23] Here's an interesting question: how about being able to write just one optimum precisely? E.g., given a problem instance, couldn't we simply denote the optimum of interest by a simple symbol, e.g. ω?

[24] In general, these results are implied by showing that some algorithm runs in $O(p(|I|, \frac{1}{\varepsilon}))$ where $|I|$ is the size of the instance and $p(\cdot, \cdot)$ is a polynomial.

[25] Linear regression generalizes projection and is possibly one of the practically most useful (and used) tools in all of mathematics. It can be cast as the following cNLP, which is in fact a cQP (see Sect. 2.9.1 below): $\min \|Ax - b\|_2$, where A is an $m \times n$ matrix, $b \in \mathbb{R}^m$, and $m > n$.

Since ℓ norms are convex functions and the constraints are convex, Eq. (2.18) is a cNLP. It is routinely used with the Euclidean norm ($\ell = 2$) to project points onto convex sets, such as cones and subspaces.

Surprisingly, the problem of interpolating an arbitrary set of points in the plane[26] by means of a given class of functions can be cast as a cNLP. Let P be a finite set of k points (p_i, q_i) in $\mathbb{R}^2$, and consider the parametrized function:

$$f_u(x) = u_0 + u_1 x + u_2 x^2 + \cdots + u_m x^m.$$

The problem is that of looking for the values $u_0, \ldots, u_m$ minimizing the norm of the error vector $F(u) = (f_u(p_i) - q_i \mid i \leq k)$, e.g.:

$$\min_u \|F(u)\|_\ell$$

for some positive integer ℓ. The point is that f_u is nonlinear in x but linear in u, which are the actual decision variables.

[26] Generalize this to $\mathbb{R}^n$.

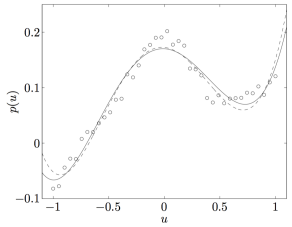


Figure 2.13: Nonlinear function interpolation [Boyd and Vandenberghe, 2004].

2.6 *Nonlinear programming*

Nonlinear Programming (NLP) consists in the minimization of any nonlinear function on any nonlinearly defined set. The standard form is

$$\left.\begin{array}{lll} \min & f(x) & \\ & g(x) & \leq \quad 0, \end{array}\right\} \tag{2.19}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^m$ are any nonlinear functions.

Adding nonconvex functions to NLP makes it much more expressive than cNLP. For example, NLP can model circle packing [Costa et al., 2013] (see Fig. 2.14).

> Packing Equal Circles in a Square (PECS). Given an integer n, find the largest radius r such that n circles of radius r can be drawn in a unit square in such a way that they intersect pairwise in at most one point.

The formulation[27] is as follows:

$$\left.\begin{array}{rrcl} \max & r & & \\ \forall i < j \leq n & (x_i - x_j)^2 + (y_i - y_j)^2 & \geq & (2r)^2 \\ \forall i \leq n & r \leq x_i & \leq & 1 - r \\ \forall i \leq n & r \leq y_i & \leq & 1 - r. \end{array}\right\} \tag{2.20}$$

Solving circle packing problems is hard in practice (see the www.packomania.com website). However, no-one ever established whether the PECS is actually **NP**-hard.[28] This is the case for many problems having small instance sizes, e.g. for the decision version of the PECS, where r is given as part of the input instead of being maximized, an instance is the pair (r, n), and the number of bits required to store them is $|n| + |r|$, where

$$\forall y = \frac{p}{q} \in \mathbb{Q} \quad |y| = \lceil \log_2 p \rceil + \lceil \log_2 q \rceil$$

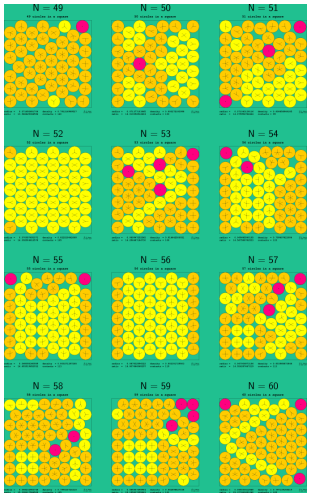


Figure 2.14: Some configurations for circle packing in a square (from www.packomania.com).

[27] Write a formulation for the variant of this problem where r is fixed and you maximize n.

[28] It is unlikely that the PECS will actually be in **NP**, since a certificate of optimality is a solution $((x_j, y_j) \mid i \leq n)$ which might have irrational components — and it might be impossible to represent them with a number of bits which is at most polynomial in the size of the input; which, in this case, is the memory necessary to store the two scalars r, n.

is the number of bits of memory required to store the floating point number y. By contrast, the size of an LP $\min\{cx \mid Ax \leq b\}$ where A is $m \times n$ is $\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}| + \sum_{i=1}^{m}|b_i| + \sum_{j=1}^{n}|c_j|$. Even taking the storage for components A, b, c to be as low as possible (1 bit), this amounts to $m + n + mn$, against $|n| + |r|$ which is of order $\log_2 n$.

The issue with having a problem Q with small instance size[29] is that **NP**-hardness is usually established by reduction (see Sect. 1.7.3) from a problem P which is already known to be **NP**-hard. All **NP**-hard problems we know have sizes at least proportional to an integer n, rather than its logarithm.

A related decision problem is the following.

FEASIBILITY KISSING NUMBER PROBLEM (fKNP).[30] Given two integers n, K, can we fit n unit balls $B_1, \ldots, B_n$ around the unit ball B_0 centered at the origin of $\mathbb{R}^K$, so that $|B_0 \cap B_i| = 1$ for all $i \leq n$ but the interiors of all the balls are pairwise disjoint?[31]

Like the PECS, the fKNP is very hard in practice [Kucherenko et al., 2007] but we do not know whether it is **NP**-hard or not, for much the same reasons as the PECS.

Another problem which can be formulated as an NLP is the following.

EUCLIDEAN DISTANCE GEOMETRY PROBLEM (EDGP). Given a positive integer K and a simple undirected graph $G = (V, E)$ with a distance function $d : E \to \mathbb{R}_+$ on the edges, determine whether there is a realization[32] $x : V \to \mathbb{R}^K$ such that:

$$\forall \{u, v\} \in E \quad \|x_u - x_v\|_2 = d_{uv}, \tag{2.21}$$

where $\forall v \in V \; x_v \in \mathbb{R}^K$.

If $K = 2$ the EDGP essentially asks to draw a given graph on the plane such that edge segments have lengths consistent with the edge function d. The EDGP has many applications in engineering and other fields [Liberti et al., 2014].

Nonconvex NLPs may have many local optima which are not global. There are therefore two categories of solvers which one can use to solve NLPs: local solvers (for finding local optima) and global solvers (for finding global optima).

We mentioned local NLP solvers (such as IPOPT or SNOPT) in connection with solving cNLP globally in Sect. 2.5 above. Unfortunately, any guarantee of global optimality disappears when deploying such solvers on nonconvex NLPs. The actual situation is even worse: most local NLP solvers are designed to simply finding the closest local optimum from a given feasible point.

In general, local NLP algorithms are iterative, and identify a sequence $x^k \in \mathbb{R}^n$ (for $k \in \mathbb{N}$) which starts from a given feasible point x^0 and by finding a feasible improving direction vector d^k by means of solving an auxiliary (but easier) optimization subproblem, they then set

$$x^{k+1} = x^k + d^k.$$

[29] **NP** reductions have to be performed in time bounded by a polynomial of the instance size of P, and have map YES (respectively, NO) instances of P to YES (respectively, NO) instances of Q. Inever had any promising intuition for encoding $O(n)$ bits of information in $O(\log_2 n)$ in such a way as to preserve YES and NO answers.

[30] This problem in $K = 3$ was originally discussed between Isaac Newton and one of his ex students, David Gregory [Liberti, 2012].

[31] Formulate the fKNP as a nonconvex NLP.

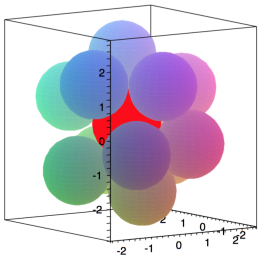


Figure 2.15: A 12-sphere packing: is there space for a 13th ball?

[32] A *realization* is a function satisfying Eq. (2.21).

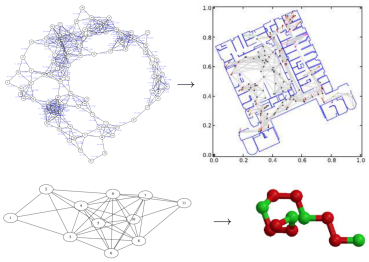


Figure 2.16: Two applications of EDGP: localization of sensor networks, and protein conformation from Nuclear Magnetic Resonance (NMR) data.

Termination is achieved whenever $\|x^{k+1} - x^k\| \leq \varepsilon$, where $\varepsilon > 0$ is a given approximation error tolerance. The auxiliary subproblem usually aims at satisfying first, and possibly second order[33] optimality conditions.

These conditions, however, are not valid everywhere: further assumptions called *constraint qualifications* must be met. Unfortunately, whether these assumptions are valid depends on the instance rather than the problem. Moreover, from a worst-case complexity point of view, finding a feasible point in a nonconvex NLP is just as hard as finding an optimal point,[34] so the assumption that a feasible point is given is also unrealistic.[35] On the other hand, if all the assumptions are verified and a feasible point exists, local NLP methods can find the closest local optimum rather efficiently.

Nonconvex NLPs can be solved globally to a given $\varepsilon > 0$ tolerance (on the optimal objective function value) by means of a BB variant called *spatial Branch-and-Bound* (sBB) [Liberti, 2006, Belotti et al., 2009].[36] The best-known sBB implementation is BARON [Sahinidis and Tawarmalani, 2005], which is a commercial solver. Recent versions of SCIP [Berthold et al., 2012] can also solve MINLP (see Sect. 2.8) and hence, in principle, NLPs as well. The only open source sBB implementation worthy of note is COUENNE [Belotti, 2011].

[33] For example, Sequential Quadratic Programming (SQP) is a class of methods which is based on solving the following auxiliary subproblem:

$$\left.\begin{array}{ll} \min\limits_{d \in \mathbb{R}^n} & f(x^k) + [\nabla_x f(x^k)]^\top d + \\ & + \frac{1}{2} d^\top \nabla_{xx} \mathcal{L}(x^k, \lambda^k) d \\ \forall i \leq m & g_i(x^k) + [\nabla_x g_i(x^k)]^\top d \leq 0 \end{array}\right\}$$

where $\mathcal{L}(x, \lambda) = f(x) + \lambda g(x)$ is the *Lagrangian* of the NLP, and λ is the vector of *Lagrange coefficients*. The solution to this problem yields d^k at each step k (λ^k are also updated at each step).

[34] Why?

[35] These methods are often used to improve existing business or production processes, where a feasible solution can be derived from the real-world setting.

[36] BB type algorithms are used to handle nonconvexity. In MILP, the integrality constraints are the only source of nonconvexity, and, accordingly, BB branches on variables which take a fractional value when solving the LP relaxation at each node. In nonconvex NLP, objective and constraints are nonconvex. Instead of branching on variables taking fractional values, we branch on variables contributing a discrepancy between an objective function value of the original problem and of the convex relaxation solved at each sBB node.

2.7 *Convex mixed-integer nonlinear programming*

CONVEX MIXED-INTEGER NONLINEAR PROGRAMMING (cMINLP) consists in the minimization of a convex function on a convex set, restricted to the cartesian product of an integer lattice and a Euclidean space. The standard form is

$$\left.\begin{array}{lrcl} \min & f(x) & & \\ & g(x) & \leq & 0 \\ \forall j \in Z & x_j & \in & \mathbb{Z}, \end{array}\right\} \tag{2.22}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^m$ are convex functions, and $Z \subseteq \{1, \ldots, n\}$.

A typical cMINLP application is mean-variance portfolio selection with cardinality constraints:

> CARDINALITY CONSTRAINED MEAN-VARIANCE PORTFOLIO SELECTION (CCMVPS).[37] Decide how to partition a unit budget into an investment portfolio which maximizes the expected return (given the return coefficient vector ρ) subject to a given bound r on the variance (given the covariance matrix Q) and to a maximum of k different investments out of a possible K.[38]

Insofar as cMINLPs are a generalization of cNLP to integer variables, all of the applications mentioned in Sect. 2.5 are relevant to this section, too. For example, the *integral* point $x^* \in \mathbb{Z}^n$ inside a given convex set $\{x \in \mathbb{R}^n \mid \forall i \leq m \; g_i(x) \leq 0\}$ which is closest (using euclidean distances) to a given point $p \in \mathbb{R}^n$ can be obtained by solving a variant of Eq. (2.18) where $x \in \mathbb{Z}^n$ and $\ell = 2$.

Figure 2.17: Optimization of water networks: deciding diameters for all pipes and water flow given water at each node over a time horizon [Bragalli et al., 2012]. Although this is actually a *nonconvex* cMINLP methods are used heuristically to solve it.

[37] This is a cardinality constrained variant of the celebrated Markowitz' portfolio selection problem [Konno and Wijayanayake, 2001]. Markowitz won the Nobel prize in Economics for his contributions on the portfolio problem.

[38] Formulate the CCPS as a cMINLP.

Also, most nonconvex functions are convex in some subdomain. This means that cMINLP algorithms can also be deployed on *nonconvex* MINLP (see Sect. 2.8 below) forsaking the global optimality guarantee (i.e. they will act as heuristics and may find local optima).

Currently, the best solver for cMINLP is the open-source BONMIN [Bonami and Lee, 2007].

2.8 Mixed-integer nonlinear programming

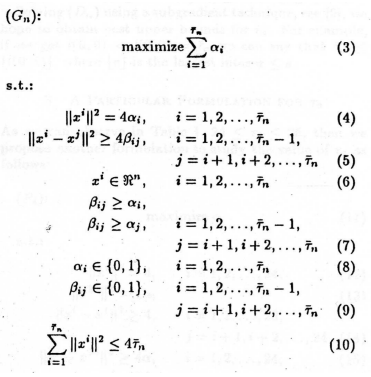
Mixed-Integer Nonlinear Programming (MINLP) consists in the minimization of any nonlinear function on any nonlinear defined set, restricted to an integer lattice cartesian product a Euclidean space. The standard form is

$$\left.\begin{array}{rlcl} \min & f(x) & & \\ & g(x) & \leq & 0 \\ \forall j \in Z & x_j & \in & \mathbb{Z}^n, \end{array}\right\} \tag{2.23}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^m$ are any nonlinear functions, and $Z \subseteq \{1, \ldots, n\}$.

MINLP is possibly the most general class of MPs. It contains all of the other classes listed in this chapter with the exception of SDP (see below). The kissing number problem (optimization version) can be formulated as a MINLP [Maculan et al., 1996]. Most industrial processes can be formulated as MINLP [Liberti, 2015]. Its expressive power is remarkable: it can encode any TM, including universal ones, such as Minsky's RM [Liberti and Marinelli, 2014], which means that *every problem* can be formulated as a MINLP (albeit one with an infinite number of variables and constraints),[39] even unsolvable ones such as the halting problem. In other words, MINLP is Turing-complete[40] as a programming language. Since MP contains MINLP as a sub-class, MP is also Turing-complete.

MINLPs can be approximated to a given $\varepsilon > 0$ tolerance using the sBB algorithm (implemented in e.g. BARON, SCIP and Couenne). See [D'Ambrosio and Lodi, 2011] for more details.



(G_n):

$$\text{maximize} \sum_{i=1}^{\bar{r}_n} \alpha_i \tag{3}$$

s.t.:

$$\|x^i\|^2 = 4\alpha_i, \quad i = 1, 2, \ldots, \bar{r}_n \tag{4}$$

$$\|x^i - x^j\|^2 \geq 4\beta_{ij}, \quad i = 1, 2, \ldots, \bar{r}_n - 1, \; j = i+1, i+2, \ldots, \bar{r}_n \tag{5}$$

$$x^i \in \mathfrak{R}^n, \quad i = 1, 2, \ldots, \bar{r}_n \tag{6}$$

$$\beta_{ij} \geq \alpha_i, \quad \beta_{ij} \geq \alpha_j, \quad i = 1, 2, \ldots, \bar{r}_n - 1, \; j = i+1, i+2, \ldots, \bar{r}_n \tag{7}$$

$$\alpha_i \in \{0, 1\}, \quad i = 1, 2, \ldots, \bar{r}_n \tag{8}$$

$$\beta_{ij} \in \{0, 1\}, \quad i = 1, 2, \ldots, \bar{r}_n - 1, \; j = i+1, i+2, \ldots, \bar{r}_n \tag{9}$$

$$\sum_{i=1}^{\bar{r}_n} \|x^i\|^2 \leq 4\bar{r}_n \tag{10}$$

Figure 2.18: The maximization formulation of the KNP [Maculan et al., 1996].

[39] In fact we only require a finite number of polynomial functions in finitely many variables for a MINLP to be able to formulate *any* problem, see Hilbert's 10th problem [Matiyasevich, 1993].

[40] A programming language is *Turing-complete* when it can describe a UTM.

2.9 Quadratic programming formulations

Quadratic programming is the easiest type of nonlinear programming. With respect to linear programming it allows a lot more flexibility in expressive power whilst keeping a lot of structure, which allows us to give stronger theoretical guarantees to formulation properties and solution algorithms.

2.9.1 Convex quadratic programming

convex Quadratic Programming (cQP) consists in the minimization of a semi-definite quadratic form subject to a polyhedron. The

standard form is

$$\left.\begin{array}{lrcl} \min & x^\top Q x & + & c^\top x \\ & Ax & \leq & b, \end{array}\right\} \tag{2.24}$$

where $c \in \mathbb{R}^n$, Q is an $n \times n$ semi-definite matrix, $b \in \mathbb{R}^m$ and A is an $m \times n$ matrix.

One of the best known application in this class Markowitz' mean-variance portfolio selection problem [Markowitz, 1952], which we already discussed in its cardinality constrained variant CCMVPS (see Sect. 2.7). Given a unit budget, decide the fractions of budget to invest in n possible stakes with returns $\rho_1, \ldots, \rho_n$, so as to minimize the risk (given the covariance matrix Q of the n investments) and subject to making at least a given mean return μ.

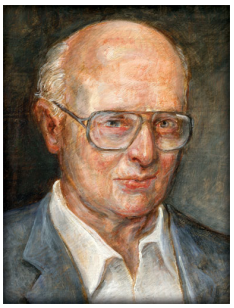


Figure 2.19: Harry Markowitz (from www.ifa.com).

$$\left.\begin{array}{lrcl} \min & x^\top Q x & & \\ & \sum_{i=1}^{n} \rho_i x_i & \geq & \mu \\ & \sum_{i=1}^{n} x_1 & = & 1 \\ & x & \in & [0,1]^n. \end{array}\right\} \tag{2.25}$$

Since covariance matrices are positive semidefinite[41] (PSD),[42] Eq. (2.25) is a cQP.

Another well known application of cQP is:

RESTRICTED LINEAR REGRESSION (rLR). Given m points $p_1, \ldots, p_m \in \mathbb{R}^n$ arranged as rows of an $m \times n$ matrix P, find the hyperplane $a^\top y = a_0$ (where $(a_0, a) = (a_0, a_1, \ldots, a_n)$ is restricted to lie in a set $X \subseteq \mathbb{R}^{n+1}$ described by linear inequalities) which minimizes[43] the error

$$\|Pa - a_0\|_2.$$

Like all convex programs, the cQP may also arise as a convex relaxation of a nonconvex NLP or QP, used at every node of a sBB algorithm.

[41] A square symmetric matrix Q is *positive semidefinite* (denoted $Q \succeq 0$) if $\forall x\ x^\top Q x \geq 0$; or, equivalently, if all of its eigenvalues are non-negative.

[42] Prove that all covariance matrices are PSD, but the converse does not hold.

[43] The rLR formulation is

$$\min_{a_0, a} \{\|Pa - a_0\|_2 \mid (a_0, a) \in X\}.$$

2.9.2 *Quadratic programming*

QUADRATIC PROGRAMMING (QP) consists in the minimization of any quadratic form subject to a polyhedron. The standard form is

$$\left.\begin{array}{lrcl} \min & x^\top Q x & + & c^\top x \\ & Ax & \leq & b, \end{array}\right\} \tag{2.26}$$

where $c \in \mathbb{R}^n$, Q is an $n \times n$ matrix, $b \in \mathbb{R}^m$ and A is an $m \times n$ matrix.

If $Q \succeq 0$ then Q is positive semidefinite the QP is actually a cQP (see Sect. 2.9.1 above). If Q only has strictly positive eigenvalues, we write $Q \succ 0$ and call Q *positive definite*. If $Q \preceq 0$ (corresponding to $-Q \succeq 0$), Q is called *negative semidefinite*, and if $Q \prec 0$ then Q is *negative definite*. The remaining case, of Q having both positive and negative eigenvalues is called *indefinite*.

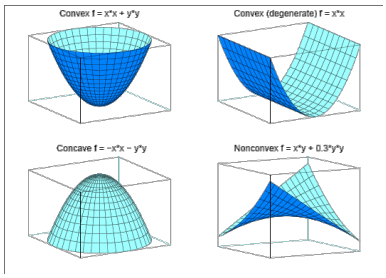


Figure 2.20: Quadratic programming (from support.sas.com).

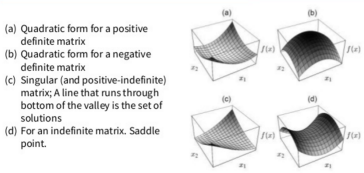


Figure 2.21: Positive/negative definite, semidefinite, and indefinite matrices (from J. Kim's slides *Solving Poisson equation using conjugate gradient method*).

As an application of indefinite QP, we introduce a QP formulation of the MAX CLIQUE problem (see Sect. 2.4.6) on a graph $G = (V, E)$, usually formulated as a BLP as:

$$\left.\begin{array}{lrcl} \max & \sum\limits_{i \in V} x_i & & \\ \forall \{i,j\} \notin E & x_i + x_j & \leq & 1 \\ & x & \in & \{0,1\}^n \end{array}\right\} \tag{2.27}$$

The reformulation below, to a *continuous* QP, is also valid:[44]

$$\left.\begin{array}{lrcl} \max & \frac{1}{2} x^\top A_G x & & \\ & \sum\limits_{i=1}^{n} x_i & = & 1 \\ & x & \geq & 0, \end{array}\right\} \tag{2.28}$$

where A_G is the adjacency matrix[45] of G.

By adding a penalty term to the objective function of Eq. (2.28), a bijection was obtained [Bomze et al., 1998, Thm. 2.6] between solutions of $\max\limits_{x \in \triangle^{n-1}} x^\top (A_G + I) x$ and the scaled characteristic vectors[46] of all the cliques in G. More precisely, the following statements hold:[47]

1. S is a maximal clique of G if and only if x_S is a local optimum of Eq. (2.28);
2. S is a maximum clique of G if and only if x_S is a global optimum of Eq. (2.28).

From this it also follows[48] that all local (and hence global) optima of Eq. (2.28) are strict and have the form x_S for some maximal clique $S \subseteq V$.

Negative semidefinite and indefinite QPs can be solved[49] to local optimality using any local NLP solver, such as IPOPT or SNOPT, and to global optimality using an sBB algorithm implementation such as BARON or COUENNE.

2.9.3 *Binary quadratic programming*

BINARY QUADRATIC PROGRAMMING (BQP) consists in the minimization of any quadratic form subject to a polyhedron. The standard form is

$$\left.\begin{array}{lrcl} \min & x^\top Q x & + & c^\top x \\ & Ax & \leq & b \\ & x & \in & \{0,1\}^n \end{array}\right\} \tag{2.29}$$

where $c \in \mathbb{R}^n$, Q is an $n \times n$ matrix, $b \in \mathbb{R}^m$ and A is an $m \times n$ matrix.

The following is a classic application of BQP.

MAX CUT. Given a simple undirected graph $G = (V, E)$ find a subset $S \subseteq V$ (called a *cut*) yielding the maximum number of edges $\{u, v\}$ with $u \in S$ and $v \notin S$.

The MAX CUT problem arises in electrical circuit layout, as well as in statistical physics.[50] It has the following formulation (also see

[44] This is known as the *Motzkin-Straus theorem*, see [Vavasis, 1991, p. 82-83] for a proof that a global optimum x^* has the form $\mathbf{1}/k$, where k is the size of a maximum clique. Vavasis proceeds by induction on $n = |V|$: start the induction at $|V| = n = 2$, then handle three cases: (i) that $\exists i \leq n\ x_i^* = 0$, and argue you can remove vertex i then use induction; (ii) that G is not complete but $x^* > 0$, and argue that from a missing edge $\{i, j\}$ you can make $x_i^* = 0$ at the expense of x_j^*, thereby moving back to case (i); (iii) that G is complete, in which case $x^* = \mathbf{1}/n$.

[45] The *adjacency matrix* of a graph is a binary square symmetric matrix which has a 1 in entry (i, j) if and only if $\{i, j\} \in E$. The feasible set

$$\{x \in \mathbb{R}^n \mid \mathbf{1}^\top x = 1 \wedge x \geq 0\}$$

is known as the *standard unit simplex* and denoted by $\triangle^{n-1}$.

[46] The *characteristic vector* of a subset $S \subseteq V$ is a vector x in $\{0,1\}^{|V|}$ with $x_i = 1$ if and only if $i \in S$. The *scaled* characteristic vector is $\frac{1}{|S|} x$.

[47] This theorem, proved in [Bomze, 1997], extends the Motzkin-Straus result [Motzkin and Straus, 1965].

[48] Why does this follow? Prove it.

[49] Whenever solutions can be irrational, such as in this case, we really mean "approximated to arbitrary accuracy".

[50] See Ch. 5 and search for the terms "VLSI" and "Ising model" in connection with "Max Cut" on Google.

Sect. 5.3):

$$\left.\begin{array}{lrcl} \max & \frac{1}{2}\sum\limits_{i<j\leq n}(1 & - & x_i x_j) \\ & x & \in & \{-1,1\}^n, \end{array}\right\} \tag{2.30}$$

where $n = |V|$. We assign a decision variable x_i to each vertex $i \in V$, and we let $x_i = 1$ if $i \in S$, and $x_i = -1$ if $i \notin S$. If $\{i,j\}$ is an edge such that $i,j \in S$, then $x_i x_j = 1$ and the corresponding term on the objective function is zero. Likewise, if $i,j \notin S$, then $x_i x_j = 1$ and, again, the objective term is zero. The only positive contribution to the objective function arises if $x_i x_j = -1$, which happens when $i \in S, j \notin S$ or vice versa, in which case the contribution of the term is 2 (hence we divide the objective by 2 to count the number of edges in the cut). Formulation (2.30) is not quite a BQP yet, since the BQP prescribes $x \in \{0,1\}^n$. However, an affine transformation takes care of this detail.[51]

BQPs are usually *linearized*[52] before solving them. Specifically, they are transformed to (larger) BLPs, and a MILP solver (such as CPLEX) is then employed [Liberti, 2007].

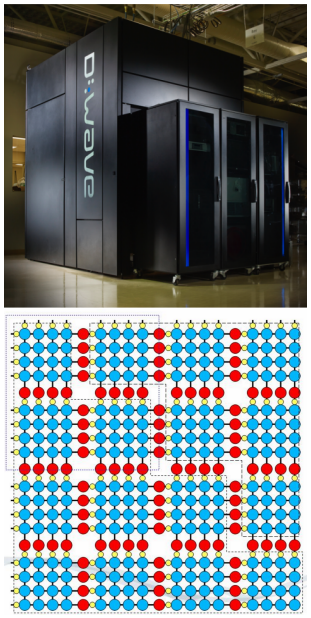


Figure 2.22: D-Wave's claimed "quantum computer" (from dwavesys.com), above. I sat at a few meetings where IBM researchers de-bunked D-Wave's claims, by solving Ising Model BQPs (see Ch. 5) defined on *chimera graphs* (below) on standard laptops *faster* than on D-Wave's massive hardware. Read Scott Aaronson's blog entry scottaaronson.com/blog/?p=1400 (it is written in inverse chronological order, so start at the bottom). By the way, I find Prof. Aaronson's insight, writing and teaching style absolutely wonderful and highly entertaining, so I encourage you to read *anything* he writes.

2.9.4 *Quadratically constrained quadratic programming*

Quadratically Constrained Quadratic Programming (QCQP) consists in the minimization of any quadratic form subject to a set of quadratic constraints. The standard form is

$$\left.\begin{array}{lrcl} \min & x^\top Q^0 x + c^0 x & & \\ \forall i \leq m & x^\top Q^i x + c^i x & \leq & b, \end{array}\right\} \tag{2.31}$$

where $c^0, \ldots, c^m$ are row vectors in $\in \mathbb{R}^n$, $Q^0, \ldots, Q^m$ are $n \times n$ matrices, and $b \in \mathbb{R}^m$.

The EDGP (see Sect. 2.6) can be written as a feasibility QCQP by simply squaring both sides of Eq. (2.21).[53] Solving this QCQP, however, is empirically very hard:[54] there are better formulations for the EDGP.[55]

Convex QCQPs can be solved to global optimality by local NLP solvers such as IPOPT and SNOPT. For nonconvex QCQPs, global solutions can be approximated to desired accuracy using the sBB algorithm (e.g. BARON or Couenne).

2.10 *Semidefinite programming*

Semidefinite Programming (SDP) consists in the minimization of a linear form subject to a semidefinite cone. The standard form is

$$\left.\begin{array}{lrcl} \min & C \bullet X & & \\ \forall i \leq m & A^i \bullet X & = & b^i \\ & X & \succeq & 0, \end{array}\right\} \tag{2.32}$$

where C and A^i (for $i \leq m$) are $n \times n$ symmetric matrices, X is an $n \times n$ matrix of decision variables, $b^i \in \mathbb{R}$ for all $i \leq m$, and for two $n \times n$ matrices $L = (\lambda_{ij})$, $M = (\mu_{ij})$ we have $L \bullet M = \sum\limits_{i,j \leq n} \lambda_{ij}\mu_{ij}$.

[51] Write down the affine transformation and the transformed formulation.

[52] This means that each term $x_i x_j$ is replaced by an additional variable y_{ij}; usually, additional linear constraints are also added to the formulation to link y and x variables in a meaningful way.

[53] The standard form can be achieved by replacing each quadratic equation by two inequalities of opposite sign.

[54] Try solving some small EDGP instances using Couenne. What kind of size can you achieve?

[55] Write and test some formulations for the EDGP.

To make Eq. (2.32) clearer, write out the componentwise product $\bullet$ of the matrices $C = (c_{jh})$, $A^i = (a^i_{jh})$ and $X = (x_{jh})$:

$$\begin{aligned} \min & \sum_{j,h \leq n} c_{jh} x_{jh} \\ \forall i \leq m \quad & \sum_{j,h \leq n} a^i_{jh} x_{jh} \quad = \quad b^i. \end{aligned}$$

This is just an LP subject to a semidefinite constraint $X \succeq 0$.

What does $X \succeq 0$ really mean? By the definition of PSD matrices, this is the same as requiring the decision variables x_{jh} to take values such that, when they are arranged in a square $n \times n$ array, the resulting matrix has non-negative eigenvalues.[56]

A basic result in linear algebra states that PSD matrices have (real) factorizations,[57] namely if $X \succeq 0$ then $X = Y^\top Y$ for some $k \times n$ where $k \leq n$. Now suppose we had a quadratic form $f(y) = y^\top Q y$ for some vector $y = (y_1, \ldots, y_n)$. Then

$$f(y) = \sum_{j,h \leq n} q_{jh} y_j y_h,$$

where $Q = (q_{jh})$, and suppose $f(x)$ appeared in some QP. We could then linearize the products $y_j y_h$ by replacing them with additional variables x_{jh}, and end up with the linear form

$$f(X) = \sum_{j,h \leq n} q_{jh} x_{jh} = Q \bullet X,$$

where $X = (x_{jh})$, and try to solve an easier LP resulting by replacing $f(y)$ with $f(X)$.

However, not all solutions X to this LP might have the property that there exists a vector y such that $y^\top y = X$, which we would need to retrieve a solution y to the original QP. On the other hand, if $X \succeq 0$ *and* $\mathsf{rk}\ X = 1$, then the existence of such a y would surely follow by the existence of the factorization $X = Y^\top Y$, since $\mathsf{rk}\ X = 1$ also implies $\mathsf{rk}\ Y = 1$, i.e. Y is a one-dimensional vector y.

In summary, provided we were able to find X having rank one, then the PSD constraint $X \succeq 0$ would automatically guarantee that for any solution of $f(X)$ we could retrieve y such that $f(y) = f(X)$. This does not really help in practice, since the *rank constraint* $\mathsf{rk}\ X = 1$ is hard to handle. But it does suggest that PSD constraints might be useful in relaxing QPs to LPs subject to $X \succeq 0$. Or, in other words, that y might be a $k \times n$ matrix instead of a vector (i.e. a $1 \times n$ matrix).

The above discussion suggests we consider MPs with quadratic forms where the decision variables are naturally arranged in a $K \times n$ array, such as for example the EDGP (see Sect. 2.6 and Sect. 2.9.4): a realization consists on vectors $y_1, \ldots, y_n$ in $\mathbb{R}^K$, which we arrange as columns of a $K \times n$ matrix Y. We rewrite Eq. (2.21) as follows:

$$\forall \{u, v\} \in E \quad y_u^\top y_u - 2 y_u^\top y_v + y_v^\top y_v = d^2_{uv}, \tag{2.33}$$

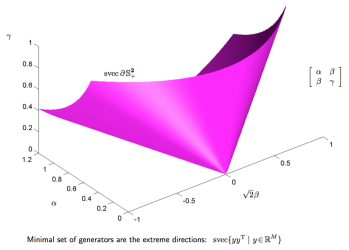


Figure 2.23: This picture of an SDP cone was borrowed from one of the most fascinating books I know [Dattorro, 2015]. This book has been growing since the early 2000s (and counting): part book and part encyclopedia, if you are looking for something in convex optimization or Euclidean geometry, this is almost sure to have it!

[56] See the lecture notes [Laurent and Vallentin, 2012] for more information about SDP.

[57] Equivalently: any PSD matrix is the *Gram matrix* of a $k \times n$ matrix Y (the converse also holds).

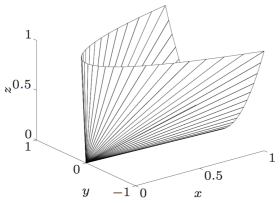


Figure 2.24: Another picture of the same cone as in Fig. 2.23 (from L. Vandenberghe's slides on *Convex optimization*).

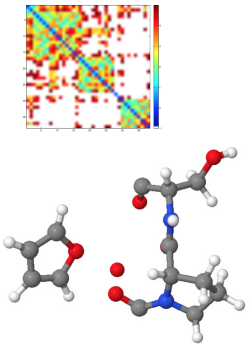


Figure 2.25: An EDGP instance (named `tiny`) from protein conformation: given some inter-atomic distances, find the positions of the atoms. Above: color map of distance matrix; below: the actual shape of the molecule.

where $G = (V, E)$ is a simple undirected graph, and $d : E \to \mathbb{R}_+$. Consider a square $n \times n$ symmetric matrix $X = (X_{uv})$: we linearize Eq. (2.33) by replacing $y_u^\top y_v$ by X_{uv}, and obtain the linear constraints

$$\forall \{u, v\} \in E \quad X_{uu} - 2X_{uv} + X_{vv} = d_{uv}^2 \tag{2.34}$$

in X.

We would now like to get as close as possible to state that there exists a $K \times n$ matrix Y such that $X - Y^\top Y = 0$. One possible way to get "as close as possible" is to replace $X - Y^\top Y = 0$ by $X - Y^\top Y \succeq 0$, which can also be trivially re-written as $X - Y^\top I_n^{-1} Y \succeq 0$. The left hand side in this equation is known as the *Schur complement* of X in the matrix

$$\mathcal{S} = \begin{pmatrix} I_n & Y \\ Y^\top & X \end{pmatrix},$$

and $X - Y^\top I_n^{-1} Y \succeq 0$ is equivalent to setting $\mathcal{S} \succeq 0$. We thus derived the SDP relaxation of the EDGP system Eq. (2.33):

$$\left.\begin{array}{rcl} \forall \{u, v\} \in E \quad X_{uu} - 2X_{uv} + X_{vv} & = & d_{uv}^2 \\ \begin{pmatrix} I_n & Y \\ Y^\top & X \end{pmatrix} & \succeq & 0. \end{array}\right\} \tag{2.35}$$

Many variants of this SDP relaxation for the EDGP were proposed in the literature, see e.g. [Man-Cho So and Ye, 2007, Alfakih et al., 1999].

SDPs are solved in polynomial time (to desired accuracy) by means of interior point methods (IPM). There are both commercial and free SDP solvers, see e.g. MOSEK [mosek7] and SeDuMi. Natively, they require a lot of work to interface to. Their most convenient interfaces are Python and the YALMIP [Löfberg, 2004] MATLAB [matlab] package. The observed complexity of most SDP solvers appears to be around $O(n^3)$, so SDP solver technology cannot yet scale to very large problem sizes. For many problems, however, SDP relaxations seem to be very tight, and help find approximate solutions which are "close to being feasible" in the original problem.

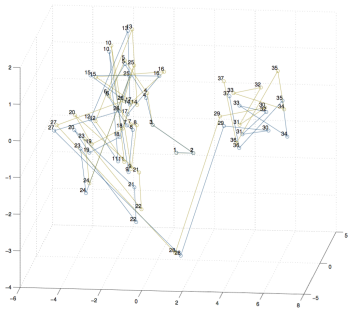


Figure 2.26: A solution to the instance in Fig. 2.25 obtained by the SDP solver MOSEK [mosek7], next to the actual shape: as you can see, they are very close, which means that the relaxation is almost precise.

2.10.1 *Second-order cone programming*

Second-Order Cone Programming (SOCP) consists in the minimization of a linear form subject to a second-order conic set. The standard form is

$$\left.\begin{array}{rcl} \min \quad \sum_{i \le r} c^i x^i & & \\ \sum_{i \le r} A^i x^i & = & b \\ \forall i \le r \qquad x_0^i & \ge & \sqrt{\sum_{j=1}^{n_i} (x_j^i)^2}, \end{array}\right\} \tag{2.36}$$

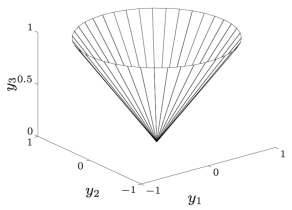


Figure 2.27: A picture of the SOCP cone $y_3 \ge \sqrt{y_1^2 + y_2^2}$, also known as *ice cream cone*, from L. Vandenberghe's slides on *Convex optimization*.

where, for each $i \leq r$, c^i is a row vector in $\mathbb{R}^{n_i}$, A^i is an $m \times n_i$ matrix, $b \in \mathbb{R}^m$, $n_1, \ldots, n_r \in \mathbb{N}$, and the decision variable vector $x \in \mathbb{R}^{\sum_i n_i}$ is partitioned into r blocks, each of size n_i (for $i \leq r$).

SOCP is a special case of SDP for which there exist more specialized solvers. Many[58] convex problems can be cast in this form after suitable (and often non-trivial) transformations are applied. See [Alizadeh and Goldfarb, 2003] for more information.

[58] For example, a convex quadratic constraint $x^\top Ax + 2b^\top x + c \leq 0$ can be transformed to

$$\|L^\top x + L^{-1}b\|_2 \leq \sqrt{b^\top A^{-1}b - c}$$

after factoring $A = LL^\top$ (why?). Also, the apparently nonconvex constraints $x^\top x \leq yz$ with $y, z \geq 0$ is actually convex, as it can be re-written as the SOCP constraint

$$y + z \geq \sqrt{4x^2 + (y - z)^2}$$

(why?).

2.11 *Multi-objective programming*

Multi-Objective Programming (MOP) consists in the identification of the Pareto[59] region (see Fig. 2.28) of many objective functions subject to a feasible set. The standard form is

$$\left.\begin{array}{rrcl} \min & f_1(x) & & \\ \vdots & \vdots & & \\ \min & f_q(x) & & \\ & g(x) & \leq & 0 \\ & x & \in & X, \end{array}\right\} \tag{2.37}$$

where $f_1, \ldots, f_q : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^m$, and there is an integer $0 \leq q \leq n$ such that $X \subseteq \mathbb{Z}^q \times \mathbb{R}^{n-q}$.

The motivation for studying MOPs is that many social or business problems have to do with some human decision maker who can authoritatively decide, after seeing the cost of an optimal solution, to accept to relax some constraints in order to further decrease the cost. In particular, decision makers often want to see what is the trade-off between a cost and some other goal.

Any constrained problem can be turned to a MOP: take e.g.

$$\min\{f(x) \mid g(x) \leq 0, \}$$

replace the zero in the right hand side by a scalar λ, and then solve

$$\min\{(f(x), \lambda) \mid g(x) \leq \lambda\}.$$

Conversely, many practical solution techniques for MOP are based on replacing all but the i-th objective $(\min f_p(x) \mid p \neq i)$ by constraints $f_p(x) \leq \lambda_p$ for $p \neq i$, then fixing $(\lambda_p \mid p \neq i)$ at some iteratively decreasing λ'. For each given λ', one solves a single-objective problem

$$\min\{f_i(x) \mid \forall p \neq i \; f_p(x) \leq \lambda_p \wedge g(x) \leq 0 \wedge x \in X\} \quad (\star)$$

in order to find minimal values for λ' as $(\star)$ goes from feasible to infeasible. For bi-objective problems, plotting the pairs $(f_i(x^*), \lambda')$ yields an approximation of the Pareto set (see Fig. 2.28).

No general-purpose established solvers currently exist for MOPs: in a certain sense, the problem is too difficult for general-purpose approaches, since the solution is not actually one point, as in single-objective problems, but a possibly infinite or even uncountable Pareto set. See [Ehrgott, 2005] for more information.

[59] A *Pareto optimum* is a solution x^* to Eq. (2.37) such that no other solution x' exists which can improve one objective without worsening another. The *Pareto set* (or Pareto region) is the set of Pareto optima.

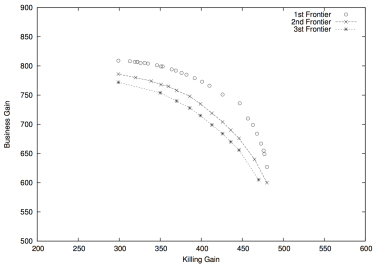


Figure 2.28: The objective function values of three Pareto sets of a MILP (based on varying one of the parameters) for deciding what products should be discontinued (or "killed"). Both killing off products and selling products can yield monetary returns (the former through cutting costs), but at the expense of each other [Giakoumakis et al., 2012].

2.11.1 *Maximum flow with minimum capacity*

We model the following problem:

> an oil company intends to maximize the flow of oil on their distribution network $G = (V, A)$ from the production node s to the customer node t, whilst minimizing the total cost of installing pipe capacities C_{ij} for each arc $(i, j) \in A$.

We suppose that arc capacities have a unit cost, i.e. the total cost of installing arc capacities is

$$\sum_{(i,j)\in A} C_{ij}. \tag{2.38}$$

If we had not been told to minimize the total installed capacity, this would be a standard MAX FLOW problem (see Sect. 1.7.2):

$$\left.\begin{array}{rrcl} \max\limits_{x\geq 0} & \sum\limits_{(s,j)\in A} x_{sj} & & \\ & \sum\limits_{(i,s)\in A} x_{is} & = & 0 \\ \forall i \in V \smallsetminus \{s,t\} & \sum\limits_{(i,j)\in A} x_{ij} & = & \sum\limits_{(j,i)\in A} x_{ji} \\ \forall (i,j) \in A & x_{ij} & \leq & C_{ij}. \end{array}\right\}$$

Now all we need to do is to consider the symbols C as decision variables instead of parameters, and minimize Eq. (2.38) as a second objective.

$$\left.\begin{array}{rrcl} \max & \sum\limits_{(s,j)\in A} x_{sj} & & \\ \min & \sum\limits_{(i,j)\in A} C_{ij} & & \\ & \sum\limits_{(i,s)\in A} x_{is} & = & 0 \\ \forall i \in V \smallsetminus \{s,t\} & \sum\limits_{(i,j)\in A} x_{ij} & = & \sum\limits_{(j,i)\in A} x_{ji} \\ \forall (i,j) \in A & x_{ij} & \leq & C_{ij} \\ \forall (i,j) \in A & x_{ij} & \geq & 0 \\ \forall (i,j) \in A & C_{ij} & \geq & 0. \end{array}\right\} \tag{2.39}$$

In order to find the Pareto set of this problem, we must change one of the two objectives, e.g. the total capacity installation cost Eq. (2.38), into a constraint:

$$\sum_{(i,j)\in A} C_{ij} \leq \lambda,$$

where λ ranges over a given interval, e.g. $[0, M|A|]$ (where M is the maximum capacity any arc can have installed).

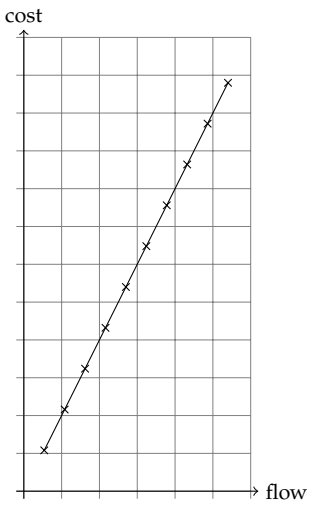


Figure 2.29: Objective values of the Pareto set of the bi-objective network flow problem. The straight line arises by the fact that the two objective functions force the constraint $x_{ij} \leq C_{ij}$ to be active on all arcs (s, j), thereby making $x_{ij} = C_{ij}$ on those arcs.

2.12 *Bilevel programming*

BILEVEL PROGRAMMING (BLevP) consists in the optimization[60] of an objective function subject to some decision variables representing the optimum of another MP formulation.[61] The standard form is:

[60] Called the *upper-level problem*.

[61] Called the *lower-level problem*.

$$\left.\begin{array}{lrcl} \min\limits_{x,y} & f(x,y) & & \\ & g(x,y) & \leq & 0 \\ & y & \in & \arg\min\{\phi_x(y) \mid \gamma_x(y) \leq 0 \wedge y \in \zeta\} \\ & x & \in & X, \end{array}\right\} \tag{2.40}$$

where x is a sequence of n decision variables, y is a sequence of p decision variables, $f : \mathbb{R}^{n+p} \to \mathbb{R}$, $g : \mathbb{R}^{n+p} \to \mathbb{R}^m$, $\phi_x : \mathbb{R}^q \to \mathbb{R}$, $\gamma_x : \mathbb{R}^q \to \mathbb{R}^\ell$, and there are integers $0 \leq s \leq n$ and $0 \leq t \leq p$ such that $\zeta \subseteq \mathbb{Z}^t \times \mathbb{R}^{p-t}$ and $X \subseteq \mathbb{Z}^s \times \mathbb{R}^{n-s}$. Note that ϕ_x, γ_x are functions which are parametrized by the decision variables x of the upper-level problem.

The motivation[62] for bilevel programs comes from studying how optimization is carried out when multiple stakeholders, ordered hierarchically, make decisions to optimize different goals. As in the case of MOP, solutions in bilevel programming are not necessarily well defined, as they may depend on whether lower-level decision makers take decisions favorably (the case of a division of a larger firm) or adversarially (competing entities ordered hierarchically). Accordingly, there are no general-purpose solvers — some assumptions must be made on the structure of the problem.[63] See [Colson et al., 2005] for more information.

[62] Typical application: a firm wants to plan operations (upper-level) subject to a certain unit revenue based on prices which are decided (lower-level) by negotiations with customers.

[63] A promising solution approach, in the case the lower-level problem is a MILP, consists in approximating the lower level problem by means of a polytope where each face is generated dynamically and added as a constraint to the upper-level problem.

2.12.1 *Lower-level LP*

Certain bilevel formulations can be transformed to single-level. Assume that:

- the objective $f(x,y)$ is only a function of x;
- among the constraints $g(x,y) \leq 0$ only one is actually a function of both x and y, specifically

  $$\min_y c(x) \cdot y \geq \alpha,$$

  where $c(x)$ is an n-vector of functions of x and α is a parameter;
- the lower-level problem has the form:

  $$\left.\begin{array}{lrcl} \min\limits_{y \geq 0} & c(x) \cdot y & & \\ & A(x) \cdot y & = & b(x), \end{array}\right\} \tag{2.41}$$

  where $A(x)$ is an $m \times n$ matrix having components in function of x, and b is an m-vector of functions of x.

Remark that, when x is fixed, Eq. (2.41) is an LP.

Under these conditions, Eq. (2.40) can be formulated as a single-level problem by replacing the lower-level LP with its dual. For x fixed, the dual of Eq. (2.41) is:

$$\left.\begin{array}{lrcl} \max\limits_{\lambda} & \lambda \cdot b(x) & & \\ & \lambda \cdot A(x) & \leq & c(x), \end{array}\right\} \tag{2.42}$$

By LP duality we can replace Eq. (2.41) by Eq. (2.42), and obtain:

$$\max_{\lambda} \lambda \cdot b(x) \geq \alpha. \tag{2.43}$$

Now stop and consider Eq. (2.43) within the context of the upper-level problem: it is telling us that the maximum of a certain function must be at least α. So if we find *any* λ which satisfies $\lambda \cdot b(x) \geq \alpha$, the maximum will certainly also be $\geq \alpha$. This means we can forget the $\max_\lambda$. The problem can therefore be written as:

$$\left.\begin{array}{rrcl} \min\limits_{x,\lambda} & f(x) & & \\ & g(x) & \leq & 0 \\ & \lambda \cdot b(x) & \geq & \alpha \\ & \lambda \cdot A(x) & \leq & c(x) \\ & x & \in & X, \end{array}\right\} \tag{2.44}$$

which is a single-level MP.

2.12.2 *Minimum capacity maximum flow*

Let us re-examine the example in Sect. (2.11.1) from the following bilevel point of view: the higher-level stakeholders (say, the Chief Financial Officer)[64] aims at reducing costs over all operations while making sure that each client receives a share of the flow by upper-bounding it by a threshold α, whereas a specific client manager wants to keep his/her own client happy and maximize the flow to that client.

The formulation is as follows:[65]

$$\left.\begin{array}{rrcl} \min\limits_{C \geq 0} & \sum\limits_{(i,j)\in A} C_{ij} & & \\ & F(C) & \leq & \alpha \\ & F(C) & = & \max\limits_{x \geq 0} \sum\limits_{(s,j)\in A} x_{sj} \\ & & & \sum\limits_{(i,s)\in A} x_{is} = 0 \\ & & & \forall i \in V \setminus \{s,t\} \quad \sum\limits_{(i,j)\in A} x_{ij} = \sum\limits_{(j,i)\in A} x_{ji} \\ & & & \forall (i,j) \in A \quad x_{ij} \leq C_{ij}. \end{array}\right\}$$

[64] Why the heck do CFOs always aim at making the shareholders richer? How about providing good service to the individual customer? How about making employees happier (and hence more productive)? Capitalism took a very nasty turn in the last, say twenty years *from this point of view* (from many other points of view the world is a better place now than in the 1960s-1980s). For this and other rants of mine, read kafkatoday.wordpress.com.

[65] Can this formulation be made single-level? If so, use AMPL or Python to solve it. If not, explain why.

2.13 *Semi-infinite programming*

Semi-Infinite Programming (SIP) consists in the minimization of an objective function subject to an infinite number of constraints. The standard form is

$$\left.\begin{array}{rrcl} \min & f(x) & & \\ \forall \sigma \in S & g_\sigma(x) & \leq & 0 \\ \forall \beta \in T & x_\beta & \in & X_\beta, \end{array}\right\} \tag{2.45}$$

where $f : \mathbb{R}^n \to \mathbb{R}$, for all $\alpha \in S$ we have $g_\sigma : \mathbb{R}^n \to \mathbb{R}$, for all $\beta \in T$, X_β is either $\mathbb{Z}$ or $\mathbb{R}$, and either S or T is an infinite or uncountable set of indices.

SIPs are mostly studied in the context of optimal control problems, where T is uncountable, as well as in robust optimization, where S is infinite or uncountable. See [López and Still, 2007] for more details.

Optimal control problems have both decision variables and state variables. One can interpret the decision variables[66] as parameters to a simulation based on the ODE/PDE constraints, which are themselves expressed in terms of decision and state variables, of which there are usually uncountably many. These problems are usually cast in terms of minimizing an integral (which might express the length of a path, or the total amount of energy consumed by a process) subject to differential equation constraints. Optimal control problems are rarely so simple that they can be solved analytically. Numerical solution methods usually call for differential equations to be replaced by finite differences equations, and for integrals to be replaced by sums. See [Evans, v. 0.2] for more information.

[66] Also known as *control variables* in this setting.

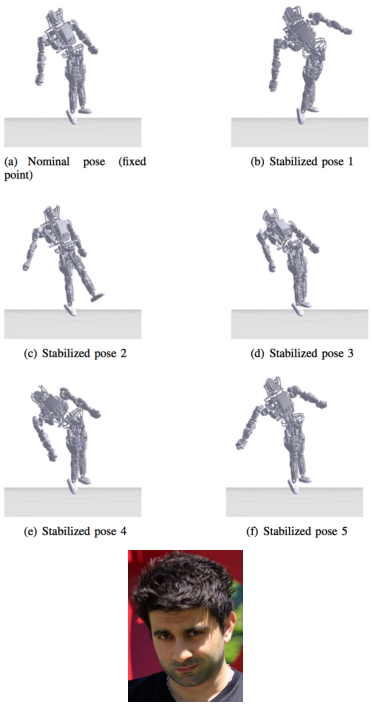


Figure 2.30: Typical application of optimal control: balancing a robot whilst it walks [Majumdar et al., 2014]. Honestly, at first sight the robot looks like it's falling. I blame the middle author (shown below the picture) ☺.

Robust optimization is a set of techniques for making sure the solutions of a MP will hold for uniform uncertainty in certain parameters. Suppose a MP formulation has the constraint $g(p, x) \leq 0$ where p is a parameter vector and x a decision variable vector. A robust counterpart to this constraint is:

$$\forall p \in [p^L, p^U]\ g(p, x) \leq 0,$$

where p is uniformly distributed in $[p^L, p^U]$. This type of problems can rarely be solved in general form. For certain type of constraints, however, ingenious solution techniques have been devised. If g is a monotonic function, for example, it suffices to impose the constraint at the interval endpoints, which reduces the uncountable set $[p^L, p^U]$ to the finite set $\{p^L, p^U\}$. Possibly the best known paper in this field is [Bertsimas and Sim, 2004].

2.13.1 *LP under uncertainty*

Consider the MP

$$\sup_x \{c^\top x \mid \forall a_j \in K_j \sum_{j \leq n} a_j x_j \leq b\}, \tag{2.46}$$

where $x \in \mathbb{R}^n$ is an n-vector of decision variables, $K = (K_j \mid j \leq n)$ is a sequence of convex subsets of $\mathbb{R}^m$, and $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$ are parameter vectors. Note that the convex sets K_j need to be encoded in a finite way (e.g., if each K_j is a polytope, as a list of vertices or facets). Note that for each $j \leq n$, K_j might be uncountable. So this is a SIP with uncountably many constraints.

Consider now the LP

$$\max_{\substack{Ax \leq b \\ x \geq 0}} c^\top x,$$

where A is an $m \times n$ matrix having a_j as its j-th column. Eq. (2.46) is a good representation of this LP whenever a_j are subject to uncertainty in the convex set K_j. It is shown in [Soyster, 1973] that

Eq. (2.46) is equivalent to the LP

$$\max\{c^\top x \mid \bar{A}x \leq b \wedge x \geq 0\}, \tag{2.47}$$

where $\bar{A} = (\bar{a}_{ij})$ is an $m \times n$ matrix such that:

$$\forall j \leq n, i \leq m \quad \bar{a}_{ij} = \sup\{a_{ij} \mid a_j \in K_j\}. \tag{2.48}$$

Hence, in order to solve Eq. (2.46), it suffices to solve the mn convex problems in Eq. (2.48), then construct the matrix $\bar{A}$ with their solutions, and finally solve Eq. (2.47).

The proof establishes an equivalence between feasible solutions of Eq. (2.47) and Eq. (2.46). Consider first x feasible in Eq. (2.47), i.e. $\bar{A}x \leq b$. By construction of $\bar{A}$, we have $A \leq \bar{A}$ for all $A \in \prod_{j \leq n} K_j$, implying $Ax \leq \bar{A}x \leq b$ for each $x \geq 0$ and hence $Ax \leq b$, meaning x is also feasible in Eq. (2.46). Conversely, if x is a solution to Eq. (2.46), $\sum_{j \leq n} a_j x_j \leq b$ holds for any $a_j \in K_j$, which means it also holds for the supremum, implying $\bar{A}x \leq b$, as claimed. Now, since the feasible regions are the same, and the objective function is the same, the optima are also the same.

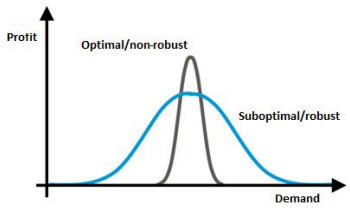


Figure 2.31: A convincing picture showing a higher profit solution traded off with a more robust but lower profit one (from www.eyeon.nl).

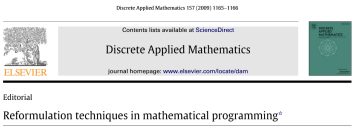
2.14 *Summary*

This chapter is a systematic study of some important MP formulation classes, with some of their properties, applications, solution methods and software packages.

- LINEAR PROGRAMMING (LP), methods, some applications
- MIXED-INTEGER LINEAR PROGRAMMING (MILP), complexity and some solution methods; some applications, the POWER GENERATION problem, the TRAVELING SALESMAN PROBLEM (TSP)
- INTEGER LINEAR PROGRAMMING (ILP), the SCHEDULING problem
- BINARY LINEAR PROGRAMMING (BLP), applications: assignment, vertex cover, set covering, set packing, set partitioning, stables and cliques, other combinatorial optimization problems
- CONVEX NONLINEAR PROGRAMMING (cNLP), complexity and some solution methods; applications: linear regression, function interpolation
- NONLINEAR PROGRAMMING (NLP), circle packing, kissing number problem, local and global solution methods
- CONVEX MIXED-INTEGER NONLINEAR PROGRAMMING (cMINLP), cardinality constrained portfolio selection, some methods
- MIXED-INTEGER NONLINEAR PROGRAMMING (MINLP), universality and some methods

- Quadratic formulations: CONVEX QUADRATIC PROGRAMMING (cQP) and restricted linear regression, (nonconvex) QUADRATIC PROGRAMMING (QP) and definite/indefinite matrices, Motzkin-Straus' clique formulation, BINARY QUADRATIC PROGRAMMING (BQP) and the MAX CUT problem, QUADRATICALLY CONSTRAINED QUADRATIC PROGRAMMING (QCQP) and Euclidean distance geometry
- SEMIDEFINITE PROGRAMMING (SDP), QP/QCQP relaxations, matrix factorization, some methods
- SECOND-ORDER CONE PROGRAMMING (SOCP)
- MULTI-OBJECTIVE PROGRAMMING (MOP) and trade-offs; some solution methods
- BILEVEL PROGRAMMING (BLevP) and hierarchical optimization
- SEMI-INFINITE PROGRAMMING (SIP) and control problems.

3
Reformulations


Discrete Applied Mathematics 157 (2009) 1985–1986

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

ELSEVIER

Editorial

Reformulation techniques in mathematical programming☆

Figure 3.1: The preface of the special issue 157 of *Discrete Applied Mathematics* dedicated to Reformulations in Mathematical Programming [Liberti and Maculan, 2009].

Reformulations are symbolic transformations in MP. We usually assume these transformations can be carried out in polynomial time. In Ch. 2 we listed many different types of MP. Some are easier to solve than others; specifically, some are motivated by a real-life application, and others by the existence of an efficient solver. Usually, real-life problems naturally yield MP classes which are **NP**-hard, while we are interested in defining the largest MP class solvable by each efficient (i.e. polytime) solver we have. Ideally, we would like to reformulate[1] a given formulation belonging to a "hard" class to an equivalent one in an "easy" class.

Hard problems will remain hard,[2] but certain formulations belong to a hard class simply because we did not formulate them in the best possible way.[3] This chapter is about transforming MPs into other MPs with desirable properties, or features. Such transformations are called *reformulations*. In this chapter, we list a few useful and elementary reformulations.

Informally speaking, a reformulation of a MP P is another MP Q, such that Q can be obtained from P in a shorter time than is needed to solve P, and such that the solution of Q is in some way useful to solving P. We list four main types of reformulation: exact, narrowing, relaxation, approximation. More information can be found in [Liberti, 2009, Liberti et al., 2009a].

[1] I do not mean that any **NP**-hard problem can just be reformulated in polytime to a problem in **P**, and no-one knows yet if this is possible or not: the $\mathbf{P} \neq \mathbf{NP}$ is the most important open problem in computer science.

[2] This is because if P is **NP**-hard and Q is obtained from P by means of a polynomial time reformulation, then by definition Q is also **NP**-hard.

[3] More precisely, **NP**-hard problems contain infinite subclasses of instances which constitute a problem in **P**, e.g. the class of all complete graphs is clearly a trivially easy subclass of Clique.

3.0.1 Definition

- *A reformulation is exact if there is an efficiently computable surjective*[4] *map $\phi : \mathcal{G}(Q) \to \mathcal{G}(P)$. Sometimes we require that ϕ also extends to local optima ($\mathcal{L}(Q) \to \mathcal{L}(P)$), or feasible solutions ($\mathcal{F}(Q) \to \mathcal{F}(P)$), or both.*
- *A reformulation is a narrowing if ϕ maps $\mathcal{G}(Q)$ to a nontrivial*[5] *subset of $\mathcal{G}(P)$. Again, we might want to extend ϕ to local optima or feasible solutions.*
- *A reformulation is a relaxation if the objective function value at the global optima of Q provides a bound to the objective function value at the global optima of P in the optimization direction.*
- *A reformulation is an approximation*[6] *if Q is a formulation of a se-*

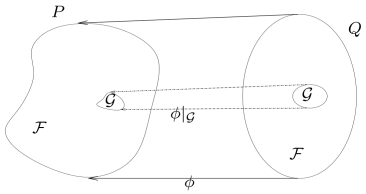


Figure 3.2: Exact reformulation.

[4] We require ϕ to be surjective so that we can retrieve all optima of P from the optima of Q.

[5] In narrowings, we are allowed to lose some optima of the original problem P; narrowings are mostly used in symmetric MPs, where we are only interested in one representative optimum per symmetric orbit.

[6] Note that, aside from the asymptotic behaviour, there may be no relation between Q and P; in other words, solving Q really only provides a heuristic method for solving P.

quence $Q_1, Q_2, \ldots$ such that $\lim_{t\to\infty} \mathcal{G}(Q_t)$ is an exact or narrowing reformulation of P.

3.1 Elementary reformulations

3.1.1 Objective function direction and constraint sense

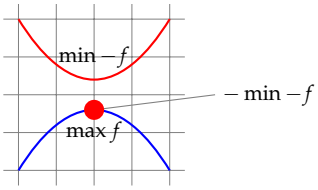


Figure 3.3: Objective function direction.

We list this for completeness, but we already mentioned that any problem in the form $\max_x f(x)$ can be written as $-\min_x(-f(x))$, and vice versa.[7] For the same reasons, any constraint in the form $g(x) \leq 0$ can be written as $-g(x) \geq 0$ (and vice versa.

[7] I often forget to take the negative of the optimal value after the optimization process. Pay attention!

3.1.2 Liftings, restrictions and projections

We define here three important classes of auxiliary problems: liftings, restrictions and projections.

- A *lifting* adds auxiliary variables to the original problem.
- A *restriction* replaces some of the variables by parameters.[8]
- A *projection* Q eliminates some variables from the original problem P in such a way that $\mathcal{F}(Q)$ is a projection[9] of $\mathcal{F}(P)$ on the subspace spanned by the variables of Q.

[8] In other words, some of the variables are *fixed*.

[9] This means that there is a mapping $\pi : \mathcal{F}(P) \to \mathcal{F}(Q)$ such that $\forall y \in \mathcal{F}(Q)$ there is $x \in \mathcal{F}(P)$ with $\pi(x) = y$.

3.1.3 Equations to inequalities

Any equality constraint $g(x) = 0$ can be replaced by pairs of inequalities[10] of opposite sign:

$$\begin{aligned} g(x) &\leq 0 \\ g(x) &\geq 0. \end{aligned}$$

[10] This can be done to conform to some problem form. For example, an LP in standard form can be reformulated to an LP in canonical form by applying this reformulation, and then by turning each constraint of the form $g(x) \geq 0$ to $-g(x) \leq 0$.

3.1.4 Inequalities to equations

An inequality $g(x) \leq 0$ in a problem P can be transformed to an equation by means of a lifting: we add an auxiliary variable to Q, called a *slack variable*[11] $s \geq 0$, constrained to be non-negative, and write $g(x) + s = 0$.

[11] Or *surplus variable*, depending on its sign restriction.

3.1.5 Absolute value terms

Consider a problem P with an absolute value term $|\alpha(x)|$. We can re-write P to a problem Q without the $|\cdot|$ operator as follows:

- add two[12] variables $t^+ \geq 0$, $t^- \geq 0$
- replace $|\alpha(x)|$ by the linear term $t^+ + t^-$

[12] If there are m absolute values in the MP, you have to carry out this reformulation m times, i.e. you have to add $2m$ additional variables.

- add the constraint $t^+ - t^- = a(x)$
- add the constraint $t^+ t^- = 0$.

The last constraint ensures that only one in t^+, t^- can be nonzero at any time, which means that if $a(x) < 0$ then $t^- = a(x)$ and if $a(x) > 0$ then $t^+ = a(x)$, and consequently that $t^+ + t^-$ has the same value as $|a(x)|$.

3.1.6 *Product of exponential terms*

We can replace a product of exponential terms[13] $\prod_i e^{f_i(x)}$ by an additional variable $w \geq 0$ and the constraint $\ln(w) = \sum_i f_i(x)$.

[13] Exponential terms like these are not so common in applications, but they may occasionally happen, e.g. when expressing probability of chains of events.

3.1.7 *Binary to continuous variables*

We can replace any binary variable $x \in \{0, 1\}$ to a continuous variable by removing the integrality constraints and adding the quadratic constraint[14] $x^2 = x$, a quadratic equation which only has $0, 1$ as solutions.

In principle, this would reduce all binary MPs to QCQPs. However, most NLP solvers do not handle the constraint $x^2 = x$ well, empirically speaking.[15] In practice, it is much better to deal with the binary constraint and use branching. One notable exception arises when the resulting QCQP is used to derive an SDP relaxation of the original BLP.

[14] This constraint is sometimes also written as $x(x-1) \geq 0 \wedge x \leq 1$ [Sutou and Dai, 2002].

[15] Many researchers have used this reformulation, but mostly for theoretical rather than practical reasons.

3.1.8 *Discrete to binary variables*

A discrete variable $x \in \{p_1, \ldots, p_k\}$ can be reformulated by relaxing it to a continuous variable,[16] then adding a vector $y = (y_1, \ldots, y_k)$ of k binary variables, as well as the constraints:

$$\begin{aligned} \sum_{h \leq k} y_h &= 1 \\ \sum_{h \leq k} p_h y_h &= x \end{aligned}$$

to the original problem.

[16] Essentially, this reformulation provides an assignment of the allowable values to the relaxed variable by means of assignment constraints.

3.1.9 *Integer to binary variables*

If $p_1, \ldots, p_k$ are consecutive integers $\{L, L+1, \ldots, L+k-1\}$, then we can write x by means of a binary encoding.[17] Let b be the smallest integer such that $k \leq 2^b$. Then add a vector $z = (z_1, \ldots, z_b)$ of b binary variables, relax x to a continuous variable, and add the constraints:

$$x = L + \sum_{h=0}^{b-1} z_h 2^h$$

to the original problem.

[17] Many researchers have used this, see e.g. [Boulle, 2004, Vielma, 2015].

3.1.10 *Feasibility to optimization problems*

WE CAN REFORMULATE a feasibility problem

$$P \equiv \{x \in \mathbb{R}^n \mid \forall i \leq m \; g(x) \leq 0\}$$

to the following optimization problem:[18]

$$\left.\begin{array}{rrcl} \min & \sum\limits_{i \leq m} y_i & & \\ \forall i \leq m & g(x) & \leq & y_i \\ & y & \geq & 0, \end{array}\right\} \tag{3.1}$$

where $y_1, \ldots, y_m$ are additional variables.

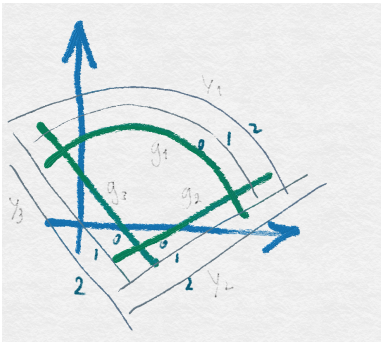


Figure 3.4: Feasibility to optimization: the feasible region $g_i(x) \leq 0$ ($i \leq 3$) is shown in dark, and the level sets for the slack variables y_1, y_2, y_3 are shown for values 1, 2.

[18] Why would anyone want to do this? Well, it so happens that most local NLP solvers are good at improving solutions which are already feasible, but terrible at finding a feasible solution from scratch. This is a way to make your solver's life easier.

3.1.11 *Minimization of the maximum of finitely many functions*

AN OBJECTIVE FUNCTION of the form[19]

$$\min \max_{i \leq n} f_i(x)$$

can be reformulated to

$$\begin{array}{rcl} \min t & & \\ \forall i \leq n \quad t & \geq & f_i(x), \end{array}$$

where t is an added (continuous) variable. The maximization of the minimum of finitely many functions can treated in a similar way.[20]

[19] Note that the first *min* operator is with respect to the decision variables x, but the second *max* is with respect to the finite set of functions $\{f_1, \ldots, f_n\}$. This is a necessary condition for applying this reformulation: if the inner optimization operator is also with respect to some decision variables, this reformulation cannot be applied.

[20] How can you reformulate the minimization of the minimum of finitely many functions, or the maximization of the maximum of finitely many functions?

3.2 *Exact linearizations*

THESE ARE REFORMULATIONS transforming various nonlinear terms to linear forms, which may involve additional mixed-integer variables.

3.2.1 *Product of binary variables*

CONSIDER A PRODUCT TERM $x_i x_j$ occurring in a MP P (for some variable indices i, j), where $x_i, x_j \in \{0, 1\}$. We can linearize this product exactly as follows:

- replace[21] $x_i x_j$ by an additional variable $y_{ij} \in [0, 1]$;
- add the following constraints:

$$y_{ij} \leq x_i \tag{3.2}$$
$$y_{ij} \leq x_j \tag{3.3}$$
$$y_{ij} \geq x_i + x_j - 1. \tag{3.4}$$

Constraints (3.2)-(3.4) are called Fortet's constraints [Fortet, 1960]. We have the following result.[22]

[21] This symbolic replacement is actually known as "linearization". More generally, a *linearization* replaces a whole nonlinear term by means of a single additional variable.

[22] Prove this proposition (hint: Fig. 3.5).

3.2.1 Proposition
For any i, j with

$$\begin{aligned} S_{ij} &= \{(x_i, x_j, y_{ij}) \in \{0,1\}^2 \times [0,1] \mid y_{ij} = x_i x_j\} \\ T_{ij} &= \{(x_i, x_j, y_{ij}) \in \{0,1\}^2 \times [0,1] \mid \textit{Eq. (3.2)-(3.4)}\} \end{aligned}$$

we have $S_{ij} = T_{ij}$.

This technique can be extended to terms of the form $y_I = \prod_{i \in I} x_i$ where $x_i \in \{0,1\}$ for each $i \in I$. The linearization constraints are:[23]

$$\forall i \leq I \quad y_I \leq x_i \tag{3.5}$$
$$y_I \geq \sum_{i \in I} x_i - |I| + 1. \tag{3.6}$$

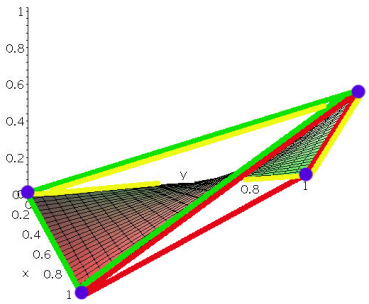


Figure 3.5: The reason why Fortet's reformulation works: the constraints form the *convex envelope* of the set $\{(x_i, x_j, y_{ij}) \mid y_{ij} = x_i x_j \wedge x_i, x_j \in \{0,1\}\}$, i.e. the smallest convex set containing the given set.

[23] Formulate and prove the generalization of Prop. 3.2.1 in the case of $|I|$ binary variables.

3.2.2 *Product of a binary variable by a bounded continuous variable*

We tackle terms of the form xy where $x \in \{0,1\}$, $y \in [L, U]$, and $L \leq U$ are given scalar parameters. It suffices to replace the term xy by an additional variable z, and then add the following linearization constraints:[24]

$$z \geq Lx \tag{3.7}$$
$$z \leq Ux \tag{3.8}$$
$$z \geq y - \max(|L|, |U|)(1 - x) \tag{3.9}$$
$$z \leq y + \max(|L|, |U|)(1 - x). \tag{3.10}$$

[24] Prove their correctness.

3.2.3 *Linear complementarity*

A *linear complementarity* constraint[25] has the form $x^\top y = 0$, where $x, y \in [0, M]^n$ are two distinct vectors in $\mathbb{R}^n$. Since $x, y \geq 0$ every term $x_j y_j$ in

$$\sum_{j \leq n} x_j y_j = 0$$

must be zero, i.e. the linear complementarity constraint is equivalent to $x_j y_j = 0$ for each $j \leq n$. This constraint actually expresses a combinatorial property, i.e. that at most one between x_j and y_j is nonzero. We can model this using binary variables $z_j \in \{0,1\}$ for each $j \leq n$, and adding the following constraints:[26]

$$x_j \leq Mz_j \tag{3.11}$$
$$y_j \leq M(1 - z_j). \tag{3.12}$$

Note that this reformulation provides an exact linearization for the reformulation in Sect. 3.1.5.[27]

[25] Linear complementarity constraints are at the basis of a standard MP formulation called Linear Complementarity Problem (LCP), consisting in an LP subject to a linear complementarity constraint. LCPs are **NP**-hard, since they can encode binary variables and hence SAT, but local optima can be obtained by means of local NLP solvers. If the LCP is a reformulation of a MILP, local optima may be feasible MILP solutions.

[26] Prove their correctness.

[27] Why?

3.2.4 *Minimization of absolute values*

Suppose the objective function of P is the minimization of a sum

$$f(x) = \cdots + |h(x)| + \cdots$$

This can be reformulated as in Sect. 3.1.5, and will result in

$$f(x) = \cdots + (t^+ + t^-) + \cdots$$

subject to the constraint $t^+ - t^- = h(x)$.

Since we are minimizing f, at least one of the global optima will have either $t^+ = 0$ or $t^- = 0$ even without the complementarity constraint $t^+t^- = 0$. Whether this optimum will be found by the solver[28] really depends on how the solver works. For example, if the problem reformulation is a MILP, and solutions at every node of the BB are obtained using the simplex method, which finds optima at vertices of the feasible polyhedron, this is going to be the case.[29]

We can also consider t^- as a surplus variable of $t^+ \geq h(x)$, and t^+ as a slack variable of $t^- \leq h(x)$, and re-write the above as

$$f(x) = \cdots + t^+ + t^- + \cdots$$

subject to

$$t^- \leq h(x) \leq t^+.$$

Finally, since $-|h(x)| \leq h(x) \leq |h(x)|$, we can simply replace t^- by $-t^+$ in the constraints (and rename t^+ as t for simplicity), to obtain

$$f(x) = \cdots + \quad t \quad + \cdots \tag{3.13}$$

$$-t \leq \quad h(x) \quad \leq t \tag{3.14}$$

$$t \quad \geq \quad 0, \tag{3.15}$$

which reduces the number of added variables by half. Note that t here represents $|h(x)|$, so we only need to sum it once on the objective function, i.e. we do not want to replace t^- in the objective, otherwise we would end up with $2t$, which would count $|h(x)|$ twice, and is therefore wrong. Note also that Eq. (3.13)-(3.15) can also be used when $|h(x)|$ appears in terms other than just the objective function: consider for example the case of diagonal dominance of an $n \times n$ matrix $A = (a_{ij})$ (see Sect. 6.6.2)

$$\forall i \leq n \quad a_{ii} \geq \sum_{j \neq i} |a_{ij}|.$$

We can reformulate this to:

$$\forall i \leq n \quad a_{ii} \quad \geq \quad \sum_{j \neq i} t_{ij} \tag{3.16}$$

$$\forall i, j \leq n \quad -t_{ij} \leq \quad a_{ij} \quad \leq t_{ij} \tag{3.17}$$

$$\forall i, j \leq n \quad t_{ij} \quad \geq \quad 0. \tag{3.18}$$

Suppose (a, t) are feasible in the reformulation ((3.16)-(3.18)): then we have

$$a_{ii} \geq \sum_{j \neq i} t_{ij} \geq \sum_{j \neq i} |a_{ij}|$$

as claimed, where the first inequality is Eq. (3.16), the second follows because Eq. (3.17) implies $t_{ij} \geq |a_{ij}|$.

[28] It would be more desirable to assert properties which hold for any solver, so as not to depend on the nature of the solver, which may vary greatly. On the other hand, getting rid of the problematic product term t^+t^- is a great benefit.

[29] Prove this.

3.2.5 *Linear fractional terms*

Consider a problem P where one of the terms is

$$\tau = \frac{a^\top x + b}{c^\top x + d},$$

where $a \in \mathbb{R}^n$, $b, d \in \mathbb{R}$ are parameters and x is a vector of n continuous decision variables which might otherwise only appear as $Ax = q$ elsewhere in the problem.

We linearize the linear fractional term by adding n variables α_j which play the role of $\frac{x_j}{c^\top x+d}$ and a further variable β which plays the role of $\frac{1}{c^\top x+d}$.

Then τ can be replaced by $a^\top \alpha + \beta b$, the system $Ax = q$ can be replaced by $A\alpha = \beta q$, and a new constraint

$$c^\top \alpha + \beta d = 1$$

can be added to the formulation.[30]

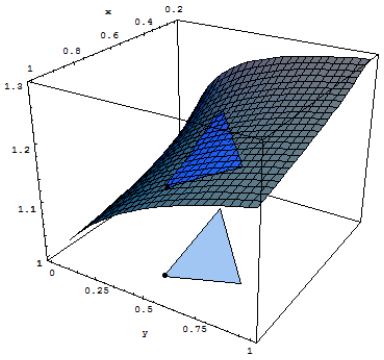


Figure 3.6: A small Linear Fractional Programming (LFP) instance. A typical application is to maximize *efficiency*: if a production line uses x_i units of product i and spends $C(x) = c_0 + \sum_i c_i x_i$ in order to make a profit $P(x) = p_0 + \sum_i p_i x_i$, then the objective function is $\max_x P(x)/C(x)$.

[30] Prove that, if the feasible set is bounded on the x variable and does not contain any x such that $c^\top x = -d$, then this reformulation is exact. Give examples of formulations where this reformulation is not exact.

3.3 *Advanced examples*

Let us see two advanced examples where we apply the reformulations given above.

3.3.1 *The Hyperplane Clustering Problem*

Consider the

> Hyperplane Clustering Problem (HCP) [Dhyani, 2007, Caporossi et al., 2004]. Given a set of points $p = \{p_i \mid 1 \leq i \leq m\}$ in $\mathbb{R}^d$, determine a set of n hyperplanes $w = \{w_{j1}\chi_1 + \ldots + w_{jd}\chi_d = w_j^0 \mid 1 \leq j \leq n\}$ in $\mathbb{R}^d$ and an assignment of points to hyperplanes such that the sum of the distances from the hyperplanes to their assigned points are minimized.

The problem can be modelled as follows:

- *Parameters*. The set of parameters is given by: $p \in \mathbb{R}^{m\times d}$, and m, n, $d \in \mathbb{N}$.
- *Variables*. We consider the hyperplane coefficient variables $w \in \mathbb{R}^{n\times d}$, the hyperplane constants $w^0 \in \mathbb{R}^n$, and the 0-1 assignment variables $x \in \{0,1\}^{m\times n}$.
- *Objective function*. We minimize the total distance, weighted by the assignment variable:

$$\min \sum_{i\leq m} \sum_{j\leq n} |w_j p_i - w_j^0| x_{ij}.$$

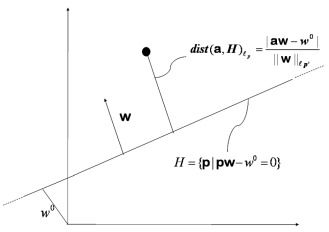


Figure 3.7: Hyperplane clustering [Dhyani, 2009].

- *Constraints*. We consider assignment constraints: each point must be assigned to exactly one hyperplane:

$$\forall i \leq m \quad \sum_{j \leq n} x_{ij} = 1,$$

and the hyperplanes must be nontrivial:

$$\forall j \leq n \quad \sum_{k \leq d} |w_{jk}| = 1$$

(since, otherwise, the trivial solution with $w = 0, w^0 = 0$ would be optimal).

This is a MINLP formulation because of the presence of the nonlinear terms (absolute values and products in the objective function) and of the binary assignment variables. We shall now apply several of the elementary reformulations presented in this chapter to obtain a MILP reformulation.

1. Because x is nonnegative and because we are going to solve the reformulated MILP to global optimality, we can linearize the minimization of absolute values as in Sect. 3.2.4:

$$\begin{aligned} \min \sum_{\substack{i \leq m \\ j \leq n}} (t^+_{ij} x_{ij} + t^-_{ij} x_{ij}) & \\ \forall i \leq m \quad \sum_{j \leq n} x_{ij} &= 1 \\ \forall j \leq n \quad |w_j|\mathbf{1} &= 1 \\ \forall i \leq m, j \leq n \quad t^+_{ij} - t^-_{ij} &= w_j p_i - w^0_j, \end{aligned}$$

where, for all $i \leq m$ and $j \leq n$, $t^+_{ij}, t^-_{ij} \in [0, M]$ are continuous additional variables bounded above by a large enough constant M.[31]

2. Reformulate the products $t^+_{ij} x_{ij}$ and $t^-_{ij} x_{ij}$ according to Sect. 3.2.2:

$$\begin{aligned} \min \sum_{\substack{i \leq m \\ j \leq n}} (y^+_{ij} + y^-_{ij}) & \\ \forall i \leq m \quad \sum_{j \leq n} x_{ij} &= 1 \\ \forall j \quad |w_j|\mathbf{1} &= 1 \\ \forall i \leq m, j \leq n \quad t^+_{ij} - t^-_{ij} &= w_j p_i - w^0_j \\ \forall i \leq m, j \leq n \quad y^+_{ij} &\leq \min(Mx_{ij}, t^+_{ij}) \\ \forall i \leq m, j \leq n \quad y^+_{ij} &\geq Mx_{ij} + t^+_{ij} - M \\ \forall i \leq m, j \leq n \quad y^-_{ij} &\leq \min(Mx_{ij}, t^-_{ij}) \\ \forall i \leq m, j \leq n \quad y^-_{ij} &\geq Mx_{ij} + t^-_{ij} - M, \end{aligned}$$

where $y^+_{ij}, y^-_{ij} \in [0, M]$ are continuous additional variables.

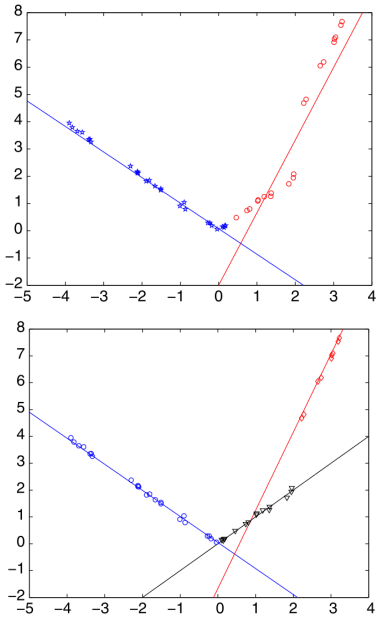


Figure 3.8: A HCP instance solved with two and three hyperplanar clusters [Amaldi et al., 2013].

[31] The upper bound M is enforced without loss of generality because w, w^0 can be scaled arbitrarily.

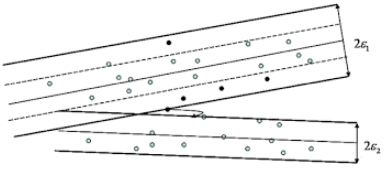


Figure 3.9: HCP with slabs of varying width [Dhyani, 2009].

3. Reformulate each term $|w_{jk}|$ (for $1 \leq k \leq d$) according to Sect. 3.1.5:

$$\begin{aligned}
\min \sum_{\substack{i\leq m\\ j\leq n}} (y^+_{ij} + y^-_{ij}) & & & \\
\forall i \leq m \quad \sum_{j\leq n} x_{ij} &= 1 \\
\forall i \leq m, j \leq n \quad t^+_{ij} - t^-_{ij} &= w_j p_i - w^0_j \\
\forall i \leq m, j \leq n \quad y^+_{ij} &\leq \min(Mx_{ij}, t^+_{ij}) \\
\forall i \leq m, j \leq n \quad y^+_{ij} &\geq Mx_{ij} + t^+_{ij} - M \\
\forall i \leq m, j \leq n \quad y^-_{ij} &\leq \min(Mx_{ij}, t^-_{ij}) \\
\forall i \leq m, j \leq n \quad y^-_{ij} &\geq Mx_{ij} + t^-_{ij} - M \\
\forall j \leq n \quad \sum_{k\leq d} (u^+_{jk} + u^-_{jk}) &= 1 \\
\forall j \leq n, k \leq d \quad u^+_{jk} - u^-_{jk} &= w_{jk} \\
\forall j \leq n, k \leq d \quad u^+_{jk} u^-_{jk} &= 0,
\end{aligned}$$

where $u^+_{jk}, u^-_{jk} \in [0, M]$ are continuous variables for all j and $k \leq d$.[32]

4. The last constraints above are complementarity constraints,[33] which we can reformulate according to Sect. 3.2.3:

$$\begin{aligned}
\min \sum_{\substack{i\leq m\\ j\leq n}} (y^+_{ij} + y^-_{ij}) & & & \\
\forall i \leq m \quad \sum_{j\leq n} x_{ij} &= 1 \\
\forall i \leq m, j \leq n \quad t^+_{ij} - t^-_{ij} &= w_j p_i - w^0_j \\
\forall i \leq m, j \leq n \quad y^+_{ij} &\leq \min(Mx_{ij}, t^+_{ij}) \\
\forall i \leq m, j \leq n \quad y^+_{ij} &\geq Mx_{ij} + t^+_{ij} - M \\
\forall i \leq m, j \leq n \quad y^-_{ij} &\leq \min(Mx_{ij}, t^-_{ij}) \\
\forall i \leq m, j \leq n \quad y^-_{ij} &\geq Mx_{ij} + t^-_{ij} - M \\
\forall j \leq n \quad \sum_{k\leq d} (u^+_{jk} + u^-_{jk}) &= 1 \\
\forall j \leq n, k \leq d \quad u^+_{jk} - u^-_{jk} &= w_{jk} \\
\forall j \leq n, k \leq d \quad u^+_{jk} &\leq Mz_{jk} \\
\forall j \leq n, k \leq d \quad u^-_{jk} &\leq M(1 - z_{jk}),
\end{aligned}$$

where $z_{jk} \in \{0, 1\}$ are binary variables for all $j \leq n$ and $k \leq d$.

[32] Again, the upper bound M does not lose generality.

[33] They can be collectively written as $\sum_{\substack{j\leq n\\ k\leq d}} u^+_{jk} u^-_{jk} = 0$. Why?

This reformulation allows us to solve P by using a MILP solver — these have desirable properties with respect to MINLP solvers, such as numerical stability, scalability and an optimality guarantee.

The AMPL `.mod` file for the MILP reformulation of the HCP is as follows.

```
## parameters
param m integer, > 0;
```

```
param n integer, > 0;
param dim integer, > 0;
set N := 1..n; # planes
set M := 1..m; # points
set D:=1..dim; # spatial dimensions
param p{M,D}; # the points
param P{d in D} := (max{i in M} p[i,d]) - (min{i in M} p[i,d]); # ranges
# used for approximate linearization
param small default 1e-6;
param big default 1e3;
# w (hyperplane coefficients) variable bounds
param wL := -1; #-big;
param w0L := -big;
param wU := 1; #big;
param w0U := big;


## variables
var w0{N} <= w0U, >= w0L;
var w{N,D} <= wU, >= wL;
var x{M,N} binary;
var tplus{M,N} >= 0, <= big;
var tminus{M,N} >= 0, <= big;
var yplus{M,N} >= 0, <= big;
var yminus{M,N} >= 0, <= big;
var uplus{N,D} >= 0, <= big;
var uminus{N,D} >= 0, <= big;
var z{N,D} binary;


## objective function
minimize fitting_error: sum{i in M, j in N} (yplus[i,j] + yminus[i,j]);

## constraints
subject to assignment {i in M} : sum{j in N} x[i,j] = 1;
subject to minabs_ref {i in M, j in N} :
  tplus[i,j] - tminus[i,j] = sum{k in D} w[j,k] * p[i,k] - w0[j];
subject to yplus_lin1 {i in M, j in N} : yplus[i,j] <= big * x[i,j];
subject to yplus_lin2 {i in M, j in N} : yplus[i,j] <= tplus[i,j];
subject to yplus_lin3 {i in M, j in N} :
  yplus[i,j] >= big * x[i,j] + tplus[i,j] - big;
subject to yminus_lin1 {i in M, j in N} : yminus[i,j] <= big * x[i,j];
subject to yminus_lin2 {i in M, j in N} : yminus[i,j] <= tminus[i,j];
subject to yminus_lin3 {i in M, j in N} :
  yminus[i,j] >= big * x[i,j] + tminus[i,j] - big;
subject to one_norm1_ref{j in N}: sum{k in D} (uplus[j,k] + uminus[j,k]) = 1;
subject to one_norm2_ref{j in N, k in D}: uplus[j,k] - uminus[j,k] = w[j,k];
subject to one_norm3_ref{j in N, k in D}: uplus[j,k] <= big*z[j,k];
subject to one_norm4_ref{j in N, k in D}: uminus[j,k] <= big*(1 - z[j,k]);
```

A small instance consisting of 8 points and 2 planes in $\mathbb{R}^2$, with $p = \{(1,7),(1,1),(2,2),(4,3),(4,5),(8,3),(10,1),(10,5)\}$ is solved to optimality by the ILOG CPLEX solver [IBM, 2014] to produce the following output:

```
Assignment of points to hyperplanar clusters:
cluster 1 = { 2 3 4 8 }
cluster 2 = { 1 5 6 7 }
The hyperplane clusters
```

```
1: (-0.272727) x_1 + (0.727273) x_2 + (-0.909091) = 0
2: (0.4) x_1 + (0.6) x_2 + (-4.6) = 0
```

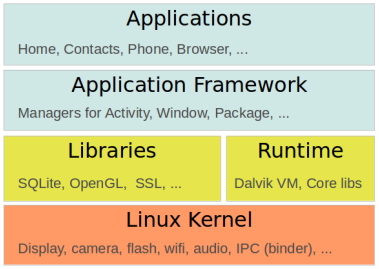


Figure 3.10: Developing software modules or selecting them off-the-shelf? (From `vogella.com`).

3.3.2 *Selection of software components*

Large software systems consist of a complex architecture of interdependent, modular software components. These may either be built or bought off-the-shelf. The decision of whether to build or buy software components influences the cost, delivery time and reliability of the whole system, and should therefore be taken in an optimal way [Cortellessa et al., 2006].

Consider a software architecture with n component slots. Let I_i be the set of off-the-shelf components and J_i the set of purpose-built components that can be plugged in the i-th component slot, and assume $I_i \cap J_i = \emptyset$. Let T be the maximum assembly time and R be the minimum reliability level. We want to select a sequence of n off-the-shelf or purpose-built components compatible with the software architecture requirements that minimize the total cost, whilst satisfying delivery time and reliability constraints.

This problem can be modelled as follows.

- *Parameters*:
  1. Let $\mathcal{N} \in \mathbb{N}$;
  2. for all $i \leq n$, s_i is the expected number of calls to the i-th component;
  3. for all $i \leq n, j \in I_i$, c_{ij} is the cost, d_{ij} is the delivery time, and μ_{ij} the probability of failure on demand of the j-th off-the-shelf component for slot i;
  4. for all $i \leq n, j \in J_i$, $\bar{c}_{ij}$ is the cost, t_{ij} is the estimated development time, τ_{ij} the average time required to perform a test case, p_{ij} is the probability that the instance is faulty, and b_{ij} the testability of the j-th purpose-built component for slot i.

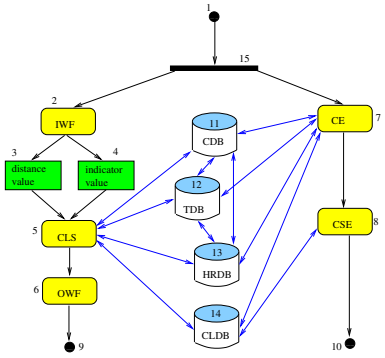


Figure 3.11: MP can be used to rationalize interfaces in software architectures, modelled as a graph, directed or undirected. After some years of development, a complex software package may start to display a "spaghetti"-like architecture such as the one shown above. This can be simplified by inserting *interface modules* whose purpose is that of dispatching to the correct module. If we identify a subgraph H which is fully connected, we can add an interface ι and replace the clique with a star centered at ι and connected to all of the nodes of H (see Fig. 3.12).

- *Variables*:
  1. Let $x_{ij} = 1$ if component $j \in I_i \cup J_i$ is chosen for slot $i \leq n$, and 0 otherwise;
  2. Let $N_{ij} \in \mathbb{Z}$ be the (non-negative) number of tests to be performed on the purpose-built component $j \in J_i$ for $i \leq n$: we assume $N_{ij} \in \{0, \ldots, \mathcal{N}\}$.
- *Objective function*. We minimize the total cost, i.e. the cost of the off-the-shelf components c_{ij} and the cost of the purpose-built components $\bar{c}_{ij}(t_{ij} + \tau_{ij}N_{ij})$:

$$\min \sum_{i \leq n} \left(\sum_{j \in I_i} c_{ij} x_{ij} + \sum_{j in J_i} \bar{c}_{ij}(t_{ij} + \tau_{ij} N_{ij}) x_{ij} \right).$$

- *Constraints*:

  1. assignment constraints: each component slot in the architecture must be filled by exactly one software component

$$\forall i \leq n \quad \sum_{j \in I_i \cup J_i} x_{ij} = 1;$$

  2. delivery time constraints: the delivery time for an off-the-shelf component is simply d_{ij}, whereas for purpose-built components it is $t_{ij} + \tau_{ij} N_{ij}$

$$\forall i \leq n \quad \sum_{j \in I_i} d_{ij} x_{ij} + \sum_{j \in J_i} (t_{ij} + \tau_{ij} N_{ij}) x_{ij} \leq T;$$

  3. reliability constraints: the probability of failure on demand of off-the shelf components is μ_{ij}, whereas for purpose-built components it is given by

$$\vartheta_{ij} = \frac{p_{ij} b_{ij} (1 - b_{ij})^{(1-b_{ij})N_{ij}}}{(1 - p_{ij}) + p_{ij}(1 - b_{ij})^{(1-b_{ij})N_{ij}}},$$

  so the probability that no failure occurs during the execution of the i-th component is

$$\varphi_i = e^{s_i \left(\sum_{j \in I_i} \mu_{ij} x_{ij} + \sum_{j \in J_i} \vartheta_{ij} x_{ij} \right)},$$

  whence the constraint is

$$\prod_{i \leq n} \varphi_i \geq R;$$

  notice we have three classes of reliability constraints involving two sets of additional variables ϑ, φ.

This problem is a MINLP P with no continuous variables. Next, we reformulate P to a MILP.

1. Consider the term $g = \prod_{i \leq n} \varphi_i$ and apply the reformulation in Sect. 3.1.6 to P, to obtain the formulation P_1 as follows:

$$\begin{aligned}
\min \sum_{i \leq n} \left(\sum_{j \in I_i} c_{ij} x_{ij} + \sum_{j \in J_i} \bar{c}_{ij} (t_{ij} + \tau_{ij} N_{ij}) x_{ij} \right) & \\
\forall i \leq n \quad \sum_{j \in I_i \cup J_i} x_{ij} &= 1 \\
\forall i \leq n \quad \sum_{j \in I_i} d_{ij} x_{ij} + \sum_{j \in J_i} (t_{ij} + \tau_{ij} N_{ij}) x_{ij} &\leq T \\
\frac{p_{ij} b_{ij} (1 - b_{ij})^{(1-b_{ij})N_{ij}}}{(1 - p_{ij}) + p_{ij}(1 - b_{ij})^{(1-b_{ij})N_{ij}}} &= \vartheta_{ij} \\
w &\geq R \\
\sum_{i \leq n} s_i \left(\sum_{j \in I_i} \mu_{ij} x_{ij} + \sum_{j \in J_i} \vartheta_{ij} x_{ij} \right) &= \ln(w),
\end{aligned}$$

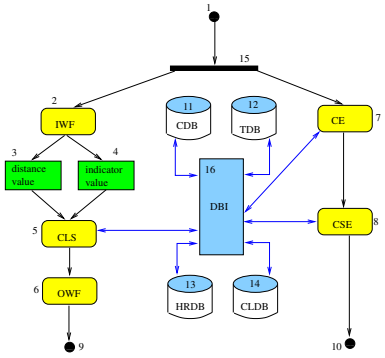


Figure 3.12: Given a software architecture (directed) graph $G = (V, A)$, we suppose that arcs are colored by a coloring function $\mu : A \to \{1, \ldots, K\}$. These colors may denote any given type of semantic relationship between the modules: they provide a way for the software architect to define relevant subgraphs which can actually be replaced by an interface. What we can do automatically with MP is to identify the densest subgraphs $H = (U, F)$, which we define as subgraphs where $|F| - |U|$ is maximum, where all the arcs are colored by a given k. To achieve this, for any given color k we solve the following BQP: $\max_{x \in \{0,1\}^{|V|}} \sum_{(u,v) \in A} x_u x_v - \sum_{v \in V} x_v$ subject to $\forall \ (u, v) \in A \quad x_u x_v \leq \min(\max(0, \mu_{uv} - k + 1), \max(0, k - \mu_{uv} + 1))$. The set of vertices U of the densest subgraph having color k is defined by the indicator vector x: if $|U| < |F|$ we replace F by the arcs $\{(\iota, u) \mid u \in U\}$. We can solve this BQP for every color $k \leq K$.

and observe that $w \geq R$ implies $\ln(w) \geq \ln(R)$ because the ln function is monotonically increasing, so the last two constraints can be grouped into a simpler one not involving logarithms of problem variables:

$$\sum_{i \leq n} s_i \left(\sum_{j \in I_i} \mu_{ij} x_{ij} + \sum_{j \in J_i} \vartheta_{ij} x_{ij} \right) \geq \ln(R).$$

2. We now make use of the fact that N_{ij} is an integer variable for all $i \leq n, j \in J_i$, and reformulate the integer variables to binary variables as per Sect. 3.1.8. For $k \in \{0, \ldots, N\}$ we add assignment variables v_{ij}^k so that $v_{ij}^k = 1$ if $N_{ij} = k$ and 0 otherwise. Now for all $k \in \{0, \ldots, N\}$ we compute the constants $\vartheta^k = \frac{p_{ij} b_{ij} (1-b_{ij})^{(1-b_{ij})k}}{(1-p_{ij}) + p_{ij}(1-b_{ij})^{(1-b_{ij})k}}$, which we add to the problem parameters. We remove the constraints defining ϑ_{ij} in function of N_{ij}: since the following constraints are valid:

$$\forall i \leq n, j \in J_i \quad \sum_{k \leq N} v_{ij}^k = 1 \tag{3.19}$$

$$\forall i \leq n, j \in J_i \quad \sum_{k \leq N} k v_{ij}^k = N_{ij} \tag{3.20}$$

$$\forall i \leq n, j \in J_i \quad \sum_{k \leq N} \vartheta^k v_{ij}^k = \vartheta_{ij}, \tag{3.21}$$

the constraints in Eq. (3.20) are used to replace N_{ij}, and those in Eq. (3.21) to replace ϑ_{ij}.[34] We obtain:

$$\begin{aligned}
\min \sum_{i \leq n} \left(\sum_{j \in I_i} c_{ij} x_{ij} + \sum_{j \in J_i} \bar{c}_{ij} (t_{ij} + \tau_{ij} \sum_{k \leq N} k v_{ij}^k) x_{ij} \right) & \\
\forall i \leq n \quad \sum_{j \in I_i \cup J_i} x_{ij} &= 1 \\
\forall i \leq n \quad \sum_{j \in I_i} d_{ij} x_{ij} + \sum_{j \in J_i} (t_{ij} + \tau_{ij} \sum_{k \leq N} k v_{ij}^k) x_{ij} &\leq T \\
\sum_{i \leq n} s_i \left(\sum_{j \in I_i} \mu_{ij} x_{ij} + \sum_{j \in J_i} x_{ij} \sum_{k \leq N} \vartheta^k v_{ij}^k \right) &\geq \ln(R) \\
\forall i \leq n, j \in J_i \quad \sum_{k \leq N} v_{ij}^k &= 1.
\end{aligned}$$

3. We distribute products over sums in the formulation to obtain the binary product sets $\{x_{ij} v_{ij}^k \mid k \leq N\}$ for all $i \leq n, j \in J_i$: by repeatedly applying the reformulation in Sect. 3.2.1 to all binary products of binary variables, we get a MILP reformulation Q of P where all the variables are binary.

We remark that Q derived above has many more variables and constraints than P. More compact reformulations are applicable in Step 3 because of the presence of the assignment constraints [Liberti, 2007].

Reformulation Q essentially rests on linearization variables w_{ij}^k which replace the quadratic terms $x_{ij} v_{ij}^k$ throughout the formulation.

[34] Constraints (3.19) are simply added to the formulation.

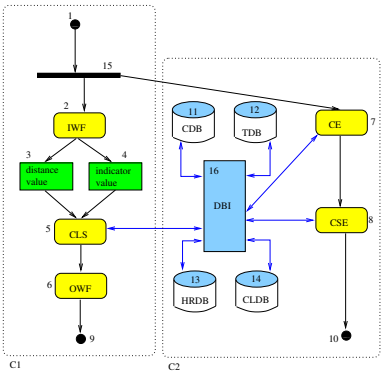


Figure 3.13: After interfacing, we can solve a graph partitioning problem on $G = (V, A)$ so as to identify semantically related clusters, for which there might conceivably exist an off-the-shelf solution.

A semantic interpretation of step 3 is as follows. We notice that for $i \leq n, j \in J_i$, if $x_{ij} = 1$, then $x_{ij} = \sum_k v_{ij}^k$ (because only one value k will be selected), and if $x_{ij} = 0$, then $x_{ij} = \sum_k v_{ij}^k$ (because no value k will be selected). This means that

$$\forall i \leq n, j \in J_i \quad x_{ij} = \sum_{k \leq N} v_{ij}^k \tag{3.22}$$

is a valid problem constraint. We use it to replace x_{ij} everywhere in the formulation where it appears with $j \in I_i$, obtaining a opt-reformulation with x_{ij} for $j \in I_i$ and quadratic terms $v_{ij}^k v_{lp}^h$. Now, because of (3.19), these are zero when $(i, j) \neq (l, p)$ or $k \neq h$ and are equal to v_{ij}^k when $(i, j) = (l, p)$ and $k = h$, so they can be linearized exactly by replacing them by either 0 or v_{ij}^k according to their indices. What this really means is that the reformulation Q, obtained through a series of automatic reformulation steps, is a semantically different formulation defined in terms of the following decision variables:

$$\forall i \leq n, j \in I_i \quad x_{ij} = \begin{cases} 1 & \text{if } j \in I_i \text{ is assigned to } i \\ 0 & \text{otherwise.} \end{cases}$$

$$\forall i \leq n, j \in J_i, k \leq N \quad v_{ij}^k = \begin{cases} 1 & \text{if } j \in J_i \text{ is assigned to } i \text{ and there} \\ & \text{are } k \text{ tests to be performed} \\ 0 & \text{otherwise.} \end{cases}$$

This is an important hint to the importance of automatic reformulation in problem analysis: it is a syntactical operation, the result of which, when interpreted, can suggest a new meaning.

3.4 Summary

This chapter is about reformulations as a means to change a formulation so it can be solved using a given solver, so that some optimality properties are preserved.

- Motivation and main definitions
- Elementary reformulations: changing objective direction and constraint sense, lifting – restrictions – projections, turning equations into inequalities and vice versa, absolute values, product of exponentials, turning binary variables to continuous, turning discrete variables to binary, turning integer variables to binary, turning feasibility to optimization problems, minimization of a maximum
- Exact linearizations: product of binary variables, product of a binary variable by a continuous one, linear complementarity, minimization of absolute values, linear fractional terms
- Applying reformulations to turn the hyperplane clustering problem from MINLP to MILP.

Part II

In-depth topics

4
Constraint programming

Constraint Programming (CP) is a declarative programming language for describing feasibility problems, such as e.g. finding the solution of a given sudoku game, or of a crossword puzzle, or of a game of *eternity* (see Fig. 4.1), or a neutral arrangement of nine queens on a chessboard.[1]

The CP language consists of sets, parameters, decision variables and constraints: like MP but without objective functions. Although today decision variables in CP can be both discrete and continuous, "native" CP only offered discrete decision variables.[2]

Let p be the parameter vector, which defines the input of the problem being solved. Let $D_1, \dots, D_n$ be the domains of the decision variables $x_1, \dots, x_n$. Let $C_1^p(x), \dots, C_m^p(x)$ be the constraints, expressed as logical statements which can evaluate to YES or NO. Denote as $x' = \mathscr{C}(p)$ a solution of the CP formulation, with $x' = \varnothing$ if $\mathscr{C}(p)$ is infeasible.

Although a constraint C_i^p (for $i \leq m$) is commonly written in a humanly readable form, such as $\mathsf{AllDifferent}(x_{i_1}, \dots, x_{i_k})$ meaning "$x_{i_1}, \dots, x_{i_k}$ must be assigned different values", C_i^p is formally defined as a k-ary relation $R_i \subseteq D_{i_1} \times \cdots \times D_{i_k}$.

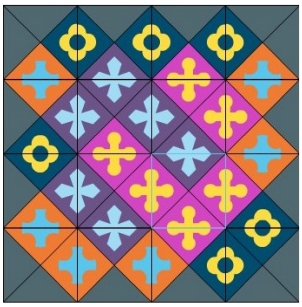


Figure 4.1: A 4×4 *eternity* board [Dürr, 2011].

[1] For some reason which completely escapes me, constraint programmers are *very* fond of games and puzzles, to the point that it is relatively hard to find any *real* application of CP, aside from scheduling that is.

[2] In fact, CP was originally only used with explicitly bounded decision variables, which prevents the language from being Turing-complete. Even with bounded variables, CP can still describe **NP**-hard problems, though: try reducing SAT to CP.

4.1 *The dual CP*

We remark that C_i^p need not be explicitly cast in function of all the decision variables. In fact, it turns out that every CP formulation $\mathscr{C}(p)$ can be reduced to one which has at most unary and binary constraints, by reduction to the *dual CP*, denoted $\mathscr{C}^*(p)$. For each $i \leq m$, consider the subset $S_i \subseteq \{1, \dots, n\}$ of variable indices that C_i^p depends on.

The dual CP has m dual variables $y_1, \dots, y_m$ with domains $R_1, \dots, R_m$ and the dual constraints:

$$\forall i < j \leq m \quad y_i[S_i \cap S_j] = y_j[S_i \cap S_j]. \tag{4.1}$$

Note that each dual variable y_i takes values from the relation R_i, which is itself a subset of a cartesian product of the D_i's. Hence you should think of each dual variable as a vector or a list, which means that $y = (y_1, \dots, y_m)$ is a jagged array.[3] The meaning of Eq. (4.1) is

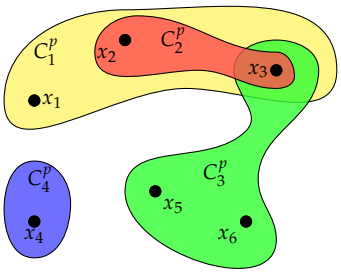


Figure 4.2: The constraint/variable incidence structure of the CP formulation can be represented as a hypergraph.

[3] A *jagged array* is a basic data structure: it consists of a list of lists of different sizes.

that, for each pair of constraints C_i^p, C_j^p, we want the corresponding dual variables y_i, y_j to take on values from R_i and R_j which agree on $S_i \cap S_j$. Note that formally each dual constraint is binary, since it only involves two dual variables. Note also that when $S_i \cap S_j = \varnothing$, the constraint is empty, which means it is always satisfied and we can formally remove it from the dual CP. Thus, Eq. (4.1) contains $O(m^2)$ constraints in the worst case, but in practice there could be fewer than that.

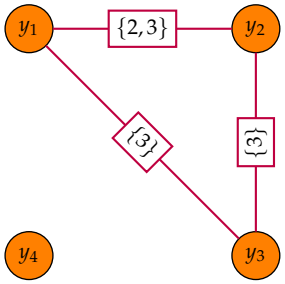


Figure 4.3: A graph representation of the constraint/variable incidence structure of the dual formulation to the CP of Fig. 4.2. Since all constraints are binary, the hyperedges are all simple edges, labeled with $S_i \cap S_j$. Note that the unary constraint C_4^p in the original CP becomes a unary constraint in the dual, which simply specifies the action of C_4^p on x_6 as the restriction of R_4 so that D_6 becomes $C_4^p(D_6)$ (the restriction of D_6 induced by C_4^p).

4.1.1 Proposition
$\mathscr{C}^*(p)$ is an exact reformulation of $\mathscr{C}(p)$ and vice versa.

Proof. Let $x' = \mathscr{C}(p)$ be a feasible solution of the original CP formulation. This means that, for all $i \leq m$, $C_i^p(x'_{i_1}, \ldots, x'_{i_k})$ evaluates to YES. Define $y'_i = (x'_{i_1}, \ldots, x'_{i_k})$. We claim $y' = (y'_1, \ldots, y'_m)$ satisfies all constraints Eq. (4.1). Consider two constraints $i < j$ with $S_i \cap S_j \neq \varnothing$. For each $h \in S_i \cap S_j$, $y'_{ih} = x'_h = y'_{jh}$, which proves the claim. Conversely, let $y' = \mathscr{C}^*(p)$ be a feasible solution for the dual CP, and aim at constructing a solution x' of $\mathscr{C}(p)$. Pick any $h \leq n$: if there are no $i < j$ such that $h \in S_i \cap S_j$, then set x'_h to any element of D_h compatible with C_i^p. Otherwise, for any $i < j$ such that $h \in S_i \cap S_j$, feasibility of y' with respect to the dual constraints ensure that $y'_{ih} = y'_{jh}$, so set $x'_h = y'_{ih}$. Now consider any constraint $C_i^p(x)$ of the original CP, for some $i \leq m$, and aim to show that $C_i^p(x')$ evaluates to YES: if $h \in S_i$ and $h \notin S_j$ for any $j \neq i$, then x'_h is, by construction, part of a satisfying solution for C_i^p; if $h \in S_i \cap S_j$ for some $j \neq i$, then $x_h = y'_{ih} = y'_{jh}$ is in the projection of both R_i and R_j to the h-th coordinate. Hence they are also part of a satisfying solution for C_i^p. Having constructed a bijection between solutions of the two CP formulations, we can use it to define the map ϕ mentioned in Defn. 3.0.1. □

4.2 *CP and MP*

CP relates to computer science as MP relates to applied mathematics. Both CP and MP are Turing complete declarative programming languages as long as their decision variables are not explicitly bounded. Both contain **NP**-hard subclasses of formulations, and hence are **NP**-hard when seen as formal decision problems. MP was born out of mathematics, whereas CP stems from the development of logic into computer science. The academic communities around CP and MP are still different, although their intersection is growing, insofar as solution methods from CP and MP can be successfully mixed.

Differently from MP, sets in CP are used both for indexing variable symbols in structured formulations and for defining *domains* where the decision variables can range. Also differently from MP, constraints in CP are put together from a richer set of operators than the one usually employed in MP. In fact, much of the research

Figure 4.4: The CPAIOR 2011 conference poster, see www.andrew.cmu.edu/user/vanhoeve/cpaior.

in CP focuses on creating new operators which can be efficiently dealt with in the typical CP solution algorithm.

Another important difference between CP and MP is that, whereas in MP we have many different solvers for different MP subclasses, in CP there is essentially one solution approach: a search based on branching and backtracking.

4.3 The CP solution method

At its most essential, branching and backtracking is limited to work on bounded variable domains, and looks like the algorithm in Fig. 4.6.

This algorithm generates a search tree where each node has as many child nodes as the number of elements in the domain of the branching variable. The algorithm can be refined by means of several per-node procedures. The most important ones are *domain reduction* and *pruning*. The former consists in various heuristic procedures for attempting to remove as many elements as possible from the variable domains without losing any part of the feasible region. As for the latter, when every element can be removed from a domain, the current search node is pruned.

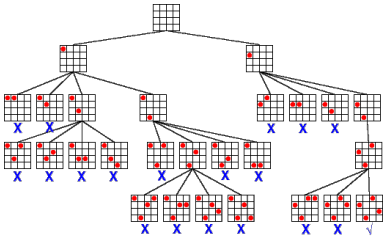


Figure 4.5: Branching/backtracking on the 4 queens problem (from ktiml.mff.cuni.cz).

function CPsolve(J, x')
// J = indices of unfixed vars
// x' = vector of fixed var. values
if $J = \varnothing$ **then**
  return x'
end if
choose branching var. index $j \in J$
for $v \in D_j$ **do**
  fix $x'_j \leftarrow v$
  if some constraint evaluates to NO **then**
    return $\varnothing$
  else
    return CPsolve($J \cup \{j\}$, x')
  end if
end for

Figure 4.6: CPsolve(J, x')

4.3.1 Domain reduction and consistency

A set of k domains $D_{i_1}, \ldots, D_{i_k}$ is k-consistent with respect to a constraint $C_i^p(x_{i_1}, \ldots, x_{i_k})$ if for each $(k-1)$-tuple

$$(a_{i_1}, \ldots, a_{i_j-1}, a_{i_j+1}, \ldots, a_{i_k}) \in D_{i_1} \times \cdots \times D_{i_j-1}, D_{i_j+1} \times \cdots \times D_{i_{k-1}}$$

there is $a_{i_j} \in D_{i_j}$ such that $C_i^p(a_{i_1}, \ldots, a_{i_k})$ evaluates to YES. A CP instance is k-consistent if all of its cartesian domain k-products are k-consistent with respect to each k-ary constraint. The cases of $k \in \{1, 2\}$ are the most important: the case $k = 1$ because it can be dealt through pre-processing, and the case $k = 2$ because every CP can be reduced (by means of duality, see Sect. 4.1) to the case of binary constraints.

Consider for example a domain $D_1 = \{2, 3, 5, 6, 9\}$ for the variable x_1, and the unary constraint $C_1^p \equiv [L \leq x_1 \leq U]$, where $L \leq U$ are parameters which are given as part of the input p. Suppose for some given $p = p'$ we have $L = 4$ and $U = 7$. Then D_1 is consistent with respect to $C_1^{p'}$ if D_1 is reduced to $\{5, 6\}$, namely only the values which satisfy $4 \leq x_1 \leq 7$.

Consistency with respect to unary constraints is also known as 1-consistency or node-consistency. As mentioned above, it can be carried out as a pre-processing step to the solution process: since unary constraints cannot encode relations between variables, once the domain becomes consistent w.r.t. a unary constraint, it must necessarily keep being node-consistent throughout the solution process. Suppose, to get a contradiction, that D_j is node-consistent with respect to the unary constraint $C_i^p(x_j)$, but that, due to some

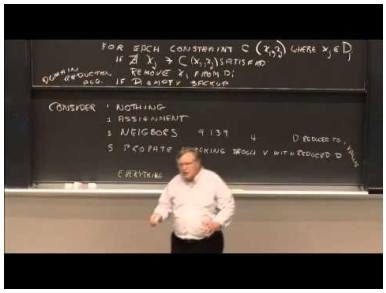


Figure 4.7: Patrick Winston explaining 1-consistency (from MITOpenCourseWare).

other constraint C_ℓ^p involving x_j, D_j becomes node-inconsistent w.r.t. C_i^p. Since constraints can never add values to domains, the effect of C_ℓ^p on D_j must yield a domain $D'_j \subseteq D_j$. Of course, if D'_j is node-inconsistent w.r.t C_i^p, any superset of D'_j must also be node-inconsistent, which means that D_j is, against the initial assumption.

Consistency with respect to binary constraints of the form $C_i^p(x_j, x_h)$ is also known as 2-consistency or arc-consistency. The domains D_j, D_h are 2-consistent with respect to C_i^p if for each $a_j \in D_j$ there is $a_h \in D_h$ such that $C_i^p(a_j, a_h)$ evaluates to YES. Consider for example the domain $D_2 = \{1,2,3,4\}$ for x_2 and the constraint $x_1 < x_2$: then D_1, D_2 are arc-consistent with respect to $x_1 < x_2$ if D_1 is reduced to $\{2,3\}$ and D_2 to $\{3,4\}$.

Domain reduction through consistency is usually performed before choosing the branching variable index in the search algorithm (Fig. 4.6).

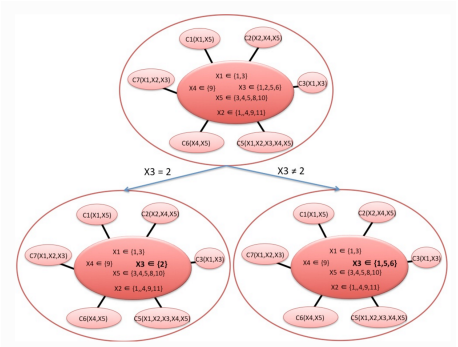


Figure 4.8: Branching in CP (from www.info.ucl.ac.be/~pschaus).

4.3.2 *Pruning and solving*

After a certain amount of branching and domain reduction, there might[4] be leaf nodes of the search tree where the domain product $\mathcal{D} = D_1 \times D_n$ is either empty, or consists of exactly one element $a = (a_1, \ldots, a_n)$. In the former case, the leaf node is *pruned*, i.e. no child nodes are further appended to the pruned node. In the latter case, the solution $x = a$ is returned to the user, and the algorithm terminates.

[4] If all the domains have a finite number of values, then you can replace "might" by "will".

Next, we illustrate how the AllDifferent constraint impacts pruning and finding solutions in CP. Suppose $D_1 = \ldots, = D_n = D$ such that $|D| = n$, and $x_1, \ldots, x_n \in D$. By stating

$$\mathsf{AllDifferent}(x_1, \ldots, x_n)$$

we are really saying that $x_1, \ldots, x_n$ encode a permutation of D. We know from Sect. 2.2 and 2.4 that permutations correspond to linear assignments, which are special cases of *matchings*.[5] Finding matchings in graphs can be done in polytime [Edmonds, 1965] *, which implies that we can efficiently test search tree nodes for pruning or finding candidate solutions.

[5] A matching M in a graph $G = (V, E)$ is a set of edges which are pairwise non-incident. The relevant optimization problem is to find matchings of maximum size. A linear assignment is a maximum perfect matching in a bipartite graph (also see Sidenote 13 in Ch. 2).

More specifically, the matching arising from an AllDifferent constraint is defined on a bipartite graph (x, D, E) between the n-variable vector x and the n-valued domain D; E denotes the set of variable/value assignments which are feasible with respect to the current node of the search tree. At this point we invoke Hall's theorem[6]

4.3.1 Theorem ([Hall, 1935])
There exists a perfect matching[7] *in a bipartite graph* (U, V, E) *if and only if*

$$\forall S \subseteq U \quad |S| \leq |\Gamma(S)|, \tag{4.2}$$

where $\Gamma(S) \subseteq V$ *is the set of vertices in* V *adjacent to the vertices in* S.

[6] Also known as *stable marriage* theorem. You can look at [Dürr, 2011] for a short proof.

[7] A matching is *perfect* if every vertex is adjacent to exactly one edge in the matching. Referred to a bipartite graph (U, V, E) the existence of a perfect matching obviously implies that $|U| = |V|$.

By Thm. 4.3.1, there is no perfect matching if the contrapositive of Eq. (4.2) is true, namely

$$\exists S \subseteq U \quad |S| > |\Gamma(S)|,$$

and it turns out that the certificate S can be found in polytime.

This offers the perfect tool for pruning a node using the AllDifferent constraint: at some level of the search tree, nodes will be created where the edge set E is sparse enough (edges having been removed at prior nodes) so as not to allow perfect matchings, with this condition being efficiently verifiable. Conversely, since maximum matchings can also be found in polytime, the AllDifferent constraint offers an efficient way to find candidate solutions to the CP.

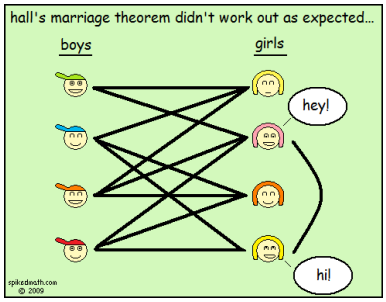


Figure 4.9: An unforeseen glitch in Hall's stable marriage theorem (from snipview.com).

4.4 *Objective functions in CP*

CP formulations do not natively have objective functions $f(x)$ to optimize, but there are at least two approaches in order to handle them.

A simple approach consists in enforcing a new constraint $f(x) \leq d$, and then update the value of d with the objective function value of the best solution found so far during the solution process (Sect. 4.3). Since a complete search tree exploration lists all of the feasible solutions, the best optimum at the end of the exploration is the global optimum.

In the case of discrete bounded variable domains another approach consists in using *bisection*, which we explain in more detail. Let $\mathscr{C}(p)$ be a CP formulation, with p a parameter vector representing the instance; and let $x' = \mathscr{C}(p)$, with $x' = \varnothing$ if p is an infeasible instance.

Given some lower bound f_L and upper bound f_U such that $f_L \leq \min_x f(x) \leq f_U$, we select the midpoint $d = \frac{f_L + f_U}{2}$ and consider the formulation $\bar{\mathscr{C}}(p, d)$ consisting of all the constraints of $\mathscr{C}(p)$ together with the additional constraint $f(x) \leq d$. We compute $\bar{x} = \bar{\mathscr{C}}(p, d)$: if $\bar{x} = \varnothing$ we update $f_L \leftarrow d$, otherwise $f_U \leftarrow d$, finally we update the midpoint and repeat. This process terminates whenever $f_L = f_U$, in which case $\bar{x}$ is a feasible point in $\mathscr{C}(p)$ which minimizes $f(x)$.

Bisection may fail to terminate unless f_L, f_U can only take finitely many values (e.g. $f_L, f_U \in \mathbb{N}$).[8] Since CP formulations are mostly defined over integer or discrete decision variables, this requirement is often satisfied.

The complexity of bisection is $O(t \log_2(|U - L|))$ where t is the complexity of calling $\bar{\mathscr{C}}(p, d)$, which makes it attractive in case $\bar{\mathscr{C}}(p, d)$ can be solved efficiently.

while $L < U$ **do**
  $d \leftarrow (L + U)/2$
  $\bar{x} \leftarrow \bar{\mathscr{C}}(p, d)$
  if $\bar{x} = \varnothing$ **then**
    $L \leftarrow d$
  else
    $U \leftarrow d$
  end if
end while
return $\bar{x}$

Figure 4.10: The bisection algorithm (see Fig. 1.16).

[8] We remark that f_L can only increase and f_U can only decrease during the bisection algorithm, hence if both are integer they can only range between their initial values.

4.5 *Some surprising constraints in CP*

Modelingwise, we already mentioned that CP offers a lot more variety than MP in the constraints it can deal with.

Name	Constraint	Why is it unusual for MP?
AllDifferent(x)	$\forall i < j \leq n \ (x_i \neq x_j)$	The $\neq$ relation is not admitted in MP. It can be modelled using $\leq, \geq$ and additional binary variables.
Implies(x, g, i, j)	$g_i(x) \leq 0 \Rightarrow g_j(x) \leq 0$	Logical connectives are not always admitted in MP (in particular, the implication is not). They can modelled using additional binary variables, linked to the truth value of each logical statement by means of "big M"s and cardinality constraints.
Disjunction(x, g)	$\bigvee_{i \leq m} (g_i(x) \leq 0)$	Constraint disjunctions is not admitted in MP. It can nonetheless be modelled using additional binary variables y_i, additional constraints $g_i(x) \leq M(1 - y_i)$, and one more additional constraint $\sum_i y_i \geq 1$.
Exactly(x, I, k, g) AtLeast(x, I, k, g) AtMost(x, I, k, g)	$\lvert\{i \in I \mid g_i(x) \leq 0\}\rvert = k$ $\lvert\{i \in I \mid g_i(x) \leq 0\}\rvert \geq k$ $\lvert\{i \in I \mid g_i(x) \leq 0\}\rvert \leq k$	Cardinality constraints on the number of satisfied constraints in a set are not admitted in MP. They can be modelled using additional slack variables $g_i(x) \leq My_i$ and imposing the cardinality constraint on $\sum_i y_i$.
Element(x, L, i, j)	$L_{x_i} = x_j$	Decision variables can never be used for indexing in MP. This situation can be modelled using additional variables $y_{\ell v} = 1$ if and only if $L_\ell = v \wedge x_i = \ell \wedge x_j = v$.

4.6 Sudoku

As a toy example, we propose a CP formulation for solving a Sudoku puzzle. This is seen as the following decision problem.

SUDOKU. Given $n \in N$ such that $\sqrt{n} \in \mathbb{N}$, and a partially defined $n \times n$ array $M = (m_{ij})$ where, for some given pairs $(i, j) \in S$, m_{ij} specifies an integer in $N = \{1, \ldots, n\}$, is there an $n \times n$ fully defined array $X = (x_{ij})$ such that:

- $\forall (i, j) \in S \quad x_{ij} = m_{ij}$
- every integer in N appears in each column of X
- every integer in N appears in each row of X
- every integer in N appears in each $\sqrt{n} \times \sqrt{n}$ sub-matrix of X whose upper-left element has indices $(k\sqrt{n}, \ell\sqrt{n})$ for some integers $k, \ell \in R = \{0, \ldots, \sqrt{n} - 1\}$?

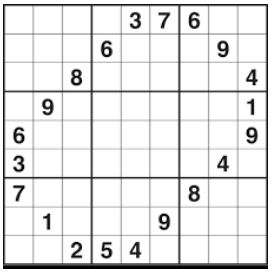


Figure 4.11: A "hard" SUDOKU instance (from theguardian.com).

We introduce integer parameters $n, r \leftarrow \sqrt{n}$, sets $N = \{1, \ldots, n\}$ and $R = \{0, \ldots, r - 1\}$, and a partially defined square array m_{ij} for $i, j \in$. The constraints are:

$$\begin{aligned}
\forall i \in N, j \in N \quad & x_{ij} = m_{ij} \\
\forall i \in N \quad & \text{AllDifferent}(x_{i1}, \ldots, x_{in}) \\
\forall j \in N \quad & \text{AllDifferent}(x_{1j}, \ldots, x_{nj}) \\
\forall k \in R, \ell \in R \quad & \text{AllDifferent}(x_{ij} \mid kr + 1 \leq i \leq kr + r \wedge \ell r + 1 \leq c \leq \ell r + r).
\end{aligned}$$

4.6.1 AMPL code

The CP formulation is encoded by means of the following AMPL `sudoku_cp.mod` file.

```
param n integer, >0;
param r integer, := sqrt(n);
set N := 1..n;
set R := 0..r-1;
param m{N,N} integer, default 0;
var x{N,N} integer, >= 1, <= n;
subject to completeM{i in N, j in N : m[i,j] > 0}: x[i,j] = m[i,j];
subject to rows{i in N}: alldiff{j in N} x[i,j];
subject to cols{j in N}: alldiff{i in N} x[i,j];
```

```
subject to squares{k in R, l in R}:
  alldiff{i in k*r+1..k*r+r, j in l*r+1..l*r+r} x[i,j];
```

We can specify an instance by means of the following `sudoku_cp.dat` file.

```
param n := 9;
param m: 1 2 3 4 5 6 7 8 9 :=
      1   2 . 4 . . . . 5 .
      2   . 9 . . . 8 . . .
      3   8 . . 1 . . . 2 .
      4   . . 7 3 6 . . 8 .
      5   . . 8 . . . 2 . .
      6   . 3 . . 2 1 7 . .
      7   . 8 . . . 3 . . 4
      8   . . . 5 . . . 6 .
      9   . 4 . . . . 8 . 5
;
```

Finally, the following `sudoku_cp.run` file calls the solver and displays the solution.

```
model sudoku_cp.mod;
data sudoku_cp.dat;
option solver ilogcp;
option ilogcp_options "logverbosity=terse";
solve;
printf "x:\n";
for{i in N} {
  for{j in N} {
     printf "%2d ", x[i,j];
  }
  printf "\n";
}
printf "\n";
```

The resulting solution is:

```
x:
 2  7  4  9  3  6  1  5  8
 6  9  1  2  5  8  4  3  7
 8  5  3  1  4  7  6  2  9
 4  2  7  3  6  9  5  8  1
 1  6  8  4  7  5  2  9  3
 9  3  5  8  2  1  7  4  6
 5  8  2  6  1  3  9  7  4
 7  1  9  5  8  4  3  6  2
 3  4  6  7  9  2  8  1  5
```

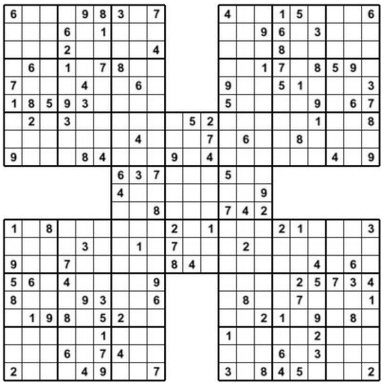


Figure 4.12: A weird SUDOKU variant (from `saxi753.deviantart.com`). Change the CP formulation in this section to handle this instance, and find a solution.

4.7 *Summary*

This chapter is about constraint programming, which is a different modeling paradigm than MP, focusing on feasibility rather than optimality, on discrete rather than continuous variables, and on a single solver parametrized on constraint structures instead of many solvers, each for a different standard form.

5
Maximum cut

The problem which names this chapter is one of the best known problems in combinatorial optimization: finding the cutset of maximum weight in a weighted undirected graph.[1] It has two main motivating applications: finding the minimum energy magnetic spin values of the atoms in a given crystal, and knowing where to drill holes in electrical circuits so that wires that must intersect do so in different board layers. The first application is commonly known as *Ising model of spin glasses*, and the second as *Very Large Scale Integration* (VLSI).[2]

First thing first: the formal definition of this problem is the following.

> Max Cut. Given a weighted undirected graph $G = (V, E, w)$ with $w : E \to \mathbb{R}$, find $S \subseteq V$ such that $\sum_{\substack{i \in S \\ j \notin S}} w_{ij}$ is maximum.

In graph theoretical terminology, a subset S of vertices is known as a *cut*, whereas the set of edges with one incident vertex in S and the other in $V \setminus S$ is a *cutset*. So the Max Cut problem calls for the cutset of maximum weight (see Fig. 5.1).

Max Cut is one of the original problems which R. Karp proved **NP**-hard [Karp, 1972], which is one of the main reasons for its importance. The reduction proof is from a problem called Max-2-Sat, which aims at determining the maximum number of a conjunction of satisfiable clauses involving a disjunction of two literals each.[3]

Another reason why Max Cut is so famous is that it is naturally modelled as a MIQP rather than a MILP, whereas most of the other "classic" combinatorial optimization problems are naturally modelled as MILP.

Most importantly, however, I believe that Max Cut is very famous because of the algorithm that Goemans and Williamson published in 1994. Theirs was the first SDP-based randomized approximation algorithm for a combinatorial optimization problem. The so-called "GW algorithm" became an instant success, and has had a deep impact in theoretical computer science for various reasons. Aside from its theoretical importance, the GW algorithm is one of the few approximation algorithms which *actually works in practice*!

[1] We already mentioned the unweighted variant in Sect. 2.9.3.

[2] Lately, another "application" came about, namely benchmarking D-Wave's claimed quantum computer, see Fig. 2.22 in Ch. 2.

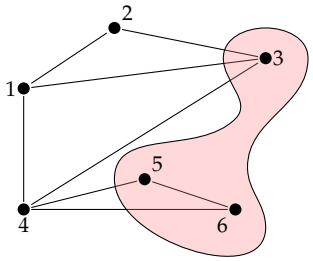


Figure 5.1: A graph with unit edge weights and its maximum cut. The corresponding cutset is the set of edges intersected by the curved perimeter.

[3] A *literal* is a boolean variable x or its complement $\bar{x}$. A *logical formula* in *conjunctive normal form* is a boolean expression in the form $C_1 \wedge \ldots \wedge C_m$, for some integer m, where each *clause* C_i is a disjunction $L_1 \vee \ldots \vee L_{n_i}$ of literals (i.e. each L_j is either a boolean variable or its complement), for each $i \leq m$. In Max-2-Sat we require that $n_i = 2$ for each $i \leq m$.

5.1 Approximation algorithms

In order to understand this mysterious business of algorithms which "surprisingly work[4] in practice", let us see what an approximation algorithm is. Suppose we are trying to solve an instance $\mathcal{I}$ of a minimization problem $P \equiv \min_{x \in \mathcal{X}} f(\mathcal{I}, x)$ having minimum objective function value f^*. We design an algorithm $\mathcal{A}$, but we cannot prove it will find an exact global optimum,[5] so it finds a feasible point $x' \in \mathcal{X}$ with objective function value $f' > f^*$. If we can prove that $\mathcal{A}$ yields a ratio $\frac{f'}{f^*}$ which is bounded *above* by a quantity ϕ for all instances of P, then $\mathcal{A}$ is a ϕ-approximation algorithm for P.

Now let $\mathcal{I}$ be an instance of a maximization[6] problem $P \equiv \max_{x \in \mathcal{X}} f(\mathcal{I}, x)$ with optimal objective function value f^*. As before, we have an algorithm $\mathcal{A}$ which is not exact, so it finds a feasible point $x' \in \mathcal{X}$ with value[7] $f' < f^*$. If we can prove that $\mathcal{A}$ yields a ratio $\frac{f'}{f^*}$ which is bounded *below* by a quantity ϕ for all instances of P, then $\mathcal{A}$ is a ϕ-approximation algorithm for P.

[4] The literature is full of approximation algorithms which prove a theoretical approximation property, but have never been tested in practice. Every time I can, I ask presenting authors of new approximation algorithms whether they have tested their algorithms computationally: more often than not, the answer is "why? what is the point in that?".

[5] In other words, the algorithm is not *exact*.

[6] Since MAX CUT is a maximization problem, this is the relevant point of view.

[7] Note the inequality sense has changed, since we are maximizing.

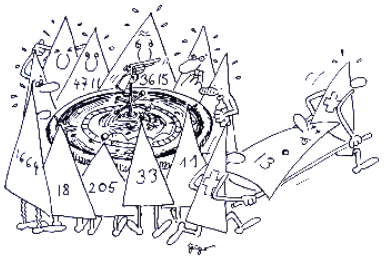
5.1.1 Approximation schemes

The quantity ϕ is a function $\phi(|\mathcal{I}|)$ of the size taken to store the instance $\mathcal{I}$. The best case occurs when ϕ is a constant, the closer to 1 the better. Among the constant approximation algorithms, those giving the best solutions are those which, for any given $\varepsilon > 0$, provide an approximation ratio $1 + \varepsilon$ (for minimization) and $1 - \varepsilon$ (for maximization) in polynomial time. Such algorithms are called *polynomial-time approximation schemes* (PTAS).

In general, we only consider approximation algorithms which terminate in polynomial time. The reason is that simple, brute-force exponential time *exact* algorithms exist to solve any **NP**-complete problems and many interesting **NP**-hard problems. Forsaking an exactness guarantee is customarily considered acceptable if, in exchange, we can find an approximate solution in polynomial time.

PTAS, specifically, can take as much as $\phi(|\mathcal{I}|^{1/\varepsilon})$ time to terminate, where ϕ is a polynomial. Obviously, as ε becomes smaller, $1/\varepsilon$ can become very large. Those PTAS which run in polynomial time of both $|\mathcal{I}|$ and $1/\varepsilon$ are called *fully polynomial time approximation schemes*, or FPTAS.

5.2 Randomized algorithms



A randomized algorithm employs a source of randomness as part of its input. We are interested in *Monte Carlo* randomized algorithms, which may output a wrong result with bounded probability.

In order to make this statement more precise, consider a decision problem P, and an algorithm $\mathcal{R}$ which, when presented with a NO instance of P, outputs NO with certainty, but when presented with a YES instance, only outputs YES with probability $p_{\mathcal{R}} > \frac{1}{2}$. The class of all such problems is called **RP** (which stands for

"randomized polynomial time").

How can $\mathcal{R}$ help solve an instance $\mathcal{I}$ of a problem P? You simply run $\mathcal{R}$ many times. For example, suppose you run it m times on $\mathcal{I}$. If you get m NO answers, the probability that the answer is YES is the same probability $\bar{p}$ that $\mathcal{R}$ gets the answer wrong m times. Since the probability of $\mathcal{R}(\mathcal{I})$ being wrong is $1 - p_{\mathcal{R}} < \frac{1}{2}$, we have

$$\bar{p} < \frac{1}{2^m},$$

which can be driven down to zero as closely as we want.[8] By contrast, if $\mathcal{R}(\mathcal{I}) =$ YES on at least one run, then $\mathcal{I}$ must be necessarily a YES instance, for if it were NO, then $\mathcal{R}$ would output NO with certainty.

[8] Note that we could replace $\frac{1}{2}$ with any other $\delta < 1$, and still get $\lim_{m\to\infty} \delta^m$. In this sense, taking $\delta = 0.5$ is arbitrary.

5.2.1 *Randomized approximation algorithms*

However, in this chapter we discuss Max Cut, which is a maximization problem rather than a decision problem. A randomized algorithm $\mathcal{R}$ in this setting would simply yield an optimum with a certain probability $p_{\mathcal{R}} > 0$. Unfortunately, the above argument no longer works: since, in general, we do not know the optimal objective function value, we cannot tell whether the solution provided by $\mathcal{R}$ is an optimum or not. Even if we go through m independent runs of $\mathcal{R}$ and take the largest value, we cannot say much about global optimality guarantees.

We can, however, exploit an upper bound U to the optimal objective function value f^*: if we can prove that $\mathcal{R}$ yields an average objective function value $\bar{f} = \delta U$ (for some $0 < \delta \leq 1$), then, because $U \geq f^*$, it follows that

$$\bar{f} \geq \delta f^*.$$

This means that, on average, $\mathcal{R}$ performs at least as well as a δ-approximation algorithm.

For minimization problems we can exploit a lower bound L to f^*, and prove that the average objective function value of $\mathcal{R}$ is $\bar{f} = \Delta L$ with $\Delta \geq 1$. By $L \leq f^*$ it follows that $\bar{f} \leq \Delta f^*$, which means that $\mathcal{R}$ performs, on average, at least as well as a Δ-approximation algorithm.

Finally, a randomized optimization algorithm which performs at least as well as a given approximation algorithm on average is a *randomized approximation algorithm*.

Figure 5.2: Karger's randomized algorithm for Min Cut contracts edges at random until only one edge remains — which corresponds to a cut in the original graph. This extremely simple algorithm returns a minimum cut with probability $2/(n(n-1))$ (where $n = |V|$), which is much better than the one obtained by sampling a cut at random (from Wikipedia). Unlike Max Cut, however, a polytime algorithm for Min Cut is known (Which one?) — this goes to show that randomized algorithms can sometimes also be useful for problems which we can solve "efficiently" (it all depends how one defines this term!). Write a Python code to evaluate whether the Karger's algorithm is better or worse than a deterministic Min Cut algorithm, on a significant test set.

5.2.1 Example

Consider the following (trivial[9]) randomized approximation algorithm $\mathcal{R}$ for the unweighted[10] version of Max Cut*: for each $i \in V(G)$, let $i \in S$ with probability 0.5. Let $\mathcal{R}(G)$ denote the cutset size corresponding to the cut S determined by $\mathcal{R}$, and $\mathsf{Opt}(G)$ be the size of the maximum cutset. We show that*

$$\mathbb{E}(\mathcal{R}(G)) \geq \frac{1}{2}\mathsf{Opt}(G),$$

[9] This algorithm is trivial because it does not even look at the edges of the graph, which actually determine the maximum cutset.

[10] I.e. Max Cut where all weights are one.

where $\mathbb{E}$ *is the expectation of* $\mathcal{R}(G)$ *over an infinite number of runs:*

$$\begin{aligned}\mathbb{E}(\mathcal{R}(G)) &= \sum_{\{i,j\}\in E(G)} \mathsf{Prob}((i \in S \wedge j \notin S) \vee (i \notin S \wedge j \in S)) \\ &= \sum_{\{i,j\}\in E(G)} \frac{1}{2} \\ &= \frac{1}{2}|E| \geq \frac{1}{2}\mathsf{Opt}(G),\end{aligned}$$

where the second line follows from the first because the only other possibilities for i, j *are* $(i \in S \wedge j \in S) \vee (i \notin S \wedge j \notin S)$*, and they are equally likely, and the third line follows because no cutset can have size larger than the total number of edges.* □

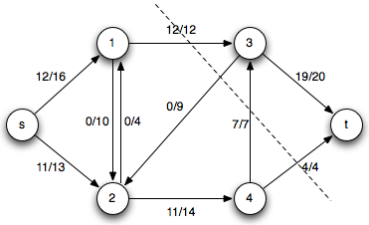


Figure 5.3: You should be all familiar with the MAX FLOW = MIN CUT theorem, which states that the maximum flow (see Sect. 1.7.2) in a directed graph with given arc capacities is the same as the capacity of a *minimum* cut. Explain why we cannot solve MAX CUT on an undirected graph by simply replacing each edge by two antiparallel arcs with capacities equal to the negative weights, and then solving a MAX FLOW LP to find the minimum cut. (Picture from `faculty.ycp.edu`).

5.3 MP formulations

Let us see a few MP formulations for the MAX CUT problem.

5.3.1 A natural formulation

We start with a "natural" formulation, based on two sets of decision variables, both for cut and cutset.

- Let V be the vertex set and E the edge set.
- Let $w : E \to \mathbb{R}_+$ be the edge weight parameters.
- For each $\{i,j\} \in E$ let $z_{ij} = 1$ if $\{i,j\}$ is in the maximum cutset, and 0 otherwise (decision variables).
- For each $i \in V$, let $y_i = 1$ if $i \in S$, and 0 otherwise (decision variables).
- The objective function is

  $$\max_{s,z\in\{0,1\}} \sum_{\{i,j\}\in E} w_{ij}z_{ij},$$

  which maximizes the size of the cutset relative to S.
- The constraints link the meaning of z and s variables:

  $$\forall\{i,j\} \in E \quad z_{ij} = y_i(1-y_j) + (1-y_i)y_j.$$

  Note that the RHS can only be zero or one. If i, j are both in S or not in S, we have $y_i = y_j = 1$ or $y_i = y_j = 0$, so the RHS evaluates to zero, which implies that the edge $\{i,j\}$ is not counted in the maximum cutset. If $i \in S$ and $j \notin S$, then the first term of the RHS is equal to 1, and the second is equal to 0, so the RHS is equal to 1, which implies that the edge $\{i,j\}$ is counted. Similarly for the remaining case $i \notin S$ and $j \in S$.

This is a Binary Quadratically Constrained Program (BQCP), which we have not really considered in past chapters. Expanding the RHS of the constraints yields

$$y_i - y_iy_j + y_j - y_iy_j = y_i + y_j - 2y_iy_j,$$

which implies that the quadratic part only arises because of a product between two binary variables, which we can linearize exactly by Sect. 3.2.1. So we can write this BQCP as a BLP and solve it using CPLEX.

5.3.2 A BQP formulation

In the formulation of Sect. 5.3.1 we can simply replace the occurrence of z_{ij} in the objective function by the RHS $y_i + y_j - 2y_iy_j$, and obtain:

$$\max_{y\in\{0,1\}^n} \sum_{\{i,j\}\in E} w_{ij}(y_i + y_j - 2y_iy_j), \tag{5.1}$$

where $n = |V|$, which is a BQP.

5.3.3 Another quadratic formulation

We are going to exploit the following equation:

$$\forall y \in \{0,1\}^n \quad 2(y_i + y_j - 2y_iy_j) = 1 - (2y_i - 1)(2y_j - 1),$$

which allows us to rewrite Eq. (5.1) as

$$\max_{y\in\{0,1\}^n} \frac{1}{2} \sum_{\{i,j\}\in E} w_{ij}(1 - (2y_i - 1)(2y_j - 1)).$$

Now, if $y_i \in \{0,1\}$ for any $i \in V$, then $x_i = 2y_i - 1$ is a variable which ranges in $\{-1,1\}$, since $x_i = 1$ if $y_i = 1$ and $x_i = -1$ if $y_i = 0$. Thus, we can formulate the Max Cut problem as:

$$\max_{x\in\{-1,1\}^n} \frac{1}{2} \sum_{\{i,j\}\in E} w_{ij}(1 - x_ix_j). \tag{5.2}$$

Note that Eq. (5.2) is also a natural formulation (see Fig. 5.4): for each $i \in V$, let $x_i = 1$ if $i \in S$ and $x_i = -1$ if $i \in V \setminus S$. Then, for each $\{i,j\} \in E$, $x_ix_j = -1$ if and only if $\{i,j\}$ is in the maximum cutset, and the weight of the corresponding edge is $\frac{1}{2}w_{ij}(1 - (-1)) = w_{ij}$.

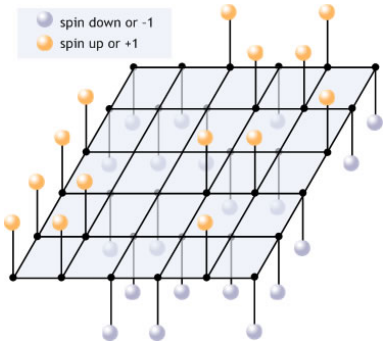


Figure 5.4: The Ising model of a spin glass (from http://www3.nd.edu/~mcbg). There are n magnetic impurities (atoms, represented by graph vertices V) in an external magnetic field, represented by a dummy vertex in V. For each $i < j$ in V the magnetic interaction is J_{ij}. The model dictates that the spin $x_i \in \{-1,1\}$ of each atom i is such that the energy of the system, given by $-\sum_{i<j} J_{ij}x_ix_j$, is minimum. A few further steps transform this to a Max Cut problem.

5.3.4 An SDP relaxation

In this section we derive a well-known SDP relaxation of the Max Cut problem. The strategy is as follows: we reformulate the problem exactly to a matrix problem with a rank constraint, which we then relax.

First, we slightly change the input of the problem: instead of a simple, undirected, edge-weighted graph $G = (V,E)$, we consider a complete simple directed graph $\bar{G} = (V,A)$, where $w_{ij} = w_{ji} = 0$ for each $\{i,j\} \notin E$. Since we are adding the weight w_{ij} twice (every edge $\{i,j\} \in E$ is replaced by two antiparallel arcs (i,j) and (j,i) in A), we have to multiply the objective function by another factor $\frac{1}{2}$. This means we can write another quadratic Max Cut formulation as follows:

$$\max_{x\in\{-1,1\}^n} \frac{1}{4} \sum_{i,j\in V} w_{ij}(1 - x_ix_j). \tag{5.3}$$

We now carry out a reformulation which we have never seen before, and which might appear very weird: we replace each scalar decision variable x_i (for $i \in V$) by a vector of decision variables $v_i \in \mathbb{S}^{n-1}$, i.e. $v_i \in \mathbb{R}^n$ and $\|v_i\|_2 = 1$. Every product $x_i x_j$ is replaced by the scalar product $v_i \cdot v_j$, thereby obtaining:

$$\max_{v \in \mathbb{S}^{n-1}} \frac{1}{4} \sum_{i,j \in V} w_{ij}(1 - v_i \cdot v_j). \tag{5.4}$$

Note that any solution to Eq. (5.4) is an $n \times n$ real symmetric matrix V having v_i as its i-th column. It should be easy to see that this is not an exact reformulation. We can make it exact, however, by enforcing the condition that, for any solution V, there exists a column vector u, with each component in $\mathbb{S}^0$, such that $uu^\top = VV^\top$. If this is the case, then for each $i \neq j$ we have $v_i^\top v_j = v_i \cdot v_j = u_i u_j$, and since $\mathbb{S}^0 = \{-1, 1\}$, we have recovered Eq. (5.4).

By Prop. 5.3.1, it suffices that $\mathsf{rank}\, V = 1$.

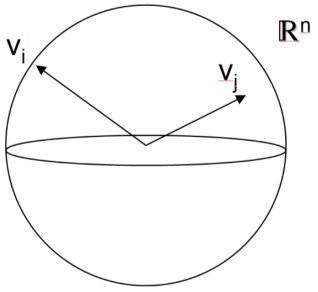


Figure 5.5: Replacing a scalar variable with a vector variable gives more degrees of freedom, so the feasible region is enlarged. In other words, it is a relaxation of the original problem (from some lecture notes of R. O'Donnell).

5.3.1 Proposition
For any $n \times n$ real symmetric matrix V, $\mathsf{rank}\, V = 1$ if and only if there exists a vector $u \in \mathbb{R}^n$ such that $uu^\top = VV^\top$.

Proof. Suppose V has rank 1. This happens if and only if every column is a scalar multiple of a single column, i.e. we can write V as $(u_1 t \; u_2 t \; \cdots \; u_n t)$ for some column vector $t \in \mathbb{R}^n$ which we can choose as having unit Euclidean norm without loss of generality.[11] In turn, this is equivalent to stating that the (i,j)-th component of $VV^\top$ is $u_i u_j \|t\|_2^2 = u_i u_j$, which happens if and only if $VV^\top = uu^\top$. The converse direction is trivial.[12] □

[11] Why is there no loss of generality in assuming $\|t\|_2 = 1$?

[12] Why? Prove it!

We now consider the Gram matrix of V (see Sect. 2.10), denoted $\mathsf{Gram}(V)$, i.e. the $n \times n$ matrix having $v_i \cdot v_j$ as its (i,j)-th component. The Gram matrix has two remarkable properties, given in Prop. 5.3.2 below.

5.3.2 Proposition
For any real $n \times n$ symmetric matrix V, we have: (i) $\mathsf{Gram}(V) \succeq 0$; (ii) $\mathsf{rank}\, \mathsf{Gram}(V) = \mathsf{rank}\, V$.

Proof. First, we remark that $\mathsf{Gram}(V) = V^\top V = VV^\top$.

(i) For any $x \in \mathbb{R}^n$, we have

$$x^\top \mathsf{Gram}(V) x = x^\top V^\top V x = (Vx)^\top Vx = \|Vx\|_2 \geq 0,$$

implying $\mathsf{Gram}(V) \succeq 0$, i.e. the Gram matrix is PSD.[13]

(ii) For any matrices A, B we know that the columns of AB are linear combinations of the columns of A, and the rows of AB are linear combinations of the rows of A: hence, the columns of $\mathsf{Gram}(V) = V^\top V$ are a linear combination of the columns of $V^\top$, i.e. the rows of V, and the rows of $\mathsf{Gram}(V)$ are a linear combination of the rows of V. Since the row rank and column rank of any matrix coincide, $\mathsf{Gram}(V)$ has the same rank as V, as claimed. □

[13] The converse also holds: any PSD matrix is the Gram matrix of some other matrix. Prove it.

5.3.3 Corollary
If $X = (X_{ij}) = \mathsf{Gram}(V)$ *and* $\mathsf{rank}\, X = 1$, *then there exists a vector* u *such that* $X_{ij} = u_i u_j$.

Proof. Since $\mathsf{rank}\, X = 1$ and X is the Gram matrix of V, then $\mathsf{rank}\, V = 1$ by Prop. 5.3.2, which, by Prop. 5.3.1 implies the existence of u such that $uu^\top = VV^\top = \mathsf{Gram}(V) = X$. □

By Cor. 5.3.3, we can reformulate Eq. (5.4) as:

$$\left.\begin{array}{lrcl} \max & \frac{1}{4}\sum\limits_{i,j\in V} w_{ij}(1-X_{ij}) & & \\ & \mathsf{Diag}(X) & = & I \\ & X & \succeq & 0 \\ & \mathsf{rank}\, X & = & 1, \end{array}\right\} \tag{5.5}$$

where $\mathsf{Diag}(X)$ is the matrix consisting of the diagonal entries of X, and I is the identity matrix.

Lastly, we look at the objective function of Eq. (5.5) more closely:

$$\begin{aligned} & \sum_{i,j\in V} w_{ij}(1-X_{ij}) \\ = \; & \sum_{i,j\in V} w_{ij}1 - \sum_{i,j\in V} w_{ij}X_{ij} \\ \overset{\mathsf{Diag}(X)=1}{\Rightarrow \forall i\in V\; X_{ii}=1} \; = \; & \sum_{i,j\in V} w_{ij}X_{ii} - \sum_{i,j\in V} w_{ij}X_{ij} \\ = \; & \sum_{i\in V}\left(\sum_{j\in V} w_{ij}\right)X_{ii} - A\bullet X \\ = \; & D\bullet X - A\bullet X \\ = \; & (D-A)\bullet X, \end{aligned}$$

where $A = (w_{ij})$ is the weighted adjacency matrix of the input digraph $\bar{G}$, and D is the matrix having $\sum\limits_{j\in V} w_{ij}$ on the i-th diagonal element, and 0 elsewhere. Note that the matrix $L = D - A$ is commonly known as the *weighted Laplacian* of the graph[14] G.

[14] Why is the weighted Laplacian defined on *undirected* graphs, yet we refer it to an input which is a directed graph?

Our matrix formulation of the Max Cut is therefore:

$$\left.\begin{array}{lrcl} \max & \frac{1}{4}L\bullet X & & \\ & \mathsf{Diag}(X) & = & I \\ & X & \succeq & 0 \\ & \mathsf{rank}\, X & = & 1. \end{array}\right\} \tag{5.6}$$

An SDP relaxation of Eq. (5.6) can now be easily derived by dropping the constraint $\mathsf{rank}\, X = 1$:

$$\left.\begin{array}{lrcl} \max & \frac{1}{4}L\bullet X & & \\ & \mathsf{Diag}(X) & = & I \\ & X & \succeq & 0. \end{array}\right\} \tag{5.7}$$

As mentioned in Sect. 2.10, SDPs can be solved in polynomial time to any desired ε accuracy by means of IPM.

5.4 The Goemans-Williamson algorithm

Since SDPs can be solved in polytime to any desired accuracy,[15] one could dream of an ideal world where one can just efficiently solve the SDP relaxation of any instance $\mathcal{I}$ of MAX CUT, and the solution X^* simply happens to have rank 1: then we can just pick the correct vector x: since, as remarked, $\|x_i\|_2 = 1$ reduces to $x_i \in \{-1, 1\}$ for each $i \leq n$, we can simply read the maximum cutset off the SDP solution. We live, alas, in an imperfect world where relaxations are often inexact, however. The main innovation of the Goemans-Williamson algorithm (GW) is a randomized algorithm for finding a decently-sized cutset from X^*. This algorithm is often called "randomized rounding", since some X^*_{ij} might well be fractional in Eq. (5.7), whereas we would need it to be either 1 or -1 in order for X^* to represent a valid solution of Eq. (5.2).

In summary, the GW is as follows:

1. Solve the SDP in Eq. (5.7) to find X^*
2. Perform "randomized rounding" to find $x' \in \{-1, 1\}^n$ representing a "good" cut.

By "good", the GW algorithm means that, on average, the randomized rounding algorithm will yield a cutset having size $\alpha \, \mathsf{Opt}(\mathcal{I})$, where $\alpha > 0.878$. Since 0.878 is quite close to 1, the GW is, at least theoretically, a pretty good randomized approximation algorithm.

[15] We remark that there exists an IPM for SDP which converges in $O(|\mathcal{I}|^{3.5} \log(1/\varepsilon))$ [Boyd and Vandenberghe, 2004], where $|\mathcal{I}|$ is the size of the SDP instance. Does this make IPM an FPTAS? It would appear so, but no-one mentions it in the literature. As far as I can tell, it is because the ε tolerance is not only applicable to optimality, but also to feasibility, i.e. IPMs for SDP might yield a solution with eigenvalues $\geq -\varepsilon$, or satisfying the constraints with a ε error. The definition of FPTAS applies the approximation to the objective function, but requires strict feasibility.

5.4.1 Randomized rounding

Consider the SDP relaxation solution X^*: since it is a symmetric matrix, it has a real factorization $P^\top \Lambda P$, where Λ has all of the eigenvalues of X^* on the diagonal (and zero elsewhere), and P is the unitary matrix formed by the corresponding eigenvectors, arranged column-wise. Since X^* is PSD, $\Lambda \geq 0$, which means that the square root matrix $\sqrt{\Lambda}$ is real. So

$$X^* = P^\top \Lambda P = P^\top \sqrt{\Lambda}^\top \sqrt{\Lambda} P = (\sqrt{\Lambda} P)^\top \sqrt{\Lambda} P = V^\top V,$$

where $V = \sqrt{\Lambda} P$, with v_i being the i-th column of V.

Note that, since X^* satisfies $\mathsf{Diag}(X^*) = 1$ (i.e. $X^*_{ii} = 1$ for each $i \leq n$), we have $X^*_{ii} = v_i \cdot v_i = \|v_i\|_2 = 1$, which implies that the vectors v_i are on $\mathbb{S}^{n-1}$. Our job would be over if we could "wisely" choose a hyperplane H, passing through the origin, which separates the vectors v_i into two parts: on the one side, say whenever $i \in S'$, we set all i's such that $x'_i = 1$, and on the other side we put the rest, namely i's with $x'_i = -1$. By "wisely", we mean of course that this magical hyperplane should yield a MAX CUT solution x' which is at least as good as α times the maximum cutset.

Geometrically, let r be the (unit) normal vector to the magical hyperplane: then those vectors v_i which form an angle ξ between -90 and 90 degrees with H will be in S', and those which form an angle ξ between 90 and 270 degrees with H will not be in S'. The

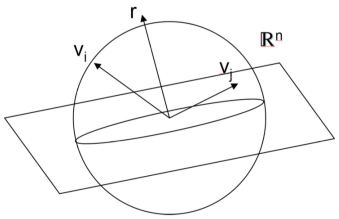


Figure 5.6: Factorizing the SDP solution yields a bunch of vectors on the spherical surface (from some lecture notes of R. O'Donnell).

algebraic connection between an angle and two vectors is the scalar product: specifically,

$$\forall i \leq n \quad r \cdot v_i = \|r\|_2 \|v_i\|_2 \cos(\xi) = \cos(\xi),$$

since r and all of the v_i's are unit vectors. Thus, $i \in S'$ if and only if $r \cdot v_i \geq 0$, which implies that we can define

$$x' = \mathsf{sgn}(r^\top V),$$

with the convention that $\mathsf{sgn}(0) = 1$.

Now, the average value of the cutset size obtained by randomized rounding is:

$$\mathbb{E}\left(\sum_{\substack{i \in S' \\ j \notin S'}} w_{ij}\right) = \sum_{i<j} w_{ij} \mathsf{Prob}(x'_i \neq x'_j). \tag{5.8}$$

So we need to evaluate the probability that, given a vector r picked uniformly at random on $\mathbb{S}^{n-1}$, its orthogonal hyperplane H forms angles as specified above with two given vectors v_i and v_j.

Since we are only talking about two vectors, we can focus on the plane η spanned by those two vectors. Obviously, the projection of $\mathbb{S}^{n-1}$ on η is a circle C centered at the origin. We look at the line λ on η (through the origin) spanned by r. The question, much simpler to behold now that it is in 2D, becomes: *what is the probability that a line crossing a circle separates two given radii v_i, v_j?*

This question really belongs to elementary geometry (see Fig. 5.7). In order to separate v_i, v_j, the radius defined by λ has to belong to the smaller slice of circle delimited by v_i and v_j. The probability of this happening is the same as the extent of the (smaller) angle between v_i and v_j divided by 2π. Only, λ really identifies a diameter, so we could turn λ around by π and get "the other side" of λ to define another radius. So we have to multiply this probability by a factor of two:

$$\forall i < j \quad \mathsf{Prob}(x'_i \neq x'_j) = \frac{\arccos(v_i \cdot v_j)}{\pi}. \tag{5.9}$$

Since we do not know $\mathsf{Opt}(\mathcal{I})$, in order to evaluate the approximation ratio of the GW we can only compare the objective function value of the rounding x' with the SDP relaxation value, which, as an upper bound to $\mathsf{Opt}(\mathcal{I})$, is such that

$$\frac{1}{2}\sum_{i<j} w_{ij}(1 - x'_i x'_j) \leq \mathsf{Opt}(\mathcal{I}) \leq \frac{1}{2}\sum_{i<j} w_{ij}(1 - v_i \cdot v_j). \tag{5.10}$$

By Eq. (5.8)-(5.9), the average value of $\frac{1}{2}(1 - x'_i x'_j)$ over all $i < j$ is $\frac{1}{\pi}(\arccos(v_i \cdot v_j))$. Now we look at Eq. (5.10) term-wise: how much worse can $\arccos(v_i \cdot v_j)/\pi$ be with respect to $(1 - v_i \cdot v_j)/2$?

Let us look at $v_i \cdot v_j$ as a quantity ρ, and plot[16] the two functions

$$\frac{1}{\pi}\arccos\rho \qquad \text{and} \qquad \frac{1}{2}(1-\rho),$$

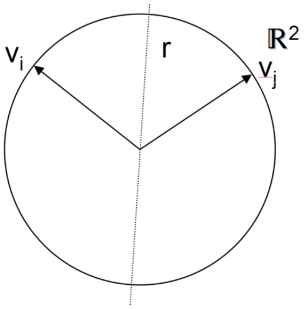


Figure 5.7: For every pair of vectors, we can look at the situation in 2D (from some lecture notes of R. O'Donnell).

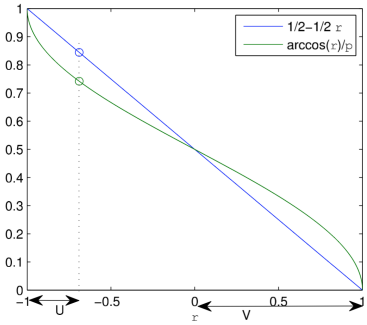


Figure 5.8: When rounding is worse than SDP: how much worse? (From some lecture notes of R. O'Donnell)

[16] See Fig. 5.8.

for $\rho \in [-1, 1]$. Note that for $\rho \geq 0$, $\frac{1}{\pi} \arccos \rho \geq \frac{1}{2}(1-\rho)$, which implies that randomized rounding performs better than the SDP solution. For $\rho < 0$ this is no longer the case. How bad can randomized rounding perform with respect to SDP? The minimum of the gap between the two functions has value:

$$\alpha = \min_{\rho \in [-1,1]} \frac{(\arccos \rho)/\pi}{(1/2)(1-\rho)} \approx 0.87854.$$

This means that, on each pair $i, j \in V$, randomized rounding can do no worse that α times the solution of the SDP. Hence:

$$\frac{1}{4}\mathbb{E}\left(\sum_{i,j \in V} w_{ij}(1 - x'_i x'_j)\right) \geq \frac{\alpha}{4} \sum_{i,j \in V} w_{ij}(1 - X^*_{ij}) \geq \alpha \, \mathsf{Opt}(\mathcal{I}), \quad (5.11)$$

as claimed.

5.4.2 *Sampling a uniform random vector from* $\mathbb{S}^{n-1}$

It only remains to discuss a method for sampling uniform random vectors from the surface of a unit sphere in $\mathbb{R}^n$. A very well known method consists in sampling n independent values $r'_1, \ldots, r'_n$ from a standard normal distribution $N(0, 1)$, then letting $r' = (r'_1, \ldots, r'_n)^\top$ and $r = \frac{r'}{\|r'\|_2}$. By the lemma below, r is uniformly distributed on $\mathbb{S}^{n-1}$.

5.4.1 Lemma
The vector r is uniformly distributed on $\mathbb{S}^{n-1}$.

Proof. Since $r'_1, \ldots, r'_n$ are independently sampled, their joint probability distribution function is simply the product of each individual distribution function:

$$\prod_{i \leq n} \frac{1}{\sqrt{2\pi}} e^{(r'_i)^2/2} = \frac{1}{(2\pi)^{n/2}} e^{-\|r'\|_2/2}.$$

The probability that r' is in some measurable set A is:

$$p_A = \frac{1}{(2\pi)^{n/2}} \int_A e^{-\|r'\|_2/2} dr'.$$

Now consider any rotation matrix U (such that $U^\top U = I$) applied to A. The probability that r' is in $U^\top A$ is:

$$\begin{aligned}
p_{UA} &= \frac{1}{(2\pi)^{n/2}} \int_{U^\top A} e^{-\|r'\|_2/2} dr' \\
&= \frac{1}{(2\pi)^{n/2}} \int_A e^{-\|U^\top r'\|_2/2} dr' \\
&= \frac{1}{(2\pi)^{n/2}} \int_A e^{-\|U^\top U\|_2 \|r'\|_2/2} dr' \\
&= \frac{1}{(2\pi)^{n/2}} \int_A e^{-\|I\|_2 \|r'\|_2/2} dr' \\
&= \frac{1}{(2\pi)^{n/2}} \int_A e^{-\|r'\|_2/2} dr' = p_A.
\end{aligned}$$

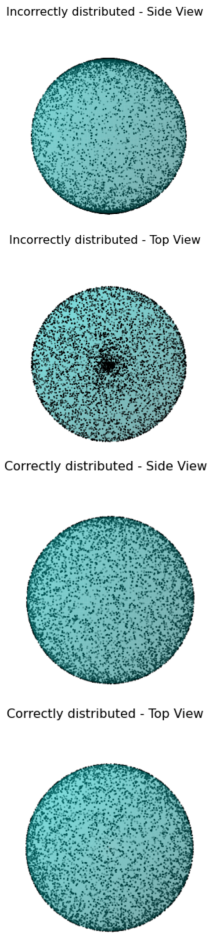


Figure 5.9: Most "wild guess methods" for uniformly sampling on a sphere turn out to be incorrect (above) or, if correct (below), inefficient (from www.bogotobogo.com).

Thus, if $s \neq t$ are points in $\mathbb{S}^{n-1}$, the probability that $r' = s$ is the same as the probability that $r' = t$. Hence r, the normalized version of r', is uniformly distributed on the unit sphere, as claimed. □

5.4.3 *What can go wrong in practice?*

We said at the beginning of this chapter that GW is an approximation algorithm which actually works in practice. We shall see this in Sect. 5.5. One word of caution, however: as mentioned in Sidenote 15, it is possible that SDP solvers would find slightly infeasible solutions, due to their error tolerance, which is in turn due to floating point computations. So X^* might have some slightly negative eigenvalues, or not satisfy $\mathsf{Diag}(X^*) = 1$ exactly. This has to be dealt with at the implementation level, rather than algorithmically.

For example, the closest PSD matrix A to a given symmetric matrix B can be obtained by factorization: let $B = P^\top \Lambda P$, where Λ is a diagonal matrix with the eigenvalues (ordered from largest to smallest) on the diagonal, and the columns of P are the corresponding eigenvectors. If $\Lambda_{ii} \geq 0$ for all $i \leq n$, we have a certificate that B is PSD. However, if there is $\ell \leq n$ such that $\Lambda_\ell < 0$ (even though perhaps $|\Lambda_\ell|$ is tiny), then we define Λ^+ as Λ, with all eigenvalues from the ℓ-th to the n-th replaced by zeros. We then compute[17] $A = P^\top \Lambda^+ P$.

Since GW really only relies in the factorization of X^* into $V^\top V$, you can also address these issues by designing a factorization algorithm which is robust to slightly negative eigenvalues in X^*. Most implementations (e.g. in NumPy or Matlab) are not robust in this sense.

[17] Note that, since this computation also involves floating point operations, some error might make some eigenvalue of A slightly negative: this is a game of hide-and-seek, where you have to artfully tune the number of very small eigenvalues which you zero in Λ^+.

5.5 *Python implementation*

We now present and discuss a Python program which solves:

- the MIQP formulation in Eq. (5.1), using Pyomo calling CPLEX[18];
- the MILP formulation derived from applying the binary variable product reformulation (Sect. 3.2.1) to Eq. (5.1), using Pyomo calling CPLEX;
- the SDP relaxation in Eq. (5.7) using the alternative[19] Picos MP modelling environment in Python, which interfaces to SDP solvers (through the Python library CvxOpt);
- the Max Cut problem using our own implementation of the GW algorithm.

[18] Note that the objective function in Eq. (5.1) is quadratic: even though CPLEX excels at solving MILPs, it can solve convex MIQPs quite well (including those with convex nonlinear constraints). I know, I know: the objective of Eq. (5.1) is not even convex! Ever since version 12.6, CPLEX has been able to solve also nonconvex MIQPs with nonconvex objective functions (the constrains have to be convex quadratic, however), by means of a special-purpose sBB-type solver.

[19] We need an alternative because Pyomo currently does not interface to SDP solvers.

5.5.1 *Preamble*

We import the following libraries:

```
import time
import sys
```

```
import math
import numpy as np
from pyomo.environ import *
from pyomo.opt import SolverStatus, TerminationCondition
import pyomo.opt
import cvxopt as cvx
import cvxopt.lapack
import picos as pic
import networkx as nx
```

We first discuss the standard libraries. We use the `time` library to collect CPU timings in order to perform an empirical comparison of the different solution approaches. The `sys` library allows us to pass a command line argument to our Python program. The `math` library gives direct access to functions such as `sqrt`, which computes the square root of its argument. The `numpy` library is necessary for many floating point computations, including linear algebra; note we import `numpy` as the shortened string `np`, which makes our code somewhat neater.

We also import some specific MP-related libraries. We already saw the import sequence for Pyomo. We then import the `cvxopt` library and its `cvxopt.lapack` linear algebra module. We do not use these libraries explicitly, but they are used by `picos`, the MP python modelling environment which is specifically suitable for convex optimization, and can interface with SDP and SOCP solvers. Finally, we store and manipulate graphs using the `networkx` library, which we import as the shortened string `nx`.

We rely on a few globally defined variables:

```
GWitn = 10000
MyEpsilon = 1e-6
cplexTimeLimit = 100
```

`GWitn` is the maximum number of iterations of the randomized rounding phase of the GW algorithm. `MyEpsilon` is used as a floating point error tolerance. `cplexTimeLimit` is the maximum number of seconds of user[20] CPU that CPLEX is allowed to run for. Note that CPLEX, as well as many other MP solvers, does not check the system clock very often, probably in an attempt to devote every single CPU cycle to finding the solution efficiently. As a result, even if you set a time limit of 100 seconds, CPLEX might well go on crunching numbers for considerably more time.

[20] CPU timings in most operating systems (OS) nowadays are reported as either *user*, *system* or *total*. The former is the actual amount of seconds that the CPU devotes to running the specific given task being monitored, whereas the latter is *user+system*. Usually, people are mostly interested in user CPU time, since the time the computer spends doing something else is not relevant to code efficiency — one might consider devoting a whole computer to running the relevant task in single-user mode. With the advent of multi-core computers, these time statistics have become fuzzier: is the time taken to transfer control or register data from one core to another classified as system or user? After all the parallelized code would not work without these control and data transfers. I believe different OSes employ different definitions of user and system times in multi-core computers.

5.5.2 *Functions*

Next, we briefly introduce the functions we use. In `create_maxcut_miqp` we take an undirected graph G (in the form of a `networkx` object) and use the Pyomo library to model a MIQP formulation of MAX CUT.

```
def create_maxcut_miqp(G):
  maxcut = ConcreteModel()
  # 0-1 vars: 0 if node in S, 1 if in \bar{S}
  maxcut.y = Var(G.nodes(), within = Binary)
```

```
  # objective: max sum_{ij in E} w_ij * (y_i + y_j - 2 y_i*y_j)
  def max_cut(model):
    return sum(G[i][j]['weight'] * (model.y[i] + model.y[j] - \
           2*model.y[i]*model.y[j]) for (i,j) in G.edges())
  maxcut.maxcutobj = Objective(rule = max_cut, sense = maximize)
  return maxcut
```

In `create_maxcut_milp` we do the same, but we add Fortet's inequalities to MIQP formulation, and linearize the products $y_i y_j$, replacing each bilinear term with a new variable z_{ij} (Sect 3.2.1).

```
def create_maxcut_milp(G):
  maxcut = ConcreteModel()
  # 0-1 vars: 0 if node in S, 1 if in \bar{S}
  maxcut.y = Var(G.nodes(), within = Binary)
  # linearization variables z_ij = y_iy_j
  maxcut.z = Var(G.edges(), within = PercentFraction)
  # objective: max sum_{ij in E} w_ij * (y_i + y_j - 2 z_ij)
  def max_cut(model):
    return sum(G[i][j]['weight'] * (model.y[i] + model.y[j] - \
           2*model.z[i,j]) for (i,j) in G.edges())
  maxcut.maxcutobj = Objective(rule = max_cut, sense = maximize)
  # Fortet's linearization constraints
  def fortet1(model, i,j):
    return model.z[i,j] <= model.y[i]
  maxcut.fortet1 = Constraint(G.edges(), rule = fortet1)
  def fortet2(model, i,j):
    return model.z[i,j] <= model.y[j]
  maxcut.fortet2 = Constraint(G.edges(), rule = fortet2)
  def fortet3(model, i,j):
    return model.z[i,j] >= model.y[i] + model.y[j] - 1
  maxcut.fortet3 = Constraint(G.edges(), rule = fortet3)
  return maxcut
```

For the SDP relaxation of MAX CUT, we employ a different MP modelling environment in Python, namely Picos. Note that the argument of `create_maxcut_sdp` is not the graph, but its Laplacian matrix. You can hopefully make out what the various instructions do by the comments.

```
def create_maxcut_sdp(Laplacian):
  n = Laplacian.shape[0]
  # create max cut problem SDP with picos
  maxcut = pic.Problem()
  # SDP variable: n x n matrix
  X = maxcut.add_variable('X', (n,n), 'symmetric')
  # constraint: ones on the diagonal
  maxcut.add_constraint(pic.tools.diag_vect(X) == 1)
  # constraint: X positive semidefinite
  maxcut.add_constraint(X >> 0)
  # objective function
  L = pic.new_param('L', Laplacian)
  # note that A|B = tr(AB)
  maxcut.set_objective('max', L|X)
  return maxcut
```

In the GW algorithm the output of the SDP relaxation phase is X^*, but the input to the randomized rounding phase is a matrix factor V of X^*. The `rank_factor` function computes this factor.

Note that we try and bound the "negative zero" eigenvalues due to floating point errors.

```
def rank_factor(X, rk):
  n = X.shape[0]
  evl, evc = np.linalg.eigh(X)       # extract eigeninfo
  # get closest nonneg eigenvals (small neg number --> 0)
  evl[evl < 0] = 0
  if rk < n:
    V = np.transpose(evc[:,-rk:])
    for k in range(rk):
      V[k] *= math.sqrt(evl[-rk-1])
  else:
    V = np.transpose(evc[::-1])
    for k in range(n):
      V[k] *= math.sqrt(evl[::-1][k])
  return V
```

The function `closest_sdp` removes small negative eigenvalues from a matrix X which should be PSD. We factor X it using eigenvectors and eigenvalues, we replace negative eigenvalues by zeros, then we multiply the factor and its transpose to retrieve an updated matrix X which is PSD. This somehow replicates some of the instructions found in the function `rank_factor` above. The point is that, as emphasized in Sect. 5.4.3, multiplying the factors involves many floating point operations, which might introduce new floating point errors. Replicating some of these "cleaning" actions helps reduce the chances of an error in the call to square root functions.[21]

[21] Honestly, I'm not really sure this further "cleaning step" is wholly justified — I must have inserted this code in consequence of some negative square root error occurring in the code on some instance I was testing. Try a version of the program where you do *not* call this function, and see what happens.

```
def closest_psd(X):
  evl, evc = np.linalg.eigh(X)  # extract eigeninfo
  # get closest nonneg eigenvals (some -1e-27 --> 0)
  evl[evl < 0] = 0
  return evc.T.dot(np.diag(evl).dot(evc))
```

We now come to the randomized rounding algorithm: the function `gw_randomized_rounding` is wholly commented, so it should be easy to understand.

```
def gw_randomized_rounding(V, objX, L, iterations):
  n = V.shape[0]
  # repeat max iterations times
  count = 0
  obj = 0
  print "maxcut(GW/rnd): improv_itn objfun"
  while (count < iterations):
    r = np.random.normal(0,1,n)  # rnd unfrm vector on unit sphere
    r /= np.linalg.norm(r)       # normalization
    x = np.sign(np.dot(V,r))     # signs of V . r
    x[x >= 0] = 1                # in case there's some zeros
    o = np.dot(x.T,np.dot(L,x))  # objective function value
    if o > obj:
      x_cut = x
      obj = o
      print count, "\t", obj
    count += 1
  return (x_cut, obj)
```

5.5.3 *Main*

We now step through the instructions of the "main" part of our Python program. First, we read the input file having name given as the first argument on the comand line.

```
# read edge list file as input
if len(sys.argv) < 2:
  # not enough args on cmd line, error
  print "error: need an edge list filename on cmd line"
  quit()
# each line of input is arc (i,j) weight w: i j w
G = nx.read_weighted_edgelist(sys.argv[1], nodetype = int)
# make G undirected
G = nx.Graph(G)
```

For example, the triangle graph with unit weights is:

```
1 2 1
1 3 1
2 3 1
```

Figure 5.10: The logo of PICOS (from `picos.zib.de`).

Secondly, we solve the MIQP formulation using the standard Pyomo constructs (see Sect. 1.8.3).

```
miqp = create_maxcut_miqp(G)
solver = pyomo.opt.SolverFactory('cplexamp', solver_io = 'nl')
solver.options['timelimit'] = cplexTimeLimit
solver.options['mipdisplay'] = 2
t0 = time.time()  # start the stopwatch
results = solver.solve(miqp, keepfiles = False, tee = True)
t1 = time.time()  # stop it
miqpcpu = t1 - t0 # this is the time taken by the solver
miqp.solutions.load_from(results) # this is an indicator vector
# retrieve the set of the indicator vect. as a dictionary (key:val)
ymiqp = {i:miqp.y[i].value for i in G.nodes()}
miqpcut = [i for i,y in ymiqp.iteritems() if y == 1]
miqpobj = miqp.maxcutobj()
```

Note the use of the `time` Python library: we call `time.time()` around the `solver.solve()` command, then we take the difference of the two timings to compute the duration of `solver.solve()`.

Third, we solve the MIQP formulation using the standard Pyomo constructs (see Sect. 1.8.3).

```
milp = create_maxcut_milp(G)
t0 = time.time()
results = solver.solve(milp, keepfiles = False, tee = True)
t1 = time.time()
milpcpu = t1 - t0
milp.solutions.load_from(results)
ymilp = {i:milp.y[i].value for i in G.nodes()}
milpcut = [i for i,y in ymilp.iteritems() if y == 1]
milpobj = milp.maxcutobj()
```

Fourth, we solve the SDP relaxation. Since we modelled the SDP using Picos, the instructions are different.

```
L = np.array(1/4.*nx.laplacian_matrix(G).todense()) # compute Laplacian
sdp = create_maxcut_sdp(L) # create max cut SDP relaxation
```

```
t0 = time.time()           # start watch
sdp.solve(verbose = 0)     # solve
t1 = time.time()           # stop watch
sdpcpu = t1 - t0           # time difference
sdpobj = sdp.obj_value()   # retrieve obj fun value
Xsdp = closest_psd(np.array(sdp.get_valued_variable('X'))) # retrieve solution
```

Fifth, we run factor `Xsdp` and apply randomized rounding.

```
t0 = time.time()
N = len(G.nodes())
V = rank_factor(Xsdp, N)
(gwcut,gwobj) = gw_randomized_rounding(V, sdp.obj_value(), L, GWitn)
t1 = time.time()
gwcpu = t1 - t0
```

Lastly, we output our comparison statistics: objective function value and CPU time.

```
print "maxcut(out): MIQPobj=", miqpobj, ", MIQPcpu=", miqpcpu
print "maxcut(out): MILPobj=", milpobj, ", MILPcpu=", milpcpu
print "maxcut(out):  SDPobj=",  sdpobj, ",  SDPcpu=",  sdpcpu
print "maxcut(out):   GWobj=",   gwobj, ",   GWcpu=",   gwcpu
```

This code resides in a text file called `maxcut.py`, and can be run using the command line

```
python maxcut.py file.edg
```

where `file.edg` is the instance file with the edge list, as detailed above (two integers and a floating point number per line, where the integers are the indices of the vertices adjacent to each edge and the floating point is its weight).

5.5.4 *Comparison on a set of random weighted graphs*

I have created[22] a set of file in `.edg` format, each storing a weighted graph generated using the Erdős-Renyi random model. This is as follows: given an integer n and a real $p \in [0,1]$, generate a graph with n vertices, where each edge has probability p of being created. I set the range of n to $\{10, 20, \ldots, 90\}$, and the range of p to $\{0.3, 0.5, 0.7\}$.

I then launched the `maxcut.py` Python program on each of the files, saved the output to a similarly-named `.out` file, and then proceeded to collect[23] the output data in the table below. Note that, since the time limit for CPLEX is set to 100s, CPLEX does not have the time to solve all instances will not be solved to global optimality.

The first two columns of the table contain the generation parameters for the random graph. The following four columns contain the objective function values of MIQP, MILP, SDP and randomized rounding (RR) applied to the SDP solution. The last four columns contain the user CPU time taken to solve MIQP, MILP, SDP and perform RR (the time for RR includes factoring the output of the SDP solution). Thus, the time taken to run the GW algorithm is

[22] You should do this too! A piece of advice: don't do this by hand — use Python, or another scripting language.

[23] Again, don't do this by hand! Parse the `.out` file using a program, and output a tabular format — or simply edit the `maxcut.py` script to write its output in tabular format.

represented by the (row-wise) sum of the values in the last two columns. Remember that only MIQP, MILP and RR actually output a feasible solution for the MAX CUT (the SDP relaxation outputs a matrix which in general does not have rank 1).

It is clear that the GW makes a very reasonable concession[24] to the quality of the obtained cut, in exchange for huge savings on the CPU time. This is why I wrote that this is an approximation algorithm which actually works in practice!

[24] Is this always the case? Generate other kinds of graphs and see for yourself! For example, how about square mesh grid graphs with unit weights?

instance		objective function value				user CPU time			
n	p	MIQP	MILP	SDP	RR	MIQP	MILP	SDP	RR
10	0.3	7.82	7.82	8.38	7.82	0.1	0.0	0.1	0.2
10	0.5	9.87	9.87	10.15	9.87	0.1	0.1	0.0	0.2
10	0.7	15.09	15.09	15.20	15.09	0.1	0.1	0.0	0.2
20	0.3	28.54	28.54	28.93	27.29	0.1	0.1	0.1	0.2
20	0.5	36.55	36.55	37.94	35.40	0.2	0.2	0.1	0.2
20	0.7	44.88	44.88	46.44	44.44	0.2	0.3	0.1	0.2
30	0.3	52.80	52.80	55.14	47.82	0.3	0.5	0.1	0.2
30	0.5	74.82	74.82	77.01	68.36	0.5	1.3	0.1	0.2
30	0.7	100.49	100.49	102.67	93.31	1.9	3.4	0.1	0.2
40	0.3	85.55	85.55	89.07	75.80	1.1	2.0	0.1	0.2
40	0.5	134.52	134.52	138.69	124.54	4.4	6.4	0.1	0.2
40	0.7	175.29	175.29	179.88	162.63	46.2	100.2	0.1	0.2
50	0.3	148.99	148.99	151.83	127.68	2.6	3.8	0.1	0.2
50	0.5	206.02	206.02	212.62	182.69	21.6	33.7	0.1	0.2
50	0.7	263.75	263.20	269.46	243.14	100.1	100.2	0.1	0.2
60	0.3	201.12	201.12	209.16	178.03	38.0	100.1	0.1	0.3
60	0.5	281.69	280.52	291.98	256.08	100.1	100.2	0.2	0.3
60	0.7	370.38	367.92	382.96	348.29	100.1	100.3	0.2	0.2
70	0.3	253.52	252.85	262.39	223.67	100.1	100.2	0.1	0.3
70	0.5	394.33	383.24	403.63	352.04	100.1	100.3	0.1	0.3
70	0.7	497.64	487.58	509.99	461.83	100.1	100.4	0.2	0.3
80	0.3	309.52	309.52	321.58	276.06	100.1	100.2	0.1	0.3
80	0.5	490.93	493.07	514.59	455.93	100.1	100.3	0.2	0.4
80	0.7	648.89	628.01	665.94	603.25	100.2	100.5	0.2	0.3
90	0.3	405.39	400.41	424.94	355.79	100.1	100.2	0.2	0.3
90	0.5	608.89	590.78	631.28	560.55	100.1	100.5	0.2	0.4
90	0.7	818.97	800.58	848.35	764.77	100.2	100.5	0.2	0.3

Implement the trivial randomized approximation algorithm from Example 5.2.1, and compare it to the other algorithms on unweighted MAX CUT instances. What is its performance like, with respect to RR? Would you define it as "an algorithm which actually works in practice"?

5.6 Summary

This chapter focuses on the MAX CUT problem. It presents applications, MP formulations and a very famous randomized approximation algorithm by Goemans and Williamson. Before delving into the thick of this algorithm, it explains what approximation algorithms are, what randomized algorithms are, and what the union of the two concepts is.

6
Distance Geometry

Distance Geometry (DG) is a sub-field of geometry based on the concept of points and *distances* rather than points, lines, planes, hyperplanes or general manifolds. By far the most common setting is where the distances are Euclidean, although early axiomatizations of DG worked with general (semi)metrics [Menger, 1931, Blumenthal, 1953]. This novel axiomatization of geometry was proposed at a time where the "old" way of doing mathematics (read: up to 1900) was being formalized[1] into our current setting. To be honest, mathematicians still use a lot of hand-waving [Krantz, 2010], but now at least they know they *could* do things completely formally if they *wanted to*.[2]

6.1 *The fundamental problem of DG*

The reason why DG and MP are related is through the fundamental problem of DG:

> Distance Geometry Problem (DGP). Given a positive integer K and a simple undirected weighted graph $G = (V, E, d)$ where $d : E \to \mathbb{R}_+$ is an edge weight function, is there a realization[3] $x : V \to \mathbb{R}^K$ such that the constraints:
>
> $$\forall \{i, j\} \in V \quad \|x_i - x_j\| = d_{ij} \tag{6.1}$$
>
> are satisfied?

When the norm in Eq. (6.1) is the 2-norm, the DGP is also called Euclidean DGP (EDGP), which was introduced in Sect. 2.6.

Because it consists of a set of constraints for which we need to find a solution, the DGP is a CP problem (see Ch. 4). On the other hand, CP problems usually have integer, bounded variables, whereas the DGP has continuous unbounded ones. So it makes more sense to see it as a feasibility NLP, i.e. an NLP with zero objective function.

As mentioned in Sect. 2.6, it is difficult to solve DGPs as systems of nonlinear constraints, as given in Eq. (6.1). Usually, MP solvers are better at improving optimality than ensuring feasibility. A nice feature of DGPs is that we can write them as a minimization of

[1] Up to 1900, many proofs had elements of "hand-waving" in them. At the end of the 19th century, David Hilbert decided he'd had enough of being unable to prove anyone wrong formally (specifically Leopold Kronecker, who kept saying that existential proofs did not prove anything – only constructive proofs were good), and decided to use his extensive influence to steer the community into defining things formally once and for all. His valiant attempt succeeded to a large extent, insofar as we now have the Zermelo-Fraenkel-Choice (ZFC) system of axioms, which appear to be enough to do *almost* anything of interest in mathematics, but was ultimately crushed by Gödel's incompleteness theorem in its aim of having a finitistic mechanical procedure for proving every true sentence from the axioms.

[2] What if the only way to write a certain proof formally takes longer than the maximum extent of the human life? Well, use computers! What about reading a proof? Today we have formal proofs, partly or wholly written out by computers, which no-one can possibly hope to read in a lifetime, much less understand. Can they still be called proofs? This is similar to the old "does the universe still exist if no-one's around to notice?"

[3] A *realization* of a graph is a function, mapping each vertex to a vector of a Euclidean space, which satisfies Eq. (6.1).

constraint errors:

$$\min_x \sum_{\{i,j\}\in E} |\,\|x_i - x_j\| - d_{ij}\,|. \tag{6.2}$$

Evidently, x^* is feasible in Eq. (6.1) if and only if it is globally optimum in Eq. (6.2), with zero optimal objective function value.

6.2 *Some applications of the DGP*

As mentioned in Sect. 2.6, the DGP has many applications to science and engineering [Liberti et al., 2014]. We always assume that the input graph G is connected (otherwise the problem can be decomposed and solved for each component separately).

6.2.1 *Clock synchronization*

Let $K = 1$ and V be a set of clocks. An edge $\{i, j\}$ is in E if the time difference between clocks i and j is d_{ij}. Suppose that each clock i knows its own time as well as the weighted graph G. Then a solution $x^* \in \mathbb{R}^1 = \mathbb{R}$ is a consistent evaluation of the absolute times of each clock in V. This application arises in network synchronization protocols [Singer, 2011].

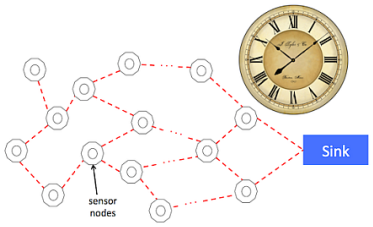


Figure 6.1: Clock synchronization (from www.mdpi.com).

6.2.2 *Sensor network localization*

Let $K = 2$ and V be a set of mobile devices in a wireless network at a certain given time t. An edge $\{i, j\}$ is in E if the distance between i and j is known to be d_{ij} at time t. Such distances can be measured up to a certain threshold, for example by estimating how much battery power a peer-to-peer communication between i and j takes. By exploiting network communication, the weighted graph G can be transmitted to a central server. A solution $x^* \in \mathbb{R}^2$ gives a consistent position of the devices at time t. This application arises naturally in the localization of sensors in wireless networks [Yemini, 1978, Biswas et al., 2006, Bachrach and Taylor, 2005, Ding et al., 2010, Cucuringu et al., 2012a].

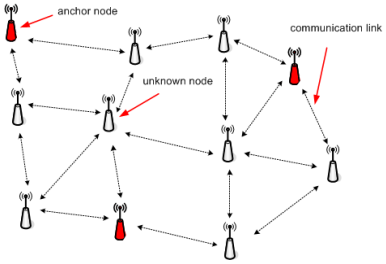


Figure 6.2: Sensor network localization (from www.commsys.isy.liu.se).

6.2.3 *Protein conformation*

Let $K = 3$ and V be a set of atoms in a molecule such as e.g. a protein. It is well known that proteins interact with cells because of their geometrical (as well as chemical) properties: in other words, their shape matters. Being so small, we cannot directly observe their shape. We can, however, measure inter-atomic distances up to a certain threshold, usually 5Å to 6Å [Schlick, 2002], using Nuclear Magnetic Resonance (NMR) [Wüthrich, 1989]. Although NMR only really gives a frequency for each triplet (*atom type*, *atom type*, *length*), which means that we can only know how frequent it is to observe a certain distance between, say, a hydrogen and a carbon, using these data we can reconstruct a weighted graph where $\{i, j\}$ is in E

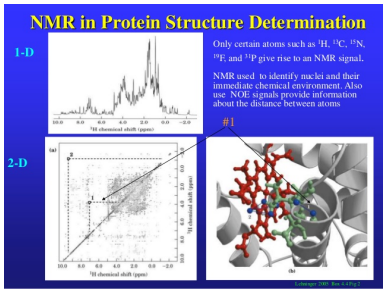


Figure 6.3: Protein conformation from distance data (slide by Dr. B. Chazotte www.campbell.edu).

if the distance d_{ij} between atom i and atom j is known. A solution $x^* \in \mathbb{R}^3$ gives a consistent conformation of the protein [Moré and Wu, 1999, Reams et al., 1999, Wu et al., 2008, Cucuringu et al., 2012b, Lavor et al., 2011, Davis et al., 2010, Cassioli et al., 2015].

6.2.4 *Unmanned underwater vehicles*

Let $K = 3$ and V be a set of unmanned submarines, such as e.g. those used by Shell when trying to fix the oil spill in the Gulf of Mexico. Since GPS cannot be used underwater, the position of each submarine must be inferred. An edge $\{i, j\}$ is in E if the distance between submarines i and j can be obtained using the sonars. A solution $x^* \in \mathbb{R}^3$ gives a consistent snapshot of the positions of the submarines at a certain time instant.

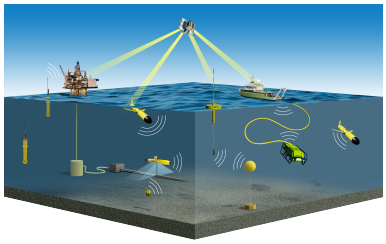


Figure 6.4: Unmanned underwater vehicles (from www.swissint.ch).

6.3 *Formulations of the DGP*

The DGP is really a whole class of problems, which depends on the type of norm used in Eq. (2.21). In general, however, if it is not explicitly given, most people would mean the Euclidean norm $\|\cdot\|_2$. In other words, most of the times you read DGP, you should think of EDGP.

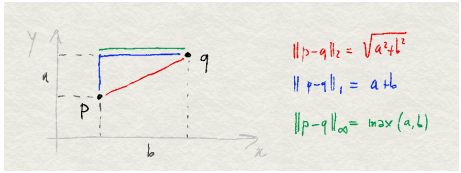


Figure 6.5: The 2-, 1- and ∞-norms.

6.4 *The 1-norm and the max norm*

We just point out that DGPs defined on the 1-norm and the ∞-norm[4] can be linearized exactly to MILPs. We show how to proceed in the case[5] $\|\cdot\| = \|\cdot\|_1$. Eq. (6.2) becomes:

$$\min_x \sum_{\{i,j\}\in E} \left| \sum_{k\le K} |x_{ik} - x_{jk}| - d_{ij} \right|.$$

By applying the reformulations in Sect. 3.2.4 and 3.1.5 we obtain the following nonconvex NLP:

$$\left.\begin{array}{rrcl} \min\limits_{x,s} & \sum\limits_{\{i,j\}\in E} (s^+_{ij} + s^-_{ij}) & & \\ \forall \{i,j\} \in E & \sum\limits_{k\le K} (t^+_{ijk} + t^-_{ijk}) - d_{ij} & = & s^+_{ij} - s^-_{ij} \\ \forall k \le K, \{i,j\} \in E & t^+_{ijk} - t^-_{ijk} & = & x_{ik} - x_{jk} \\ \forall k \le K, \{i,j\} \in E & t^+_{ijk}\, t^-_{ijk} & = & 0 \\ \forall \{i,j\} \in E & s^+_{ij}, s^-_{ij} & \ge & 0 \\ \forall k \le K, \{i,j\} \in E & t^+_{ijk}, t^-_{ijk} & \ge & 0, \end{array}\right\}$$

where the only nonlinear constraints are the products $t^+_{ijk}\, t^-_{ijk} = 0$. In theory, we cannot apply the reformulation in Sect. 3.2.3 since the t variables are unbounded. On the other hand, their unboundedness stems from the x variables, which are themselves unbounded — but simply because we did not bother looking at the problem closely enough. Suppose that the weighted diameter[6] of the input graph G is γ. Then no realization can have $\|x_i - x_j\| \ge \gamma$, which also implies[7]

[4] Also known as *max norm*.

[5] By adapting the MILP reformulation for the $\|\cdot\|_1$ case, derive a MILP reformulation for the $\|\cdot\|_\infty$ case.

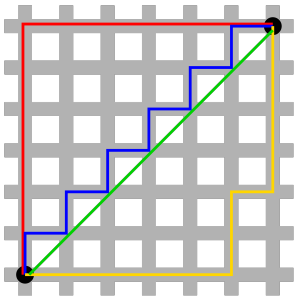


Figure 6.6: The 1-norm between these two points is 12 (from Wikipedia).

[6] The *weighted diameter* of a graph is the maximum, over all pairs of vertices u, v in the graph, of the weight of the shortest path connecting u and v.

[7] This implication holds since all translations are congruences, but it requires us to add another family of (linear) constraints to the formulation. Which ones?

$\|x_i\| \leq \gamma$ for all $i \in V$, and in turn $-\gamma \leq x_{ik} \leq \gamma$ for all $i \in V$ and $k \in K$. So, using interval arithmetic on the linear constraints, we can bound all variables x, s, t by a suitably large[8] constant M. We can now apply the reformulation in Sect. 3.2.3, which yields the MILP:

[8] For best computational results, this constant should be as small as possible for each instance to be solved.

$$\left.\begin{array}{rrcl} \min\limits_{x,s} & \sum\limits_{\{i,j\}\in E} (s^+_{ij} + s^-_{ij}) & & \\ \forall \{i,j\} \in E & \sum\limits_{k\leq K} (t^+_{ijk} + t^-_{ijk}) - d_{ij} & = & s^+_{ij} - s^-_{ij} \\ \forall k \leq K, \{i,j\} \in E & t^+_{ijk} - t^-_{ijk} & = & x_{ik} - x_{jk} \\ \forall k \leq K, \{i,j\} \in E & t^+_{ijk} & \leq & M z_{ijk} \\ \forall k \leq K, \{i,j\} \in E & t^-_{ijk} & \leq & M(1 - z_{ijk}) \\ \forall \{i,j\} \in E & s^+_{ij}, s^-_{ij} & \geq & 0 \\ \forall k \leq K, \{i,j\} \in E & t^+_{ijk}, t^-_{ijk} & \geq & 0 \\ \forall k \leq K, \{i,j\} \in E & z_{ijk} & \in & \{0,1\}. \end{array}\right\} \quad (6.3)$$

For the rest of this chapter we shall focus on the 2-norm, i.e. we shall mean EDGP whenever we write DGP.

6.4.1 *The 2-norm*

We can square both sides of Eq. (6.1) and obtain another NLP with exactly the same set of solutions:

$$\forall \{i,j\} \in V \quad \|x_i - x_j\|_2^2 = d_{ij}^2, \quad (6.4)$$

which we recognize as a QCQP with zero objective function: indeed, the right hand side of Eq. (6.4) can be written as:

$$\begin{aligned} \|x_i - x_j\|_2^2 &= (x_i - x_j) \cdot (x_i - x_j) \\ &= x_i \cdot x_i + x_j \cdot x_j - 2x_i \cdot x_j \\ &= \|x_i\|_2^2 + \|x_j\|_2^2 - 2x_i \cdot x_j \\ &= \sum_{k\leq K} x_{ik}^2 + \sum_{k\leq K} x_{jk}^2 - 2 \sum_{k\leq K} x_{ik} x_{jk}, \end{aligned}$$

which is a multivariate polynomial of second degree. An SDP relaxation of Eq. (6.4) was derived in Sect. 2.10, see Eq. (2.35).

Applying Eq. (6.2) to Eq. (6.4), we obtain the Global Optimization (GO) formulation[9]

[9] Note that the absolute values in Eq. (6.2) have been replaced by a square: obviously the set of global optima of Eq. (6.2) is the same as the one for Eq. (6.5).

$$\min_x \sum_{\{i,j\}\in E} (\|x_i - x_j\|_2^2 - d_{ij}^2)^2, \quad (6.5)$$

which is an unconstrained quartic polyonomial minimization problem. This formulation is at the basis of several different solution methods for the DGP [Lavor et al., 2006, Liberti et al., 2009b].

6.5 *Euclidean Distance Matrices*

A *Euclidean distance matrix* (EDM) is an $n \times n$ zero-diagonal symmetric matrix $D = (d_{ij})$ for which there exist an integer K and a realization $x = (x_1, \ldots, x_n)^\top \in \mathbb{R}^{nK}$ such that $\forall i, j \leq n$ we have $\|x_i - x_j\|_2 = d_{ij}$. Oftentimes, people write EDM when they really mean *square EDM*, i.e. $D = (d_{ij}^2)$.

6.5.1 *The Gram matrix from the square EDM*

Since

$$\forall i \leq j \leq n \quad \|x_i - x_j\|_2^2 = \|x_i\|_2^2 + \|x_j\|_2^2 - 2x_i \cdot x_j, \tag{6.6}$$

the square EDM D is intimately related to the Gram matrix $B = (x_i \cdot x_j)$ (see Sect. 2.10 and 5.3.4) of the realization $x \in \mathbb{R}^{nK}$. In this section we show[10] how to compute B in function of D.

We first translate x so that its barycenter[11] is at the origin:

$$\sum_{i \leq n} x_i = 0. \tag{6.7}$$

Now we remark that, for each $i, j \leq n$, we have:

$$d_{ij}^2 = \|x_i - x_j\|^2 = (x_i - x_j) \cdot (x_i - x_j) = x_i \cdot x_i + x_j \cdot x_j - 2x_i \cdot x_j. \tag{6.8}$$

Next, we "invert" Eq. (6.8) to express $x_i \cdot x_j$ in function of d_{ij}^2: we sum Eq. (6.8) over all values of $i \in \{1, \ldots, n\}$, obtaining:

$$\sum_{i \leq n} d_{ij}^2 = \sum_{i \leq n} (x_i \cdot x_i) + n(x_j \cdot x_j) - 2 \left(\sum_{i \leq n} x_i \right) \cdot x_j. \tag{6.9}$$

By Eq. (6.7), the rightmost term in the right hand side of Eq. (6.9) is zero. On dividing through by n, we have

$$\frac{1}{n} \sum_{i \leq n} d_{ij}^2 = \frac{1}{n} \sum_{i \leq n} (x_i \cdot x_i) + x_j \cdot x_j. \tag{6.10}$$

Similarly for $j \in \{1, \ldots, n\}$, we obtain:

$$\frac{1}{n} \sum_{j \leq n} d_{ij}^2 = x_i \cdot x_i + \frac{1}{n} \sum_{j \leq n} (x_j \cdot x_j). \tag{6.11}$$

We now sum Eq. (6.10) over all j, getting:

$$\frac{1}{n} \sum_{\substack{i \leq n \\ j \leq n}} d_{ij}^2 = n \frac{1}{n} \sum_{i \leq n} (x_i \cdot x_i) + \sum_{j \leq n} (x_j \cdot x_j) = 2 \sum_{i \leq n} (x_i \cdot x_i) \tag{6.12}$$

(the last equality in Eq. (6.12) holds because the same quantity $f(k) = x_k \cdot x_k$ is being summed over the same range $\{1, \ldots, n\}$, with the symbol k replaced by the symbol i first and j next). We then divide through by n to get:

$$\frac{1}{n^2} \sum_{\substack{i \leq n \\ j \leq n}} d_{ij}^2 = \frac{2}{n} \sum_{i \leq n} (x_i \cdot x_i). \tag{6.13}$$

We rearrange Eq. (6.8), (6.11), (6.10) as follows:

$$2x_i \cdot x_j = x_i \cdot x_i + x_j \cdot x_j - d_{ij}^2 \tag{6.14}$$

$$x_i \cdot x_i = \frac{1}{n} \sum_{j \leq n} d_{ij}^2 - \frac{1}{n} \sum_{j \leq n} (x_j \cdot x_j) \tag{6.15}$$

$$x_j \cdot x_j = \frac{1}{n} \sum_{i \leq n} d_{ij}^2 - \frac{1}{n} \sum_{i \leq n} (x_i \cdot x_i), \tag{6.16}$$

[10] The sequence of transformations that follow is fastidiously detailed and precise. It is more or less taken from [Cox et al., 1997], with some added steps of my own. The reason I tried to put in as many details as possible is that I was tired to re-work the details out every time I read this proof with some missing steps.

[11] The *barycenter* of a sequence $(x_1, \ldots, x_n)$ of points in $\mathbb{R}^K$ is the vector $\frac{1}{n} \sum_{i \leq n} x_i$. The barycenter is also known as *centroid*.

and replace the left hand side terms of Eq. (6.15)-(6.16) into Eq. (6.14) to obtain:

$$2x_i \cdot x_j = \frac{1}{n}\sum_{k\leq n} d_{ik}^2 + \frac{1}{n}\sum_{k\leq n} d_{kj}^2 - d_{ij}^2 - \frac{2}{n}\sum_{k\leq n}(x_k \cdot x_k), \tag{6.17}$$

whence, on substituting the last term using Eq. (6.13), we have:

$$2x_i \cdot x_j = \frac{1}{n}\sum_{k\leq n}(d_{ik}^2 + d_{kj}^2) - d_{ij}^2 - \frac{1}{n^2}\sum_{\substack{h\leq n\\ k\leq n}} d_{hk}^2. \tag{6.18}$$

Eq. (6.18) can also be written[12] in matrix form as:

[12] Prove it.

$$B = -\frac{1}{2}JDJ, \tag{6.19}$$

where $J = I_n - \frac{1}{n}\mathbf{1}\cdot\mathbf{1}^\top$ and $\mathbf{1} = \underbrace{(1,\ldots,1)}_{n}$.

6.5.2 *Is a given matrix a square EDM?*

We can now establish that the problem of determining whether a given matrix is a square EDM can be solved in polytime as follows:

1. given D, compute B as per Sect. 6.5.1
2. compute the eigenvalues of B
3. if all eigenvalues are non-negative, then $B \succeq 0$, so we can factor B as in Sect. 5.4.1, to obtain x such that $B = xx^\top$, as required: output YES[13]
4. if at least one eigenvalue is negative, then output NO.

[13] This proves that any PSD matrix is a Gram matrix of some realization.

We note that all of the steps can be carried out in polynomial time, and remark that the above algorithm can also be used to ascertain whether D is a (non-square) EDM or not.[14]

[14] Why?

6.5.3 *The rank of a square EDM*

EDMs and their square counterparts have a fundamental difference, as remarked in the recent survey [Dokmanić et al., 2015]: whereas EDMs may have any rank greater or equal to $\mathsf{rank}(x)$, square EDMs have rank in the set[15] $\{\mathsf{rank}(x), \ldots, \mathsf{rank}(x)+2\}$. By Eq. (6.6), we can write the EDM D of the realization $x \in \mathbb{R}^{nK}$ as:

[15] So, if x has full rank K, the square EDM of x has rank in $\{K, K+1, K+2\}$.

$$D = \mathbf{1}\mathsf{diag}(xx^\top)^\top - 2xx^\top + \mathsf{diag}(xx^\top)\mathbf{1}^\top,$$

where $\mathbf{1}$ is the (column) vector of all ones, and $\mathsf{diag}(v)$ is the (column) vector of the diagonal of v. This implies[16]

[16] Note that Eq. (6.20) is the matrix equation inverse of Eq. (6.19).

$$D = \mathbf{1}\mathsf{diag}(B)^\top - 2B + \mathsf{diag}(B)\mathbf{1}^\top, \tag{6.20}$$

where B is the Gram matrix of x. Now, by Prop. 5.3.2(ii), we know that $\mathsf{rank}(B) = \mathsf{rank}(x)$, and both the first and the last term in the RHS of Eq. (6.20) have rank 1. The claim follows by rank inequalities.[17]

[17] I.e., if $M = M' + M''$ then $\mathsf{rank}(M) \leq \mathsf{rank}(M') + \mathsf{rank}(M'')$. In other words, the $\mathsf{rank}(\cdot)$ function is *sub-additive*.

The reason why this matters in practice is the following. Usually, given an EDM, you really want to find the realization it corresponds to. If you square every component of your given EDM, and then compute its rank r (perhaps only considering eigenvalues having absolute value larger than a given "floating point error tolerance"), $r-2$ will be a lower bound on the affine dimension K' of the realization giving rise to your EDM. In most applications, n will be very large, but K will be very small, see e.g. Sect. 6.2 where $K \in \{1,2,3\}$. If you find $r-2 \gg K$, and you cannot impute this to a floating point error in eigenvalue computations, chances are your given EDM is wrong.

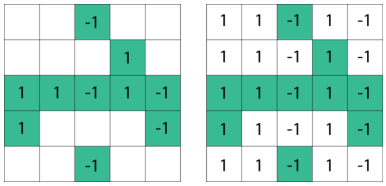


Figure 6.7: There are many different completion problems: see above an example of rank-1 completion (from Wikipedia.

6.6 *The EDM Completion Problem*

The following problem is closely related to the EDGP.

> Euclidean Distance Matrix Completion Problem (EDMCP). Given a partially specified[18] $n \times n$ zero-diagonal symmetric matrix, determine whether it can be completed to an EDM (or to a square EDM).

It is easy to show that a partially specified zero-diagonal square symmetric matrix D over $\mathbb{R}_+$ carries the same information as a simple undirected edge-weighted graph $G = (V, E, d)$: every component $d_{ij} \neq \star$ in D corresponds to an edge $\{i,j\} \in E$ with weight d_{ij}, whereas $\star$ components corresponds to edges not in E. So is the EDMCP just another name for the EDGP? The difference between EDGP and EDMCP is in the integer K: in the former, $K > 0$ is given as part of the input, while in the latter it is part of the output. Specifically, while the graph information is the same, the EDGP is also given K, and then asks if, for that K, the given partial matrix can be completed to a distance matrix of a realization in $\mathbb{R}^K$. The EDMCP, on the other hand, asks if there is *any* K such that the given partial matrix can be completed to a distance matrix of a realization in $\mathbb{R}^K$.

This difference is subtle, but has a very deep implication: the EDGP is known to be **NP**-hard,[19] whereas we do not know the complexity status of the EDMCP: no-one has found a polytime algorithm, nor a proof of **NP**-hardness.

[18] A *partially specified* matrix (or *partial matrix*) over a number field $\mathbb{F}$ is a matrix over $\mathbb{F} \cup \{\star\}$. By convention, a $\star$ entry is unspecified. Here is an example of a partial matrix:

$$\begin{pmatrix} 0 & \star & 1 & 1 \\ \star & 0 & 4 & 9 \\ 1 & 4 & 0 & 4 \\ 1 & 9 & 4 & 0 \end{pmatrix}.$$

Can this matrix be completed to an EDM?

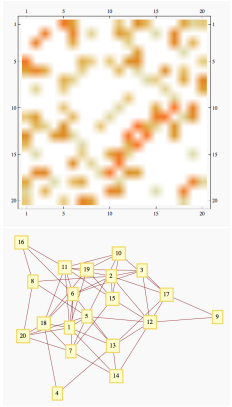


Figure 6.8: An EDMCP instance (above) and the equivalent EDGP instance (below).

[19] By reduction from the Partition problem, see [Saxe, 1979].

6.6.1 *An SDP formulation for the EDMCP*

By Prop. 5.3.2, every Gram matrix is PSD, and, by Sidenotes 13 in Ch. 5 and 13 in this chapter, every PSD matrix is the Gram matrix of some other matrix (or realization). If we were given a partially specified Gram matrix $B = (b_{ij})$, we could complete[20] it by solving the following SDP:

$$\left.\begin{array}{rcl} \forall i,j \leq n : b_{ij} \neq \star \quad X_{ij} & = & b_{ij} \\ X & \succeq & 0. \end{array}\right\} \tag{6.21}$$

[20] This is called the PSD Completion Problem (PSDCP).

Any solution X^* to Eq. (6.21) is a PSD matrix having b_{ij} in the components specified by B.

However, we are not given B but the square EDM matrix $D = (d_{ij}^2)$. We therefore use Eq. (6.6) and replace $x_i \cdot x_j$ by X_{ij} (for all $i, j \leq n$):

$$\left.\begin{array}{lll} \forall i,j \leq n : d_{ij}^2 \neq \star \quad X_{ii} + X_{jj} - 2X_{ij} & = & d_{ij}^2 \\ X & \succeq & 0. \end{array}\right\} \tag{6.22}$$

Note the similarity of the first constraint of Eq. (6.22) with Eq. (2.34), written in terms of weighted graphs instead of partial matrices. Any solution X^* to Eq. (6.22) is a PSD matrix, and hence a Gram matrix, such that $X_{ii} + X_{jj} - 2X_{ij} = d_{ij}^2$ in the components specified by D. In other words, if we apply Eq. (6.20) with B replaced by X^*, we get an EDM with the specified entries.

Compare the two SDP formulations for the EDGP (Eq. (2.35)) and for the EDMCP (Eq. (6.22)). The former is different, in that we use the Schur complement:[21] while in the EDGP we require a PSD solution *of a given rank* K, in the EDMCP we only want a PSD solution of *any rank* (or, in other words, any PSD solution). This is why Eq. (2.35) is only a relaxation of the EDGP, whereas Eq. (6.22) is an exact formulation of the EDMCP.

[21] The Schur complement being PSD states $X \succeq xx^\top$, which is a relaxation of $X = xx^\top$. The fact that x (and hence its rank) is explicitly mentioned in the SDP makes the relaxation tighter with respect to the rank constraint.

Since SDPs can be solved in polytime and the SDP in Eq. (6.22) is an exact formulation for the EDMCP, why am I saying that no-one found a polytime algorithm for solving the EDMCP yet? Essentially, because of floating point errors: the IPMs that solve SDPs are *approximate* methods: they find a matrix which is approximately feasible with respect to the constraints and approximately PSD (see Sect. 5.4.3). Worst-case complexity classes such as **P** or **NP**, however, are defined for precise solutions only. As discussed in Sidenote 15 in Ch. 5, approximate feasibility disqualifies the algorithm from being defined as an "approximation algorithm".

6.6.2 *Inner LP approximation of SDPs*

SDPs are nice because they usually provide tight relaxations and can be solved in polytime. On the other hand, current technology can solve relatively small SDPs, say with matrices of up to around 1000 components. In practice, it is not uncommon to see DGP applications where $K \in \{1, 2, 3\}$ but $n \geq 1000$. This would yield an SDP variable matrix with at least one million components. Since SDP solvers are usually based on IPMs, at each iteration the solver has to deal with $O(10^6)$ floating point numbers. Unless the structure of the problem can be exploited to simplify the formulation, SDP solvers will generally fail. In Sect. 6.7 we will present a heuristic algorithm which scales better.

Figure 6.9: Amir Ali Ahmadi is the main architect behind the work in SDP approximations by DD matrices. You've already seen his mug in Fig. 2.30.

Here we discuss an extremely recent method [Majumdar et al., 2014] which can be used to approximate an SDP by means of an LP. Since the LP is based on an *inner approximation*, any solution X' found by the LP is also a feasible solution for the SDP. This method

is based on diagonal dominance.[22]

Recall that an SDP is simply an LP defined on a matrix X of decision variables, subject to a further PSD constraint $X \succeq 0$. We replace the PSD constraint by the DD constraint:

$$\forall i \leq n \quad a_{ii} \geq \sum_{j \neq i} |a_{ij}|, \tag{6.23}$$

and remark that any matrix X which is DD is also SDP; this follows directly from Gershgorin's theorem.[23] Lastly, we reformulate the absolute values in Eq. (6.23) using Eq. (3.16)-(3.18) in Sect. 3.2.4, obtaining the following LP, which will yield solutions of the SDP relaxation in Eq. (2.35):

$$\left.\begin{array}{rrcl} \forall \{i,j\} \in E & X_{ii} + X_{jj} - 2X_{ij} & = & d_{ij}^2 \\ \forall i \in V & \sum_{j \neq i} t_{ij} & \leq & X_{ii} \\ \forall i \neq j \in V & -t_{ij} \leq X_{ij} & \leq & t_{ij}. \end{array}\right\} \tag{6.24}$$

Note that this is a feasibility LP. We can solve it "as such", or try to make the entries of X as small as possible (which might reduce its rank if many of them become zero) by using $\min \sum_{ij} t_{ij}$.

[22] An $n \times n$ matrix $A = (a_{ij})$ is *diagonally dominant* (DD) if Eq. (6.23) holds.

[23] The proof on Wikipedia is very clear, search for "Gershgorin circle theorem".

6.6.3 *Refining the DD approximation*

An iterative refinement method for improving the approximation quality of the solution X of Eq. (6.24) with respect to the SDP is described in [Ahmadi and Hall, 2015]. We present it here in full generality, applied to any SDP.

For any $n \times n$ matrix U, we have $U^\top U \succeq 0$. This follows because any Gram matrix is PSD, and, trivially, $U^\top U = \mathsf{Gram}(U)$. Moreover, for any $X \succeq 0$, we have

$$U^\top X U \succeq 0,$$

since X being PSD means that it is the Gram matrix of some other matrix V, so

$$U^\top X U = U^\top V^\top V U = (VU)^\top VU \succeq 0$$

as the Gram matrix of VU must be PSD. This means that the set

$$\mathcal{D}(U) = \{U^\top X U \mid X \text{ is DD}\}$$

only contains PSD matrices.

Let us go back to the general SDP formulation in Eq. (2.32). As mentioned above, we obtain an inner approximation of this SDP by replacing $X \succeq 0$ by "X is DD", which can be reformulated to an LP. This LP has no "adjustable parameters" by which we can try and improve the approximation; in other words, it is an LP rather than a family of LPs — and this is one of those cases where having a parametrized family of LP approximations could help in adapting the approximation to the original SDP. By replacing "X is DD" by

Figure 6.10: An example of diagonal dominance (from `www.voonik.com`).

$X \in \mathcal{D}(U)$ we achieve exactly this, and let our LP approximation depend on the matrix U.

$$\left.\begin{array}{rrcl} \min & C \bullet X & & \\ \forall i \leq m & A^i \bullet X & = & b^i \\ & X & \in & \mathcal{D}(U), \end{array}\right\} \tag{6.25}$$

We can now leverage the parameter matrix U to attempt to improve the approximation.

The iterative algorithm proposed in [Ahmadi and Hall, 2015] defines a sequence of matrices $U^0, U^1, \ldots, U^\ell, \ldots$ as follows:

$$U^0 = I \tag{6.26}$$

$$U^\ell = \mathsf{factor}(X^{\ell-1}), \tag{6.27}$$

where, factor denotes a factor[24] of the argument matrix, and for each $k \in \mathbb{N}$, X^ℓ is the solution to the LP

$$\left.\begin{array}{rrcl} \min & C \bullet X & & \\ \forall i \leq m & A^i \bullet X & = & b^i \\ & X & \in & \mathcal{D}(U^\ell), \end{array}\right\} \tag{6.28}$$

which is the same as Eq. (6.25) with U replaced by U^ℓ. Eq. (6.28) actually defines a sequence of approximating LPs to Eq. (2.32).

6.6.1 Proposition
The approximation X^ℓ to the solution of Eq. (2.32) is no worse than $X^{\ell-1}$.

Proof. It suffices to show that $X^{\ell-1}$ is feasible for the ℓ-th approximating LP in the sequence, since this shows that the feasible sets $\mathcal{F}_\ell$ of the approximating LP sequence form a chain

$$\mathcal{F}_0 \subseteq \cdots \subseteq \mathcal{F}_\ell \subseteq \cdots$$

Since all are inner approximations of the feasible region of Eq. (2.32), the larger the feasible region, the better the approximation. Now, by Eq. (6.27) we have that $X^{\ell-1} = (U^\ell)^\top U^\ell = (U^\ell)^\top I U^\ell$, and since the identity matrix I is trivially DD, we have that $X^{\ell-1} \in \mathcal{D}(U^\ell)$. Moreover, since $X^{\ell-1}$ solves the $(\ell-1)$-th approximating LP in the sequence Eq. (6.28), we have $\forall i \leq m \; A^i \bullet X^{\ell-1} = b^i$, which proves the claim. □

6.6.4 *Iterative DD approximation for the EDMCP*

The application of the results of Sect. 6.6.3 to Eq. (6.24) yields the following iterative algorithm.

1. Initialize $U^0 = I$ and $\ell = 0$
2. Solve the following LP in the matrix variables X, Y and $T \geq 0$:

$$\left.\begin{array}{rrcl} \forall i,j : d_{ij}^2 \neq \star & X_{ii} + X_{jj} - 2X_{ij} & = & d_{ij}^2 \\ \forall i \leq n & \sum_{j \neq i} t_{ij} & \leq & Y_{ii} \\ & -T \leq Y & \leq & T \\ & (U^\ell)^\top Y U^\ell & = & X \end{array}\right\} \tag{6.29}$$

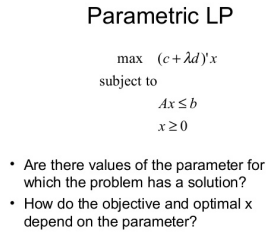


Figure 6.11: An introductory slide on parametric LP (from users.soe.ucsc.edu/~kross).

[24] The factor can be found in any number of ways. The paper suggests using *Cholesky factors* where

$$X^{\ell-1} = LL^\top,$$

with L a lower triangular matrix.

and let (X^ℓ, Y^ℓ, T^ℓ) be its solution

3. Let $U^{\ell+1}$ be a Cholesky factor of of X^ℓ
4. Increase ℓ and loop from Step 2.

The termination conditions are based on either ℓ being smaller than a pre-determined $\ell_{\max}$ threshold, or CPU time, or $\|X^\ell - X^{\ell-1}\|$ being too small.

6.6.5 *Working on the dual*

What if the original SDP formulation is used to compute a guaranteed bound to some problem P? Then inner approximations of the SDP feasible region will not help, since only an optimum of the SDP (or a bound to this optimum) will provide the bound guarantee for P.

In such cases, we leverage the fact that both LPs and SDPs have a strong duality theory. In particular:

- the dual of an LP is another LP, and the dual of an SDP is another SDP;
- the optimal objective function values of the LP and its dual (respectively of the SDP and its dual) are equal;
- any feasible solution of the LP (respectively, SDP) dual is a guaranteed bound on the objective function value of the primal.

So all we need to do is to compute the dual of the original SDP, and then apply the DD inner approximation to the dual. This will yield a feasible solution in the dual SDP, which provides a valid guaranteed bound for the original SDP (and hence, in turn, for P).

6.6.6 *Can we improve the DD condition?*

All DD matrices are SDP, but not every SDP matrix is DD. In fact, X being DD is quite a restrictive condition on X. It can be relaxed somewhat by requiring that there is a diagonal matrix Δ with strictly positive diagonal components such that $\Delta X \Delta$ is DD. A matrix X with this property is called *scaled diagonally dominant* (SDD).

The relevant property of SDD matrices to this discussion is that every SDD matrix is PSD, and every DD matrix is also (trivially, with $\Delta = I$) an SDD matrix. So they represent a set of matrices between DD and PSD matrices.

Luckily, it turns out that X is SDD can be written as a SOCP (see Sect. 2.10.1). Methods similar to those discussed above can also be derived, with LPs replaced by SOCPs. The trade-off is that, in exchange for improving the approximation of the PSD matrix cone, and hence the solution quality, we have to spend more CPU time computing the solution of each SOCP.

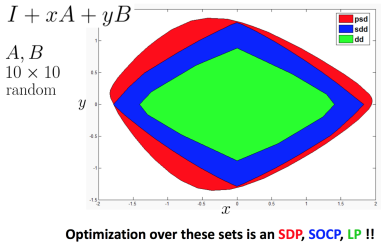


Figure 6.12: DD and SDD inner approximations of the PSD set

$$I + xA + yB \succeq 0$$

(from aaa.princeton.edu).

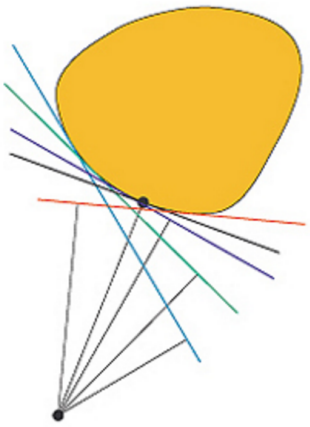


Figure 6.13: A geometric view of a minimization SDP dual: maximizing over the direction vectors perpendicular to the outer approximation (from [Dattorro, 2015]).

6.7 The Isomap heuristic

The Isomap algorithm is usually known as a dimensionality reduction method, i.e. an algorithm which transforms a set $X \subseteq \mathbb{R}^n$ into another set $Y \subseteq \mathbb{R}^m$ with $m < n$, and such that some properties of X are preserved, or approximately preserved, in Y. In summary, the Isomap method is as follows: given X,

1. compute a partial EDM D on X (for example by only considering distances smaller than a certain threshold);
2. complete D to some approximate distance matrix $\bar{D}$, for example by running a weighted shortest path algorithm on the weighted graph corresponding to D;
3. use Sect. 6.5.1 to find the (approximate) Gram matrix $\bar{B}$ corresponding to $\bar{D}$;
4. in general, since $\bar{D}$ is usually not an EDM, $\bar{B}$ is not a Gram matrix, in particular it is not PSD and hence fails to have a real factoring $YY^\top$; so we find the closest PSD matrix B' to $\bar{B}$ (see Sect. 5.4.3) and then factor it as $B' = YY^\top$.

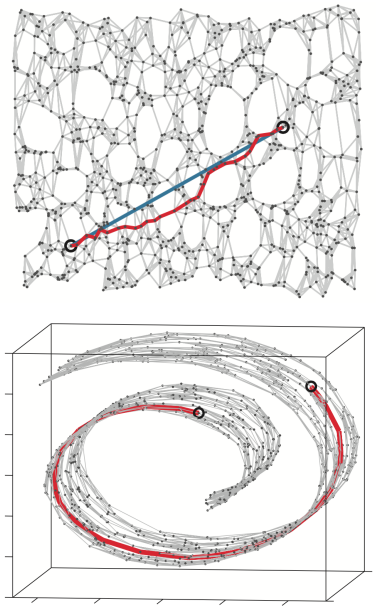


Figure 6.14: The Isomap method for DG: complete a partial distance matrix using shortest paths, above; then find an approximate realization, below (from [Tenenbaum et al., 2000]).

This algorithm is successful as long as

$$m = \mathsf{rank}(Y) < \mathsf{rank}(X) = n.$$

In practice, though, we would prefer $m \ll n$.

6.7.1 Isomap and DG

In order to employ Isomap as an algorithm for the EDMCP we simply start it from Step 2, so that its input is the partial EDM D.

If we want to apply it to the EDGP, we replace B' by the closest PSD matrix to $\bar{B}$ having given rank K, namely,[25] we only keep at most K non-negative eigevalues.

In general, the Isomap algorithm is neither exact nor approximate, and hence only qualifies as a *heuristic algorithm*. It is nonetheless very successful in practice [Tenenbaum et al., 2000] since it scales well to large sizes.

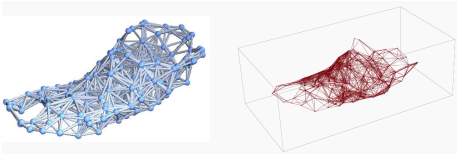


Figure 6.15: The Isomap algorithm solving a DGP instance (left). The shape of the solution (right) looks convincingly similar to the original shape.

[25] To make this summary explanation somewhat more explicit: we factor $\bar{B}$ into eigenvectors P and eigenvalues Λ listed in decreasing order, we zero all negative ones, and possibly all the positive ones up until we leave at most K non-negative ones Λ^+, then we compute $B' = P^\top \Lambda^+ P$.

6.8 Random projections

An important feature of the 2-norm is that it can be approximately preserved by random projections. Let $X \subseteq \mathbb{R}^m$ with $X = \{x_1, \ldots, x_n\}$, and think of X as an $m \times n$ matrix having n columns $x_1, \ldots, x_n$ each defining a point in $\mathbb{R}^m$. Then there is a randomized algorithm which outputs a $k \times m$ matrix T, where k is $O(\frac{1}{\varepsilon^2} \log n)$, such that the probability that:

$$\forall i < j \leq n \; (1-\varepsilon)\|x_i - x_j\|_2 \leq \|Tx_i - Tx_j\|_2 \leq (1+\varepsilon)\|x_i - x_j\| \quad (6.30)$$

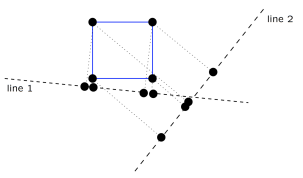


Figure 6.16: The JLL won't work in low dimensions.

holds can be made arbitrarily close to 1. This is known as the *Johnson-Lindenstrauss Lemma* (JLL) [Johnson and Lindenstrauss, 1984], and is one of the cornerstones of clustering large-dimensional datasets (such as image databases).

The most striking feature[26] of the JLL is that k is not a function of m. In other words, the actual number of dimensions you need in order to "pack" the projections of the vectors in X, so that the pairwise distances are approximately preserved to within ε, does not depend on the number of dimensions of X. Another striking feature is that the number of dimensions grows only logarithmically in the number of vectors.[27]

A few comments on the JLL are in order.

- $O(\varepsilon^{-2} \log n)$ is not really a number, but a short-hand for $C\varepsilon^{-2} \log n$ for some constant C which does not depend on n or m. To use the JLL in practice, this constant must be estimated somehow.[28]
- $C\varepsilon^{-2} \log n$ is not always smaller than m — aside from C, it depends on ε and n, of course. But asymptotically, the term in $\log n$ grows very slowly with respect to how fast ε^{-2} grows.
- If you want the approximation to be very accurate, then ε should be very small, so $1/\varepsilon^2$ is going to be very large. If m is huge, there will be a saving. The trade-off is between ε and m. For smaller m's, you would need a larger error tolerance ε.

The randomized algorithm to find T is very simple.

1. Sample each of the km components of T independently from a normal distribution with mean 0 and standard deviation $\frac{1}{\sqrt{k}}$
2. Does TX verify Eq. (6.30)? If not, repeat from Step 1.

The elementary proof of the JLL given in [Dasgupta and Gupta, 2002] shows that the number of iterations of this algorithm is $O(n)$.

6.8.1 *Usefulness for solving DGPs*

A possible use of random projections for solving EDGPs is given by the observation that it is easier to satisfy Eq. (6.1) if K is large than if K is small. On the other hand, most applications require a small K. So, for extremely large-scale EDGP instances, one might try deploying any heuristic method to find an acceptably good solution in a high-dimensional space, and then use a random projection matrix T to scale the dimensions down to a more acceptable level. One would then need a heuristic such as Isomap (Sect. 6.7) to drive the dimensions down to a given K.[29]

6.9 *How to adjust an approximately feasible solution*

Working with SDP and heuristic solutions exposes us to the issue of infeasibility: what if our solution x to a EDGP does not satisfy Eq. (6.1)? A practical and usually acceptably fast way to do so is

[26] Edo Liberty (of Yahoo! Labs) once told me that the way he sees the JLL is that, in any Euclidean space, you can fit *exponentially many* (in function of the dimension) "approximately pairwise orthogonal" vectors . Prove that this is equivalent to the JLL.

[27] The upshot is that the perfect application setting for the JLL is when X contains a moderate to large number of vectors in a huge dimensional space. Image or movie databases have this property, which is why the random projections are used when clustering images.

[28] There is not much literature on this, unfortunately [Venkatasubramanian and Wang, 2011].

[29] More in general, the JLL is applicable to any algorithm which only (or mostly) uses Euclidean distances, such as e.g. the popular k-means heuristic for solving Euclidean clustering problems.

to use x as a starting point for a local search with respect to the GO formulation Eq. (6.5). A good, open-source local NLP solver which can be employed for this purpose is IPOPT. There is still no guarantee of feasibility, but at least the infeasibility error should decrease.

6.10 Summary

This chapter is about Distance Geometry, and introduces the fundamental problem of DG. It argues that it is useful for scientific and engineering applications, it presents formulations for the 1-, ∞- and 2-norms. It introduces Euclidean distance matrices and draws a parallel between the DGP and the completion problem for partially specified EDMs. It presents an SDP formulation for this problem, and discusses a very new method for finding feasible SDP solutions by solving an LP. It then presents the famous dimensionality reduction Isomap heuristic as a solution algorithm for the DGP, and briefly introduces random projections which approximately preserve Euclidean distances.

Bibliography

A. Ahmadi and G. Hall. Sum of squares basis pursuit with linear and second order cone programming. Technical Report 1510.01597v1, arXiv, 2015.

A. Alfakih, A. Khandani, and H. Wolkowicz. Solving Euclidean distance matrix completion problems via semidefinite programming. *Computational Optimization and Applications*, 12:13–30, 1999.

F. Alizadeh and D. Goldfarb. Second order cone programming. *Mathematical Programming B*, 95:3–51, 2003.

E. Amaldi, K. Dhyani, and L. Liberti. A two-phase heuristic for the bottleneck k-hyperplane clustering problem. *Computational Optimization and Applications*, 56:619–633, 2013.

D. Applegate, R. Bixby, V. Chvátal, and W. Cook. *The Traveling Salesman: a Computational Study*. Princeton University Press, Princeton, 2007.

J. Bachrach and C. Taylor. Localization in sensor networks. In I. Stojmenović, editor, *Handbook of Sensor Networks*, pages 3627–3643. Wiley, 2005.

P. Belotti. Couenne*: a user's manual*. Dept. of Mathematical Sciences, Clemson University, 2011.

P. Belotti, J. Lee, L. Liberti, F. Margot, and A. Wächter. Branching and bounds tightening techniques for non-convex MINLP. *Optimization Methods and Software*, 24(4):597–634, 2009.

T. Berthold, G. Gamrath, A. Gleixner, S. Heinz, T. Koch, and Y. Shinano. *Solving mixed integer linear and nonlinear problems using the SCIP Optimization Suite*. ZIB, `http://scip.zib.de/`, 2012.

D. Bertsimas and M. Sim. The price of robustness. *Operations Research*, 52(1):35–53, 2004.

P. Biswas, T. Lian, T. Wang, and Y. Ye. Semidefinite programming based algorithms for sensor network localization. *ACM Transactions in Sensor Networks*, 2:188–220, 2006.

L. Blumenthal. *Theory and Applications of Distance Geometry*. Oxford University Press, Oxford, 1953.

I. Bomze. Evolution towards the maximum clique. *Journal of Global Optimization*, 10:143–164, 1997.

I. Bomze, M. Budinich, P.M. Pardalos, and M. Pelillo. The maximum clique problem. In D.-Z. Du and P.M. Pardalos, editors, *Handbook of Combinatorial Optimization, Supp. A*, volume supp. A, pages 1–74. Kluwer Academic Publishers, Dordrecht, 1998.

P. Bonami and J. Lee. `BONMIN` user's manual. Technical report, IBM Corporation, June 2007.

M. Boulle. Compact mathematical formulation for graph partitioning. *Optimization and Engineering*, 5:315–333, 2004.

S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, 2004.

C. Bragalli, C. D'Ambrosio, J. Lee, A. Lodi, and P. Toth. On the optimal design of water distribution networks: a practical minlp approach. *Optimization and Engineering*, 13:219–246, 2012.

R. Burkard, M. Dell'Amico, and S. Martello. *Assignment problems*. SIAM, Providence, 2009.

E. Candès. The mathematics of sparsity. In S.Y. Jang, Y.R. Kim, D.-W. Lee, and I. Yie, editors, *Proceedings of the International Congress of Mathematicians*, volume I. Kyung Moon SA, Seoul, 2014.

G. Caporossi, D. Alamargot, and D. Chesnet. Using the computer to study the dyamics of the handwriting processes. In *DS 2004 Proceedings*, volume 3245 of *LNAI*, pages 242–254. Springer, 2004.

A. Cassioli, B. Bordeaux, G. Bouvier, A. Mucherino, R. Alves, L. Liberti, M. Nilges, C. Lavor, and T. Malliavin. An algorithm to enumerate all possible protein conformations verifying a set of distance constraints. *BMC Bioinformatics*, page 16:23, 2015.

B. Colson, P. Marcotte, and G. Savard. Bilevel programming: a survey. *4OR*, 3:87–107, 2005.

V. Cortellessa, F. Marinelli, and P. Potena. Automated selection of software components based on cost/reliability tradeoff. In V. Gruhn and F. Oquendo, editors, *EWSA 2006*, volume 4344 of *LNCS*, pages 66–81. Springer, 2006.

A. Costa, P. Hansen, and L. Liberti. On the impact of symmetry-breaking constraints on spatial branch-and-bound for circle packing in a square. *Discrete Applied Mathematics*, 161:96–106, 2013.

D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties and Algorithms*. Springer, Berlin, second edition, 1997.

M. Cucuringu, Y. Lipman, and A. Singer. Sensor network localization by eigenvector synchronization over the Euclidean group. *ACM Transactions on Sensor Networks*, 8:1–42, 2012a.

M. Cucuringu, A. Singer, and D. Cowburn. Eigenvector synchronization, graph ridigity and the molecule problem. *Information and Inference: a journal of the IMA*, 1:21–67, 2012b.

N. Cutland. *Computability: an introduction to recursive function theory*. Cambridge University Press, Cambridge, 1980.

C. D'Ambrosio and A. Lodi. Mixed-integer nonlinear programming tools: a practical overview. *4OR*, 9:329–349, 2011.

G. Dantzig, A. Orden, and P. Wolfe. The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, 5(2):183–196, 1955.

S. Dasgupta and A. Gupta. An elementary proof of a theorem by johnson and lindenstrauss. *Random Structures and Algorithms*, 22: 60–65, 2002.

J. Dattorro. *Convex Optimization and Euclidean Distance Geometry*. $\mathcal{M}\epsilon\beta oo$, Palo Alto, 2015.

R. Davis, C. Ernst, and D. Wu. Protein structure determination via an efficient geometric build-up algorithm. *BMC Structural Biology*, 10(Suppl 1):S7, 2010.

Ph. Delsarte. Bounds for unrestricted codes by linear programming. *Philips Research Reports*, 27:272–289, 1972.

K. Dhyani. Personal communication, 2007.

Kanika Dhyani. *Optimization models and algorithms for the hyperplane clustering problem*. PhD thesis, Politecnico di Milano, 2009.

Y. Ding, N. Krislock, J. Qian, and H. Wolkowicz. Sensor network localization, Euclidean distance matrix completions, and graph realization. *Optimization and Engineering*, 11:45–66, 2010.

I. Dokmanić, R. Parhizkar, J. Ranieri, and M. Vetterli. Euclidean distance matrices: Essential theory, algorithms and applications. *IEEE Signal Processing Magazine*, 1053-5888:12–30, Nov. 2015.

C. Dürr. Programmation par contraintes et programmation mathématique, 2011. Lecture notes, École Polytechnique.

J. Edmonds. Paths, trees and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

M. Ehrgott. *Multicriteria Optimization*. Springer, New York, 2005.

L. Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii Academiæ Scientiarum Imperialis Petropolitanæ*, 8:128–140, 1736.

L. Evans. *An introduction to mathematical optimal control theory*. v. 0.2. `math.berkeley.edu/ evans/control.course.pdf`.

M. Fischetti and A. Lodi. Local branching. *Mathematical Programming*, 98:23–37, 2005.

M. Fischetti, F. Glover, and A. Lodi. The feasibility pump. *Mathematical Programming*, 104(1):91–104, 2005.

M. Flood. The traveling salesman problem. *Operations Research*, 4 (1):61–75, 1956.

L. Ford and D. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

R. Fortet. Applications de l'algèbre de Boole en recherche opérationelle. *Revue Française de Recherche Opérationelle*, 4:17–26, 1960.

R. Fourer and D. Gay. *The AMPL Book*. Duxbury Press, Pacific Grove, 2002.

V. Giakoumakis, D. Krob, L. Liberti, and F. Roda. Technological architecture evolutions of information systems: trade-off and optimization. *Concurrent Engineering Research and Applications*, 20 (2):127–147, 2012.

K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1930.

R.E. Gomory. Essentials of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64(5): 256, 1958.

G. Gutin and A. Punnen, editors. *The traveling salesman problem and its variations*. Kluwer, Dordrecht, 2002.

P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, 10(1):26–30, 1935.

D. Harel. On folk theorems. *Communications of the ACM*, 23(7): 379–389, 1980.

W. Hart, C. Laird, J.-P. Watson, and D. Woodruff. *Pyomo — Optimization modelling in Python*. Springer, New York, 2012.

IBM. *ILOG CPLEX 12.2 User's Manual*. IBM, 2010.

IBM. *ILOG CPLEX 12.6 User's Manual*. IBM, 2014.

W. Johnson and J. Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. In G. Hedlund, editor, *Conference in Modern Analysis and Probability*, volume 26 of *Contemporary Mathematics*, pages 189–206, Providence, 1984. American Mathematical Society.

N. Karmarkar. A new polynomial time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

R. Karp. Reducibility among combinatorial problems. In R. Miller and W. Thatcher, editors, *Complexity of Computer Computations*, volume 5 of *IBM Research Symposia*, pages 85–104, New York, 1972. Plenum.

L. Khachiyan. Polynomial algorithms in linear programming. *USSR Computational Mathematics and Mathematical Physics*, 23(1):53–72, 1980.

H. Konno and A. Wijayanayake. Portfolio optimization problem under concave transaction costs and minimal transaction unit constraints. *Mathematical Programming, Series B*, 89(2):233–250, 2001.

S. Krantz. *The Proof is in the pudding*. Springer, New York, 2010.

S. Kucherenko, P. Belotti, L. Liberti, and N. Maculan. New formulations for the kissing number problem. *Discrete Applied Mathematics*, 155(14):1837–1841, 2007.

M. Laurent and F. Vallentin. Semidefinite optimization. Technical Report Lecture Notes, CWI and TU Delft, 2012.

C. Lavor, L. Liberti, and N. Maculan. Computational experience with the molecular distance geometry problem. In J. Pintér, editor, *Global Optimization: Scientific and Engineering Case Studies*, pages 213–225. Springer, Berlin, 2006.

C. Lavor, A. Mucherino, L. Liberti, and N. Maculan. On the computation of protein backbones by using artificial backbones of hydrogens. *Journal of Global Optimization*, 50:329–344, 2011.

L. Liberti. Writing global optimization software. In L. Liberti and N. Maculan, editors, *Global Optimization: from Theory to Implementation*, pages 211–262. Springer, Berlin, 2006.

L. Liberti. Compact linearization of binary quadratic problems. *4OR*, 5(3):231–245, 2007.

L. Liberti. Reformulations in mathematical programming: Definitions and systematics. *RAIRO-RO*, 43(1):55–86, 2009.

L. Liberti. Symmetry in mathematical programming. In J. Lee and S. Leyffer, editors, *Mixed Integer Nonlinear Programming*, volume 154 of *IMA*, pages 263–286. Springer, New York, 2012.

L. Liberti. Optimization and sustainable development. *Computational Management Science*, 12(3):371–395, 2015.

L. Liberti and N. Maculan. Preface to special issue on reformulations in mathematical programming. *Discrete Applied Mathematics*, 157(6), 2009.

L. Liberti and F. Marinelli. Mathematical programming: Turing completeness and applications to software analysis. *Journal of Combinatorial Optimization*, 28(1):82–104, 2014.

L. Liberti, S. Cafieri, and F. Tarissan. Reformulations in mathematical programming: A computational approach. In A. Abraham, A.-E. Hassanien, P. Siarry, and A. Engelbrecht, editors, *Foundations of Computational Intelligence Vol. 3*, number 203 in Studies in Computational Intelligence, pages 153–234. Springer, Berlin, 2009a.

L. Liberti, C. Lavor, N. Maculan, and F. Marinelli. Double variable neighbourhood search with smoothing for the molecular distance geometry problem. *Journal of Global Optimization*, 43:207–218, 2009b.

L. Liberti, C. Lavor, N. Maculan, and A. Mucherino. Euclidean distance geometry and applications. *SIAM Review*, 56(1):3–69, 2014.

J. Löfberg. YALMIP: A toolbox for modeling and optimization in MATLAB. In *Proceedings of the International Symposium of Computer-Aided Control Systems Design*, volume 1 of *CACSD*, Taipei, 2004. IEEE.

M. López and G. Still. Semi-infinite programming. *European Journal of Operational Research*, 180(2):491–518, 2007.

N. Maculan, P. Michelon, and J. MacGregor Smith. Bounds on the kissing numbers in $\mathbb{R}^n$: Mathematical programming formulations. Technical report, University of Massachusetts, Amherst, USA, 1996.

A. Majumdar, A. Ahmadi, and R. Tedrake. Control and verification of high-dimensional systems with dsos and sdsos programming. In *Conference on Decision and Control*, volume 53, pages 394–401, Los Angeles, 2014. IEEE.

A. Makhorin. *GNU Linear Programming Kit*. Free Software Foundation, `http://www.gnu.org/software/glpk/`, 2003.

A. Man-Cho So and Y. Ye. Theory of semidefinite programming for sensor network localization. *Mathematical Programming B*, 109: 367–384, 2007.

H. Markowitz. Portfolio selection. *The Journal of Finance*, 7(1):77–91, 1952.

matlab. *MATLAB R2014a*. The MathWorks, Inc., Natick, MA, 2014.

Y. Matiyasevich. *Hilbert's Tenth Problem*. MIT Press, Boston, 1993.

J. Matoušek and B. Gärtner. *Understanding and using Linear Programming*. Springer, Berlin, 2007.

K. Menger. New foundation of Euclidean geometry. *American Journal of Mathematics*, 53(4):721–745, 1931.

M. Minsky. *Computation: Finite and infinite machines*. Prentice-Hall, London, 1967.

J. Moré and Z. Wu. Distance geometry optimization for protein structures. *Journal of Global Optimization*, 15:219–234, 1999.

mosek7. *The* `mosek` *manual, Version 7 (Revision 114)*. Mosek ApS, 2014. (`www.mosek.com`).

T. Motzkin and E. Straus. Maxima for graphs and a new proof of a theorem of Turán. *Canadian Journal of Mathematics*, 17:533–540, 1965.

J. Munkres. Algorithms for the assignment and transportation problems. *Journal of SIAM*, 5(1):32–38, 1957.

R. Reams, G. Chatham, W. Glunt, D. McDonald, and T. Hayden. Determining protein structure using the distance geometry program APA. *Computers and Chemistry*, 23:153–163, 1999.

N.V. Sahinidis and M. Tawarmalani. *BARON 7.2.5: Global Optimization of Mixed-Integer Nonlinear Programs*, User's Manual, 2005.

J. Saxe. Embeddability of weighted graphs in k-space is strongly **NP**-hard. *Proceedings of 17th Allerton Conference in Communications, Control and Computing*, pages 480–489, 1979.

T. Schlick. *Molecular modelling and simulation: an interdisciplinary guide*. Springer, New York, 2002.

A. Singer. Angular synchronization by eigenvectors and semidefinite programming. *Applied and Computational Harmonic Analysis*, 30: 20–36, 2011.

A. Soyster. Convex programming with set-inclusive constraints and applications to inexact linear programming. *Operations Research*, 21 (5):1154–1157, 1973.

A. Sutou and Y. Dai. Global optimization approach to unequal sphere packing problems in 3D. *Journal of Optimization Theory and Applications*, 114(3):671–694, 2002.

J. Tenenbaum, V. de Silva, and J. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290: 2319–2322, 2000.

A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(1):230–265, 1937.

R. Vanderbei. *Linear Programming: Foundations and Extensions*. Kluwer, Dordrecht, 2001.

S.A. Vavasis. *Nonlinear Optimization: Complexity Issues*. Oxford University Press, Oxford, 1991.

S. Venkatasubramanian and Q. Wang. The Johnson-Lindenstrauss transform: An empirical study. In *Algorithm Engineering and Experiments*, volume 13 of *ALENEX*, pages 164–173, Providence, 2011. SIAM.

J.P. Vielma. Mixed integer linear programming formulation techniques. *SIAM Review*, 57:3–57, 2015.

D. Wu, Z. Wu, and Y. Yuan. Rigid versus unique determination of protein structures with geometric buildup. *Optimization Letters*, 2(3): 319–331, 2008.

K. Wüthrich. Protein structure determination in solution by nuclear magnetic resonance spectroscopy. *Science*, 243:45–50, 1989.

Y. Yemini. The positioning problem — a draft of an intermediate summary. In *Proceedings of the Conference on Distributed Sensor Networks*, pages 137–145, Pittsburgh, 1978. Carnegie-Mellon University.

Index