Politecnico di Torino

Corso di Laurea Magistrale in

Ingegneria Matematica

Lagrangian Heuristic for Capacitated Lot Sizing Problems

Relatore:
Paolo Brandimarte

Candidato:
Federico Paolucci

Correlatore:
Edoardo Fadda

Anno Accademico 2020/2021

Contents

Chapter 1

Introduction

In today's world, inventory management is an increasingly important issue. To increase competitiveness, companies need to minimize production costs, among others. There are many things to take into consideration: on the one hand it is important that the customers' demand is met, on the other hand, to do this, sometimes you risk incurring huge inventory costs. Answering questions like when and how much to produce is the purpose of inventory management. In the field of operations research, the Lot Sizing Problem (LSP) is introduced by Wagner and Whitin [1958]. It aims to satisfy the demand of clients while minimizing setup and inventory holding costs. In this model the time is discretized in periods, whose duration, depending on the need, can vary from a few hours to several weeks. The reason for considering setup costs is that starting production in a given period may involve costs due to the ignition of the machine, the transport of material or other. These costs are paid each time a production starts, whatever the number of units produced. Obviously, the real cost that a company has to bear should also take into account the quantity produced. In models where demand is considered to be known and has to be met, however, these costs can be considered fixed, since we know the exact quantity to be produced from the beginning. In most studies, a unit inventory cost or holding cost is charged to carry one unit into stock from one period to another. Inventory costs model the cost of renting space, rather than the loss of value of goods or other.

The aim of this dissertation is to propose efficient tools and investigate different optimization methods to solve the LSP in different variants. In Chapter 2 the discussion focuses on the Deterministic Capacitated Lot Sizing Problem (CLSP) on multiple items. As proved in Bitran and Yanasse [1981]

the CLSP is NP-Hard if at least two items are considered. In our thesis work we compare the exact implementation of the model, which corresponds to a Mixed-Integer Linear Programming problem, with a Lagrangian heuristic that will be better described in the next chapter. In Chapter 3 we deal with the Stochastic Capacitated Lot Sizing Problem (SCLSP) but we will limit ourselves to the case of a single item. As you can imagine, in a stochastic environment things become more complex. Even just satisfying the demand, a constraint in the deterministic model, is no longer possible. For this second part of the discussion we will focus on Dynamic Programming(DP) and Reinforcement Learning(RL) models to deal with the problem. As for the first part, the different algorithms proposed are compared and the strengths of each is underlined.

In this thesis the mathematical and programming aspects will go hand in hand. The validation of the proposed solutions is in fact carried out through a Python code, freely accessible on request.

Chapter 2

Deterministic problem

In this chapter the CLSP is deepened and several solutions proposals are implemented. Depending on the parameters in play, the best ones will be identified and used to solve the problem. In Section 2.1 we present the mathematical model of the problem and its possible variants. This model can be implemented without too many difficulties on a commercial optimization solver. Our implementation, based on Gurobi Optimizer, is used as a benchmark to assess the quality of the solutions we find with heuristics methods. In Section 2.2 there are theoretical references to Lagrange multipliers and Lagrangian relaxation methods. The treatment is rather general and unrelated to the problem under consideration, but being a widely used method we thought it might be useful to give an overview. Finally, the relaxed model is built and examined in the following sections. In Section 2.3 we solve the model on the single item and without capacity constraint. Two different algorithms are proposed for solving the problem: the first, exact and deterministic is based on the well-known Wagner-Whitin(WW) algorithm, while the second, stochastic and approximate, is based on the Cross Entropy(CE) method. Finally in Section 2.4 we deal with the capacitated problem on multi item. The blocks exposed in the previous sections are put together to solve the CLSP by a Lagrangian relaxation procedure. The update of the multipliers and the stop conditions are investigated in detail. With Section 2.5 we conclude the chapter, showing all the results obtained with the different methods and the different choices of parameters. By helping us with graphs and tables we analyze in which situations it is better to use the procedure implemented by us and when the solver of Gurobi out performs our results.

2.1 Model

To make this work readable even for less experienced readers, before introducing the model, we give some basic information. In mathematics and computer science, an optimization problem is the problem of finding the best solution among all feasible solutions. In the simplest case, it consists of maximizing or minimizing a real function by systematically choosing input values from a set, named feasible set. Mathematically, given a function $f : \mathcal{A} \to \mathbb{R}$ from a set $\mathcal{A}$ to real number, we seek an element $x_0 \in \mathcal{A}$ such that $f(x_0) \leq f(x) \, \forall x \in \mathcal{A}$ (if minimizing), or such that $f(x_0) \geq f(x) \, \forall x \in \mathcal{A}$ (if maximizing). Function f is called objective function, while $\mathcal{A}$, the feasibile set, is often specified by a set of constraints, equalities or inequalities that its members have to satisfy.

Optimization models can be divided according to the nature of the variables, continuous or discrete and depending on the type of objective function and constraints. We can roughly say that a problem is linear when both its objective function and all its constraints are expressed by a linear relation; otherwise we will say that the problem is not linear. Moreover we can say that a problem is continuous if all its variables are continuous, that is if they assume real values. Continuous Linear programming problems can be solved efficiently using, for example, the simplex algorithms or the interior point methods. For an introduction to mathematical optimization, linear problems and simplex algorithm we recommend to see Dantzig [1965]. Interior point methods, however, are a wide class of algorithms that solve both linear and nonlinear convex optimization problems. Their use in linear programming problems can be seen in Marsten et al. [1992], while a detailed treatment can be found in Roos et al. [2005].

The CLSP can be expressed via a Mixed Integer Linear Programming (MILP) model. It is an optimization problem in which some of the variables are constrained to be integers and in which the objective function and the constraints (other than the integer constraints) are linear. The introduction of some integer variables (or binaries, as in our case), make the model extremely difficult to solve. In the paper of Bitran and Yanasse [1981] the authors study the computational complexity of CLSP under the assumptions of particular cost structures. Collateral evidence also shows that if two or more items are considered the CLSP is a nondetermistic polynomial-time hard (NP-Hard) problem. In our dissertation we consider a much more difficult version of the problem in which also setup times are considered. In this case, also test whether or not a feasible solution exists is an NP-Complete problem (see Trigeiro et al. [1989] for further informations). To deepen the

topic of computational complexity the reader can see Du and Ko [2011], while for a list of NP-Complete problem (s)he can refers to Garey and Johnson [1990].

In our problem we want to minimize the sum of inventory costs and setup costs regarding I items, over a finite time horizon discretized into T intervals. For each item the unit cost of setup f_i and the unit cost of inventory h_i are known. A setup time r'_i and a unit production time r_i are also present. The demand d_{it} for product i in the time period t has to be met. The problem is capacitated, that is we have a capacity limit, which can be seen, for example, as the number of working hours available. The number of hours available is an aggregate value and part of the goal of our work is to divide it between the production times of the items. The capacity in general may depend on the time interval in which we are and is referred to as R_t. The variable of the model are of three type: x_{it}, δ_{it}, I_{it} are the produced quantity, the setup and the inventory level, respectively, for product i in time t.

Given this notation, a natural model for the CLSP problem is

$$\begin{aligned}
\min \quad & \sum_{i=1}^{I}\sum_{t=1}^{T}(f_i\delta_{it} + h_i I_{it}), && && && (2.1)\\
\text{s.t.} \quad & I_{it} = I_{i,t-1} + x_{it} - d_{it}\,, && i = 1,\dots,I, && t = 1,\dots,T, && (2.2)\\
& \sum_{i=1}^{I}(r_i x_{it} + r'_i\delta_{it}) \le R_t\,, && && t = 1,\dots,T, && (2.3)\\
& x_{it} \le M_{it}\delta_{it}\,, && i = 1,\dots,I, && t = 1,\dots,T, && (2.4)\\
& I_{i0} = 0\,, && i = 1,\dots,I, && &&\\
& x_{it}, I_{it} \ge 0\,, && i = 1,\dots,I, && t = 1,\dots,T, &&\\
& \delta_{it} \in \{0,1\}\,, && i = 1,\dots,I, && t = 1,\dots,T. &&
\end{aligned}$$

Our goal is minimize the objective function (2.1) in which the sum of setup and holding cost is considered. Constraint (2.2) represents the evolution of the inventory. For each item separately, the inventory evolves in a natural way: the inventory level at time instant t is equal to the sum of its level at time instant $t-1$ and the produced quantity in period t, to which the demand occurred in period t must be subtracted. Note the use of the two terms *period* (or *interval*) and *instant*. Inventory is observed in the instant of time t, while production and demand occur during the time period t, which joins the time instants t and $t+1$. Finally the new inventory is observed at instant $t+1$. According to this notation we will have T time intervals and $T+1$ time instants. We also note that as the model is made, the demand

that occurred in the period t can be satisfied by using the production of the same period. This means that the lead time is equal to zero, i.e., the product is delivered immediately to customers. Constraint (2.3) represents the capacity constraint. In each time period the time spent in make the setup and produce products cannot exceed the number of available capacity R_t. The constraint (2.4) is a big M constraint used to relate variables x_{it} and δ_{it}: if for a given item in a given time period the production is strictly positive, the correspondent setup variable must be set to 1. From an optimization point of view is important make the big M values, M_{it}, as small as possible, remembering however that M_{it} has to be an upper bound on x_{it}. In fact the constraint must be active when $\delta_{it} = 0$ and it should not have effect when $\delta_{it} = 1$. This consideration leads us to think that the M value should be as large as possible. On the other hand, however, the larger the value, the more difficult the problem becomes. The reason is that the software solves the problem by relaxing the binary setup variables to continuous values and solving the resulting LP problem. If the big-M value is too large, weak linear programming relaxations are obtained, and this results in a poorly pruned branch and bound tree (Camm et al. [1990], Klotz and Newman [2013]). It is therefore important to find the minimum value of M_{it} so the constraint does not affect the production when the setup has been made. Of course, it wouldn't make sense to produce more than demand from here to the end of the time horizon, so it is possible to write

$$x_{it} \leq \left(\sum_{\tau=t}^{T} d_{i\tau} \right) \delta_{it} \,, \tag{2.5}$$

i.e., $M = \sum_{\tau=t}^{T} d_{i\tau}$. The remaining constraints give us some conditions on the domain of the variables. The number of units produced and the inventory on hand have to be positive, while the setup variables have to be binary. Surely someone might have something to say about the feasible region of the production and inventory variables. If we exclude the uncapacitated case, where it is easy to see that if the demand takes integer values then the optimal solution will have integer values of production and inventory also, in general we cannot exclude that in the optimal solution there are some fractional variables. So strictly speaking it is not correct to consider these variables as continuous, since it is not possible to produce, for example, a light bulb and a half. Imposing that the variables are integer would, however, makes the model unnecessarily slower as it is possible to see from comparisons of the resolution times of the model with continuous (MILP) or integer(ILP) variables in Table 2.1. Anyway, besides the difficulty, the

	MILP Time (s)	ILP Time (s)
$I = 100, \quad T = 30$	1.15	2.21
$I = 300, \quad T = 30$	7.93	11.22
$I = 500, \quad T = 30$	9.65	17.99
$I = 1000, \quad T = 30$	28.48	38.30

Table 2.1: Time comparison in solving MILP and ILP model. Average results on 10 instances.

model with integer variables is not treated because normally CLSP is used in reference to companies with high production levels and producing 1000 pieces instead of 1001 might not be such a major problem.

Sometimes, as in Brandimarte [2006], CLSP is implemented using plant location formulation. As we mentioned earlier, in a MILP problem, in the presence of a constraint like $x \leq M\delta$, we would like to find the smallest possible value for M. A classic plant location model is a model whose objective is to decide whether or not to build some factories to serve a number of retailers. In addition to the decision on the plants construction, we must also decide how much each plant have to produce to serve every retailer. The CLSP can be seen from a plant location model perspective if we think that production should be transported in time, rather than in space. To do this, we disaggregate the production variable x_{it} into other variables y_{itp}, denoting the amount of item i produced during time period t in order to meet demand in the current or in a future time period p ($t \leq p$). In this view the setup variable represent the opening of a plant while the variable x_{it} would represent the amount produced in the factory t and that must be divided between retailers to meet the demand, i.e., $x_{it} = \sum_{t \leq p} y_{itp}$. The setup cost during a time period corresponds to the fixed cost of opening a plant, and transportation instead of inventory costs incur. This result in the following model

$$\min \quad \sum_{i=1}^{I}\sum_{t=1}^{T}\left(f_i\delta_{it}+\sum_{p\geq t}(p-t)h_i y_{itp}\right), \tag{2.6}$$

$$\sum_{i=1}^{I}\left(\sum_{p\geq t} r_i y_{itp}+r_i'\delta_{it}\right)\leq R_t, \quad t=1,\ldots,T, \tag{2.7}$$

$$\sum_{t\leq p} y_{itp}=d_{ip}, \quad i=1,\ldots,I, \quad p=1,\ldots,T, \tag{2.8}$$

$$y_{itp}\leq d_{ip}\delta_{it}, \quad i=1,\ldots,I, \quad t=1,\ldots,T, \quad p=t,\ldots,T, \tag{2.9}$$

$$y_{itp}\geq 0, \quad i=1,\ldots,I, \quad t=1,\ldots,T, \quad p=t,\ldots,T,$$

$$\delta_{it}\in\{0,1\}, \quad i=1,\ldots,I, \quad t=1,\ldots,T.$$

With this formulation we can also get rid of the inventory variables. We can understand the inventory level immediately from the disaggregated variables: in objective function (2.6) we use the fact that the amount y_{itp} remains in stock for $p-t$ periods to compute the holding costs. Constraint (2.7) takes the role of constraint (2.3) limiting the working hours used. Constraint (2.8) tell us that the demand for item i in time period p have to be satisfied using the quantity produced in the current or in past time periods. The big M coefficient in (2.9) has a smaller value of M, in fact with this formulation we can simply say that the quantity produced at time period t to met demand in time period p must be less or equal to the demand in time period p.

The main drawback of this model, which surely is evident to a careful reader, is the need to use many more variables and many more constraints than before. In fact, compared to $3IT+I$ variables and $5IT+I+T$ constraints of the first model, the $\frac{1}{2}IT^2+IT$ variables and $IT^2+\frac{3}{2}IT+T$ constraints of the second, are of larger order of magnitude. In general, when T increases, on the one hand the plant location formulation has an increasingly more convenient big M constraint than the classic formulation, on the other it has an increasing number of variables and constraints. In our experiments the number of time periods considered is about 30, while the number of products is in the order of thousands. In this case the first model has about 90000 variables and 151000, constraints, while the second has 480000 and 945000 respectively. The high number of constraints of the plant location formulation means that the model, although very fast in being solved, is slow to build. Moreover, in recent years optimization software, such as Gurobi, have made great strides forward, and are now able to work with the original model without having performance problems in solving the prob-

		Normal formulation time (s)			Strong formulation time (s)		
		Build	Solve	Total	Build	Solve	Total
$I = 100,$	$T = 30$	0.42	1.46	1.88	5.01	0.69	$5,70$
$I = 300,$	$T = 30$	1.33	4.74	6.07	15.47	2.40	$17,87$
$I = 500,$	$T = 30$	2.25	9.13	11.38	26.01	3.07	29.08
$I = 1000,$	$T = 30$	4.54	19.77	24.32	51.56	6.94	58.51

Table 2.2: Time comparison for the two different formulations of the CLSP.

lem. In Table 2.2 a time comparison between the original model and the one based on the plant location formulation is made. As we can see, despite the resolution time of the second model is much lower than that of the first, if we consider the overall time to build and solve the model, the first is preferable. Precisely for this reason, for all the subsequent treatment, we always use the classical formulation.

2.2 Lagrangian Relaxation

In this section our intention is to give a narrow, but rather general treatment of the Lagrangian relaxation method and then see how to apply it to our problem. In order to have a general treatment, let's imagine that we consider the problem

$$\begin{aligned} \min \quad & f(\boldsymbol{x}), \\ \text{s.t.} \quad & h_i(\boldsymbol{x}) \leq 0, \qquad i = 1, \ldots, m, \\ & \boldsymbol{x} \in \mathcal{S} \subseteq \mathbb{R}^n, \end{aligned} \tag{2.10}$$

where f is a scalar function and $\boldsymbol{x}$ is a vector. Lagrangian relaxation is a procedure that involves building a lower bound of the objective function and trying to improve it until it is arbitrarily close to a feasible solution. Suppose that the problem is difficult to solve and that in particular the difficulty is in meeting the inequality constraints $h_i(\boldsymbol{x}) \leq 0$ and not in making $\boldsymbol{x}$ belongs to $\mathcal{S}$. The idea that applies, classic in the field of optimization, is to pretend that there are no constraints, but to penalize in some way the fact that you are not meeting them. The most obvious way to act is therefore to insert each constraint in the objective function and penalize with a coefficient when it is exceeded. So we can introduce the Lagrangian function

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\mu}) = f(\boldsymbol{x}) + \boldsymbol{\mu}^T \boldsymbol{h}(\boldsymbol{x}), \tag{2.11}$$

where $\boldsymbol{\mu}$ is the m-vector of Lagrangian multipliers and $\boldsymbol{h}$ is the m-vector of constraints. Obviously, although minimizing the Lagrangian function is not the same, it is somewhat related to the initial problem. The Lagrangian function, however, depends on two variables. If we fix the value of $\boldsymbol{\mu}$ and minimize it respect to the variable $\boldsymbol{x}$, what we get is a function called dual function.

$$\begin{aligned} w(\boldsymbol{\mu}) = \min \quad & \mathcal{L}(\boldsymbol{x}, \boldsymbol{\mu}), \\ \text{s.t.} \quad & \boldsymbol{x} \in \mathcal{S}. \end{aligned} \tag{2.12}$$

Theorem 2.1. *If $\boldsymbol{x}^*$ is an optimal value for the Problem* (2.10) *and $w(\boldsymbol{\mu})$ is the related dual function then*

$$w(\boldsymbol{\mu}) \leq f(\boldsymbol{x}^*), \qquad \forall \boldsymbol{\mu} \geq 0.$$

Proof. The following inequalities hold

$$\begin{aligned} f(\boldsymbol{x}^*) &= \begin{pmatrix} \min & f(\boldsymbol{x}), \\ \text{s.t.} & h_i(\boldsymbol{x}) \leq 0, & i = 1, \ldots, m, \\ & \boldsymbol{x} \in \mathcal{S}. \end{pmatrix} \\ &\geq \begin{pmatrix} \min & f(\boldsymbol{x}), \\ \text{s.t.} & \mu_i h_i(\boldsymbol{x}) \leq 0, & i = 1, \ldots, m, \\ & \boldsymbol{x} \in \mathcal{S}. \end{pmatrix} \\ &\geq \begin{pmatrix} \min & f(x) + \sum_i \mu_i h_i(x), \\ \text{s.t.} & \mu_i h_i(\boldsymbol{x}) \leq 0, & i = 1, \ldots, m, \\ & \boldsymbol{x} \in \mathcal{S}. \end{pmatrix} \\ &\geq \begin{pmatrix} \min & f(\boldsymbol{x}) + \sum_i \mu_i h_i(\boldsymbol{x}), \\ \text{s.t.} & \boldsymbol{x} \in \mathcal{S}. \end{pmatrix} \\ &= w(\boldsymbol{\mu}). \end{aligned}$$

The first inequality is due to the fact that when multiplying the function h_i for the scalar μ_i we expand the feasible region, in fact if $\mu_i > 0$ the constraint does not change, while if $\mu_i = 0$ the constraint is automatically satisfied. If the region of feasibility is larger, the minimum of the new problem is smaller or at most equal to the minimum of the old problem. The second inequality is due to the fact that we are adding a non positive quantity to the objective function, so it decreases. At the end we remove the constraint, expanding

much more the feasible space. The last problem is, by definition, the dual function, and so we prove the theorem. □

So, the idea is to define a second problem, called dual problem with the objective of pushing the dual value up. In fact thanks to the theorem, we are sure that it will never exceed the optimal value of the primal problem. Then we want to solve the following dual problem

$$\begin{aligned} \max \quad & w(\boldsymbol{\mu}), \\ \text{s.t.} \quad & \boldsymbol{\mu} \geq 0. \end{aligned}$$

It would be a great result if we could say that the maximum of the dual problem was equal to the minimum of the primal problem, i.e., if

$$f(\boldsymbol{x}^*) = w(\boldsymbol{\mu}^*)$$

but this is not true in general. If not, we can anyway maximize the dual function and find a good lower bound for the primal problem.

From a computation point of view often to maximize the dual objective function an iterative procedure is used. Obviously this approach make sense if and only if we are able to solve the problem (2.12) much more easily than the original problem (2.10).

Lagrangian procedure

1: Chose an initial vector $\boldsymbol{\mu}^{(0)}$,
2: $k = 0$,
3: **repeat**
4: solve the problem (2.12) for a given $\boldsymbol{\mu}^{(k)}$,
5: find new multipliers $\boldsymbol{\mu}^{(k+1)}$,
6: $k = k + 1$,
7: **until** some termination condition holds.

In general, also if we are able to minimize in a proper way, such a method could suffer from the problem of local maximum, but this is not the case, in fact the following applies

Theorem 2.2. *What ever the data $\mathcal{S}, f, \boldsymbol{h}$ in Problem* (2.10)*, the dual function $w(\boldsymbol{\mu})$ is concave (but not necessarily differentiable).*

Proof. It is enough to notiche that $\mathcal{L}(\boldsymbol{x}, \boldsymbol{\mu})$ is an affine function of $\boldsymbol{\mu}$. Then the dual function is the pointwise infimum of a family of affine functions.

But the pointwise infimum of concave functions is concave (see Niculescu and Persson [2004] or Borwein and Lewis [2010] for further information), so $w(\boldsymbol{\mu})$ is concave. □

Thanks to this theorem the problem of local maximum does not arise and one can be sure, if the procedure converges, to converge at the maximum of the dual function. The critical point in the procedure is how to update the multipliers, while normally the choice of their initial values does not affect the final result significantly: multipliers are usually initialized all equal to zero or randomly chosen. To update them, the basic idea on which most of methods are based, is that when the solution found does not respect a constraint, the associated multiplier is increased to further penalize capacity utilization, while when the solution respects a constraint, multipliers shall be kept constant or decreased to try to make the solution less conservative. The literature is plenty of methods to update multipliers. Here we propose two classes, the subgradient methods and the bundle methods. For a comprehensive treatment of Lagrangian relaxation reader can see Lemaréchal [2013], while in Boyd et al. [2003] subgradient methods are thrash out.

Subgradient Method

Definition 2.1. *Let f be a function and $\boldsymbol{x}$ a point in its domain $\mathcal{D}$. The subgradient of f at $\boldsymbol{x}$ is any vector $\boldsymbol{g}$ that satisfies the inequality*

$$f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \boldsymbol{g}^T(\boldsymbol{y} - \boldsymbol{x}) \quad \forall \boldsymbol{y} \in \mathcal{D}.$$

Equivalently is possible to define the supergradient of f at $\boldsymbol{x}$, that is any vector $\boldsymbol{g}$ that satisfies the inequality

$$f(\boldsymbol{y}) \leq f(\boldsymbol{x}) + \boldsymbol{g}^T(\boldsymbol{y} - \boldsymbol{x}) \quad \forall \boldsymbol{y} \in \mathcal{D}.$$

The subgradient extends the gradient for non-differentiable functions. The subgradient method is an iterative procedure for minimizing a non differentiable convex function ϕ. At each iteration the subgradient in the current point is computed and a new point is found according with the direction of the subgradient. The simple updating rule performed is

$$\boldsymbol{y}^{(k+1)} = \boldsymbol{y}^{(k)} - \alpha_k \boldsymbol{g}^{(k)},$$

where $\boldsymbol{g}^{(k)}$ is a subgradient of function ϕ in point $\boldsymbol{y}^{(k)}$. If, as in our case, we wish to maximize a concave function, instead, we can use the supergradient. The update becomes

$$\boldsymbol{y}^{(k+1)} = \boldsymbol{y}^{(k)} + \alpha_k \boldsymbol{g}^{(k)},$$

where $\boldsymbol{g}^{(k)}$ is a supergradient of function ϕ in point $\boldsymbol{y}^{(k)}$. In the case of the function $w(\boldsymbol{\mu})$ the supergradient is easy to find, in fact the following theorem holds.

Theorem 2.3. *Let $\boldsymbol{x}^*$ be an optimal solution of Problem (2.12) for a multiplier vector $\hat{\boldsymbol{\mu}}$. Then $\boldsymbol{h}(\boldsymbol{x}^*)$ is a supergradient of the dual function at $\hat{\boldsymbol{\mu}}$.*

Proof. Let $\boldsymbol{\mu} \geq 0$ be any vector of multipliers and consider the quantity $\mathcal{L}(\boldsymbol{x}^*, \boldsymbol{\mu})$. $\boldsymbol{x}^*$ was optimal for $\hat{\boldsymbol{\mu}}$ but in general it isn't optimal for $\boldsymbol{\mu}$. So it follows that

$$\begin{aligned} w(\boldsymbol{\mu}) &= \begin{pmatrix} \min & \mathcal{L}(\boldsymbol{x}, \boldsymbol{\mu}), \\ \text{s.t.} & \boldsymbol{x} \in \mathcal{S}. \end{pmatrix} \\ &\leq \mathcal{L}(\boldsymbol{x}^*, \boldsymbol{\mu}) \\ &= f(\boldsymbol{x}^*) + \boldsymbol{\mu}^T \boldsymbol{h}(\boldsymbol{x}^*) \\ &= f(\boldsymbol{x}^*) + \boldsymbol{\mu}^T \boldsymbol{h}(\boldsymbol{x}^*) + \hat{\boldsymbol{\mu}}^T \boldsymbol{h}(\boldsymbol{x}^*) - \hat{\boldsymbol{\mu}}^T \boldsymbol{h}(\boldsymbol{x}^*) \\ &= f(\boldsymbol{x}^*) + \hat{\boldsymbol{\mu}}^T \boldsymbol{h}(\boldsymbol{x}^*) + \boldsymbol{h}^T(\boldsymbol{x}^*)(\boldsymbol{\mu} - \hat{\boldsymbol{\mu}}) \\ &= w(\hat{\boldsymbol{\mu}}) + \boldsymbol{h}^T(\boldsymbol{x}^*)(\boldsymbol{\mu} - \hat{\boldsymbol{\mu}}) \end{aligned}$$

Then for any vector $\boldsymbol{\mu}$

$$w(\boldsymbol{\mu}) \leq w(\hat{\boldsymbol{\mu}}) + \boldsymbol{h}^T(\boldsymbol{x}^*)(\boldsymbol{\mu} - \hat{\boldsymbol{\mu}}),$$

i.e., $\boldsymbol{h}(\boldsymbol{x}^*)$ is a subgradient of the dual function $w(\boldsymbol{\mu})$ at $\hat{\boldsymbol{\mu}}$. □

So at iteration k we can use the update formula

$$\boldsymbol{\mu}^{(k+1)} = \boldsymbol{\mu}^{(k)} + \alpha_k \boldsymbol{h}(\boldsymbol{x}^*), \tag{2.13}$$

To avoid confusion, also if we have to maximize the dual function and so we use the supergradient, we talk about the subgradient method.

One of the most challenging problem in using this algorithm is decide which value of step lengths α_k to use. One of the most popular policies is the square summable but not summable step size, i.e.,

$$\sum_{k=1}^{\infty} \alpha_k = \infty, \qquad \sum_{k=1}^{\infty} \alpha_k^2 < \infty.$$

Following this rule a possible choice could be $\alpha_k = \frac{a}{b+k}, \quad a > 0, \quad b \geq 0$. Different policies can be found in Boyd et al. [2003]. An interesting choice

is the one used in Süral et al. [2009] where the step length size is calculated according to the primal and dual functions with the following formula:

$$\alpha_k = \alpha \frac{f(\bar{\boldsymbol{x}}) - w(\boldsymbol{\mu}^{(k)})}{||\boldsymbol{g}||}, \tag{2.14}$$

where $\bar{\boldsymbol{x}}$ is the best current feasible solution and $w(\boldsymbol{\mu}^{(k)})$ is the current objective value of the relaxed problem, i.e., it is a lower bound for the optimal value. The idea is that when the gap between the lower bound and the upper bound is very small, it means that we are close to the optimal, so the step size becomes smaller so as to not get too far from the current solution. At the same time, if $\boldsymbol{\mu}^{(k)}$ is a point where the gradient has a large norm, we would like to move slowly, because the objective function changes quickly, whereas if $\boldsymbol{\mu}^{(k)}$ is a point where the gradient has a small norm, means that the objective function is almost flat and it is possible to use a larger step length. Note that to use this rule, we need to have a feasible primal solution $\bar{\boldsymbol{x}}$. If this is not available, we could use an infeasible solution, but of course we would no longer be sure that $f(\bar{\boldsymbol{x}})$ is an upper bound for the optimal value $f(\boldsymbol{x}^*)$. In this case the step length would be underestimated.

In this work we do not discuss the convergence of the procedure, but in the literature there are several results in this regard (Boyd et al. [2003]). If f is regular enough and square summable but not summable step size are chosen, the algorithm is guaranteed to converge to the optimal value $\boldsymbol{\mu}^*$.

In this sense, for the termination of the procedure, a reasonable condition could be

$$\frac{|w(\boldsymbol{\mu}^{(k+1)}) - w(\boldsymbol{\mu}^{(k)})|}{|w(\boldsymbol{\mu}^{(k)})|} \leq \epsilon.$$

With this formula, we stop the algorithm when the percentage change in the dual function falls below a tolerance ϵ. If the convergence to the optimal value is not guaranteed, the stopping criterion may be replaced by

$$\frac{||\boldsymbol{\mu}^{(k+1)} - \boldsymbol{\mu}^{(k)}||}{||\boldsymbol{\mu}^{(k)}||} \leq \epsilon.$$

A different convergence criterion that can be used is the following:

$$\frac{|w(\boldsymbol{\mu}^{(k+1)}) - f(\bar{\boldsymbol{x}})|}{|w(\boldsymbol{\mu}^{(k+1)})|} \leq \epsilon.$$

This condition is stronger, but if it holds, it guaranteed a bound in term of

f. In fact, if we are able to say that $f(\boldsymbol{x}^*) = w(\boldsymbol{\mu}^*)$ it is possible to write

$$\begin{aligned}
\epsilon &\geq \frac{|w(\boldsymbol{\mu}^{(k+1)}) - f(\bar{\boldsymbol{x}})|}{|w(\boldsymbol{\mu}^{(k+1)})|} \\
&= \frac{|w(\boldsymbol{\mu}^{(k+1)}) - w(\boldsymbol{\mu}^*) + f(\boldsymbol{x}^*) - f(\bar{\boldsymbol{x}})|}{|w(\boldsymbol{\mu}^{(k+1)})|} \\
&= \frac{|w(\boldsymbol{\mu}^{(k+1)}) - w(\boldsymbol{\mu}^*)|}{|w(\boldsymbol{\mu}^{(k+1)})|} + \frac{|f(\boldsymbol{x}^*) - f(\bar{\boldsymbol{x}})|}{|w(\boldsymbol{\mu}^{(k+1)})|} \\
&\geq \frac{|w(\boldsymbol{\mu}^{(k+1)}) - w(\boldsymbol{\mu}^*)|}{|w(\boldsymbol{\mu}^*)|} + \frac{|f(\boldsymbol{x}^*) - f(\bar{\boldsymbol{x}})|}{|f(\boldsymbol{x}^*)|},
\end{aligned}$$

then

$$\frac{|f(\boldsymbol{x}^*) - f(\bar{\boldsymbol{x}})|}{|w(\boldsymbol{x}^*)|} \leq \epsilon.$$

If the strong duality does not holds, i.e. if $w(\boldsymbol{\mu}^*) < f(\boldsymbol{x}^*)$, similar reasoning can be made.

Bundle Methods

There are numerous approach for maximizing non smooth concave functions. Here we propose an example of bundle methods. These methods are also based on the concept of subgradient: subgradient directions from past iterations are collected in a bundle and from them, performing an easy optimization problem, a new direction is obtained. To explain the method we require the definition of ϵ - subdifferential

$$\partial_\epsilon \mathcal{L}(\boldsymbol{\mu}) \equiv \{\boldsymbol{g} \in \mathbb{R}^n | \mathcal{L}(\bar{\boldsymbol{\mu}}) \leq \mathcal{L}(\boldsymbol{\mu}) + \langle \boldsymbol{g}, \bar{\boldsymbol{\mu}} - \boldsymbol{\mu} \rangle + \epsilon \quad \forall \bar{\boldsymbol{\mu}} \in \mathbb{R}^n\}. \tag{2.15}$$

Element in $\partial_\epsilon \mathcal{L}(\boldsymbol{\mu})$ are called ϵ - subgradients, while the ϵ - directional derivative along the direction $\boldsymbol{d}$ at $\boldsymbol{\mu}$ is defined as

$$\mathcal{L}'_\epsilon(\boldsymbol{\mu}, \boldsymbol{d}) \equiv \sup_{t>0} \frac{\mathcal{L}(\boldsymbol{\mu} + t\boldsymbol{d}) - L(\boldsymbol{\mu}) - \epsilon}{t}.$$

As is possible to see in Zhao and Luh [2003] the directional derivative can be rewrite as

$$\mathcal{L}'_\epsilon(\boldsymbol{\mu}, \boldsymbol{d}) = \inf_{g \in \partial_\epsilon \mathcal{L}(\boldsymbol{\mu})} g'\boldsymbol{d},$$

so if a direction $\boldsymbol{d}$ such that $\mathcal{L}'_\epsilon(\boldsymbol{\mu}, \boldsymbol{d}) > 0$ is found, this means that the dual cost can be increased by at least ϵ. Therefore, it is desirable to select a search

direction $\boldsymbol{d}^*$ such that the directional derivative is maximized:

$$\begin{aligned}
\boldsymbol{d^*} &= \arg\left\{\max_{||\boldsymbol{d}||=1} \mathcal{L}'_\epsilon(\boldsymbol{\mu}, \boldsymbol{d})\right\} \\
&= \arg\left\{\max_{||\boldsymbol{d}||=1} \inf_{\boldsymbol{g}\in\partial_\epsilon\mathcal{L}(\boldsymbol{\mu})} \boldsymbol{g}'d\right\} \\
&= \arg\left\{\inf_{\boldsymbol{g}\in\partial_\epsilon\mathcal{L}(\boldsymbol{\mu})} \max_{||\boldsymbol{d}||=1} \boldsymbol{g}'\boldsymbol{d}\right\} \\
&= \arg\left\{\inf_{\boldsymbol{g}\in\partial_\epsilon\mathcal{L}(\boldsymbol{\mu})} ||\boldsymbol{g}||\right\}
\end{aligned}$$

Therefore, this $\boldsymbol{d^*}$ is the ϵ - subgradient with the smallest norm. Generally the ϵ - subgradient is hard to obtain and the idea is to approximate it through a sequence of subgradients. At every iteration the subgradient is accumulate in a bundle $B = \{g_1, \ldots, g_b\}$ and to approximate $\partial_\epsilon\mathcal{L}(\mu)$ the convex hull of bundle's elements is used

$$P_b = \left\{\boldsymbol{g}|\boldsymbol{g} = \sum_{i=1}^{b} \alpha_i \boldsymbol{g_i},\ \boldsymbol{g_i} \in B,\ \alpha_i \geq 0, \quad \sum_{i=1}^{b} \alpha_i = 1, \quad \sum_{i=1}^{b} \alpha_i e_i \leq \epsilon\right\},$$

where e_i is the linearization error for element i,

$$e_i = \mathcal{L}(\mu_i) + \langle \boldsymbol{g_i}, \boldsymbol{\mu} - \boldsymbol{\mu_i}\rangle - \mathcal{L}(\boldsymbol{\mu}).$$

Once $\boldsymbol{d}^*$ was found, we use it to update lagrangin multiplier

$$\boldsymbol{\mu}^{(k+1)} = \boldsymbol{\mu}^{(k)} + \alpha_k \boldsymbol{d}^{*(k)}, \tag{2.16}$$

where the step size α_k is chosen in the same way like in the subgradient method. If the problem we build is infeasible and no $\boldsymbol{d^*}$ can be found, the lagrangian multipliers are not updated. In any case at any iteration the subgradient is computed and it is added to the bundle P_b. This method, by accumulating gradients from past iterations and choosing the direction to take, manages to get a better Lagrange multiplier update, compared to a simple subgradient method. However, numerous iterations may be required before multipliers are updated, which means that the benefit of a better upgrade is lost. From our tests the subgradient method was preferable, but in the literature there are several techniques to try to reduce the number of iterations in which the bundle method fails to update the multipliers, to make it more competitive (for more information we always recommend to read Zhao and Luh [2003]). The stop condition can be chosen with the same criteria as for the subgradient method.

2.3 Uncapacitated Single Item Problem

The uncapacitated LSP (ULSP) on a single item is a classic optimization problem, introduced by Harvey M. Wagner and Thomson M. Whitin in 1958. It can be formulated as follow

$$\begin{aligned} \min \quad & \sum_{t=1}^{T}(f_t\delta_t + h_t I_t), && (2.17)\\ \text{s.t.} \quad & I_t = I_{t-1} + x_t - d_t, && t = 1,\dots,T,\\ & x_t \le M_t\delta_t, && t = 1,\dots,T,\\ & I_0 = 0, && \\ & x_t, I_t, \ge 0, && t = 1,\dots,T,\\ & \delta_t \in \{0,1\}, && t = 1,\dots,T. \end{aligned}$$

The resolution of the ULSP is not only preparatory to the CLSP, but often is the main building block for its resolution. Also our procedure is based on this philosophy: the problem is first decomposed through the Lagrangian relaxation and is then solved individually on each product without considering any capacity constraint. That is why in this section we will focus on the resolution of the ULSP. In particular we will propose two different methods, which will then be compared. The first method is based on the original algorithm proposed in Wagner and Whitin [1958] which uses dynamic programming to solve the problem. Some theoretical notions about DP can be found in the second chapter, otherwise we suggest to see Brandimarte [2021]. The second method is a probabilistic algorithm that is based on the concept of cross entropy, of which we will give some theoretical rudiments.

2.3.1 Wagner Whitin Algorithm

One method of solving the ULSP is to enumerate the 2^T combinations of either order or not in each period. The Wagner Whitin algorithm evolves from a DP characterization of an optimal policy and it is able to find an optimal solution in a maximum of $T(T+1)/2$ steps.

We start from the known functional equation

$$g_t(s) = \min_{\substack{x_t \ge 0 \\ s+xt \ge d_t}} h_t s + f_t\delta_t + g_t(s + x_t - d_t), \qquad (2.18)$$

representing the minimal cost policy from the period t to the period T, when

the inventory at the beginning of time t is $I_{t-1} = s$. In period T we have

$$g_T(s) = \min_{\substack{x_T \geq 0 \\ s + xT \geq d_T}} h_T s + f_T \delta_T.$$

Using this functional equation it is possible to build a DP algorithm that works backward in time. However, taking into account the properties of our problem, it is possible to build an even simpler approach. Let us suppose for the sake of simplicity $d_1 > 0$, assumption that we eliminate later.

Theorem 2.4. *Consider the Problem* (2.17). *There exists an optimal program such that* $I_{t-1}x_t = 0$, $t = 1, \ldots, T$.

Proof. Assume for contradiction that exist an optimal program for which at beginning of time period t we have $I_{t-1} > 0$ unity in the inventory and we place and order of $x_t > 0$ (i.e., $I_{t-1}x_t > 0$). Then there is a cheaper policy that consists of including I_{t-1} units in the production of the period t, saving a cost of $I_{t-1}h_{t-1}$. □

Note that the theorem is given in the case in which we assume that the producing and selling cost are constant through the time. Anyway, is possible to show that a sufficient condition for which the theorem is satisfied is that the structure of costs is concave, i.e., if the marginal costs are not increasing. In particular is not relevant that production costs are the same in different time periods (see Wagner [1959] and Zangwill [1968]).

Corollary 2.4.1. *There exists an optimal program such that for all* t

$$x_t = 0 \quad or \quad x_t = \sum_{i=t}^{k} d_i, \quad t \leq k \leq T.$$

Proof. Since demand must be met for each period, any other value of x_t implies that exists a period $t^* \geq t$ such that $I_{t^*-1}x_{t^*} > 0$, that is not possible for Theorem 2.4. □

The implication of Corollary 2.4.1 is that we can limit the values of s in Eq. (2.18) for period t between zero and the cumulative demand for t up to T, and the only $T(T+1)/2$ different values of s are examined over the entire T periods. Thanks to these observations (and some other considerations that can be found in Wagner and Whitin [1958]), if we denote as $F(t)$ the

minimal cost program for period 1 through t, it is possible to write

$$F(t) = \min \left\{ \min_{1 \le j < t} \left\{ f_j + \sum_{l=j}^{t-1} \sum_{k=l+1}^{t} h_l d_k + F(j-1) \right\}, \quad f_t + F(t-1) \right\} \tag{2.19}$$

where $F(0) = 0$ e $F(1) = f_1$. That is, the minimum cost for the first t periods includes a fixed cost in the period j, plus the cost of meeting demand in the periods $k = j+1, \ldots, t$, by carrying inventory from period j, plus the cost of adopting an optimal policy in periods 1 through $j-1$. Adopting this functional equation is possible starting at time $t = 1$ and go forward in time. At any time instant t only t different policies need to be considered. Moreover, as shown in the paper, if at period $\bar{t}$ the minimum in Eq. (2.19) occurs for $j = t^* \le \bar{t}$ then in periods $t > \bar{t}$ it is sufficient to consider only $t^* \le j \le t$. This means that if it is optimal to have a setup cost at time $t^* \le \bar{t}$ when periods 1 through t^* are considered, then we may let $x_{t^*} > 0$ in the T period model without loosing optimality. This statement, known as the Planning Horizon Theorem, can be used to reduce the number of policies to compute.

In our problem we always consider the holding unitary cost as constant in time, and so Eq. (2.19) can be rewrite as

$$F(t) = \min \left\{ \min_{1 \le j < t} \left\{ f_j + h \sum_{k=j+1}^{t} (k-j) d_k + F(j-1) \right\}, \right.$$
$$\left. f_t + F(t-1) \right\}, \tag{2.20}$$

if, as done so far, we assume that in the first step the demand is not zero. Suppose now that $d_1 = 0$, and the first non zero demand appear in time period $\bar{t}$. It is sufficient to set $x_t = 0$, $t \le \bar{t}$, $F(\bar{t}) = f_{\bar{t}}$ and consider Eq. (2.20) only for periods from $\bar{t}$ to T. Alternatively is also possible to slightly modify the recursion formula and find a valid formulation. Let s_{jt} be 1 if a positive demand occur between time j and time t, 0 otherwise. Then $F(t)$ can be computed as

$$F(t) = \min \left\{ \min_{1 \le j < t} \left\{ f_j s_{jt} + h \sum_{k=j+1}^{t} (k-j) d_k + F(j-1) \right\}, \right.$$
$$\left. s_{tt} f_t + F(t-1) \right\}.$$

The term s_{jt} prevent to mistakenly add a set up cost when there is no demand in period between j and t. Anyway, if we use this recursion, the Planning Horizon Theorem does not hold, so the Eq. (2.20) is preferable, with the care of start the algorithm in the time period in which the first non zero demand occurs.

Wagner Whitin algorithm

Set $F_0 = 0$,
for $t \in [1, \dots, T]$ **do**
    for $j = 1 \in [1, \dots, t]$ **do**
        $c_j = f_j + h \sum_{k=j+1}^{t} (k-j) d_k + F_{j-1}$,
    end for
    $F_t = \min_{1 \leq j \leq t} c_j$,
end for

To help the reader to better understand the algorithm, here is a small numerical example, that can be read in Table 2.3. As done from Wagner and Whitin we use the notation $(1, 2, \dots, t^*), t^* + 1, t^* + 2, \dots, \bar{t}$ to indicate that an order is placed in period $t^* + 1$ to cover the demand in periods $t^* + 1, t^* + 2, \dots, \bar{t}$ and the optimal policy is adopted for periods 1 through t^*. Let considering a constant set up cost $f = 6$ and an holding cost $h = 1$, while the demand is specified in the table. We write in the third row the possible cost for different strategies and in the last row the minimum cost and the optimal policy, computing using Eq. (2.20).

From the last column of Table 2.3 we know that the optimal solution has a cost of 16 and we know that we have to order in period 6 and follow the optimal policy for the first 5. In the column related to $t = 5$ we understand that the optimal policy is to order in time $t = 3$ to cover demand until period 5 and follow the optimal policy for the first 2 periods. In the column related to $t = 2$ we discover that there is nothing to do, in fact in period 1 and 2 the demand is zero, so there are no cost. In the example we compute the cost of each possible strategy, but the ones in square brackets are the policies that it is possible to avoid to compute using the Planning Horizon Theorem. Despite Wagner Whitin's algorithm exploits the good properties of the problem to decrease computational effort, several improvements to the implementation have been proposed over the years. In Heady and Zhu [1994] an algorithm with execution time that is approximately linear in the number of time periods is proposed. Similar result are found also in Sajadi et al. [2009], where the Planning Horizon Theorem and the Economic Part Period concept are used to reduce the burden of the computations. In our

t	1	2	3	4	5	6
demand	0	0	14	0	2	5
	0	0	[34]	[34]	[42]	[67]
		0	20	[20]	[26]	[46]
			6	6	10	25
				12	14	24
					12	21
						16
Minimum cost	0	0	6	6	10	16
Optimal policy	-	-	(12)$\underline{3}$	(12)$\underline{34}$	(12)$\underline{345}$	(12345)$\underline{6}$

Table 2.3: Example of Wagner Whitin algorithm.

work, to maintain a good readability of the code, it was decided to implement the basic algorithm of Wagner Whitin, but with a little bit of programming effort you could switch to an improved implementation and save computing time.

2.3.2 Cross Entropy

Sometimes, when a deterministic algorithm is computationally expensive, it might be convenient to use a stochastic algorithm: accuracy is sacrificed to get results faster. On this track we implement an algorithm based on the concept of Cross Entropy.

The CE method was proposed by Rubinstein [1997] as an adaptive importance sampling procedure for the estimation of rare event probabilities. It is used to change the sampling distribution of the random search so that the rare event is more likely to occur. The goal of this method is estimates a sequence of sampling distributions that converges to a distribution with probability mass concentrated in a region of near-optimal solutions. The explanation of the CE procedure is taken from De Boer et al. [2005].

Consider the estimation of the probability

$$\ell = \mathbb{P}\left(S(\boldsymbol{X}) \leq \gamma\right) = \mathbb{E}\left[\mathbb{I}_{\{S(\boldsymbol{X})\leq\gamma\}}\right] = \int \mathbb{I}_{\{S(\boldsymbol{x})\leq\gamma\}} f(\boldsymbol{x}, \boldsymbol{u})\, dx,$$

where S is a real-valued function, γ is a threshold or level parameter, and the random variable $\boldsymbol{X}$ has probability density function $f(\cdot, \boldsymbol{u})$, which is parameterized by a finite-dimensional real vector $\boldsymbol{u}$. In an important sampling procedure we need to find a pdf g such that if $g = 0$ then $\mathbb{I}_{\{S(\boldsymbol{X})\leq\gamma\}} f(\boldsymbol{x}, \boldsymbol{u}) = 0$.

We can now represent ℓ using the pdf g

$$\ell = \int \frac{f(\boldsymbol{x},\boldsymbol{u})\mathbb{I}_{\{S(\boldsymbol{x})\leq\gamma\}}}{g(\boldsymbol{x})} g(\boldsymbol{x})\, dx = \mathbb{E}\left[\frac{f(\boldsymbol{X},\boldsymbol{u})\mathbb{I}_{\{S(\boldsymbol{X})\leq\gamma\}}}{g(\boldsymbol{X})}\right], \quad \boldsymbol{X}\sim g.$$

If now we consider $\boldsymbol{X}_1,\ldots,\boldsymbol{X}_N \overset{\text{iid}}{\sim} g$, an unbiased estimator of ℓ is given by

$$\hat{\ell} = \frac{1}{n}\sum_{i=1}^{N} \mathbb{I}_{\{S(\boldsymbol{X}_i)\leq\gamma\}} \frac{f(\boldsymbol{X}_i,\boldsymbol{u})}{g(\boldsymbol{X}_i)},$$

it is know that the pdf g^* for which the variance of $\hat{\ell}$ is minimal is

$$g^*(\boldsymbol{x}) = \frac{f(\boldsymbol{x},\boldsymbol{u})\mathbb{I}_{\{S(\boldsymbol{x})\leq\gamma\}}}{\ell},$$

where however the value ℓ is unknown. The idea behind the CE methods is to choose g between a class of parametric densities $\{f(\cdot,\boldsymbol{v}),\, \boldsymbol{v}\in\mathcal{V}\}$ such that the Kullback Leibler divergence between the optimal importance sampling pdf g^* and g is minimal. The Kullback Leibler divergence, which is also termed the cross-entropy, between g and g^* is given by

$$\mathcal{D}(g^*,g) = \int g^*(\boldsymbol{x}) \ln\frac{g^*(\boldsymbol{x})}{g(\boldsymbol{x})}\, dx = \mathbb{E}\left[\ln\frac{g^*(\boldsymbol{X})}{g(\boldsymbol{X})}\right], \quad \boldsymbol{X}\sim g^*.$$

The procedure reduce to find an optimal parameter vector $\boldsymbol{v}^*$ that minimize the cross entropy

$$\begin{aligned}
\boldsymbol{v}^* &= \arg\max_{\boldsymbol{v}} \mathcal{D}(g^*, f(\cdot,\boldsymbol{v})) \\
&= \arg\max_{\boldsymbol{v}} \int g^*(\boldsymbol{x}) \ln\frac{g^*(\boldsymbol{x})}{f(\boldsymbol{x},\boldsymbol{v})}\, dx \\
&= \arg\max_{\boldsymbol{v}} \int g^*(\boldsymbol{x}) \ln g^*(\boldsymbol{x})\, dx - \int g^*(\boldsymbol{x}) \ln(\boldsymbol{x},\boldsymbol{v})\, dx \\
&= \arg\min_{\boldsymbol{v}} \int g^*(\boldsymbol{x}) \ln f(\boldsymbol{x},\boldsymbol{v})\, dx \\
&= \arg\min_{\boldsymbol{v}} \int \frac{f(\boldsymbol{x},\boldsymbol{u})\mathbb{I}_{\{S(\boldsymbol{x})\leq\gamma\}}}{\ell} \ln f(\boldsymbol{x},\boldsymbol{v})\, dx \\
&= \arg\min_{\boldsymbol{v}} \int \ln f(\boldsymbol{x},\boldsymbol{v})\mathbb{I}_{\{S(\boldsymbol{x})\leq\gamma\}} f(\boldsymbol{x},\boldsymbol{u})\, dx \\
&= \arg\min_{\boldsymbol{v}} \mathbb{E}_u\left[\ln f(\boldsymbol{X},\boldsymbol{v})\mathbb{I}_{\{S(\boldsymbol{X})\leq\gamma\}}\right]
\end{aligned}$$

Using again importance sampling, with a change of measure $f(\cdot, \boldsymbol{w})$ we can write

$$\ln f(\boldsymbol{x}, \boldsymbol{v}) \mathbb{I}_{\{S(\boldsymbol{x}) \leq \gamma\}} f(\boldsymbol{x}, \boldsymbol{u}) = \mathbb{I}_{\{S(\boldsymbol{x}) \leq \gamma\}} \ln f(\boldsymbol{x}, \boldsymbol{v}) \frac{f(\boldsymbol{x}, \boldsymbol{u})}{f(\boldsymbol{x}, \boldsymbol{w})} f(\boldsymbol{x}, \boldsymbol{w}),$$

and so

$$\boldsymbol{v}^* = \arg\min_{\boldsymbol{v}} \mathbb{E}_w \left[\mathbb{I}_{\{S(\boldsymbol{X}) \leq \gamma\}} \ln f(\boldsymbol{X}, \boldsymbol{v}) \frac{f(\boldsymbol{X}, \boldsymbol{u})}{f(\boldsymbol{X}, \boldsymbol{w})} \right].$$

This v^* can be estimated via

$$\hat{\boldsymbol{v}} = \arg\min_{\boldsymbol{v}} \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}_{\{S(\boldsymbol{X}_i) \leq \gamma\}} \frac{f(\boldsymbol{X}_k, \boldsymbol{u})}{f(\boldsymbol{X}_k, \boldsymbol{w})} \ln f(\boldsymbol{X}_k, \boldsymbol{v}), \tag{2.21}$$

where $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N \overset{\text{iid}}{\sim} f(\cdot, \boldsymbol{w})$. A complication in Eq. (2.21) is that for a rare event probability ℓ, most of indicators $\mathbb{I}_{\{S(\boldsymbol{X}_i) \leq \gamma\}}$ are zero and it is not to have a good estimation. In that case is possible to build a sequence of vector $\{\hat{\boldsymbol{v}}_t\}$ and levels $\{\hat{\gamma}_t\}$ with the aim to converge to $\boldsymbol{v}^*$ and γ. At each iteration t we simulate N independent random variables $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_N$ from the current importance sampling density $f(\cdot, \hat{\boldsymbol{v}}_{t-1})$ and let $\hat{\gamma}_t$ be the $(1-\rho)$ quantile of the performances values $S(\boldsymbol{X}_1), \ldots, S(\boldsymbol{X}_N)$ where ρ is called rarity parameter. We then update the value of $\hat{\boldsymbol{v}}_{t-1}$ using the cross entropy minimization based on the $N^e = \lceil N\rho \rceil$ random variables for which $S(\boldsymbol{X}_i) \geq \hat{\gamma}_t$.

Cross Entropy procedure for rare event estimation

1: Choose $\boldsymbol{v_0}$, $N \in \mathbb{N}$, $\rho \in \mathbb{R}$, $N^e = \lceil N\rho \rceil$, max_iter, $k = 1$,
2: **repeat**
3: generate $\boldsymbol{X_1}, \ldots, \boldsymbol{X_N} \overset{iid}{\sim} f(\cdot, \boldsymbol{v_{k-1}})$,
4: compute $S(\boldsymbol{X_i})$ for all i,
5: order them from the smallest to largest: $S_{(1)}, \ldots, S_{(N)}$,
6: let γ_t be the sample $(1-\rho)$ - quantile, i.e., $\gamma_t = S(X_{N-N^e})$,
7: $\hat{\boldsymbol{v}} = \arg\min_{\boldsymbol{v}} \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}_{\{S(\boldsymbol{X}_i) \leq \gamma\}} \frac{f(\boldsymbol{X}_k, \boldsymbol{u})}{f(\boldsymbol{X}_k, \boldsymbol{w})} \ln f(\boldsymbol{X}_k, \boldsymbol{v})$,
8: **until** $\hat{\gamma}_t \leq \gamma$ or $k =$ max_iter
9: Let K the final counter. Generate $\boldsymbol{X_1}, \ldots, \boldsymbol{X_N} \overset{iid}{\sim} f(\cdot, \boldsymbol{v_K})$,
10: estimate ℓ with $\hat{\ell} = \frac{1}{n} \sum_{i=1}^{N} \mathbb{I}_{\{S(\boldsymbol{X}_i) \leq \gamma\}} = \frac{f(\boldsymbol{X}_i, \hat{\boldsymbol{v}}_K)}{g(\boldsymbol{X}_i)}$.

Why it could be useful for optimization purposes? Assume that our problem has the form

$$\begin{aligned} \min \quad & S(\boldsymbol{x}), \\ \text{s.t.} \quad & \boldsymbol{x} \in \mathcal{X}, \end{aligned}$$

and admit only one minimizer $\boldsymbol{x^*}$. As shown in Botev et al. [2013], if we denote $\gamma^* = S(\boldsymbol{x^*})$ we can associate with the above optimization problem the estimation of the probability $l = \mathbb{P}\left(S(\boldsymbol{X}) \leq \gamma\right)$ where $\boldsymbol{X}$ has some probability density $f(\boldsymbol{x}, \boldsymbol{u})$ with $\boldsymbol{u}$ a real vector and γ close to the unknown value γ^*. Typically, l is a rare-event probability, and in this sense is possible to use the CE approach to find sampling distribution that concentrates all its mass in a neighborhood of the point $\boldsymbol{x^*}$. In this way we would achieve optimal or near optimal solutions. Note that, in contrast of what happen in the rare event simulating setting, here the final level γ^* is not known in advance but ideally the method produce a sequence of $\hat{\gamma}_t$ that converge to the optimum and a sequence of $\boldsymbol{v_t}$ that converges to $\boldsymbol{v}$ and such that $f(\boldsymbol{x}, \boldsymbol{u})$ is a probability distribution that concentrates all its mass in a neighborhood of the point $\boldsymbol{x^*}$.

To run the algorithm one needs to propose a class of parametric sampling densities $\{f(\cdot, \boldsymbol{u}),\, \boldsymbol{u} \in \mathcal{U}\}$, the initial vector $\boldsymbol{u_0}$, the sample size N and the rarity parameter ρ. Obviously the most challenging choice is the selection of an appropriate class of parametric sampling densities. It has to be flexible enough to estimate rare-events but it has to be simple enough to allow fast random variable generation and closed-form solutions to the maximum likelihood estimation program. For many details on the CE method in an optimization setting the reader can consult Rubinstein and Kroese [2016].

We now try to understand how we can exploit this idea in our case. The key step is to observe that is easy to find the optimal ULSP solution once set up variables are fixed. So our goal is using cross entropy to choose the binary variables. If we put ourselves in the situation where there is only one product, a solution can be described by a binary vector. But a vector of binary values can be seen as a sequence of realizations from a Bernoulli variable. So the parametric sampling densities that we can use for our problem are

$$\phi(\boldsymbol{\delta}, \boldsymbol{u}) = \prod_t \left[(u_t)^{\delta_t} + (1 - u_t)^{1-\delta_t} \right],$$

where $\delta_t = 1$ if production takes place in period t and $u_t \in [0, 1]\, \forall t$. In this case our objective function can be viewed as

$$z^* = \min_{\boldsymbol{\delta} \in \mathcal{X}} f(\boldsymbol{\delta}). \tag{2.22}$$

Using CE method we can associate to the optimization problem (2.22) a stochastic estimation problem

$$\mathbb{P}_{\boldsymbol{u}}(f(\boldsymbol{\Delta}) \leq z) = \sum_{\boldsymbol{\delta} \in \mathcal{X}} \mathbb{I}_{\{f(\boldsymbol{\delta}) \leq z\}} \phi(\boldsymbol{\delta}, \boldsymbol{u}),$$

where $\mathbb{P}_{\boldsymbol{u}}$ is the probability measure that a random state $\boldsymbol{\delta}$ drawn under $\phi(\boldsymbol{\delta}, \boldsymbol{u})$ has a performance function value less than or equal to a given threshold value z. As in the general case, given $\boldsymbol{\Delta_1}, \ldots, \boldsymbol{\Delta_n}$, randomly sampled from density functions $\phi(\boldsymbol{\delta}, \boldsymbol{p})$, our goal is to determine $\boldsymbol{p}$, that can be estimated by solving the problem

$$\hat{\boldsymbol{p}} = \arg\max_{\boldsymbol{p}} \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}_{\{f(\boldsymbol{\Delta}_i) \leq z\}} \ln \phi(\boldsymbol{\Delta}_i, \boldsymbol{p}).$$

Since $\phi(\boldsymbol{\delta}, \boldsymbol{p})$ is the density of a Bernoulli finding $\hat{\boldsymbol{p}}$ corresponds to solving

$$\frac{\partial}{\partial p_j} \sum_{i=1}^{N} \mathbb{I}_{\{f(\boldsymbol{\Delta}_i) \leq z\}} \ln \phi(\boldsymbol{\Delta}_i, \boldsymbol{p}),$$

which gives the optimal updating rule:

$$\hat{p}_j = \frac{\sum_{i=1}^{N} \mathbb{I}_{\{f(\boldsymbol{\Delta}_i) \leq z\}} \delta_{ij}}{\sum_{k=1}^{N} \mathbb{I}_{\{f(\boldsymbol{\Delta}_i) \leq z\}}} \quad j = 1, \ldots, T, \tag{2.23}$$

where $\boldsymbol{\Delta}_i$ indicates a binary vector and δ_{ij} is the j-th component of the i-th point of the CE population. In the absence of information a starting vector $\boldsymbol{p^{(0)}}$ for which each value is equal to $1/2$ can be used, and rule of Eq. (2.23) can be iteratively applied with the aim of generating a sequence of increasing threshold values $z_0, z_1, \ldots$, converging either to the global optimum z^* or to a value close to it. At each iteration, the new value of z is used to generate a better vector $\boldsymbol{p}$. The new parameter is used to draw a better population from $\phi(\boldsymbol{\delta}, \boldsymbol{p})$ which will lead to a better value. The process stops when either we have no improvement in the value of z or the vector $\boldsymbol{p}$ converges to a vector in $\mathcal{X}$, which implies that any random state drawn under $\phi(\cdot, \boldsymbol{p})$ will converge to the same solution in $\mathcal{X}$.

We do not underline how to use the binary variable to find the value z of the objective function, but similarly to what we have seen in Wagner Whitin algorithm, once we have the periods in which to order, is easy to find the production and the inventory for any time using Corollary 2.4.1.

Let's see a pseudocode for the algorithm:

Cross Entropy algorithm

1: Choose $\boldsymbol{p^{(0)}}$, $N \in \mathbb{N}$, $\rho \in \mathbb{R}$, max_iter,
2: set k=1
3: **repeat**
4: draw a sample population $\Omega^{(k)} = \{\boldsymbol{\Delta}_1, \ldots, \boldsymbol{\Delta}_N\} \sim b(\boldsymbol{p^{(k)}})$,
5: compute $f(\boldsymbol{\Delta}_i)$ for each $\boldsymbol{\Delta}_i \in \Omega_k$,
6: sort $\Omega^{(k)}$ in ascending order with respect to $f(\boldsymbol{\Delta}_i)$,
7: $p_j^{(k)} = \dfrac{\sum_{i=1}^{\lceil \rho N \rceil} \delta_{ij}}{\lceil \rho N \rceil} \quad j = 1, \ldots, T,$
8: **until** $\boldsymbol{p}^{(k)} \notin \{0,1\}^T \wedge k < \text{max\_iter}$

2.4 Capacitated Multi Item Problem

What we have said so far is about a single item uncapacitated lot sizing problem. In this section we explain how, thanks to the Lagrangian relaxation, it is possible to switch from a problem constrained on several items, to a succession of problems not constrained on the single item. If we consider our original problem formulation and we build the Lagrangian function as shown in Eq. (2.11) relaxing the capacity constraint, the problem (2.12) can be rewritten as

$$
\begin{aligned}
w(\boldsymbol{\mu}) = \min \quad & \sum_{t=1}^{T} \left(\sum_{i=1}^{I} \left(h_i I_{it} + (f_i + \mu_t r_i') \delta_{it} + \mu_t r_i x_{it} \right) - \mu_t R_t \right), \qquad (2.24) \\
\text{s.t.} \quad & I_{it} = I_{i,t-1} + x_{it} - d_{it}\,, && i = 1, \ldots, I, && t = 1, \ldots, T, \\
& x_{it} \leq M_{it} \delta_{it}\,, && i = 1, \ldots, I, && t = 1, \ldots, T, \\
& I_{i0} = 0\,, && i = 1, \ldots, I, && \\
& x_{it}, I_{it} \geq 0\,, && i = 1, \ldots, I, && t = 1, \ldots, T, \\
& \delta_{it} \in \{0,1\}\,, && i = 1, \ldots, I, && t = 1, \ldots, T.
\end{aligned}
$$

The problem is now uncapacitated, but the objective function (2.24) is multi item. It is enough to rewrite in the form

$$
\sum_{i=1}^{I} \sum_{t=1}^{T} \left(h_i I_{it} + (f_i + \mu_t r_i') \delta_{it} + \mu_t r_i x_{it} \right) - \sum_{t=1}^{T} \mu_t R_t \qquad (2.25)
$$

to notice that the term $\sum_{t=1}^{T} \mu_t R_t$ does not give any contribution on the choice of the minimum (remember we are minimizing in $\boldsymbol{I}$, $\boldsymbol{x}$, $\boldsymbol{\delta}$, while Lagrangian multipliers are fixed). Therefore the problem can be decoupled for

items, with respect to both the objective function and the constraints, and is possible to solve separately I problems. For each of them we find the best production plan and then, to compute the objective function all we need is sum the different pieces and subtract the term $\sum_{t=1}^{T} \mu_t R_t$.

So once $\boldsymbol{\mu}$ is fixed, for each $i = 1, \ldots, I$ we solve

$$\begin{aligned} w_i(\boldsymbol{\mu}) = \min \quad & \sum_{t=1}^{T} \left(h_i I_{it} + (f_i + \mu_t r_i') \delta_{it} + \mu_t r_i x_{it} \right) \\ \text{s.t.} \quad & I_{it} = I_{i,t-1} + x_{it} - d_{it}\,, && t = 1, \ldots, T, \\ & x_{it} \leq M_{it} \delta_{it}\,, && t = 1, \ldots, T, \\ & I_{i0} = 0 \\ & x_{it}, I_{it} \geq 0\,, && t = 1, \ldots, T, \\ & \delta_{it} \in \{0, 1\}\,, && t = 1, \ldots, T, \end{aligned}$$

and then we compute the objective function of the problem as

$$w(\boldsymbol{\mu}) = \sum_{i=1}^{I} w_i - \sum_{t=1}^{T} \mu_t R_t. \tag{2.26}$$

Once we call $f_t = f + \mu_t r'$ and $r_t = \mu_t r$, we can write the single item model in the form

$$\begin{aligned} \min \quad & \sum_{t=1}^{T} (h I_t + f_t \delta_t + r_t x_t) \\ \text{s.t.} \quad & I_t = I_{t-1} + x_t - d_t, && t = 1, \ldots, T, \\ & x_t \leq M_t \delta_t, && t = 1, \ldots, T, \\ & I_0 = 0, \\ & x_t, I_t, \geq 0, && t = 1, \ldots, T, \\ & \delta_t \in \{0, 1\}, && t = 1, \ldots, T, \end{aligned}$$

which is very similar to the model (2.17), considered in the Wagner Whitin discussion, with the single addition of production cost. In this case, even if the unitary costs change from one time interval to another, the cite Theorem 2.4 holds. So we could use the recursive formula

$$F(t) = \min \left\{ \min_{1 \leq j < t} \left\{ f_j + d_j r_j + \sum_{k=j+1}^{t} \left[h(k-j) + r_j \right] d_k + F(j-1) \right\}, \right.$$
$$\left. f_t + d_t r_t + F(t-1) \right\},$$

where we add the production cost term. Obviously we can also solve the problem using the CE method and then, once we find the periods in which to produce, calculate the optimal quantities to be produced using the Corollary 2.4.1.

Once the optimal production plan for each product is calculated, we can simply combine the solutions and calculate the value of the dual function as shown in (2.26). Unfortunately, even after several iterations with Lagrange multipliers, we have no guarantee that the solution found is feasible for the original problem, as the capacity constraint may not be met. There is also no need to build large-scale problems to find an example in which the dual problem fails to achieve the optimal solution of the primal and in particular cannot even find a feasible solution.

Example 2.1. Consider an instance with 2 item e 4 time steps. Costs and demand are

```
"set_up_cost": [10, 9],
"inventory_cost": [2,3],
"processing_time": [2,3],
"set_up_time": [7,3],

"time_capacity": 31,
"demand": [[7,8,1,5], [2,2,6,4]].
```

Solving the problem exactly we found that an optimal solution is given by

```
production
 [[7.5 7.5 1.5 4.5]
  [2   2   6   4  ]]
inventory
 [[0   0.5 0   0.5 0.0 ]
  [0   0   0   0   0   ]]
cost
 78.0
```

It is easy to see that for item 1 the Theorem 2.4 does not hold, while we show that for the relaxed problem the Wagner Whitin property is satisfied. This means that there is no hope of finding the optimal solution using this procedure. In this particular case, the example is built in such a way that no feasible solution that respects the property of Wagner Whitin exists. In fact if we use a Lagrangian procedure to solve the problem we found the solution

```
production
 [[ 7   8   6   0]
  [ 4   0  10   0]]
inventory
 [[0  0  0  5  0]
```

```
   [0   2   0   4   0]]
cost
76.0
```

for which the capacity constraint is broken in time steps 1 and 3.

2.4.1 Feasibility Issues

It is therefore necessary to build a procedure to transform a good near feasible solution in a feasible one. This operation can be seen as a transition from a dual solution to a primal solution, with an obvious increase in the objective function (remember Theorem 2.1), and it is therefore important to find a methodology that makes it possible to make the solution feasible by modifying it as little as possible, so as not to increase its cost too much and not spend a lot of time. However, sometimes, without more radical intervention it may not be possible to restore the feasibility of the solution. It is therefore necessary to find a compromise between the ability to make the solution feasible and the time of execution of the process. In general, however, this re-feasibility method is only applied at the end of the entire problem, once the Lagrange multipliers have reached a certain degree of convergence or the maximum number of iterations has been reached. At that moment it is therefore necessary to build a feasible solution, even at the cost of using more computational effort. We build three different methods, to be applied in sequence. Each one is more expensive and more invasive than the previous one, but it allows with greater probability to find a feasible solution. The third method in particular guarantees that a feasible solution is found (if it exists), but at a much higher cost than the other two methods.

Linear Programming Heuristic

The simplest and faster approach we can perform is to take fixed the binary variables and try to optimize only in the continuous ones, using all the original constraints. In this case we have a linear programming problem that has a polynomial complexity instead of an exponential one. If Y_{it} is the value of

the fixed variable δ_{it} we have to solve the problem

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{I}\sum_{t=1}^{T} h_i I_{it} \\
\text{s.t.} \quad & I_{it} = I_{i,t-1} + x_{it} - d_{it}\,, && i = 1,\dots,I, && t = 1,\dots,T, \\
& \sum_{i=1}^{I}(r_i x_{it} + r_i' Y_{it}) \le R_t\,, && && t = 1,\dots,T, \\
& x_{it} \le M_{it} Y_{it}\,, && i = 1,\dots,I, && t = 1,\dots,T, \\
& I_{i0} = 0\,, && i = 1,\dots,I, && \\
& x_{it}, I_{it} \ge 0\,, && i = 1,\dots,I, && t = 1,\dots,T.
\end{aligned}
$$

This method would appear to be the perfect candidate, but the problem is that fixing binary variables does not guarantee that a feasible solution can be found, as can we notice if we analyze Example 2.1. We could try to perform a sort of fix and optimize heuristic (see Güner Gören and Tunali [2018]) but we will loose the sense of using a Lagrangian heuristic. Another idea could be to fix only a part of binary variables: the model will be more complex without the guarantee of finding a feasible solution. In any case we will need an heuristic to build a feasible solution, as for example in Dillenberger et al. [1994], losing again the sense of a Lagrangian optimization.

Production Swap Heuristic

Trigeiro et al. [1989] propose a method that tries to modify as little as possible the solution, but allow to change the binary variables. A similar approach was followed by Caserta and Quinonez [2009]: they define a greedy heuristic scheme that attempts to project the infeasible solution back to the feasible space. To understand what they do let us suppose that the capacity constraint of period t is not respected, which means the production plan suggested by the Wagner Whitin algorithm uses more capacity than available in period t. Their heuristic scheme attempts to restore feasibility by moving backward or forward a certain amount of production, until overload production of period t is eliminated. We tweak their approach slightly to suit our needs.

Let us first consider the case in which we want to move backward production of a certain item i from period t to period $t' < t$. To understand how much production $p_i^{tt'}$ for the item i we can move from time t to time t' we have to consider the minimum between the production we currently make in

time t and the production we are able to add in period t' without exceeding the capacity constraint, i.e.

$$p_i^{tt'} = \max\left\{\min\left\{\lfloor\frac{R_{t'} - \sum_{k=1}^{I}(r_k x_{kt'} + r'_k\delta_{kt'}) - r'_i(1-\delta_{it'})}{r_i}\rfloor,\ x_{it}\right\},\ 0\right\}$$

The first term can be explained as the current available capacity in period t' minus, if needed, the capacity spend to activate the production of item i, divided by the capacity spent for a unit of production of item i. If for some reason $r_i = 0$ the approach is different: in fact we need just the possibility to activate the production and it is not important how many unit we transfer. So we have

$$p_i^{tt'} = x_{it}\mathbb{I}_{(R_{t'} - \sum_{k=1}^{I}(r_k x_{kt'} + r'_k\delta_{kt'}) - r'_i(1-\delta_{it'})}$$

or better $p_i^{tt'} = x_{it}$ if it is possible to starting produce item i in period t', zero otherwise. Now we change the production plan in the following way:

- if $p_i^{tt'} = x_{it}$ then $x_{it} = 0$, $\delta_{it} = 0$, $\delta_{it'} = 1$ and $x_{it'}$ += x_{it}
- if $0 < p_i^{tt'} < x_{it}$ then x_{it} −= $p_i^{tt'}$, $\delta_{it'} = 1$ and $x_{it'}$ += $p_i^{tt'}$
- if $p_i^{tt'} = 0$ no changes are needed
- for any $k \in \{t', \ldots, t\}$: $\quad I_{i,k+1}$ += $p_i^{tt'}$

Let us now consider the case in which we want to move forward production of a certain item i from period t to period $t'' > t$. As before it is possible to move production in period t'' only if there is enough capacity to produce it. In this case, however, we must act carefully. If we move the production from time t to time t'' without paying attention we could find that in some period t^*, $t \leq t^* < t''$ it is no more possible to satisfy the demand. We have to notice that the quantity we move can not be larger both than the minimum inventory level between time period t and time period t'' and than the production x_{it}. So the amount of production that is possible to move from time t to time t'' is

$$p_i^{tt''} = \max\left\{\min\left\{\lfloor\frac{R_{t''} - \sum_{k=1}^{I}(r_k x_{kt''} + r'_k\delta_{kt''}) - r'_i(1-\delta_{it''})}{r_i}\rfloor,\right.\right.$$
$$\left.\left.x_{it},\quad \min_{t\leq k<t''} I_{ik}\right\},\quad 0\right\}$$

when $r_i > 0$. If the time used for producing item i is zero then

$$p_i^{tt''} = \min\left\{x_{it}, \quad \min_{t \le k < t''} I_{ik}\right\} \mathbb{I}_{(R_{t''} - \sum_{k=1}^{I}(r_k x_{kt''} + r'_k \delta_{kt''}) - r'_i(1-\delta_{it''})}$$

Now we have to update the variables like before:

- if $p_i^{tt''} = x_{it}$ then $x_{it} = 0$, $\delta_{it} = 0$, $\delta_{it''} = 1$ and $x_{it''}$ += x_{it}
- if $0 < p_i^{tt''} < x_{it}$ then x_{it} −= $p_i^{tt''}$, $\delta_{it''} = 1$ and $x_{it''}$ += $p_i^{tt''}$
- if $p_i^{tt''} = 0$ no changes are needed
- for any $k \in \{t', \dots, t\}$: $\quad I_{i,k+1}$ −= $p_i^{tt'}$

To try to make the solution feasible, firstly we look at the periods for which capacity constraint is not satisfied. All products for which production take place are considered and for each of them a forward or backward swap is sought. If no swap is possible it means that we are not able to restore feasibility in that period, while if at least a swap is perform we check the feasibility. If the capacity constraint is not satisfied we try to make more swap, whereas if it is satisfied we move in another time period for which excess capacity is used, until either capacity is restored in each period or no more swap are possible.

Production swap heuristic

```
Let 𝒯 the set of periods for which exist feasibility issues,
while |𝒯| ≠ 0 do
    let t the first element of 𝒯
    for i ∈ [1, ..., I] do
        if δ_it = 1 then
            for t' ∈ [t − 1, ..., 1] do
                try to perform a backward swap
                if feasibility in time t is restored then
                    t is removed from 𝒯, we check the next element in 𝒯
                end if
            end for
            for t'' ∈ [t + 1, ..., T] do
                try to perform a forward swap
                if feasibility in time t is restored then
                    t is removed from 𝒯, we check the next element in 𝒯
                end if
```

```
            end for
        end if
    end for
    if no swap are performed then
        feasibility can not be restored: STOP
    end if
end while
solution is feasible
```

This method is more powerful than the other but it takes more time to run. In any case also this approach do not ensure us to find a feasible solution. In fact, it is possible that to find a feasible solution we have to make joint changes to the solution, which are not provided for by this algorithm.

MILP Approach

We therefore need to find at least one method that guarantees us to obtain a feasible solution and that is less complicated than finding the optimal solution from scratch. Unfortunately, as we have mentioned, it is possible to show that even just finding a feasible solution to the CLSP is an NP-hard problem. Obviously a method that guarantee us to build a feasible solution (if the problem admits it) is to impose all the constraints of the original problem. If we want a new solution that is similar to the one we have, we can penalize the deviations from the original solution. If the starting solution is near feasible, find a feasible one could be easier to solve the original problem from scratch. The model to solve is

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{I}\sum_{t=1}^{T} |Y_{it} - \delta_{it}| + |X_{it} - x_{it}| \\
\text{s.t.} \quad & I_{it} = I_{i,t-1} + x_{it} - d_{it}\,, && i = 1,\dots,I, && t = 1,\dots,T, \\
& \sum_{i=1}^{I} (r_i x_{it} + r'_i \delta_{it}) \le R_t\,, && && t = 1,\dots,T, \\
& x_{it} \le M_{it}\delta_{it}\,, && i = 1,\dots,I, && t = 1,\dots,T, \\
& I_{i0} = 0\,, && i = 1,\dots,I, && \\
& x_{it}, I_{it} \ge 0\,, && i = 1,\dots,I, && t = 1,\dots,T, \\
& \delta_{it} \in \{0,1\}\,, && i = 1,\dots,I, && t = 1,\dots,T,
\end{aligned}
$$

where Y_{it} is the value of the setup variable and X_{it} is the value of the production variable for item i in period t for the starting solution.

2.4.2 Solution Procedure

After finishing defining the fundamental blocks of our algorithm, let's now see how to proceed to solve the problem. Although in the previous section we took a peek at how to solve the CLSP, there are still some important details to discuss, depending on the success of the algorithm. Basically the procedure for solving the problem is as follows. As a first step we choose the initial multipliers. At this point using these multipliers the problem is decomposed on the single product and solved using one of the two methods proposed. The solutions found are combined to have a candidate CLSP solution. The multipliers are updated via the subgradient method or the bundle method and finally the stop conditions are checked. If they occur, the algorithm ends, otherwise we solve the problem again with the new multipliers. If after a number of iterations no feasible solutions have been found yet, the three heuristics to make a solution feasible are applied in sequence.

For the success of the algorithm it is important to choose the right number of maximum iterations, understand how to initialize multipliers and what step size use to update them, understand what stopping conditions to use and in general find the precautions to improve performances. Most of the choices described later arise from numerous tests on different procedures. We try to make the choice of parameters as general as possible, so that it does not depend on the size of the problem or how tight the capacity constraint is.

Two different procedures are proposed which have different strengths and weaknesses. Of fundamental influence on the performance of both of them is the step size α_k to be used in the multipliers update in Eq. (2.13) or in Eq. (2.16). To choose the value of α_k we decide to use the approach following by Süral et al. [2009], i.e., use the formula (2.14) that we rewrite here for clarity specifying the norm used by us

$$\alpha_k = \alpha \, \frac{f(\bar{\boldsymbol{x}}) - w(\boldsymbol{\mu}^{(k)})}{||\boldsymbol{g}||_1}.$$

The idea, already explained above, is that when the numerator $f(\bar{\boldsymbol{x}}) - w(\boldsymbol{\mu}^{(k)})$ is small it means that we are close to convergence and therefore it is right to take small values so as not to move too quickly. The denominator $||\boldsymbol{g}||_1$ instead takes into account that if the gradient is very large, we are at a point where the function is very steep and it is better to move slowly. Let us first assume that the solution $\bar{\boldsymbol{x}}$ is feasible and when this formula can give us problems. The two special cases to consider are when the numerator or the

denominator are zero. Recall that a subgradient of the dual function

$$w(\boldsymbol{\mu}^{(k)}) = \sum_{i=1}^{I}\sum_{t=1}^{T}(f_i\delta_{it}^{(k)} + h_i I_{it}^{(k)}) + \sum_{t=1}^{T}\left(\sum_{i=1}^{I}(r_i x_{it}^{(k)} + r_i'\delta_{it}^{(k)}) - R_t\right)\mu_t^{(k)}$$

is

$$\mathbf{g}^{(k)} = \left(\sum_{i=1}^{I}(r_i x_{i1}^{(k)} + r_i'\delta_{i1}^{(k)}) - R_1, \ldots, \sum_{i=1}^{I}(r_i x_{iT}^{(k)} + r_i'\delta_{iT}^{(k)}) - R_T\right).$$

If the subgradient norm is zero, it would mean that any component $g^{(k)}$ is zero, i.e., there is a feasible solution such that every constraint is satisfied at the limit. If we try to change in any way the production plan, we found an infeasible solution at least in one time period. This means that $\overline{x}$ is the only feasible solution of the problem. If instead the numerator was null, it would mean that we found the pair $\boldsymbol{x}^*$ and $\boldsymbol{\mu}^*$ of optimal solution of primal and dual. Again there is nothing to do: $\boldsymbol{x}^*$ is the optimum of the problem. In general, however, the current solution $\bar{\boldsymbol{x}}$ is not feasible. In this case it is not possible that the subgradient norm is zero, because it would mean that all constraints are respected. For the numerator, obviously for construction $w(\boldsymbol{\mu}^{(k)}) \leq f(\bar{\boldsymbol{x}})$ still applies, which means that anyway we will never look for solutions in the wrong area, but it may be possible that $w(\boldsymbol{\mu}^{(k)}) = f(\bar{\boldsymbol{x}})$. In particular this is the case when multipliers are all zeros. And that's why it's important, in our algorithm, that we don't start with multipliers equal to zero, because we can't move away from the first solution found.

In all this, the only parameter to calibrate is α. It would be optimal to find a value that fits all the instances of the problem, but it is not possible. In fact, when the number of products and cost parameters increase, also the difference between $f(\bar{\boldsymbol{x}})$ and $w(\boldsymbol{\mu}^{(k)})$ increases and there is a need to use a smaller value of α. Several tests have led us to understand that the value of α should be inversely proportional to the number of products and the average demand. If we let D be

$$D = \frac{1}{IT}\sum_{i=1}^{I}\sum_{t=1}^{T} d_{it},$$

then we found that a good choice for α is

$$\alpha^* = \frac{1}{2DI}.$$

Another choice to take that is common to both algorithms is that of multipliers. As said before, the choice, which is sometimes made, and which we will also propose in the stochastic case, to start with a vector of multipliers null, is not effective in our case. After several tests, we realized that to achieve convergence faster it was good to start with small multipliers: in fact we notice that in the optimal solutions the multipliers always were not too large. After some attempts we decided to opt for random multipliers evenly distributed between 0 and 2. In the procedures shown below, some of the results are shown in order to motivate the choices made for some of the parameters in play. A more detailed analysis of the results is done in the next section. The tables present in this section were built using the WW algorithm and updating multipliers with the subgradient method. We will then see the comparison with the other methods of resolution and update of multipliers.

Classical Procedure

The first approach that we tried to use is what we call Classic Procedure. Lagrange multipliers are updated and the relaxed problem is solved until one of the stop conditions is met.

Classical Procedure

Choose K, α, b_o, b_μ
Initialize $\boldsymbol{\mu}^{(1)} : \mu_1^{(1)}, \ldots, \mu_T^{(1)} \overset{iid}{\sim} U(0,2)$
Initialize s_f, the best feasible solution to $None$
Initialize $\ell^{(0)}$, the lower bound for optimal cost at iteration 0 to $+\infty$
Let $c(s)$ a function that return the cost of the solution s
Let $d(s)$ a function that return the dual value of the solution s
Let $k = 1$
while (s_f is $None$ or $\frac{c(s_f)-\ell^{(k-1)}}{\ell^{(k-1)}} > b_o$) and $\frac{||\boldsymbol{\mu}^{(k-1)}-\boldsymbol{\mu}^{(k)}||_\infty}{||\boldsymbol{\mu}^{(k-1)}||_\infty} > b_\mu$ and $k < K$ **do**
    Let $s^{(k)}$ the solution found using $\boldsymbol{\mu}^{(k)}$
    $\ell^{(k)} = \max\{\ell^{(k-1)}, d(s^{(k)})\}$
    if $s^{(k)}$ is feasible and $c(s^{(k)}) < c(s_f)$ **then**
        $s_f = s^{(k)}$
    end if
    find the gradient $\boldsymbol{g}^{(k)}$ using the solution $s^{(k)}$
    $\boldsymbol{\mu}^{(k+1)} = \boldsymbol{\mu}^{(k)} + \alpha \dfrac{c(s^{(k)}) - d(s^{(k)})}{||\boldsymbol{g}^{(k)}||_1} \boldsymbol{g}^{(k)}$

16: **end while**
17: **if** s_f is $None$ **then**
18: Try to use the heuristic to restore the feasibility
19: **end if**

As is possible to see from the pseudocode, our algorithm, in addition to the aforementioned $\boldsymbol{\mu}$ and α, also needs parameters K, b_0 and $b_{\boldsymbol{\mu}}$. K is the maximum number of iterations, while b_0 and $b_{\boldsymbol{\mu}}$ are two thresholds used to decide when to end the procedure. If the maximum number of iterations K has not been reached, the algorithm continues to modify the multipliers and search for a solution until a feasible one is found. From the moment that a feasible solution is found, the algorithm starts to check the other stop conditions and the procedure ends when the relative difference between the upper bound and the lower bound is less than b_0. If the maximum relative difference between two multipliers is less than $b_{\boldsymbol{\mu}}$, the algorithm stops, also if a feasible condition was not found at the moment. Obviously the thresholds must be chosen according to how good the approximation of the optimal solution is wanted. The lower the thresholds, the more iterations are needed to complete the procedure. One by one we see how different choices of parameters change the execution times and the quality of the solutions found and we try to find the set of values that allows to solve the problem for instances very different from each other.

As said before thanks to several tests we found that a good value of α is $\alpha^* = 1/(2DI)$. Higher values lead to moving too fast in multiplier space. As a result, too large multipliers may be chosen and the dual function may diverge (negatively), Smaller values, on the other hand, cause the optimization of dual function to progress too slowly. To show how the algorithms behave at α variation we extracted a pool of instances quite varied and ran the algorithm for α^*, $\frac{1}{5}\alpha^*$ and $5\alpha^*$. The results in Table 2.4 confirm what was said: for α too small the algorithm is slower, while for α too large it diverges, without finding satisfactory solutions.

	Approach 1	
	Time(s)	Gap (%)
$\frac{1}{5}\alpha^*$	33	$1,43$
α^*	23	0.77
$5\alpha^*$	99	$60,0$

Table 2.4: Algorithm performances when changing α. Average results on 50 instances.

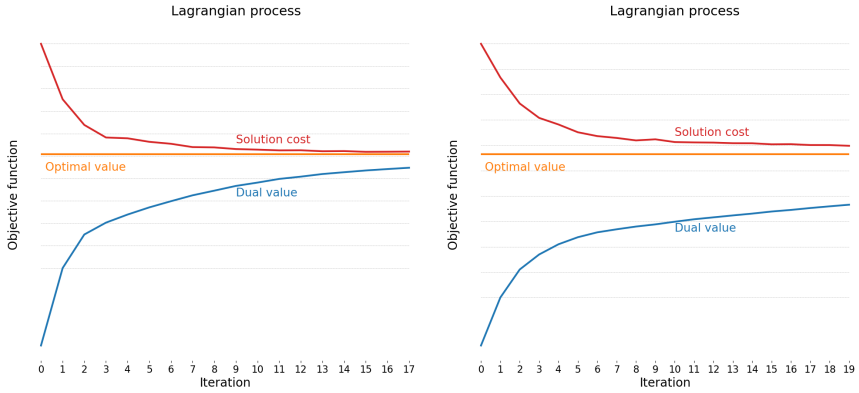

Figure 2.1: Cost and dual objective function curves when $\alpha = \alpha^*$ (left) and $\alpha = \frac{1}{5}\alpha^*$ (right).

In Figure 2.1 we show the curves of the dual objective function and the cost of the solution through the iterations. On the left is shown the curve for $\alpha = \alpha^*$, while on the right is shown for $\alpha = \frac{1}{5}\alpha^*$. It is easily to see by eye that the function on the right grows more slowly.

Normally, thanks to the smart update of the Lagrangian multipliers, it is possible to find a feasible solution without having to use heuristics. It may happen, however, that even if you are very close to a feasible solution, you still need numerous iterations to find one. In this case it may be convenient to truncate the algorithm and try to build a solution using heuristics. This is one of the reasons why it is useful to insert a maximum number of iterations, after which, if a feasible solution has not been found, we try to build one using an heuristic. Another reason to enter a maximum number of iterations is that, using a Lagrangian relaxation to solve the problem, we have no way of understand if the problem admits solution. We could then continue to look for a solution, when this does not exist. The main problem in truncating the process too early is that if the solution found is still too far from be feasible, heuristics become time consuming and may fail to find a feasible solution in a reasonable time. In particular the first two heuristics presented by us, the Linear Programming heuristic and the Production swap heuristic are still able to give a positive or negative outcome in a short time, while the MILP heuristic, which is always able to find a solution, may take a really long time to find one. For this reason when executing this method, a time limit is set. In the table is possible to see some result obtained by two bunch of 30 instances. The first group is composed by quite easy instances, while

	Easy instances			Difficult instances		
	Solved	Time(s)	Gap(%)	Solved	Time(s)	Gap(%)
$K = 20$	28	24	$1,05$	16	25	$3,68$
$K = 100$	30	46	0.07	30	137	0.07

Table 2.5: Algorithm performances when changing K. Time and error refer only to the solved instances.

the second is composed by difficult ones. We show the differences when we choose $K = 20$ or $K = 100$. As we can see, in the difficult instances, when 20 iterations are used, we fail in building a feasible solution 16 times out of 30 (300 seconds are used as time limit for the MILP heuristic), while when using 100 iteration we are always able to find a feasible solution. From the easy instances we can understand that, even if with 20 iterations we are almost always able to find a feasible solution, the gap from the optimal solution is bigger. Obviously, however, better performance comes at the cost of a longer calculation time. If in the case of the difficult instances we are more than happy to use more time to be able to solve them all, in the case of the easy instances, we could be satisfied with the error of 1%, but saving almost 50% of the time. In order not to spend too much time unnecessarily, but trying to limit the risk of failing to solve the problem, the algorithm can be modified slightly. In particular, when we realize that the dual objective function is close to convergence, we could try to make the solution feasible, but without ending the Lagrangian relaxation procedure. If we fail to re-establish eligibility by using one of the two fastest heuristics, we will continue to iterate by changing the multipliers. One way to tell when the dual function is converging is to see when it starts to increase more slowly. To understand when it's time to try to make the solution feasible, we decided to enter an additional threshold b_1 and to test at iteration k if

$$\frac{c^{(k)} - \ell^{(k)}}{\ell^{(k)}} < b_1,$$

where $c^{(k)}$ is the cost of the best infeasible solution at iteration k. Note that, unlike what was done to decide whether to terminate the algorithm, here $c^{(k)}$ is not an upper bound for the optimal value, because the solution is not feasible. In any case, this ratio is a good way to understand how close we are to convergence. To avoid that, after the first time, this attempt to make the solution feasible is made at each iteration, slowing down the execution of the algorithm, each time the threshold b_1 is lowered by a factor of 10. In Table 2.6 we show the result obtained on the same instances of Table 2.5, when

	Easy instances			Difficult instances		
	Solved	Time(s)	Gap(%)	Solved	Time(s)	Gap(%)
$K = 100$ modified	30	31	0.24	30	125	0.36
$K = 100$ classic	30	46	0.07	30	137	0.07

Table 2.6: Algorithm performances classic versus modified, when $K = 100$. Average results on 10 instances.

using the modified algorithm with a threshold $b_1 = 0.02$. The smaller the threshold, the more the procedure tends to the classic one. The goal of this modification is to save time without loose too much on performance side. As we can see in some cases this approach can be of help, as in the case of easy instances, where we can save about a third of the time, while in other cases, its contribution is insignificant, as in the case of difficult instances, where we save less than 10% of the time.

The last two parameters we need to discuss are b_0 and $b_{\boldsymbol{\mu}}$. If the values of these parameters are too low, you may never meet the stop condition and continue to iterate even if you have a very good feasible solution. If their values are too high, on the contrary, there is a risk of getting stuck too early and accepting a solution that, although feasible, has a much higher cost than the optimal one. Actually, the problem of choosing too high a threshold is not very felt, in fact very often the first feasible solution found by the algorithm is already within the $1 - 2\%$ gap from the optimal. By doing several tests we found that a pair of convenient values is $b_0^* = 0.02$ and $b_{\boldsymbol{\mu}}^* = 0.01$. In Table 2.7 there are three tests. In the first column we use values that are a third of those chosen above, in the second column we use the value b_0^* and $b_{\boldsymbol{\mu}}^*$, while in the third column we use values that are three times of those proposed before. As we can see, in passing from the third column to the second, there is a remarkable improvement in performance at the cost of, on average, only one more iteration, while if we try to decrease more the parameters values, we have to pay the improvement with a much higher number of iterations

Restoration Procedure

The second approach we tried to use does not rest on solid theoretical bases. It consists of a modification of the Classic Procedure, in which, after having solved the dual problem, we try to make the solution feasible.

The main difference in this method, whereby a pseudo code is shown below, is than, when changing the solution, we use a pseudo gradient instead

	high b_0 and $b_{\boldsymbol{\mu}}$		medium b_0 and $b_{\boldsymbol{\mu}}$		low b_0 and $b_{\boldsymbol{\mu}}$	
	Iterations	Gap(%)	Iterations	Err (%)	Iterations	Gap(%)
Inst 1	42	0.06	29	0.11	28	0.90
Inst 2	34	0.06	19	0.18	18	0.34
Inst 3	33	0.08	18	0.22	17	1.36
Inst 4	32	0.04	19	0.22	18	0.95
Inst 5	37	0.10	22	0.23	21	0.67
Inst 6	32	0.04	19	0.08	18	0.39
Inst 7	42	0.07	28	0.37	27	1.32
Inst 8	34	0.07	21	0.09	20	0.36
Inst 9	33	0.07	21	0.14	20	0.43
Inst 10	39	0.04	28	0.49	26	1.51
Average	35.8	0.06	22.4	0.21	21.3	0.82

Table 2.7: Algorithm performances when changing b_0 and $b_{\boldsymbol{\mu}}$.

Restoration Procedure

Choose K, α, b_o, b_μ
Initialize $\boldsymbol{\mu}^{(1)} : \mu_1^{(1)}, \ldots, \mu_T^{(1)} \overset{iid}{\sim} U(0,2)$
Initialize s_f, the best feasible solution to $None$
Initialize $\ell^{(0)}$, the lower bound for optimal cost at iteration 0 to $+\infty$
Let $c(s)$ a function that return the cost of the solution s
Let $d(s)$ a function that return the dual value of the solution s
Let $k = 1$
while (s_f is $None$ or $\frac{c(s_f)-\ell^{(k-1)}}{\ell^{(k-1)}} > b_o$) and $\frac{||\boldsymbol{\mu}^{(k-1)}-\boldsymbol{\mu}^{(k)}||_\infty}{||\boldsymbol{\mu}^{(k-1)}||_\infty} > b_\mu$ and $k < K$ **do**
    Let $s^{(k)}$ the solution found using $\boldsymbol{\mu}^{(k)}$
    $\ell^{(k)} = \max\{\ell^{(k-1)}, d(s^{(k)})\}$
    if $s^{(k)}$ is not feasible **then**
        let $\overline{s}^{(k)}$ be the modified solution using the swap heuristic
    else
        let $\overline{s}^{(k)} = s^{(k)}$
    end if
    if $\overline{s}^{(k)}$ is feasible and $c(\overline{s}^{(k)}) < c(s_f)$ **then**
        $s_f = s^{(k)}$
    end if
    find the pseudo gradient $\overline{\boldsymbol{g}}^{(k)}$ using the solution $\overline{s}^{(k)}$
    $\boldsymbol{\mu}^{(k+1)} = \boldsymbol{\mu}^{(k)} + \alpha \frac{c(s^{(k)}) - d(s^{(k)})}{||\overline{\boldsymbol{g}}^{(k)}||_1} \overline{\boldsymbol{g}}^{(k)}$
end while
if s_f is $None$ **then**

23: Try to use the heuristic to restore the feasibility
24: **end if**

of the gradient of the dual function. In fact, also if we are not able to restore completely the feasibility of the solution $\boldsymbol{x}$, we modify it. If we call $\overline{\boldsymbol{x}}$ the new solution, than we can define the pseudo gradient as

$$\overline{\boldsymbol{g}} = \left(\sum_{i=1}^{I} (r_i \overline{x}_{i1} + r_i' \overline{\delta}_{i1}) - R_1, \ldots, \sum_{i=1}^{I} (r_i \overline{x}_{iT} + r_i' \overline{\delta}_{iT}) - R_T \right),$$

and use it to update the multiplier. We talk about pseudo gradient because the solution $\overline{\boldsymbol{x}}$ is no more the optimal solution for the dual problem, and so Theorem 2.3 does not hold. But let us try to understand what the meaning of this reasoning may be. Our idea was as follows. Suppose that at the time t the constraint is not satisfied in solving the dual problem. Using the swap heuristic there is a new solution in which the same constraint may or may not be met. If the constraint is not satisfied, even the pseudo gradient, like the gradient, has the t-th component with positive sign and the corresponding multiplier will increase. If instead the constraint is satisfied for the new solution, then the pseudo gradient will have the t-th component with negative sign and the multiplier will decrease. This would seem a contradiction: to decrease the multiplier at the time t means to look for a solution that exploits more the constraint, despite already the solution of the previous iteration did not respect it. Actually the idea is that if the method was able to re-establish the feasibility with that multiplier, we would like to see if it can do that with a set of even smaller multiplier. In other words, we are trying to give importance to the multipliers corresponding to those constraints that cannot be restored. The advantages of this approach is that, at the beginning of the procedure it is easier to find feasible solutions, although not with a costs close to the optimal. However, of course, trying to re-establish the constraint at each iteration involves a larger computing time.

The parameters to use are always the same. For b_0 and b_μ in particular it was decided to use the same pair of values. Even the value of α has been kept constant. In any case, this approach is less related to the maximization of the dual, so it tends to work well in a larger range of values. Instead, compared to the Classical Procedure, this algorithm works better with a lower number of maximum iterations. The algorithm in fact tends to reach good feasible solutions from the first iterations, but then has some problems to close the gap and get closer to the optimal. It can therefore be considered a good

	Easy instances			Difficult instances		
$K = 20$	Solved	Time(s)	Gap(%)	Solved	Time(s)	Gap(%)
Classical Procedure	28	24	1, 05	16	25	3, 68
Restoration Procedure	30	30	0.93	30	36	3.14
$K = 100$	Solved	Time(s)	Gap(%)	Solved	Time(s)	Gap(%)
Classical Procedure	30	33	0.56	30	137	0.07
Restoration Procedure	30	46	0.07	30	94	1.67

Table 2.8: Performances comparison between the two procedures when K change. Time and error refer only to the solved instances.

algorithm when we are not interested in having a very precise solution, but we are interested in having a first approximation in a short time.

We can notice from Table 2.8 how in the case of the difficult instances, different from the classical procedure, this algorithm is able, even with only 20 iterations, to always find a solution, even if the error is higher.

2.5 Numerical Tests

After having presented in broad outline the two procedures we use to solve the problem and having highlighted the pros and cons, we finally arrive at the part of the results, where our methods are compared with the resolution of Gurobi Optimizer.

First we focus on how we generated instances. There are a lot of data that can take on a wide range of values, and it's important to understand how our algorithm performs with varying parameters. Parameters such as the number of products and the number of time intervals are obviously chosen a priori, while the costs and the demand are sampled. Most of our tests were done for the pair $I = 1000, T = 30$. In fact is important to have results that are valid when the size of the products becomes considerable. For time intervals, assuming that each time period corresponds to 1 week, we preferred not to exceed $25 - 30$ weeks, which corresponds to a time horizon of about 6 months. However we also show how these parameters influence the execution time and the quality of our algorithms and the procedure implemented by Gurobi Optimizer (version 9.1.0) on a personal computer with 16GB of RAM and a processor Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59GHz.

The cost and time parameters of each product are distributed according to a uniform distribution between a minimum and a maximum value. With regard to demand, each product is associated with a mean and a standard deviation, also sampled by two uniform distributions. The demand for in-

dividual time intervals is then sampled from a normal truncated with the average and standard deviation sampled before. Being in the deterministic, the form of the distribution of the demand is not fundamental, but in order to make the data more plausible, the truncated normal distribution has been chosen. For the available capacity, we have used a constant capacity, as it is assumed that a company always has the same number of available manpower hours. Obviously this value is fundamental to define the difficulty of the problem. If the available capacity is high, the constraint is never violated and it is simple to build a feasible solution. If the constraint is very tight, both the MILP resolution, and the one based on Lagrangian relaxation, need much more time to find a solution. In particular we define 3 levels of capacity: low, medium and high. We define the necessary capacity as

$$C = \frac{1}{T} \sum_{t=1}^{T} \sum_{i=1}^{I} (r_i d_{it} + r'_i)$$

and 3 capacity factor level $c_{\text{low}} = 1.01$, $c_{\text{medium}} = 1.03$ and $c_{\text{high}} = 1.05$. The constant capacity level is then calculate as the product of the necessary capacity and the capacity factor level. It may seem that the capacity so defined is actually excessive. In fact we are defining C so that on average we can bear the setup time and the time for the production of the demand of that period. The fact is that this is only true on average and in those periods of time when demand is high there will always be production capacity in default. In practice what happens is that the production plan is very complicated especially in the first periods, where, starting with the empty warehouse, there is no possibility of producing in excess for the following periods. Obviously our choice is not the only one possible nor probably the most realistic, but it manages effectively to build complex problems, which allow our algorithm to show its effectiveness.

Before comparing our algorithm with the solution of the problem given by commercial software, we need to understand which of our strategies is the best. We have two options both to solve the problem (WW and CE), and two options to update the Lagrange multipliers, the subgradient method and the bundle method. Let's start by analyzing how best to update Lagrange multipliers. As we mentioned earlier, the bundle method looks for better directions than the subgradient method, and a small linear optimization problem is solved to find the right step size. However, the main drawback of the method is that multipliers are often not updated for several iterations. This leads the algorithm to need many more iterations to converge, and this results in a generally slower resolution of the problem. If we leave the same

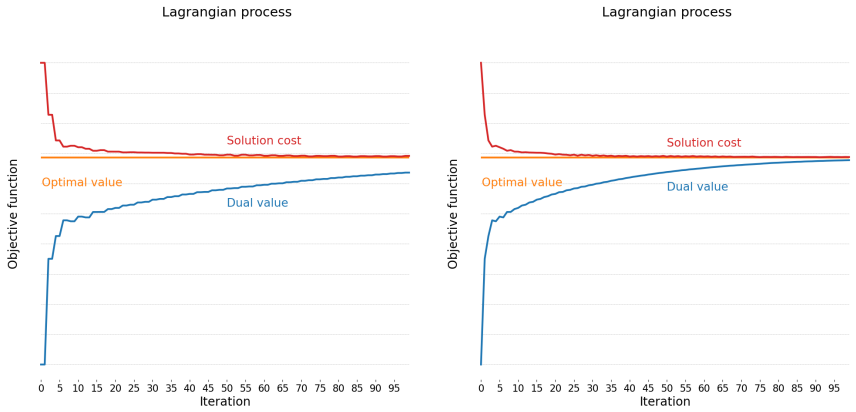

Figure 2.2: Cost and dual objective function curves. Bundle method on the left, subgradient method on the right.

	Bundle Method		Subgradient Method	
	Time(s)	Gap(%)	Time(s)	Gap(%)
$I = 50$	15 (4)	2.59 (2.74)	8 (3)	1.44 (1.99)
$I = 100$	25 (14)	1.98 (2.13)	17 (5)	1.38 (2.47)
$I = 200$	30 (11)	2.44 (1.79)	28 (9)	0.81 (1.50)
$I = 500$	81 (29)	1.59 (1.39)	71 (20)	0.17 (0.24)
$I = 750$	140 (24)	3.59 (2.12)	113 (18)	0.07 (0.02)
$I = 1000$	250 (55)	3.91 (1.19)	210 (66)	0.21 (0.25)

Table 2.9: Performances comparison between Bundle Method and Subgradient Method (Classical Procedure is used to solve the problem). Standard deviation in brackets.

number of iterations that are used with the subgradient, we find poor results, as these iterations are not enough. We found then that the best choice of the direction in which to move in the space of the multipliers fails to compensate the high number of iterations without displacements. We also note that if the bundle method is used to update multipliers, the stop conditions must be changed. In fact it is no longer possible to keep the stop condition on the relative deviation of the multipliers, as in numerous iterations these are not updated, and therefore the deviation would be zero. As you can see in Figure 2.2, the subgradient method is performing much better especially in the first iterations, managing to get close to the convergence and then slowly refining the solution. In Table 2.9 instead it is possible to see how for instances with different size the bundle method turns out to be slower and to have a larger gap from the optimal solution with respect to the subgradient method.

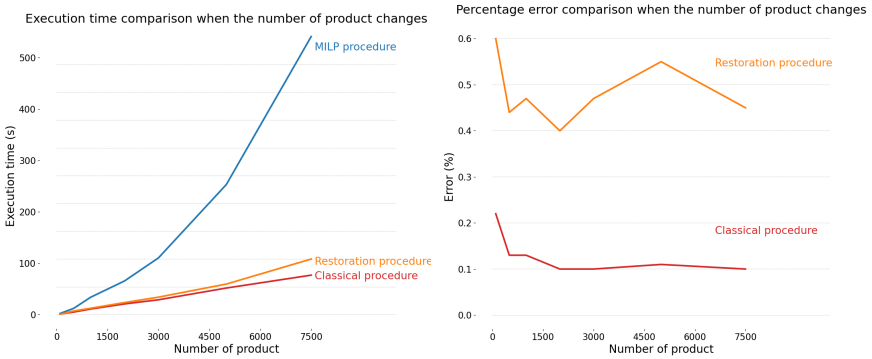

Figure 2.3: Comparison between the performances of MILP approach and our algorithms.

The second choice that needs to be taken is the one regarding the method to be used to solve the uncapacitated single item problem. Although the method of cross entropy seemed promising, unfortunately the parameters to estimate are too many and the problem requires too many iterations to achieve convergence. We tried to refine the parameters of the problem to obtain better results, but in no way the method seemed to be able to compete with the algorithm of WW. In the next chapter, however, we will see how this approach can be used when the parameters to be estimated are less, and in particular we use it to estimate the values s and S of the policy (s, S) in the stochastic case.

So the best solution to proceed to compare our algorithm with that of commercial software is to use Wagner Whitin to solve the unconstrained problem and the subgradient method to update multipliers. Firstly, we would like to show that both of our procedures scale better than the MILP approach when the number of products increases. To do this we used an example where costs and also average demand are low. As far as that is not credible, we used these values to make the resolution fast enough and we could test for very high number of products. As you can see in Figure 2.3, our algorithm turns out to have a linear complexity in the number of products, while the MILP resolution seems to depend on the number of products with a higher grade term. Another thing we can see is that, in the area of 1000 products, the MILP approach is quite similar, so in many of our subsequent tests, there seems to be no substantial difference between our methods and the exact one. The tests were done for this number of products because it seemed to us a reasonable compromise between choosing a high value and being able

Execution time (s)			Gap (%)	
Exact	Classical	Restoration	Classical	Restoration
200	122	55	0.20	3.89

Table 2.10: Average results on 50 instances.

to get results in a short time. It should be remembered, however, that with a view to trying to solve the problem on larger and larger instances, our approach becomes more and more favourable.

Let us now consider a more realistic problem, on which we will make our analyses and evaluate the results. We consider a number of products equal to 1000 and a number of periods equal to 30. Set up costs are uniformly distributed between 1750 and 2550, while inventory costs between 2 and 10. In addition, set-up times are uniformly distributed between 30 and 250, while production times are between 1 and 4. Demand for the single product is distributed as a truncated normal with the mean uniformly distributed between 100 and 1000 and the standard deviation between 30 and 70. The capacity factor level is set as medium.

We generate 50 instances of the problem with this data, and we got that in all 30 cases, both of our algorithms end before the MILP procedure. As we see in the table, the restaurant procedure is much faster, but has an average gap of about 4%, while the classic procedure has a gap below the 1, but uses a longer calculation time. We now show how changing some parameters affects the performance of all three methods. In particular, in Figure 2.4 we can note how, even in the case of higher costs, our algorithms step better than the software resolution.

Let's now analyze how performance changes as the relationship between set up cost and holding cost varies. As is possible to see in Özdamar and Bozyel [2000], Trigeiro et al. [1989], Thizy and Wassenhove [1985], the difficulty of an instance depends on this ratio. The bigger the problem, the more difficult it is. Once the demand is fixed, the larger the ratio, the more difficult the problem becomes. To increase this ratio we kept all the parameters fixed and changed the set up costs. It's possible to notice from the table how as set up costs increase, the problem takes longer to be solved by all three methods. Despite this, the classical procedure remains competitive, both in terms of execution time and performance.

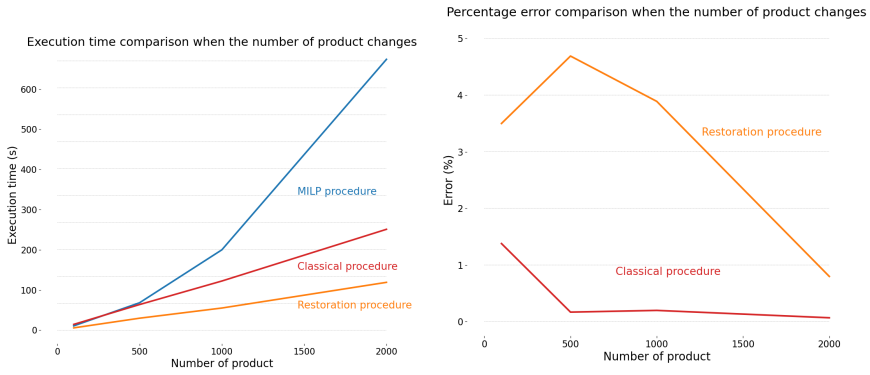

Figure 2.4: Comparison between the performances of MILP approach and our algorithms in case of large setup-holding costs ratio.

	Execution time(s)			Gap(%)	
	Exact	Classical	Restoration	Classical	Restoration
$f_i \in [0, 500]$	16(2)	97(6)	31(4)	0.04(0.01)	0.25(0.04)
$f_i \in [500, 1000]$	20(2)	43(5)	22(4)	0.06(0.01)	0.14(0.04)
$f_i \in [1000, 1500]$	41(6)	31(6)	21(5)	0.10(0.04)	0.19(0.09)
$f_i \in [1500, 2000]$	132(27)	64(26)	33(11)	0.52(0.44)	0.57(0.26)
$f_i \in [2000, 2500]$	277(84)	162(29)	56(24)	0.10(0.05)	3.03(2.50)
$f_i \in [2500, 3000]$	729(210)	203(38)	49(26)	0.22(0.06)	3.80(4.52)

Table 2.11: Average performances comparison when the ratio between set up cost and holding cost varies. Ten repetition are used, standard deviation in brackets.

Chapter 3

Stochastic Problem for Single Item Uncapacitated Lot Sizing

After solving the capacitated lot sizing problem (CLSP) in a deterministic setting, we consider its stochastic version. Before starting to solve the problem is better understand what we mean with *stochastic* and which are the main issues that come around in this context. In the deterministic case we have set up a model without needing any particular reasoning. The stochastic model, on the other hand, requires a series of modeling choices before being built, which can lead to different models. Also for this reason, we prefer to focus first on the uncpacitated single item lot sizing problem and, once set and solved, we proceed by inserting the capacity constraint. In this context we do not enter into the complications due to multiple items, but we try to fully understand how to efficiently solve the problem for the single product, with the hope that these bases can be used in the future to proceed to treat a multi item extension. We try to compare different approaches from very different fields, such as Dynamic Programming (DP) or Reinforcement Learning(RL). In Section 3.1 explains what is meant by stochastic. There are highlighted the difficulties that leads to consider this new version of the LSP and why it is no longer possible to solve the model as it was done previously. It is also highlighted how crucial it is to properly model the flow of time and information. In Section 3.2 we talk about the strategies we can implement to solve the problem. After a fairly theoretical discussion of DP approaches, we see how this can be used to build different resolution policies. We apply this technique to the ULSP on a single item. We start from small size problems to understand the advantages and drawbacks of each method. Then we implement approximate methods to solve the problem when the

size grows. Solutions are proposed both in case it is feasible to use a tabular representation of the problem and when we are forced to find continuous approximations. In Section 3.3 we try to use different approach and to solve the problem a Q-factors procedure is proposed, while in Section 3.4 numerical results are shown. Finally in Section 3.5 we deal with the stochastic CLSP on a single item, where a Lagrangian procedure is built to find a solution.

3.1 Uncertainty

It is not fundamental in the deterministic case, but for a model with uncertainty is important to understand how information changes over time. We have already mentioned the difference between *time instants* and *time intervals*, but it is better to emphasize it so as not to create confusion in the reader. Decisions, like how much to produce, are made in a time instant; state variables, like the inventory on hand, are checked in a time instant, while the demand is realized in a time interval.

- Time instants are indexed by $t = 0, \ldots, T$. In this moments we look at the system and make a decision. Observe that there are no decision to take at time $t = T$, while the value of the state variable could be relevant.
- A time interval is the time elapsed between two time instants. They are indexed by $t = 1, \ldots, T$. After taking a decision at time instant t the system evolves during the time interval $t + 1$.

This aspect is fundamental if consider a stochastic evolution for the system. In fact at time instant t we observe the state of the system, we choose a value for the decision variable and only later we will discover the risk factor (the demand).

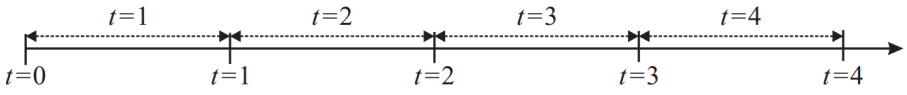

Figure 3.1: Illustration of time conventions taken from Brandimarte [2021].

After making these clarifications, the deterministic problem should be written in slightly differentway, compared to what has been done in Chapter

2

$$\begin{aligned}
\min \quad & \sum_{t=0}^{T-1} f\delta_t + \sum_{t=1}^{t=T} hI_t, && && (3.1)\\
\text{s.t.} \quad & I_{t+1} = I_t + x_t - d_{t+1}, && t = 0, \dots, T-1, && (3.2)\\
& x_t \leq M_t \delta_t, && t = 0, \dots, T-1,\\
& x_t, I_t, \geq 0, && t = 0, \dots, T-1,\\
& \delta_t \in \{0, 1\}, && t = 0, \dots, T-1.
\end{aligned}$$

where the cost involving the inventory on hand at time $t = 0$ is not consider because I_0 is given. Note that in the deterministic case normally the objective function does not contains the cost due to the quantity produced, in fact when demand is deterministic and we fully satisfying it, the total ordered amount is

$$\sum_{t=0}^{T-1} x_t = \sum_{t=1}^{T} d_t - I_0 + I_T \,.$$

Since in the optimal solution the ending inventory on hand is zero and I_0 is given, the total cost given from the amount produced is constant. A priori this is not true in the stochastic case, due to a well known issue: we may not guarantee demand satisfaction in each period (or sometimes it might be possible if the demand is finite, but only at a very high cost). There are two main situations that can happen when demand is not satisfy completely: if customers are not willing to wait for delivery at a later time, the excess demand is lost, if customers are patient, we may satisfy demand at a later time. In our model assumption we decide to deal with lost sales.

The deterministic model does not therefore have sense in the stochastic case. In fact we cannot be sure that the constraints $I_{t+1} = I_t + x_t - d_{t+1}$ and $I_{t+1} \geq 0$ are verified at the same time. Obviously, we can not delete constraint (3.2): in that cases the optimal solution will be a solution for which all variables are always zero. We need to try to meet the demand, and penalize lost sale. To do it we can add a new variable, the lost sale z_t. In this way it is possible to rewrite the constraint (3.2) like

$$I_{t+1} - z_{t+1} = I_t + x_t - d_{t+1} \qquad t = 0, \dots, T-1$$

where $z_t \geq 0$ is indexed by $t = 1, \dots, T$. Clearly, we have also to penalized the lost sales in the objective function with a penalty p for each unit of unsatisfied demand. Notice that p and h are in some sense related: the higher the ratio $\frac{p}{h}$, the greater the importance of satisfying demand. With

some caution, we could avoid using a penalty for the amount produced x_t, because if production is too little we have a penalty due to the term z_t, while if production is too large we have a penalty due to the term I_t. In any case we prefer adding this term for completeness. Another problem with the stochastic model is that if the demand is not known, it makes no sense to write in the model d_{it}, in fact this would be a random variable. What we can do, anyway, is minimize the expected value of the cost function. So the model in its stochastic version can be write as

$$\begin{aligned}
\min \quad & \mathbb{E}\Big[\sum_{t=0}^{T-1}(f\delta_t + cx_t) + \sum_{t=1}^{t=T}(hI_t + pz_t)\Big], \\
\text{s.t.} \quad & I_{t+1} - z_{t+1} = I_t + x_t - d_{t+1}, \qquad t = 0, \ldots, T-1, \\
& x_t \leq M_t\delta_t, \qquad t = 0, \ldots, T-1, \\
& x_t, I_t, z_t \geq 0, \\
& \delta_t \in \{0, 1\},
\end{aligned}$$

where the constraint should be understood in the sense of the recourse models. In the recourse programming the problem is divided into stages (in the simple case into two). In the first stage we must take a decision (in these case the production), while in the second stage, after seeing the realization of stochastic factors (the demand), we can take further decisions (inventory on hand and lost sales) to prevent the constraints of the problem from becoming infeasible. In other words, in the second phase an additional degree of flexibility is used to preserve feasibility (but at a cost). Note in particular that in this second phase the decisions we take depends on the particular realization of the stochastic elements observed, while in the first phase it does not.. Normally in these models the uncertainty is modeled through a scenario tree, while in very simple case an exact expected value can be computed.

An equivalent approach can be followed also without introducing the variable z_t. In fact, under the lost sales assumption, the state transition equation becomes

$$I_{t+1} = \max\{0,\, I_t + x_t - d_{t+1}\}$$

and in the objective function we can add the term

$$\sum_{t=1}^{T} p \max\{0,\, d_t - (I_{t-1} + x_{t-1})\}.$$

The model, in a recourse sense can be written as

$$\begin{aligned}
\min \quad & \mathbb{E}\Big[\sum_{t=0}^{T-1}(f\delta_t + cx_t) \\
& \quad + \sum_{t=1}^{t=T}(hI_t + p\max\{0,\, d_t - (I_{t-1} + x_{t-1})\})\Big], && (3.3)\\
\text{s.t.} \quad & I_{t+1} = \max\{0,\, I_t + x_t - d_{t+1}\}, \qquad t = 0, \ldots, T-1, && (3.4)\\
& x_t \leq M_t\delta_t, \qquad t = 0, \ldots, T-1, \\
& x_t, I_t \geq 0, \\
& \delta_t \in \{0, 1\}.
\end{aligned}$$

Note that in our previous consideration we assume that the only risk factor is demand: but we may also have to deal with uncertainty about costs. Considering other factors of uncertainty can complicate the calculation of the expected value and make simulation necessary even when using simple demand distributions.

In the literature there are numerous techniques to try to solve the stochastic lot sizing problem. One of the approaches used is the aforementioned programming with recourse. Otherwise our approach is based on finding a policy, i.e., a rule of action for every possible state of the system. Obviously to use such an approach it is necessary to build a model in which a state variable is present. In our case the inventory level on hand can be naturally used as state variable. In the following paragraphs we see different approaches to build this policy, based on both Dynamic Programming and Reinforcement Learning.

3.2 Dynamic programming

Dynamic Programming (DP) is a principle for solving challenging optimization problems based on the decomposition of a multistage problem by breaking it down into simpler subproblems and using the property that the optimal solution to the overall problem is composed by the optimal solutions of its subproblems. The result of a dynamic programming approach is often a strategy, i.e., a recipe to make decisions after observing random realizations (in a stochastic setting, as in our case) of risk factors and their impact on the system state. The concept of system state is central to dynamic programming and we need to represent its evolution over time, as a function of decisions and additional external inputs. However, dynamic programming

requires a specific model structure: in order to use it we need that the state of the system at time $t+1$ should depend only on the state observed at time t, the decision made at time t after observing the state, and the realization of external inputs during the subsequent time interval, i.e., the system must have a Markovian representation.

Among the whole range of problems to which dynamic programming can be applied, in the easy case ours belongs to those stochastic problems for which the state space is discrete, the decision variables are discrete and the time horizon is finite. Let's see in general how such a problem can be modeled. If we consider the time conventions from Figure 3.1 we can introduce the following notation.

- The vector of state variables at time instant t is denoted by $\boldsymbol{s}_t$. State variables contain all the information we need to control the evolution of the system (this does not mean that we are able to predict how it evolves). $\boldsymbol{s}_0$ is the initial state, which is typically known, whereas $\boldsymbol{s}_T$ is the terminal state. According to our time convention, $\boldsymbol{s}_t$ is the value of the state variables at the end of the time interval t, as a result of what happened between time instants $t-1$ and t.

- The vector of decision/control variables at time instant t, is denoted by $\boldsymbol{x}_t$. Decisions are based on the knowledge of the current state $\boldsymbol{s}_t$ (we assume that the state is perfectly observable).

- The vector of external factors at time period $t+1$ is denoted by $\boldsymbol{\xi}_{t+1}$. The state at time instant $t+1$ depends on the state and the selected decision at time t, but also on the realization of the external factor during the time interval $t+1$. The convention on the index is used to remember that $\boldsymbol{\xi}_{t+1}$ is observed after making decision $\boldsymbol{x}_t$.

The system dynamics is represented by a state transition equation like

$$\boldsymbol{s}_{t+1} = g_{t+1}(\boldsymbol{s}_t, \boldsymbol{x}_t, \boldsymbol{\xi}_{t+1}) \tag{3.5}$$

where the function g_{t+1} (that could not depend from time) explains how the transition from one state to the next takes place on the basis of the decisions made and the risk factors realization.

In the general case, a stochastic problem, with finite horizon T, might be stated as

$$\min \mathbb{E}_0 \left[\sum_{t=0}^{T-1} \gamma_t f_t(\boldsymbol{s}_t, \boldsymbol{x}_t) + F_T(\boldsymbol{s}_T) \right] \tag{3.6}$$

where $\gamma_t \in (0,1)$ is a discount factor, $f_t(\boldsymbol{s}_t, \boldsymbol{x}_t)$ is the immediate cost which we incur when we make decision $\boldsymbol{x}_t$ in state $\boldsymbol{s}_t$, while $F_T(\boldsymbol{s}_T)$ is the terminal cost, which only depend on $\boldsymbol{s}_T$. Generally we use $\gamma = 1$, because we want to give the same importance to immediate and future costs, but for example in financial applications you need to choose a value $\gamma < 1$. The notation $\mathbb{E}_0[\,\cdot\,]$ is used to point out that the expectation is taken at time $t = 0$; hence, it is an unconditional expectation, as we did not observe any realization of the risk factors yet. The dependence from the risk factor $\boldsymbol{\xi}_{t+1}$ is hidden but present: in fact the immediate cost at time t can be seen as:

$$f_t(\boldsymbol{s}_t, \boldsymbol{x}_t) = \mathbb{E}_t\left[h_t(\boldsymbol{s}_t, \boldsymbol{x}_t, \boldsymbol{\xi}_{t+1})\right]$$

where we suppose we're able to compute the expectation. We could take advantage on the additive form of Eq. (3.6) to devise a quick and dirty decision rule: when we are at state $\boldsymbol{s}_t$, we could simply solve the greedy problem

$$\min_{\boldsymbol{x}_t \in \mathcal{X}(\boldsymbol{s}_t)} f_t(\boldsymbol{s}_t, \boldsymbol{x}_t)$$

where $\mathcal{X}(\boldsymbol{s}_t)$ is the set of feasible decision at state $\boldsymbol{s}_t$. In general, as we imagine, this approach perform poorly, but we could use knowledge of the next state's value to balance short and long term goals. If we knew a suitable function $V_t(\,\cdot\,)$ mapping every state $\boldsymbol{s}_t$ at time t into its value, we could apply a decomposition strategy leading to a sequence of single stage problems.The value $V_t(\boldsymbol{s})$ should be the expected cost obtained when we apply an optimal policy from time t onwards, starting from state $\boldsymbol{s}$. More specifically, when in state $\boldsymbol{s}_t$ at time t, we should select the current decision $\boldsymbol{x}_t \in \mathcal{X}(\boldsymbol{s}_t)$ that optimizes the sum of the immediate contribution to performance and the discounted expected value of the next state:

$$V_t(\boldsymbol{s}_t) = \min_{\boldsymbol{x}_t \in \mathcal{X}(\boldsymbol{s}_t)} \left\{ f_t(\boldsymbol{s}_t, \boldsymbol{x}_t) + \mathbb{E}\left[V_{t+1}(g_{t+1}(\boldsymbol{s}_t, \boldsymbol{x}_t, \boldsymbol{\xi}_{t+1}) | \boldsymbol{s}_t, \boldsymbol{x}_t\right]\right\} \tag{3.7}$$

This recursive functional equation, known as Bellman's equations, is at the core of DP. The optimal decision $\boldsymbol{x}_t^*$ is obtained by solving an optimization problem parameterized by the current state $\boldsymbol{s}_t$, based on the knowledge of the value function $V_{t+1}(\,\cdot\,)$. In the simple case of a small finite state space, we may directly associate the optimal decision with each state. In this lucky case, we may represent the optimal decision policy in a tabular form: we just have to store, for each time instant t and state $\boldsymbol{s}_t$, the corresponding optimal decision. When the space is continuous or even discrete but of huge dimension this is not possible but we can find the policy in an implicit form,

i.e.,

$$\boldsymbol{x}_t^* = \mu_t^*(\boldsymbol{s}_t) \in \mathcal{X}(\boldsymbol{s}_t)$$

where $\mu_t(\,\cdot\,)$ is a function mapping the state at time t into a feasible decision. Note that if we denote $\mathcal{M}$ like the set of the feasible policy we can rewrite the problem like

$$\min_{\boldsymbol{\mu}\in\mathcal{M}} \mathbb{E}_0 \left[\sum_{t=0}^{T-1} f_t(\boldsymbol{s}_t, \mu_t(\boldsymbol{s}_t)) + F_T(\boldsymbol{s}_T) \right] \tag{3.8}$$

In this sense our goal is to find a set of functions, the overall optimal policy, $\boldsymbol{\mu}^* = (\mu_0^*, \mu_1^*, \cdots, \mu_{T-1}^*)$.

Like other functional equations, we need boundary conditions to solve Eq. (3.7). In our case, the natural solution process goes backward in time, starting from the terminal condition

$$V_T(\boldsymbol{s}_T) = F_T(\boldsymbol{s}_T) \qquad \forall \;\; \boldsymbol{s}_T$$

At the last decision time instant, $t = T-1$, we should solve the single-stage problem

$$V_{T-1}(\boldsymbol{s}_{T-1}) = \min_{\boldsymbol{x}_{T-1}\in\mathcal{X}(\boldsymbol{s}_{T-1})} \Big\{ f_{T-1}(\boldsymbol{s}_{T-1}, \boldsymbol{x}_{T-1}) + \mathbb{E}\left[V_T(g_T(\boldsymbol{s}_{T-1}, \boldsymbol{x}_{T-1}, \boldsymbol{\xi}_T) | \boldsymbol{s}_{T-1}, \boldsymbol{x}_{T-1} \right] \Big\},$$

for every possible state $\boldsymbol{s}_{T-1}$. This is a static, but not myopic problem, since the terminal value function $V_T(\,\cdot\,)$ also accounts for the effect of the last decision $\boldsymbol{x}_{T-1}$ on the terminal state. By solving the problem recursively backward in time, at the end, given the initial state $\boldsymbol{s}_0$, we find the first optimal decision by solving the single-stage problem

$$V_0(\boldsymbol{s}_0) = \min_{\boldsymbol{x}_0\in\mathcal{X}(\boldsymbol{s}_0)} \left\{ f_0(\boldsymbol{s}_0, \boldsymbol{x}_0) + \mathbb{E}\left[V_1(g_1(\boldsymbol{s}_0, \boldsymbol{x}_0, \boldsymbol{\xi}_1) | \boldsymbol{s}_{T-1}, \boldsymbol{x}_0 \right] \right\}.$$

Once we have found the cost of the overall policy $V_0(\boldsymbol{s}_0)$ for a given state $\boldsymbol{s}_0$ if we want to check the policy, we may proceed as follows:

- Given the initial state $\boldsymbol{s}_0$ and the value function $V_1(\,\cdot\,)$ solve the first stage problem and find $\boldsymbol{x}_0^*$
- Observe the random risk factors realization $\boldsymbol{\xi}_1$ and use the transition function to generate the next state, $\boldsymbol{s}_1 = g_1(\boldsymbol{s}_0, \boldsymbol{x}_0^*, \boldsymbol{\xi}_1)$

- Given the state $\boldsymbol{s}_1$ and the value function $V_2(\,\cdot\,)$, solve the second-stage problem and find $\boldsymbol{x}_1^*$.
- Repeat the process until we generate the last decision $\boldsymbol{x}_{T-1}$ and the terminal state $\boldsymbol{s}_T$.

The S and (s,S) Policies

Dynamic programming may be applied to prove that the optimal decision strategy for a given problem has a specific form. This means that maybe we cannot easily find the numerical values that define an optimal strategy, but we may use DP to infer its structure. The structure of an optimal policy may be characterized in different ways but, sometimes, we may be able to come up with an optimal decision rule depending on a small set of unknown parameters. This is the case of the stochastic single item lot sizing problem.

As we seen in the first part of this thesis, the Wagner–Whitin condition provides us with an efficient approach to solve a deterministic and uncapacitated lot-sizing problem. As mentioned above, in the stochastic case we have to accept that we are not always able to meet demand. Let us assume that the excess demand is lost, so that there is a lost sales penalty and the total cost function includes a term like

$$q(s) = h\max\{0, s\} + p\max\{0, -s\}$$

We can obtain this form with a rewriting of Eq. (3.3). It is enough to remember that $I_{t+1} = \max\{0,\, I_t + x_t - d_{t+1}\}$ and so

$$hI_{t+1} + p\max\{0,\, d_{t+1} - (I_t + x_t)\}$$

becomes

$$h\max\{0,\, I_t + x_t - d_{t+1}\} + p\max\{0,\, d_{t+1} - (I_t + x_t)\}$$

that is in the form of $h\max\{0,\, s\} + p\max\{0,\, -s\}$ where we have chosen $s = I_t + x_t - d_{t+1}$. For sake of simplicity we disregard fixed cost in a first moment, but we include a linear variable cost, with unit ordering cost c. Hence, the overall problem requires to find a policy minimizing the expected total cost over T time periods:

$$\mathbb{E}_0\left[\sum_{t=0}^{T-1}\left\{cx_t + q(I_t + x_t - d_{t+1})\right\}\right]$$

We may write the DP recursion as

$$V_t(I_t) = \min_{x_t \geq 0} \Big\{ cx_t + H(I_t + x_t) + \mathbb{E}\big[V_{t+1}(I_t + x_t - d_{t+1})\big] \Big\}$$

where we define $H(y_t) \coloneqq \mathbb{E}\big[q(y_t - d_{t+1})\big] = h\,\mathbb{E}\big[\max\{0, y_t - d_{t+1}\}\big] + p\,\mathbb{E}\big[\max\{0, d_{t+1} - y_t\}\big]$ We assume zero lead time, so $y_t = I_t + x_t$ is the available inventory after ordering and immediate delivery. Note that if, as done in Brandimarte [2021], we define

$$G_t(y_t) = cy_t + H(y_t) + \mathbb{E}\big[V_{t+1}(y_t - d_{t+1})\big] \tag{3.9}$$

we can conveniently rewrite the recursion as

$$V_t(I_t) = \min_{y_t \geq I_t} G_t(y_t) - cI_t$$

From the property of q is possible to show that $V(\,\cdot\,)$ and $G(\,\cdot\,)$ are convex and they go to $+\infty$ when $y \to \pm\infty$. See Bertsekas [1995] for more information. Now we can claim that $G_t(\,\cdot\,)$ has a finite unconstrained minimizer

$$S_t = \operatorname*{argmin}_{y_t \in \mathbb{R}} G_t(y_t)$$

However, we have the constraint $y_t \geq I_t$. If we define unconstrained and constrained minimizers, S_t and y_t^*, respectively. There are two possibilities

- if the constraint is not active, $y_t^* > I_t$, then the constrained and unconstrained minimizer are equal: $S_t = y_t^*$
- if the constraint is active, $y_t^* = I_t$, then the inventory levels before and after ordering are the same, which clearly implies $x_t^* = 0$

Then, the optimal policy follows the rule:

$$x_t^* = \mu_t^*(I_t) = \begin{cases} S_t - I_t, & I_t < S_t \\ 0, & I_t \geq S_t \end{cases}$$

S_t is like a target inventory levels: whenever we reach the critical threshold, we should produce. All we have to do is finding the optimal sequence of target inventory levels S_t, although it may not be trivial. If we now consider the setup costs, as demonstrated in Bertsekas [1995], the optimal policy changes slightly in

$$x_t^* = \mu_t^*(I_t) = \begin{cases} S_t - I_t, & I_t < s_t \\ 0, & I_t \geq s_t \end{cases}$$

depending on two sequences of parameters s_t and S_t, where $s_t \leq S_t$. Moreover in a stationary environment, we find that a stationary (s, S) policy is optimal. To find these two values we will use some variation of Montecarlo simulation. The results found with (s, S) policy will be used as a benchmark for the other methods we will implement. Furthermore, this policy is also be applied in case in which a capacity constraint in present to understand how our algorithms perform.

3.2.1 Discrete Demand with Small Support

In principle, if we assume a discrete random demand, we may adopt a tabular representation of the value function. However, in practice, this can be done only when a few values are possible for the demand. In fact, one should build a $T \times I_{\max}$ matrix and solve $T \times I_{\max}$ optimization problems, where $I_{\max}$ is an upper bound on the state variable that depends from D, the cardinality of the demand state space.

We consider lost sales, so the transition equation is Eq. (3.4) that we rewrite here

$$I_{t+1} = \max\{0,\, I_t + x_t - d_{t+1}\}$$

where in this case $(d_t)_{t=1,\dots,T}$ is a sequence of i.i.d. discrete random variable and we consider zero lead time. We repeat the exact sequence of events

- At time t we observe the inventory on hand I_t.
- We make production decision x_t.
- We observe the realization of demand, d_{t+1} and update the inventory on hand.

Note that we need for programming purposes to set a limitation on inventory: in fact, when we tabulate the value function, we need a value for each time instant and for each inventory level.

$$I_t \leq I_{\max}$$

This also imposes a limitation on the quantity produced which must necessarily be positive, but which must be at most equal to quantity $I_{\max} - I_t$ in order to be crammed into the warehouse. So the feasible set for the produced quantity is

$$\mathcal{X}_t(I_t) = \big\{0, 1, \dots, I_{\max} - I_t\big\}.$$

To construct the value function, we must analyze the immediate cost at the generic instant of time t. This is made up of different contributions. The first

is a linear production cost cx_t to which it can also be added a fixed cost $f\delta_t$. Then, we must take into account cost related to the inventory and the lost sales. If a lost sale and an extra item in stock had the same weight, we could use a term like $\beta|I_t + x_t - d_{t+1}|$ or $\beta(I_t + x_t - d_{t+1})^2$, but in a more general situation the two penalties are not symmetrical. Hence in the end, we have an immediate cost term that depends on the realization of the risk factor during the time period $t+1$ after making the decision x_t. So the recursion of dynamic programming have an immediate cost which is stochastic:

$$V_t(I_t) = \min_{x_t \in \mathcal{X}_t(I_t)} \mathbb{E}\Big[c\, x_t + f\, \delta_t + h \max\{0, I_t + x_t - d_{t+1}\} \\ + p \max\{0, d_{t+1} - (I_t + x_t)\} + V_{t+1}(I_{t+1})\Big]$$

for $t = 0, 1, \ldots, T-1$ and $I_t = 0, 1, \ldots, I_{\max}$. For simplicity, we choose $V_T(I_T) = 0$ for every value of the inventory. The only risk factor is simply a sequence of i.i.d. discrete random variables, then all we need to model uncertainty is a probability vector π_k for each possible value of the demand $k = 0, 1, \ldots, d_{\max}$.

Example 3.1. Suppose to use the simplified version of the recursion equation in which no set up costs incurs and a linear non-symmetric cost is given for inventory on hand and lost sales:

$$V_t(I_t) = \min_{x_t \in \mathcal{X}_t(I_t)} \mathbb{E}\Big[c\, x_t + h\, I_{t+1} + pI^-_{t+1} + V_{t+1}(I_{t+1})\Big]$$

where, to simplify the notation, we indicated $\max\{0, d_{t+1} - (I_t + x_t)\}$ with I^-_{t+1}. Furthermore we set $T = 4$, $I_{\max} = 4$ and $d_{max} = 3$, with $\pi_k = (0.2, 0.3, 0.4, 0.1)$, $h = 1$, $c = 2$ and $p = 5$.

The dynamic programming algorithm gives the following results:

```
value_table
 [[16.6  12.8   9.    5.2   0.  ]
  [14.6  10.8   7.    3.2   0.  ]
  [12.6   8.8   5.    1.2   0.  ]
  [11.36  7.62  4.08  1.6   0.  ]
  [10.97  7.4   4.4   2.6   0.  ]]

action_table
 [[2. 2. 2. 1.]
  [1. 1. 1. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]
```

As we can see, in this simple case, dynamic programming gives us as a result a policy with a single value of S. At each instant of time, if the inventory on hand is below the threshold $S = 2$, we produce until we have two pieces. To be precise in this example, the value of S is constant except for the last instant of time, but this is due to the fact that our dynamic programming assigns very weak end-of-inventory conditions and therefore has a myopic behavior in the last period. Let us now try to simulate the results obtained both using the action table and using the S policy. By sampling 10000 times the demand from the discrete distribution used earlier and calculating the average cost when the inventory at the time instant 0 is 2, we get:

```
average_dp_cost
 12.56

average_s_cost
 12.62
```

It is not now in our interest to say whether statistically the two results are equivalent, but by eye we see that we have obtained the results expected from the value table.

Example 3.2. Consider now a full version of the recursion equation with a linear non-symmetric cost is given for inventory on hand and lost sales:

$$V_t(I_t) = \min_{x_t \in \mathcal{X}_t(I_t)} \mathbb{E}\Big[c\, x_t + f\delta_t + h\, I_{t+1} + pI_{t+1}^{-} + V_{t+1}(I_{t+1})\Big]$$

Furthermore, we set $T = 5$, $I_{\max} = 6$ and $d_{max} = 4$, with $\pi_k = (0.1, 0.2, 0.15, 0.3, 0.25)$, $h = 3$, $c = 2$, $f = 1$ and $p = 5$. The dynamic programming algorithm gives the following results:

```
value_table
 [[45.66 36.77 27.88 19.   10.2   0.  ]
  [43.66 34.77 25.88 17.    7.8   0.  ]
  [41.12 32.23 23.34 14.42  5.2   0.  ]
  [38.66 29.77 20.88 12.    3.8   0.  ]
  [38.14 29.25 20.38 11.71  4.8   0.  ]
  [39.54 30.67 21.88 13.62  7.8   0.  ]
  [41.58 32.75 24.15 16.6  10.8   0.  ]]

action_table
 [[3. 3. 3. 3. 2.]
  [2. 2. 2. 2. 0.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]]
```

In this case, looking at the action table, we can see that the best policy, as we expected, is of type (s, S), with $s = 2$ and $S = 3$: whenever the inventory drops below threshold 2, produce until it reaches the level 3. As in the previous example, we note that, due to the constraint $V_T(I_T) = 0$, the values of s and S depend on the instant of time that is considered and in particular they are equal to 1 and 2 respectively in the last instant of time. Like before we can try to simulate the results obtained both using the action table and using the (s, S) policy. By sampling 10000 times the demand from the discrete distribution used earlier and calculating the average cost when the inventory at the time instant 0 is 3, we get:

```
average_dp_cost
 38.61

average_s_cost
 39.20
```

The cost of policy (s, S) is slightly higher because we used constant values of s and S. If, on the other hand, we make them vary in the last instant of time, the result is

```
average_s_cost
 38.65
```

in line with that obtained with the action table.

The majority of the literature that treats the lot sizing problem tends not to consider and to undervalue the problem of the so-called "end of inventory conditions". All deterministic models in fact in the optimal solution, leave the inventory empty. This condition is almost improbable: just because we are doing an analysis over six months, does not mean that the company will close in six months. The justifications for this lack of care are the fact that the model, built for six months, is revised every month and therefore the effect of not modeling the inventory level at the end of the time horizon, is mitigated. In the stochastic case the situation is slightly different, we are not sure of the value of the inventory at the end of the time horizon, but, as for example in the proposed dynamic programming algorithm, the fact that the value function in the last instant is always zero, it makes sure that ending with an empty or very low level of inventory is not penalizing: for this reason, as we have seen before, in the last period our algorithm tells us to produce less than normal. It happens that the problem of the value of the level of the terminal inventory is by no means trivial and it is not our aim to deal with it here. However, we would like not to ignore the problem

completely, so we have developed a series of heuristics to be used to assign a value to the value function at the terminal instant.

The first heuristic we developed is very rough, but as we notice from the results shown below, it can give a first help to mitigate the problem of terminal inventory and in fact it is the one we use in the rest of the work. What we do is to look at the average demand and figure out how much it would cost to meet it depending on the level of stock we have at the end of the time horizon. Suppose a moment for ease that the average demand is an integer d. If we had an inventory level d, we would not have to spend something to satisfy it. If we had less than d, we would have to spend to produce, then the fixed cost plus the unit cost multiplied by the quantity produced. If we had more than d, instead, we would not have production costs, but we would have inventory costs the next instant, because we would have something in stock. If we apply this approach to the previous toy examples we obtain in the first case

```
action_table
 [[2. 2. 2. 2.]
  [1. 1. 1. 1.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]
```

and in the second case

```
action_table
 [[3. 3. 3. 3. 3.]
  [2. 2. 2. 2. 2.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]]
```

that is by modifying the value function heuristically we get the approach s or (s, S) as expected. Even in the simple case in which we are, however, our heuristic is not enough to always bring decisions to those of optimal policy (s, S). If we choose $T = 5$, $I_{\max} = 10$ and $d_{max} = 9$, with $\pi_k = (0.05, 0.05, 0.1, 0.1, 0.2, 0.15, 0.1, 0.1, 0.10, 0.05)$ and $h = 3$, $c = 2$, $f = 7$ and $p = 5$ we obtain the following action table

```
action_table
 [[6. 6. 5. 5. 0.]
  [0. 0. 0. 4. 0.]
  [0. 0. 0. 0. 0.]
        ...
```

```
       ...
  [0. 0. 0. 0. 0.]]
```

which shows a behavior far from that of a policy (s, S). If we use our heuristic to change the value function in the last time instant, the table change in

```
action_table
 [[6. 6. 6. 6. 5.]
  [0. 0. 0. 5. 0.]
  [0. 0. 0. 0. 0.]
       ...
       ...
  [0. 0. 0. 0. 0.]]
```

which once again does not correspond to a policy (s, S) but certainly comes closer, suggesting that it could be optimal to choose $S = 6$ and $s = 1$.

Of course, the first heuristic method presented in this section does not take the form of demand distribution into account at all and assumes that it is optimal to produce to meet the average demand. This, as it is widely known (newsvendor problem) is only true if there are no fixed costs and if the penalty for lost sales is equal to the inventory cost. However, before we start building complicated methods that allow us to accurately estimate the value function in the final time interval, let's ask ourselves what it would mean to do it: it would simply mean adding a column to the value table. In fact it would simply mean moving the column with all zeros to the right. This suggests a second, rather obvious heuristic. If I have to choose production from here to 6 months, I simulate production for $8 - 9$ months and then I choose the best actions to implement only for the first 6, which is what is done in practice. Below the three action table of before when using $T + 2$ periods.

```
action_table
 [[2. 2. 2. 2. 2. 1.]
  [1. 1. 1. 1. 1. 0.]
  [0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0.]]

action_table
 [[3. 3. 3. 3. 3. 3. 2.]
  [2. 2. 2. 2. 2. 2. 0.]
  [0. 0. 0. 0. 0. 0. 0.]
           ...
           ...
  [0. 0. 0. 0. 0. 0. 0.]]
```

```
action_table
 [[6. 6. 6. 6. 5. 5. 0.]
  [0. 0. 0. 0. 0. 4. 0.]
  [0. 0. 0. 0. 0. 0. 0.]
            ...
            ...
  [0. 0. 0. 0. 0. 0. 0.]]
```

If we remove the last two periods, we can mitigated the lack of management for the final value of the value function

Finding the Optimal (s,S) Policy

In the example presented before, after looking at the action table, is possible understand the right value for S or for the couple (s, S). In general, however, we would like to calculate the optimal value of S without having to use the action table. In case the support is small it is also possible to run a simulation for all values and choose the one that gets the best results, but when the possible stock values grow and especially when using the set up costs (and therefore both s and S), we must find a less expensive way to proceed.

Since the 1960s, a wide range of methods have been developed for finding the optimal policy (s, S). Most, such as Arthur F. Veinott and Wagner [1965], despite obtaining the optimal pair of values, were unusable, because they were prohibitively expensive for computation. In the mid-1980s, efficient exact algorithms began to be developed, mostly in the wake of the work of Federgruen and Zipkin [1984]. Zheng and Federgruen [1991] defines $G(y)$ the one-period expected costs, including only holding and lost sales penalty costs, when starting with an inventory position y

$$G(y) = h\,\mathbb{E}\big[\max\{0, y-d\}\big] + p\,\mathbb{E}\big[\max\{0, d-y\}\big],$$

and he finds the optimal pair of value (s, S) using a minimizer of function G.

We, on the other hand, preferred to implement a statistical approach to find the desired pair (s, S). In particular, we can use the cross entropy method, already presented in the first section, in which we treated the deterministic case. This time, however, the parameters to be estimated are no longer binary but positive integers. As done in Costa et al. [2007] we create a map from integers to binaries (simply using the binary representation) and apply the cross entropy algorithm again on 0/1 variables. To understand the number of bits needed, we use the inventory capacity $I_{\max}$, since $s < S < I_{\max}$. So if $I_{\max}$ can be represented on n bits, we are sure that n bit will suffice both for s and for S.

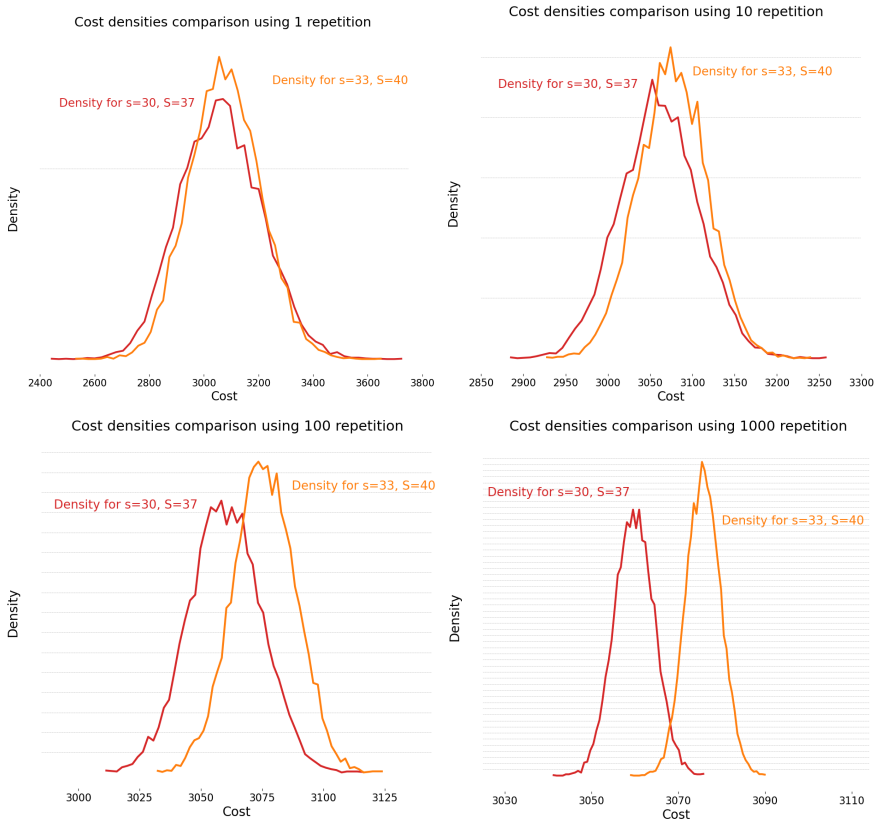

Figure 3.2: Graph shows how densities become more defined when increasing the number of repetitions.

The main drawback in this way of proceeding is that in this case the objective function involve an expected value: to understand how well each pair of proposed values (s, S) perform, we have to start a simulation. Obviously, the number of repetitions used affect both the speed and accuracy of the simulation: it is a well known fact that in the limit the sample average policy cost $\overline{c}(s, S)$ converges to the expected value $\mathbb{E}\big[c(s, S)\big]$ and that its standard deviation will depend from a factor $\frac{1}{\sqrt{n}}$, where n is the sample size.

Therefore the danger consist in choosing s_1, S_1 instead of s_2, S_2 relying on the fact that $\overline{c}(s_1, S_1) < \overline{c}(s_2, S_2)$, unaware that $\mathbb{E}\big[c(s_2, S_2)\big] < \mathbb{E}\big[c(s_1, S_1)\big]$. To better visualize the problem, we consider an example where $I_{\max} = 100$ and we show the sample density distribution of $\overline{c}(30, 37)$ and $\overline{c}(33, 40)$ when $n = 1, 10, 100, 1000$ repetition are performed. As is possible to see from the graphs, when n is small it's hard to tell if an observation comes from one

density or another. As n grows, the task becomes easier and easier, and as a result it is possible to order alternatives more easily. From our point of view we can claim that if in the cross entropy method, to calculate the cost of each alternative we use a number of high repetitions, we are more certain of its reliability. On the other hand, during the method, such simulation goes done again and again, and it is not possible to increase too much the number of repetitions if one wants to obtain a result in short times.

3.2.2 Discrete Random Demand with Large Support

The main difference from the previous paragraph is that we are now considering a support too large to deal with the problem using an exact dynamic programming. What we do to decrease the execution time is, instead of considering all the values between 0 and $I_{\max}$ for the state variable I_t at time time t, use only a subset of them. Specifically, once we have chosen a simplification factor k, we consider only the levels of the status variable $0, k, 2k, \ldots, \lfloor \frac{I_{\max}}{k} \rfloor k$. In this way the state variable takes only $\lfloor \frac{I_{\max}}{k} \rfloor + 1$ different values. If now we start the dynamic programming procedure we run into the problem of not knowing the value function $V(s)$ for each s, as we would need. In fact at step t we should evaluate $V_{t+1}(\max\{0, I_t + x_t - d_{t+1}\}$, but $V(\,\cdot\,)$ is defined only when $I_t + x_t - d_{t+1}$ is a multiple of k. In all other cases we are not able to compute the value function. A widely used approach in this context (Cervellera et al. [2007], Trick and Zin [1997]) is to approximate the value function by cubic splines.

Then we calculate the value function only in the points of the grid chosen, but, in computing the expected value use an approximation of the value function defined for each possible level of the inventory. Usually in the approximate dynamic programming the goal of the procedure is only to find the value function in the selected points. Our algorithm, however, also gives output an action table at the same points, useful to simplify calculations during the simulation.

Approximate dynamic programming: value and action table generation

input: $T, I_{\max}, k$
output: V, A
initialize the value table V and the action table A
for $t \in [T-1, \dots, 0]$ **do**
    Let $\widetilde{V}$ be the value table approximation at time t
    for $s \in [0\,,\, k\,,\, 2k\,,\, \dots\,,\, \lfloor \frac{I_{\max}}{k} \rfloor k]$ **do**
        $c^* = +\infty$, $a^* = \text{None}$
        for $x_t \in [0, \dots, I_{\max} - s]$ **do**
            $c_x = \mathbb{E}\Big[c\,x_t + f\delta_t + h\,I_{t+1} + pI^-_{t+1} + \widetilde{V}_{t+1}(I_{t+1})\Big]$
            if $c_x < c^*$ **then**
                $c^* = c_x$, $a^* = x_t$
            end if
        end for
        $V(s/k, t) = c^*$, $A(s/k, t) = a^*$
    end for
end for

Approximate dynamic programming: simulation

input: $T, I_{\max}, k, n, V, A, s$
output: $\overline{c}$
initialize: c vector of size n
for $i \in [1, \dots, n]$ **do**
    $c^*(i) = +\infty$
    for $t \in [0, \dots, T]$ **do**
        Let $\widetilde{V}$ be the value table approximation at time t
        $x^*_t = \text{None}$
        if $\frac{s}{k} = \lfloor \frac{s}{k} \rfloor$ **then**
            $x^*_t = A(\frac{s}{k}, t)$
        else
            if $\lceil \frac{s}{k} \rceil = \lfloor \frac{s}{k} \rfloor$ **then**
                $x^*_t = A(\lfloor \frac{s}{k} \rfloor, t)$
            else
                $m = A(\lceil \frac{s}{k} \rceil, t)$
                $M = \min\{A(\lfloor \frac{s}{k} \rfloor, t),\, I_{\max} - s\}$
                $c^{**} = +\infty$
                for $x_t \in [m, \dots, M]$ **do**
                    $c_x = \mathbb{E}\Big[c\,x_t + f\delta_t + h\,I_{t+1} + pI^-_{t+1} + \widetilde{V}_{t+1}(I_{t+1})\Big]$
                end for

21: **if** $c_x < c^{**}$ **then**
22: $c^{**} = c_x$, $x_t^* = x_t$
23: **end if**
24: **end if**
25: **end if**
26: $c^*(i) = c^*(i) + \mathbb{E}\Big[c\, x_t^* + f\delta_t + h\, I_{t+1} + pI_{t+1}^- + \widetilde{V}_{t+1}(I_{t+1})\Big]$
27: $s = \max\{0,\, x_t^* - d_{i\,t+1}\}$
28: **end for**
29: **end for**
30: $\overline{c} = \frac{1}{n}\sum_i c^*(i)$

Before proceeding with the case in which we consider the demand continuously distributed, let's do some tests to understand how the different proposed procedures perform in case of discrete demand, but with very large support. We compare the time and quality of the results of the algorithm based on exact dynamic programming with the approximate one (with $k = 5, 10, 20, 50$) and with the one in which the policy (s, S) is used, where the parameters s and S are estimated with the cross entropy. We divide the simulation into two tranches. The first with $d_{\max} = 120$ and $I_{\max} = 250$, the second with $d_{\max} = 300$ and $I_{\max} = 500$. The time horizon is $T = 30$, while all cost parameters are sampled between a minimum and a maximum value. Two different analysis are performed, one in which the value function in the last time instant is always zero and one in which the heuristic mentioned above is used to fill the table in the last period. Ten instances are generated, and for each of them the evaluation of the result is made on 1000 repetitions. The evaluation measure is the per thousand variation from the optimal cost, where as optimal cost is considered the one obtained from the exact dynamic programming.

Although the tests done are not sufficient in number and variety, we can see that the implementation time of the dynamic programming procedure becomes 4 times bigger when the values that can obtain the demand are doubled. In fact, even if the complexity of the algorithm that implements dynamic programming is not directly quadratic in the demand, in it is $O(I_{\max} d_{\max} T)$, and when we increase $d_{\max}$ there is a need to also increase the maximum inventory level $I_{\max}$. We also note that the duration of the algorithm with the implementation of approximate dynamic programming is about a k-th of that of the original algorithm. This is because obviously the value table is calculated in a simplified grid of a factor k. The last algorithm tested, the one concerning the choice of parameters s and S of

	Gap (‰) from exact DP				
	Appr_5	Appr_{10}	Appr_{20}	Appr_{50}	(s, S)
$I_{\max} = 250$ e.o.v. = F	0.01	0.03	0.03	0.07	4.13
$I_{\max} = 250$ e.o.v. = T	0.01	0.03	0.03	0.16	3.31
$I_{\max} = 500$ e.o.v. = F	0	0	0.01	0.61	3.35
$I_{\max} = 500$ e.o.v. = T	0.01	0.01	0.01	0.02	2.23

Table 3.1: Average Gap on 10 instances of the problem.

	Elapsed time (s)					
	Exact	Appr_5	Appr_{10}	Appr_{20}	Appr_{50}	(s, S)
$I_{\max} = 250$ e.o.v. = F	$17,7$	$5,2$	$2,6$	$1,4$	$0,6$	$1,7$
$I_{\max} = 250$ e.o.v. = T	$17,4$	$5,1$	$2,6$	$1,3$	$0,59$	$1,8$
$I_{\max} = 500$ e.o.v. = F	$72,0$	$22,6$	$11,3$	$5,7$	$2,4$	$3,3$
$I_{\max} = 500$ e.o.v. = T	$72,0$	$21,6$	$10,6$	$5,4$	$2,2$	$4,3$

Table 3.2: Average elapsed time on 10 instances of the problem.

the method (s, S) has an execution time that depends a lot on the internal parameters. These parameters were chosen so that the algorithm could get good values for the reorder thresholds and so that it was competitive with the approximated dynamic programming for k large. To be fair, it should be noted that although approximate dynamic programming is very fast in building the model, the simulation time increases dramatically. This is not a problem, however, because our objective is to build a model quickly and that can occupy little memory. The simulation is certainly important, but once the model is validated there will be no need to test it on millions of instances. A company could have dozens, hundreds of cost forecasts, and you need to test the model on those. As for the results, however, we can see that the approximate dynamic programming manages to perform at the level of the exact one. The cost deviation from the exact dynamic programming implementation is on average less than 0.001%. Even the results obtained with the (s, S) policy are not daunting, departing from the optimal of about 0.3%. In particular, we note that when we use our heuristic to choose the value function in the instant terminal, the performance obtained with the (s, S) policy improves by about 30%. In all subsequent tests, the end of inventory value heuristic will be applied.

3.2.3 Continuous Random Demand

When dealing with a situation where average demand values are very large, any tabular approximation becomes too expensive to be used, both from the computational and memory point of view. Even when we talk about companies that produce units, and therefore we should talk about a demand with integer values, often in literature in these cases we prefer to work with a demand distributed continuously. The idea we apply is very similar to that of approximate dynamic programming. In this case, however, we do not save a value function in tabular form, although small in size, but only the coefficients of splines, one for each time interval, through which the value function is approximated.

Obviously, the fact of having only the spline coefficients, causes that during the simulation phase, you have to solve a small optimization problem. Furthermore, since it is no longer possible to represent the probability density of demand through a vector, the expected value of the value function shall be calculated by simulating the demand. If we not consider setup costs, the cost function at time t for producing a quantity x_t is given by

$$\begin{aligned} C_t(x_t) = {} & cx_t + h\,\mathbb{E}\big[\max\{0, I_t + x_t - d_{t+1}\}\big] + p\,\mathbb{E}\big[\max\{0, d_{t+1} - I_t - x_t\}\big] \\ & + \mathbb{E}\big[V_{t+1}(I_t + x_t - d_{t+1})\big]. \end{aligned}$$

Minimizing this function would seem a hopeless task, but we can use the fact that the function $G_t(\cdot)$, defined in Eq. (3.9), that we rewrite for convenience

$$\begin{aligned} G_t(y_t) = {} & cy_t + h\,\mathbb{E}\big[\max\{0, y_t - d_{t+1}\}\big] + p\,\mathbb{E}\big[\max\{0, d_{t+1} - y_t\}\big] \\ & + \mathbb{E}\big[V_{t+1}(y_t - d_{t+1})\big] \end{aligned}$$

is convex in $y_t = I_t + x_t$. Now, it is enough to notice that the value I_t is already known when we make the decision and that consequently cI_t is a constant term, to claim that the cost function, which can be written as

$$C_t(x_t) = G(x_t + I_t) - cI_t,$$

is a convex function in the variable x_t. The fact that set up costs are present, does not create complications, in fact the function $C_t(\cdot)$ has a shift upwards of a constant term for each value of x_t, excluding $x_t = 0$. Then the function remains convex for $x_t > 0$ and to find the optimal production plan it is enough to calculate the minimum of the function for $x_t > 0$ and compare it with the option not to produce at all.

In this case, however, we have to complication: the first is that we do not know the value function, but only its approximation by spline, the second is

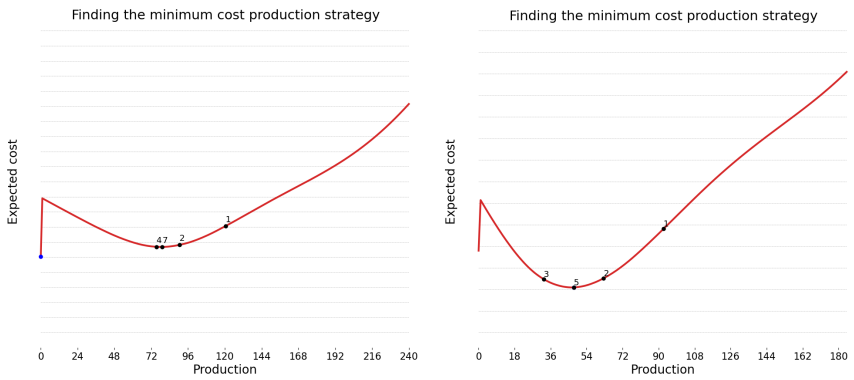

Figure 3.3: Procedure to find the minimum of the function. Black point represent the best value at that iteration. The blue point is the zero value, which is checked at the end.

that we can not compute the exact expected values. We have no guarantee that the approximation is still a convex function, and indeed, from the tests done, it appears that in general it may present region in which the function is not convex. Despite this, in the many tests we have done, the non convexity of the function has never given problems in its minimization. So we decided to use an algorithm that works for convex functions. To find the minimum point of the function, we build a raw algorithm, which allows, however, to obtain results with a good approximation in a short time. The approach is as follows. We starts considering a certain production value x, and compares to the values of $x-d$ and $x+d$, where d is a deviation value chosen beforehand. If producing x is the least expensive strategy, I decrease d and recalculate the cost of the two extreme strategies. If instead, for example, the less expensive strategy consists in producing $x-d$, we set $x = x-d$ and continue the algorithm without changing the value of d. When d fall below a certain tolerance, we stop the routine. The best production found is now compared with the alternative of no produce at all and the best strategy is selected.

Besides the fact that this approach can be used when demand is continuous or when we are not able to determine a density, the advantage of the algorithm is also at the level of calculation time. The time spent building the model depends on how many points we use to interpolate the spline. The smaller the number of points, the shorter the time used, at a cost, however, of less precision. The simulation time, however, depends directly only on the sample size used to calculate the expected values. Obviously exists, even if hidden, the dependence from the maximum available inventory, since the

	Build Time			
	$I_{\max} = 150$	$I_{\max} = 400$	$I_{\max} = 800$	$I_{\max} = 1500$
Exact	5.09	41.89	190.80	792.14
Appr_5	1.52	13.52	67.00	335.55
Appr_{50}	0.19	1.45	7.17	34.17
(s,S)	1.39	5.46	6, 42	8.40
Cont_{50}	0.19	1.32	6.81	33.12

Table 3.3: Average time in building the model for different methods and different instance dimensions.

proposed algorithm of minimization uses more iterations to converge if the range of possible values of the demand is greater. In general, however, by changing the tolerance and making it proportional to the maximum inventory level, the computing time can be reduced, but at a cost of less precision. Let's remember instead that, in the case of the algorithm based on the dynamic programming approximated in the discrete case, a reduction in the building time of the model, always corresponds to an increase in the simulation time. Unlike the continuous case we cannot decide to resort to a greater level of approximation without incurring an increase of the simulation time.

As we can see from the Table 3.3, for all dynamic programming algorithms, as we expected, the time of building the model has a quadratic trend with respect to the maximum value of inventory. The advantage of the approximate algorithms, however, is that is possible too choose a greater approximation factor to reduce this computing time. Distinctly more efficient is the search for the pair values s and S, which have an about linear dependence on the size of the instance. The (s, S) policy is the best even when it comes to simulating the model: in fact the application of the policy does not depend in any way on the inventory values and has therefore constant time, as it is possible to see in Table 3.4. Theoretically, even the exact dynamic programming algorithm has this property, but the calculation time increases because there are much heavier data structures and therefore there is a slowdown that depends on the access to the data and the passage of such data to the methods. Although in a small way, this slowdown is present also in the algorithms of approximate dynamic programming, where moreover the simulation time has a dependence on the factor k, in fact, as previously specified, the k values present between two entries of the approximated action table, are checked one by one.

Finally, in Table 3.5, we show, in case the inventory available is very high, how the algorithms of approximated dynamic programming in discrete and

	Simulation Time			
	$I_{\max} = 150$	$I_{\max} = 400$	$I_{\max} = 800$	$I_{\max} = 1500$
Exact	0.51	1.12	1.81	2.93
Appr_5	3.42	3.84	6.14	8.99
Appr_{50}	24.64	29.13	41.56	53.01
(s,S)	0.12	0.12	0.12	0.12
Cont_{50}	23.09	29.60	38.10	39.85

Table 3.4: Average time in simulating the model for different methods and different instance dimensions.

	Gap (%)	Build Time (s)	Simulation Time (s)
Exact	–	1592	4
Appr_{50}	0.000	69	46
Appr_{100}	0.000	36	90
Cont_{50}	0.038	65	40
Cont_{100}	0.036	33	38
(s,S)	0.018	10	0

Table 3.5: Average results on 5 instances with $I_{\max} = 2000$.

continuous change when the approximation index is doubled. In this case the percentage errors are also reported, to show how a higher approximation index does not degrade the quality of the algorithm too much.

In the next section, we present an alternative to dynamic programming to solve the stochastic LSP. A reinforcement learning approach based on Q factors is considered. The results obtained from this approach are not comparable with those of dynamic programming, neither from a quality point of view, nor from a speed point of view. However we wanted to present them because they could be used as a starting point for future work. At the end of the section we also propose an idea to solve the problem of computational heaviness of the algorithm.

3.3 Q-factors

The application of the DP principle may require a too expensive effort from the computational and model point of view. Sometimes an appropriate reformulation can be adopted to exchange expectation and optimization, avoiding the resolution of a difficult stochastic optimization problem. This situation occurs in a Q-learning context, a form of reinforcement learning where the state value function $V(s)$ is replaced by Q-factors $Q(s, a)$ representing the

value of state-action pairs. A Q-factor $Q(i,a)$ measures the value of taking action a when in state i.

Let's consider the case in which we have to deal with an infinite horizon DP problem. The functional equation is

$$V(s) = \min_{x \in \mathcal{X}(s)} \Big\{ f(s,x) + \gamma \mathbb{E}\big[V\big(g(s,x,\xi)\big)\big] \Big\},$$

or, if the immediate contribution is itself stochastic

$$V(s) = \min_{x \in \mathcal{X}(s)} \Big\{ \mathbb{E}\big[h(s,x,\xi) + \gamma V\big(g(s,x,\xi)\big)\big] \Big\}.$$

If the decision spaces and the state spaces are discrete it is possible to use a Markov decision process (MDP) approach. After changing a little bit the notation we can represent dynamics by a matrix of transition probabilities between pair of states. Probabilities are influenced by our decisions (called actions), so we have to use a three dimensional array to store the probabilities of a transition from state i to state j after choosing action a, $\pi(i,a,j)$. In this case the expectation boils down in a sum and we can write the functional equation like

$$V(i) = \min_{a \in \mathcal{A}(i)} \sum_{j \in S} \pi(i,a,j) \Big\{ h(i,a,j) + \gamma V(j) \Big\}.$$

If we name

$$Q(i,a) = \sum_{j \in S} \pi(i,a,j) \Big\{ h(i,a,j) + \gamma V(j) \Big\}, \tag{3.10}$$

we may observe that

$$V(j) = \min_{a \in \mathcal{A}(j)} Q(j,a). \tag{3.11}$$

Hence, plugging Eq. (3.11) into Eq. (3.10) it is possible rewrite the DP recursion in terms of Q-factors:

$$Q(i,a) = \sum_{j \in S} \pi(i,a,j) \Big\{ h(i,a,j) + \gamma \min_{a \in \mathcal{A}(j)} Q(j,a) \Big\}.$$

In this way we have swapped expectation and optimization: we have to solve many deterministic optimization problems instead than a single stochastic one. On the other hand instead of a state value function $V(i)$, now we have state-action value function $Q(i,a)$. If the size of the problem becomes large, we may be in trouble. Of course, in general we do not know the probabilities of transition, or in any case it might be impractical to use

them. The strength of this approach, however, is that we may learn the Q-factors by statistical sampling.

Imagine that at iteration k we are about to choose action $a^{(k)}$ and we have a current set of estimates of Q-factors $\widehat{Q}^{(k-1)}(i,a)$. Now, at state $s^{(k)}=i$ we select the next action by considering what looks best:

$$a^{(k)} \in \arg\min_{a\in\mathcal{A}(i)} \widehat{Q}^{(k-1)}(i,a).$$

After applying the selected action we observe the next state $s^{(k+1)}=j$ and the immediate contribution $h(i,a^{(k)},j)$. Then, we can use this information to update the estimate of $Q(s^{(k)},a^{(k)})$. Using the new observation obtained we have to make a trade off between the short term objective and the long term objective when we are at state $s^{(k)}$ and select action $a^{(k)}$. If we call

$$\widetilde{q} = h(i,a^{(k)},s^{(k+1)}) + \gamma \min_{a'\in\mathcal{A}(s^{(k+1)})} \widehat{Q}^{(k-1)}(s^{(k+1)},a'),$$

the update is the following:

$$\widehat{Q}^{(k)}(s^{(k)},a^{(k)}) = \alpha\widetilde{q} + (1-\alpha)Q^{(k-1)}(s^{(k)},a^{(k)}).$$

A similar procedure can be built in a time dependent environment. In this case Q is a three dimension matrix:

$$Q_t(i,a) = \sum_{j\in S} \pi_t(i,a,j)\Big\{h(i,a,j) + \min_{a\in\mathcal{A}(j)} Q_{t+1}(j,a)\Big\},$$

Adapting this procedure to our case is quite immediate. In the pseudocode of the algorithm it is sufficient to replace the generic state variable s with the inventory I and the generic immediate cost function h with the sum of production cost, set up cost, inventory on hand cost and lost sales cost.

The dynamic programming algorithm has a running time that is $O(TI^2)$. This algorithm instead requires nT iterations. If n were small, the algorithm would certainly be preferable to dynamic programming, but unfortunately it is not so. In fact Q is an object with $O(TI^2)$ elements, so only to initialize it you need TI^2 operations (however obviously they can be vectorized). The problem arises from the fact that Q have to be estimated, so $Q_t(s,a)$ needs to be touched several times, for each possible pair. For this n must be chosen as $\bar{n}\frac{I^2}{2}$, with $\bar{n}$ at least $100,1000$.

So as much as using methods of reinforcement learning greatly simplifies the procedures, for such an easy problem, in which you have only one risk

Qfactors - Finite time horizon

input: α, n
output: Q
sample the demand values d_{kt}, $t = 1, \dots, T$, $n = 1, \dots, n$
initialize $Q_t^{(0)}(s, a)$, $t = 0, \dots, T$, $s = 0, \dots, I$, $a = 0, \dots, I - s$. Set $k = 1$
for k in $[1, \dots, n]$ **do**
    sample $s_0^{(k)}$
    for t in $[0, \dots, T-1]$ **do**
        $a_t^{(k)} \in \arg\min_{a \in \mathcal{A}(s_t^{(k)})} Q_t^{(k-1)}(s_t^{(k)}, a)$
        $s_{t+1}^{(k)} = \max\{0, s_t^{(k)} + a_t^{(k)} - d_{t+1}\}$
        $q = h(s_t^{(k)}, a_t^{(k)}, s_{t+1}^{(k)}) + \min_{a' \in \mathcal{A}(s_{t+1}^{(k)})} Q_{t+1}^{(k-1)}(s_{t+1}^{(k)}, a')$,
        $Q_t^{(k)}(s_t^{(k)}, a_t^{(k)}) = \alpha q + (1 - \alpha) Q_t^{(k-1)}(s_t^{(k)}, a_t^{(k)})$
    end for
end for

factor, it seems to be more convenient an exact approach rather than a simulation.

In case the application has small support, we did some tests to effectively understand the feasibility of the algorithm, but the results, in line with our expectations, showed not bearable calculation times. In Table 3.6 we show the result of this procedure. After some test α was fixed to 0.02 while we change I and $\bar{n}$ to see the effects. Gap is the percentage error in evaluating the system with respect to the DP result. The second column is ratio between the execution time of the procedure and the execution time of the exact DP algorithm. A number 100 means that the algorithm based on Q factor is 100 times slower than the DP one. The last column is a measure of the strategy risk. The semi IC ratio is calculate as follow

$$\frac{Q_{95} - \overline{DP}}{DP_{95} - \overline{DP}},$$

where Q_{95} is the 0.95 quantile of Qfactors algorithm results, while DP_{95} is the 0.95 quantile of DP algorithm results and $\overline{DP}$ is the average value of DP algorithm results. We can notice that when we use $\bar{n} = 1000$, although the performance is lower than the DP-based algorithm, the expected cost distribution is closer to the average value.

In Table 3.7 we show results found with a slightly different version of the algorithm. In this case we use a starting value for α which is bigger

$T=6,\quad \alpha=0.02$	Gap (%)	Time Ratio	Semi IC Ratio (%)
$I=4,\quad \bar{n}=300$	2.32	233	101.4
$I=4,\quad \bar{n}=1000$	0.58	800	97.5
$I=10,\quad \bar{n}=300$	3.71	374	105.0
$I=10,\quad \bar{n}=1000$	3.03	1266	94.6
$I=20,\quad \bar{n}=300$	3.03	433	100.8
$I=20,\quad \bar{n}=1000$	1.71	1422	88.4
$I=50,\quad \bar{n}=300$	2.60	511	104.7
$I=50,\quad \bar{n}=1000$	1.58	1764	89.5

Table 3.6: Result of base algorithm: α is fixed, $toll = 0$.

with respect of before, but it is decreased along the iterations. Moreover a tolerance parameter is used: at iteration k when we ave to choose $a_t^{(k)}$ we sampling a random number from a uniform distribution: if it is smaller than the tolerance we select $a_t^{(k)}$ at random from the possible value $a \in \mathcal{A}(s_t^{(k)})$. In terms of performance the two algorithms are quite similar, but the second is much faster than the first. However, there is no longer any narrowing of the confidence interval.

To reduce the enormous computational effort that this method expect, what is possible to do is using a linear regression to estimate the Q-factors. The main idea is the following: consider a set of state-action pairs and for each of these initialize the Q-factors. Now we use a regression to express Q-factors by a linear combination of function depending by states and actions. Using regression coefficient when we sampling new state-action pairs we can update the Q-factors. From the new Q-factors is possible to find new coefficients and go on like this.

Imagine we are a iteration k. The current state-action pair are $(s_j^{(k)}, a_j^{(k)})$, $j = 1, \ldots, n$. while the estimates of the Qfactors are $\widehat{Q}^{(k-1)}(s_j^{(k)}, a_j^{(k)}) = 0$, $j = 1, \ldots, n$. Is possible to impose the following linear regression

$$\begin{pmatrix} \widehat{Q}^{(k-1)}(s_1^{(k)}, a_1^{(k)}) \\ \widehat{Q}^{(k-1)}(s_2^{(k)}, a_2^{(k)}) \\ \cdots \\ \widehat{Q}^{(k-1)}(s_n^{(k)}, a_n^{(k)}) \end{pmatrix} = \begin{pmatrix} 1 & s_1^{(k)} & a_1^{(k)} \\ 1 & s_2^{(k)} & a_2^{(k)} \\ \cdots & \cdots & \cdots \\ 1 & s_n^{(k)} & a_n^{(k)} \end{pmatrix} \begin{pmatrix} \beta_0^{(k)} \\ \beta_1^{(k)} \\ \beta_2^{(k)} \end{pmatrix} + \begin{pmatrix} \epsilon_1^{(k)} \\ \epsilon_2^{(k)} \\ \cdots \\ \epsilon_n^{(k)} \end{pmatrix}$$

and find β via least square

$$\min_{\beta} \sum_{j=1}^{n} \left[\widehat{Q}^{(k-1)}(s_j^{(k)}, a_j^{(k)}) - \left(\beta_0^{(k)} + \beta_1^{(k)} s_j^{(k)} + \beta_2^{(k)} a_j^{(k)} \right) \right]^2 .$$

$T=6, \quad \alpha=0.02$			Gap (%)	Time Ratio	Semi IC Ratio (%)
$I=4,$	$\bar{n}=75$	$toll=0.50$	3.06	58	104.9
$I=4,$	$\bar{n}=75$	$toll=0.75$	0.88	49	97.2
$I=4,$	$\bar{n}=100$	$toll=0.50$	3.01	67	105.2
$I=4,$	$\bar{n}=100$	$toll=0.75$	0.49	67	106.0
$I=10,$	$\bar{n}=75$	$toll=0.50$	3.11	91	101.38
$I=10,$	$\bar{n}=75$	$toll=0.75$	3.20	80	104.2
$I=10,$	$\bar{n}=100$	$toll=0.50$	1.37	121	98.9
$I=10,$	$\bar{n}=100$	$toll=0.75$	1.27	106	105.8
$I=20,$	$\bar{n}=75$	$toll=0.50$	2.70	103	108.8
$I=20,$	$\bar{n}=75$	$toll=0.75$	1.94	95	100.8
$I=20,$	$\bar{n}=100$	$toll=0.50$	1.26	145	100.3
$I=20,$	$\bar{n}=100$	$toll=0.75$	1.71	124	98.0
$I=50,$	$\bar{n}=75$	$toll=0.50$	2.05	127	102.4
$I=50,$	$\bar{n}=75$	$toll=0.75$	4.57	115	112.1
$I=50,$	$\bar{n}=100$	$toll=0.50$	1.28	169	100.9
$I=50,$	$\bar{n}=100$	$toll=0.75$	1.86	156	102.4

Table 3.7: Result of modified algorithm: α decrease along the iteration.

In general is possible to use different functions of s and a and write

$$\min_{\beta_l} \sum_{j=1}^{n} \left[\widehat{Q}^{(k-1)}(s_j^{(k)}, a_j^{(k)}) - \Big(\sum_{l=1}^{m} \beta_l^{(k)} \phi_l(s_j^{(k)}, a_j^{(k)}) \Big) \right]^2$$

Now we generate new pairs $(s_j^{(k+1)}, a_j^{(k+1)})$, $j = 1, \ldots, n$. $s_j^{(k)}$ is generate from $s_j^{(k)}$ and $a_j^{(k)}$ using the state equation. Now we need to update Q-factors. The formula was

$$\begin{aligned} \widehat{Q}(s^{(k)}, a^{(k)}) = \alpha \Big[h(s^{(k)}, a^{(k)}, s^{(k+1)}) + \gamma \min_{a' \in \mathcal{A}(s^{(k+1)})} \widehat{Q}^{(k-1)}(s^{(k+1)}, a') \Big] \\ + (1-\alpha) Q^{(k-1)}(s^{(k)}, a^{(k)}) \end{aligned}$$

but this time we do not know compute the minimum. We could try all possible actions, but in this way we need to solve an optimization problem with requires time (we have a gain only from a memory point of view, in fact we save only β parameters). In this case the update is

$$\begin{aligned} \widehat{Q}(s_j^{(k)}, a_j^{(k)}) = \alpha \Big[h(s_j^{(k)}, a_j^{(k)}, s_j^{(k+1)}) + \gamma \min_{a' \in \mathcal{A}(s^{(k+1)})} \sum_{l=1}^{m} \beta_l^{(k)} \phi_l(s_j^{(k+1)}, a') \Big] + \\ (1-\alpha) \sum_{l=1}^{m} \beta_l^{(k)} \phi_l(s_j^{(k)}, a_j^{(k)}) \end{aligned}$$

What we can do is sample the a values or use the sub gradient method.

Qfactor regression

input: ϕ
output: ϕ, β
sample $(s_j^{(0)}, a_j^{(0)})$ and initialize $Q^{(k)}(s_j^{(0)}, a_j^{(0)}) = 0$, $j = 1, \ldots, n$. Set $k = 1$
for k in $[1, \ldots, K]$ **do**
    $\beta^{(k)} = \arg\min_\beta \sum_{j=1}^n \left[\widehat{Q}^{(k-1)}(s_j^{(k)}, a_j^{(k)}) - \left(\sum_{l=1}^m \beta_l \phi_l(s_j^{(k)}, a_j^{(k)}) \right) \right]^2$
    Sample $s_j^{(k+1)}$, $j = 1, \ldots, n$
    for j in $[0, \ldots n]$ **do**
        $a_j^{(k+1)} = \arg\min_{a \in \mathcal{A}(s^{(k+1)})} \sum_{l=1}^m \beta_l^{(k)} \phi_l(s_j^{(k+1)}, a)$
        $\widehat{Q}(s_j^{(k)}, a_j^{(k)}) = \alpha \Big[h(s_j^{(k)}, a_j^{(k)}, s_j^{(k+1)}) + \gamma \sum_{l=1}^m \beta_l^{(k)} \phi_l(s_j^{(k+1)}, a_j^{(k+1)}) \Big] + (1 - \alpha) \sum_{l=1}^m \beta_l^{(k)} \phi_l(s_j^{(k)}, a_j^{(k)})$
    end for
end for

3.4 Numerical Result

In this section we show the results obtained with the different dynamic programming algorithms. First, we focus on how to generate demand. Unlike in the deterministic case in this case, the form of demand may affect the stability of the different methods. In particular, if the demand is normal, for example, the risk of being on the tails is low and therefore the expected values can be estimated without too many observations. We have tested in three different cases. In the first the distribution of demand is given through a density vector. In this case the demand is discrete and the supports is on the integer numbers. In the second we used a normal truncated distribution, while in the third we used a gamma distribution. Despite the continuously approximated dynamic programming algorithm works when the demand is distributed continuously, in order to compare it with the other approaches, the demand is always discretized on integer values in the tests we make.

The tests we show are divided into two parts. The first part is based on instances of medium size. In these instances the exact dynamic programming procedure is compared with two different discrete approximate dynamic pro-

	Gap (%)	Time (s)	90%$I.C.$
Exact	–	12.9	–
Appr_5	−0.004	4.0	[−0.026, 0.013]
Appr_{50}	0.205	0.5	[0.022, 0.333]
Cont_{50}	0.334	0.4	[0.022, 0.718]
(s,S)	0.515	1.9	[0.242, 0.967]

Table 3.8: Average results on 10 instances with $I_{\max} = 250$ and normal demand.

gramming algorithms with approximation factors $k = 5$ and $k = 50$, with the continuous approximate dynamic programming with $k = 50$ and with the (s, S) policy. In the second part of the tests are instead used instances in which the maximum level of inventory is large. In this case the exact dynamic programming is too expensive, so the (s, S) policy is used as a benchmark and both methods of approximated DP, discrete and continuous, are tested with values of $k = 50$ and $k = 100$. As in the deterministic case, inventory, production, setup and lost sales costs are uniformly distributed between a minimum and a maximum value. All tests have a time horizon of 15 periods and the proposed heuristic to determine the end-of inventory value is used.

For the first part of the tests we use the following values: $h \in [1, 10]$, $c \in [1, 15]$, $p \in [16, 30]$, $f \in [300, 500]$, while the maximum level of inventory is $I_{\max} = 250$. We generated 10 instances and 1000 scenarios are used to estimate the cost of each strategy. Average costs and a 90% confidence interval are collected. The results shown in Table 3.8 refer to the case where demand is normal, while those in Table 3.9 refer to the case in which demand has a gamma distribution . In both cases there the mean of the demand is 100 and its standard deviation is 30.

From Table 3.8, we observe that in general, all approximate methods have really amazing results. The approximations with a factor of $k = 50$, manage to save more than 95% of running time compared to the exact method and to obtain results that differ in average of about 0.3%. A clarification is needed to explain the results of the second row. In fact, it would seem that the approximate algorithm performs better than the exact one: in reality this is not really true, but since the policies generated by the two algorithms are very similar, approximate algorithm performance may be better on the limited number of scenarios generated to simulate the results (only 1000). Even in the case of demand distributed as a gamma, the results are satisfactory. It is not shown in the table, but it must be said that in this case, there are greater deviations from the expected cost, that is the one obtained from the

	Gap (%)	Time (s)	$90\% I.C.$
Exact	–	13.4	–
Appr_5	0.020	4.1	[0.002, 0.049]
Appr_{50}	0.384	0.5	[0.165, 0.604]
Cont_{50}	0.253	0.4	[−0.156, 0.799]
(s,S)	0.617	2.9	[0.132, 0.776]

Table 3.9: Average results on 10 instances with $I_{\max} = 250$ and gamma demand.

table value of the exact dynamic programming. This is certainly due to the fact that the gamma distribution has heavier tails than normal and therefore a sample of 1000 observations is not the most suitable method to estimate the average value. It should not surprise us that there are cases in which the approximated continuous DP has better performances than the exact programming. Although the method is approximate, it has the flexibility to choose production values in continuous, while all other methods can only use integer production values.

For the second part of the tests we maintain the same value for all the cost parameters except for the setup cost which are distributed between 1500 and 2000. Now the maximum inventory value is set to 2000, while the demand has a mean of 1000 and a standard deviation of 300. With such a large inventory level, the exact DP algorithm, takes about 30 minutes to build the model. For this reason it was decided to use the (s, S) policy as a benchmark, where s and S are estimated with the cross entropy method using more observations than normal. Obviously in this case it is possible that the algorithms of approximate DP perform better than the benchmark. Some might argue that without implementing the exact DP, we don't have a true measurement of the error, but we did some tests (with only 5 repetitions), respectively with an inventory level of 800, 1500, 2000 and we have obtained that the performances of the approximate discrete algorithm are identical to those of the exact DP algorithm. This makes us believe that in reality, having set the degree of approximation to k, the algorithm of approximated discrete DP has a smaller relative error when the maximum inventory level is larger.

As the results in Table 3.10 show, the discrete DP approximation procedure perform better than the benchmark. In general, anyway, all the proposed procedures perform very well, but none can compete at the level of computing time with the (s, S) policy. However in the next section we show, even if only with some examples, how the performances of this policy are degraded when we consider the capacitated problem. Also in Table 3.11, we

	Gap (%)	Time (s)	90%$I.C.$
Benchmark	–	200	–
Appr_{50}	−0.040	69	[−0.228, 0.140]
Appr_{100}	−0.039	35	[0. − 228, 0.139]
Cont_{50}	0.009	66	[−0.209, 0.227]
Cont_{100}	0.012	33	[−0.207, 0.227]
(s,S)	0.174	9	[−0.115, 0.456]

Table 3.10: Average results on 10 instances with $I_{\max} = 2000$ and normal demand.

	Gap (%)	Time (s)	90%$I.C.$
Benchmark	–	668	–
Appr_{50}	−0.097	90	[−0.480, 0.460]
Appr_{100}	−0.079	46	[−0.490, 0.460]
Cont_{50}	0.069	87	[−0.480, 0.760]
Cont_{100}	0.048	44	[−0.520, 0.690]
(s,S)	0.098	14	[−0.360, 0.790]

Table 3.11: Average results on 10 instances with $I_{\max} = 2000$ and gamma demand.

can see that all methods have similar performance. Even if not underlined in the table, as told before, the main advantage of the continuous approximation is that the simulation time remains constant and the higher the approximation factor k, the more the advantage with respect to the discrete approximation is sensitive.

3.5 Stochastic Problem for Single Item Capacitated Lot Sizing

In this part of the work, we use the concepts seen in the previous sections to solve the problem of lot sizing subject to a capacity constraint. As done in the deterministic case our goal is to dualize the capacity constraint, solve many capacitated problems decoupled and finally, thanks to a gradient algorithm adjust the multipliers until converge to a good feasible solution. Actually, in this case there is nothing to decouple, since we consider CLSP on the individual product. On the single item the capacity constraint in the deterministic case takes the form

$$r\, x_t + r'\, \delta_t \leq R_t, \quad t = 0, \ldots, T-1,$$

but obviously written like this, the constraint is meaningless in the stochastic version of the problem. It should be remembered that x_t depends on the state of the system in which we are, or in our case, on the state of the inventory at the time t. If we rewrite the constraint like

$$r\, x_t^s + r'\, \delta_t^s \leq R_t, \quad t = 0, \ldots, T-1,\ s = 0, \ldots, I_{\max},$$

we understand that the Lagrangian multipliers must depend both on time and on inventory level. We can remove the constraint and plug it in the objective function, i.e., in a DP approach we can insert it in the recursion equation, that become

$$V_t(I_t) = \min_{x_t \in \mathcal{X}_t(I_t)} \mathbb{E}\Big[c\, x_t + f\delta_t + h\, I_{t+1} + pI_{t+1}^- + \mu_t^{I_t}\big(r\, x_t + r'\delta_t - R_t\big) + V_{t+1}(I_{t+1})\Big]$$

or equivalently

$$V_t(I_t) = \min_{x_t \in \mathcal{X}_t(I_t)} \mathbb{E}\Big[(c + \mu_t^{I_t}\, r)x_t + (f + \mu_t^{I_t}\, r')\delta_t + h\, I_{t+1} + pI_{t+1}^- - \mu_t^{I_t}\, R_t + V_{t+1}(I_{t+1})\Big]$$

and the update rule for the Lagrangian multipliers is

$$\mu_t^s = \mu_t^s + \alpha\big(r\, x_t^s + r'\delta_t^s - R_t\big), \quad t = 0, \ldots, T-1,\ s = 0, \ldots, I_{\max}, \quad (3.12)$$

where we omit the iteration index for readability.

For the capacitated problem we decided to focus only on discrete DP. In fact in the case in which it is decided to use a continuous approximation, we would have in some way to build the multipliers like a function of the inventory on hand. As far as this is feasible, we preferred to remain grounded and analyze the immediate complications that the capacity constraint entails. We note that so far we have always used models in which the penalty for lost sales and for items in stock is piecewise linear. This implies that if production costs are higher than those of lost sales, it is always optimal not to produce. In itself this is not a problem, in fact, for the model to make sense, the relation $c < p$ must apply. But when we are going to use a procedure with Lagrange multipliers, the relationship becomes $c + \mu r < p$, which is obviously not verified when μ grows.

To understand why this is a problem, let's suppose to have a solution that exceeds the capacity constraint. The multiplier μ is increased and it happens that $c + \mu r > p$ Then the solution of the next iteration will correspond to never producing, even if actually the constraint would allow to produce.

To avoid this inconvenience, which often does not allow to find the right production plan, we therefore decided to use quadratic penalties, both for lost sales and for holding costs.

The procedure we propose is to start with a matrix of multipliers all null, to find the value and the action table and update the multipliers according to the Eq. (3.12). We choose a step lenght of type

$$\alpha^{(k)} = \frac{a}{b + ck},$$

where a depend on the average demand and the production cost, while $b = 1$ and $c = 0.3$. After comparing different stop techniques we decide opt for a convergence on multipliers of type

$$\frac{||\boldsymbol{\mu}^{(k-1)} - \boldsymbol{\mu}^{(k)}||_\infty}{||\boldsymbol{\mu}^{(k-1)}||_\infty} < b_\mu.$$

Note that in a certain sense the procedure is easier to build with respect to the deterministic case. In fact now we are not forced to meet the demand. In particular, although the policy that we find not always guarantee that we meet the constraint, we can always produce less at any time we fail to meet the constraint of capacity. In practice the policy found can always be adjusted as needed. In particular, what is done during the simulation is, after implementing the optimal choice, to check the capacity constraint. If it is not respected, production will decrease until it is.

While in the uncapacitated problem we are certain that the exact DP approach, at least in the case where the probability distribution is given in discrete, is the best viable policy, here we no longer have this guarantee. In the deterministic case we implement the model and solve it by Gurobi's solver in order to have a comparison of the goodness of our results, but here it is not so trivial to do the same thing. What we do, however, is implement our algorithm and compare it with the (s, S) policy. This policy, which is optimal when the capacity constraint is not present, can still be used, but it does not give us guarantees of good results. In the uncapacitated case the results between our algorithms and the (s, S) policy are very similar. What we expect now is that our performance is considerably better. In this case to implement the (s, S) policy what we do is look for values for s and S such that $s \leq S \leq P_{\max}$, where $P_{\max}$ is the maximum value that is possible to produce without exceeding the capacity constraint.

To test the algorithms in a first moment we used a normal (discretized) demand with mean 70 and standard deviation equal to 20. The maximum

	Gap (%)	Time (s)	90%$I.C.$
Exact	–	46.6	–
Appr_5	0.811	5.8	[−0.56, 2.54]
Appr_{50}	0.998	0.5	[−4.70, 8.62]
(s,S)	29.573	0.8	[−0.70, 91.80]

Table 3.12: Average results on 30 instances with $I_{\max} = 150$.

	Gap (%)	Time (s)	90%$I.C.$
Exact	–	269	–
Appr_5	−0.099	84	[−0.93, 0.32]
Appr_{50}	2.451	5	[−3.77, 10.55]
(s,S)	64.5	2	[22.28, 136.05]

Table 3.13: Average results on 30 instances with $I_{\max} = 500$.

inventory level is $I_{\max} = 150$. All the cost and time parameters are uniformly distributed between the following values: $c \in [1, 15]$, $h \in [1, 10]$, $p \in [16, 30]$, $f \in [75, 200]$, $r \in [1, 3]$, $r' \in [130, 190]$, $R_t \in [100, 150]$. Both approximate DP algorithms seem to have good performance, but we can see how increasing the degree of approximation also increases the variability of the results. The approximate algorithm, in this case, has a double advantage regarding the execution time, compared to the exact one. In fact on the one hand we have to build smaller tables, so fewer operations are done, on the other we also have to use less multipliers, and the convergence is faster. The (s, S) policy, instead, despite remaining the fastest, has very poor performances. This is due to the fact that when the capacity constraint severely limits production, it is optimal to produce even when we have so much in stock, to be ready when there will be a peak of demand and this is an aspect that (s, S) policy cannot take into account.

For the second part of tests, given the long computation time required to find a solution when a larger maximum inventory level is used, we decided to decrease the time horizon from $T = 15$ to $T = 5$. Also this time we use a truncated normal distribution for the demand but with mean 250 and standard deviation 60. The maximum inventory level is $I_{\max} = 500$. All the cost and time parameters are uniformly distributed between the following values: $c \in [1, 15]$, $h \in [1, 10]$, $p \in [16, 30]$, $f \in [150, 400]$, $r \in [1, 3]$, $r' \in [375, 550]$, $R_t \in [375, 550]$. The results are more or less in line with the case where the maximum inventory is lower. The highest variability of the performances, in our opinion, is due to the fact that exact DP does not always

	Gap (%)	Time (s)	90%$I.C.$
Exact	–	269	–
Appr_5	0.467	49	[0.00, 2.45]
Appr_{50}	0.830	2	[0.00, 4.12]
(s,S)	44.9	3	[1.65, 96.10]

Table 3.14: Average results on 30 instances with $I_{\max} = 500$ and low setup times.

find the best solution, and this can have a different effect on the approximate algorithms. One group of instances where things work very well is when the setup times are very low. As we can see in Table 3.14, in fact, in this case the variation of performance of the approximated algorithms and their execution times are lower.

Chapter 4

Conclusion

In conclusion our work proposes different resolution techniques for LSP. In the deterministic case, our approach has been found to be clearly superior to that of the commercial software, above all in the more interesting cases in literature, that is when the number of items is large and the ratio between setup costs and inventory costs of increases. A strong point of our work is to have developed several procedures that can be applied to a wide range of instances and can be used both when we want to get an accurate result, and when we want to have answers in a short time, even if with less precision. In addition, after numerous tests, we we found an optimal choice of values for the problem parameters regardless of the size of the instance or the costs involved. This is obviously a huge advantage, as there is never a need to recalibrate the algorithm. Despite this, we are convinced that is possible to achieve better results, in particular it is possible to work on the implementation of the WW algorithm to try to get further improvements on the computing time. Another improvement that we can make to the algorithm is to build a model in which we can start with a non-empty inventory on hand. This choice, which could be very useful from a practical point of view, was not made from the beginning because for historical reasons, lot sizing models use an empty inventory at the beginning of the time horizon.

As far as the stochastic problem is concerned, our objective has been less ambitious. Although we can be satisfied with the results obtained, the second part of the thesis in fact only lays the basis for the solution of the CLSP on multiple items. In the single item ULSP, the algorithms of approximated DP have brought excellent results, obtaining in fact the same performances of the exact DP, but with much shorter times. In this simple case, however, the (s, S) policy appears to be the best alternative to use. The strength of

our algorithms, however, is not to deteriorate when a capacity constraint is considered. Although the stochastic capacitated problem is dealt with briefly, we can observe how in this case the (s, S) policy has costs on average 50% higher than those found by our approximate DP algorithms. The effort that must still be made is to find more computationally advantageous approximations to be able to deal with problems with multiple products.

Appendix A

Python Code for Deterministic CLSP

A summary of all classes and methods used is given.

DetLotSizing

- *solve*(dict_data:*Dict*, mu:*numpy.array*, time_limit:*float*, gap:*float*, verbose:*bool*) -> (solution:*Solution*, comp_time:*float*)

  This method solves the CLSP via MILP. Gurobi is used to build the model and to solve it. If mu is given it solves a model in which the capacity constrained has been relaxed.

- *solve_wagner_whitin* (dict_data:*Dict*, mu:*numpy.array*) -> (solution:*Solution*, comp_time:*float*)

  This method solves the ULSP via DP. If *mu* is given it solves the CLPS for which the capacity constraint has been relaxed. What this method does is solve at the same time I different problems, one for each product. Then production, set up periods and inventory are put together to build the solution. For each product the method *wagner_whitin* is invoked.

- *wagner_whitin* (set_up_cost:*float*, inventory_cost:*float*, demand:*numpy.array*, set_up_time:*float*, processing_time:*float*, mu:*numpy.array*) -> (production:*numpy.array*, set_up:*numpy.array*, inventory:*numpy.array*)

  This method solves the ULSP via DP. If *mu* is given it solves the CLPS for which the capacity constraint has been relaxed. What this method

does is solve at the same time I different problems, one for each product. Then production, setup periods and inventory are put together to build the solution. For each product the step to perform are the following:

  - A matrix $T \times T$ is build. The element ij contains the cost of produce at time i to cover demand until time j. Only the lower half is filled.
  - Two matrix $T \times T$ is build. The first is a matrix of *float* which contains the value function. Element ij is the best value when we consider the cost of production from tme i to time j. The second is a matrix of *List* which contains as element ij a *List* of one or two elements. If only one element is present it means that the best policy when we consider the cost of production from time i to time j is produce at time i only the demand for that period. If two elements are present it means that the best policy when we consider the cost of production from time i to time j is produce at time i to satisfy demand until time k, where k is the second element of the list. For both matrices only the upper half is filled.
  - From the first matrix the best overall cost is extracted. From the second matrix the optimal ordering policy is found.
  - Setup times, production quantity and holding inventory are found.

- `solve_cross_entropy (dict_data:Dict, mu:numpy.array) -> (solution:Solution, comp_time:float)`

  This method find a solution to the ULSP via a Cross Entropy procedure. If mu is given it find a solution to the CLPS for which the capacity constraint has been relaxed. What this method does is solve at the same time I different problems, one for each product. Then production, setup periods and inventory are put together to build the solution. This method do not guarantee to find the optimal solution, but using an importance sampling Monte Carlo it is looking for a quite good solution. For each product an iterative procedure is performed:

  - From a vector of T Bernoulli with given parameter p, n different observation are generated. Each vector represents one possible solution, in fact any solution can be represented from its binary variables.
  - The candidates with best results are used to generate a new parameter p.

  - The algorithm stops if p is a vector of 0/1 values or if the maximum number of iteration is reached.

- *sub_gradient* (dict_data:*Dict*, mu:*numpy.array*, solution:*Solution*, alpha:*float*, diff_obj:*float*) -> (mu:*numpy.array*, grad:*numpy.array*)

  This method perform the subgradient technique explain in Section 2.2 to update the Lagrangian multipliers.

- *bundle_sub_gradient*(dict_data:*Dict*, mu:*numpy.array*, obj:*float*, gradients_bundle:*numpy.array*, dual_values:*numpy.array*, solution:*Solution*, alpha:*float*, diff_obj:*float*, time_limit:*float*, gap:*float*, verbose:*bool*) -> (mu:*numpy.array*, grad:*numpy.array*)

  This method perform a different way to update lagrangian multipliers. In this case a simple optimization problem is solved to found an approximation to the sub differential, that is used to update mu. If it is not possible to find the subdifferential, the subgradient is used.

Solution

- Solution(problem:*DetLotSizing*, production:*numpy.array*, set_up:*numpy.array*, inventory:*numpy.array*, cost:*float*, mu:*numpy.array*, dual_value:*float*, feasible:*bool*)

  Constructor.

- *set_cost*(dict_data:*Dict*) -> (cost:*float*)

  This method compute the cost of the solution.

- *make_feasible_LPH*(dict_data:*Dict*, time_limit:*float*, gap:*float*, verbose:*bool*) -> (solution:*Solution*, comp_time:*float*)

  This method perform a first attempt to make the solution feasible. It fix the set up and try to modified production and inventory.

- *make_feasible_swap*(dict_data:*Dict*) -> (solution:*Solution*, comp_time: *float*)

  This method try to make a solution feasible moving production backward or forward. As first thing period in which an infeasibility is present are stored in a vector. Then they are considered one by one and an attempt is made to remove the infeasibility. The attempt consists of the following step:

- Given a product we try to anticipate the production. If this swap (backward) allows to remove the infeasibility, the procedure is stopped and next time period is considered.
- If a swap is not possible we try instead to postpone the production. If this swap (forward) allows to remove the infeasibility the procedure is stopped and next time period is considered.
- If no swap has been made considering all products we stop. We are not able to restore infeasibility. If at least one swap has been made, we restart from the first product to try to move production.

At the end if we're able to restore feasibility for each period, we found a feasible solution.

- *make_feasible*(dict_data:*Dict*, time_limit:*float*, gap:*float*, verbose:*bool*) -> (solution:*Solution*, comp_time:*float*)

  This method is always able to find a feasible solution for the problem if it exists. It use the MILP model explained in Section 2.4 and solve it via Gurobi to restore the feasibility.

- *check_feasibility*(dict_data:*Dict*) -> (feasible:*bool*)

  This method return True if the capacity constraint is satisfied.

Appendix B

Python Code for Stochastic LSP

SingleItemStochasticLotSizing

A summary of all classes and methods used is given.

- *dynamic_programming*(dict_data:*Dict*, mu:*numpy.array*) -> (value_table:*numpy.array*, action_table:*numpy.array*)

  This method finds the ValueTable and the ActionTable for the single item uncapacitated stochastic LSP in the case in which the demand has a small discrete support. If mu is given it solves the CLPS for which the capacity constraint has been relaxed.

- *dynamic_programming_simulation* (dict_data:*Dict*, num_scenarios:*int*, action_table:*numpy.array*) -> (cost:*float*)

  This method simulate the performance of a dynamic programming approach on a single item uncapacitated stochastic LSP in the case in which the demand has a small discrete support. If time capacity is given it is able to work with the single item stochastic CLSP.

- *dynamic_programming_spline* (dict_data:*Dict*, k:*int*, mu:*numpy.array*) -> (value_table:*numpy.array*, action_table:*numpy.array*)

  This method perform an approximate dynamic programming approach to finds a discretization of the value table and the action table for the single item uncapacitated stochastic LSP also in the case in which the demand has a big (but discrete) support. The k value is a sort of

"simplification factor", if the inventory capacity is I, only $(I+1)/k$ level are considered. To evaluate, when needed, the value function in the points in which it is not defined, a spline approximation is used. If mu is given it solves the CLPS for which the capacity constraint has been relaxed.

- *dynamic_programming_simulation_spline* (dict_data:*Dict*, num_scenarios:*int*, value_table:*numpy.array*), action_table:*numpy.array*, *k:int*) -> (cost:*float*)

  This method simulate the performance of an approximate dynamic programming approach on a single item uncapacitated stochastic LSP in the case in which the demand has a big (but discrete) support. In this case we have a discretization of the action table. If the state in which we are is on the action table, we use it to decide what to do. If not, we solve a small optimization problem in which we use the approximation of the value function made with the use of the splines. If time capacity is given it is able to work with the single item stochastic CLSP.

- *continuous_dynamic_programming_spline*(dict_data:*Dict*, k:*int*) -> (spline_coefficient_list:*List*)

  This method work similarly to dynamic_programming_spline, but it saves only the spline coefficients.

- *continuous_dynamic_programming_simulation_spline*(dict_data:*Dict*, num_scenarios:*int*, spline_coefficient_list:*List*) -> (cost:*float*)

  This method simulate the performance of an approximate dynamic programming approach on a single item uncapacitated stochastic LSP in the case in which the demand has a continuous support. At any time step we solve a small optimization problem in which we use the approximation of the value function made with the use of the splines.

- *finds_s_S_cross_entropy*(dict_data:*Dict*) -> (s:*int*, S:*int*)

  This method uses a cross entropy approach to find the s and S value of the (s, S) policy for the single item uncapacitated stochastic LSP in the case in which the demand has a discrete support. We use a binary representation for s, S and also for the inventory capacity.

- *s_S_simulation*(dict_data:*Dict*, num_scenarios:*int*, s:*int*, S:*int*, seed:*int*) -> (cost:*float*)

This method simulate the implementation of a (s, S) policy.

- *sub_gradient* (dict_data:*Dict*, mu:*numpy.array*, action_table:*numpy.array*, alpha:*float*) -> (mu:*numpy.array*)

  This method perform the subgradient technique to update the Lagrangian multipliers in the case of discrete support.

- *q_factors* (dict_data:*Dict*, alpha:*float*, repetition:*int*) -> (q:*List*)

  The Q-factos algorithm presented in Section 3.3 is implemented.

Bibliography

J. Arthur F. Veinott and H. M. Wagner. Computing optimal (s, s) inventory policies. *Management Science*, 11(5):493–615, 1965.

D. Bertsekas. *Dynamic Programming and Optimal Control*, volume 1, chapter 1. Athena scientific Belmont, MA, 1995.

G. Bitran and H. Yanasse. Computational complexity of the capacitated lot size problem. *Management Science*, 28, 02 1981.

J. Borwein and A. S. Lewis. *Convex analysis and nonlinear optimization: theory and examples*, pages 76–77. Springer Science & Business Media, 2010.

Z. I. Botev, D. P. Kroese, R. Y. Rubinstein, and P. L'Ecuyer. The cross-entropy method for optimization. In *Handbook of statistics*, volume 31, pages 35–59. Elsevier, 2013.

S. Boyd, L. Xiao, and A. Mutapcic. Subgradient methods. *lecture notes of EE392o, Stanford University, Autumn Quarter*, 2004:2004–2005, 2003.

P. Brandimarte. Multi-item capacitated lot-sizing with demand uncertainty. *International Journal of Production Research*, 44:2997–3022, 2006.

P. Brandimarte. *From Shortest Paths to Reinforcement Learning*. Springer, 2021.

J. Camm, A. Raturi, and S. Tsubakitani. Cutting big m down to size. *Interfaces*, 20:61–66, 1990.

M. Caserta and E. Quinonez. A cross entropy-lagrangean hybrid algorithm for the multi-item capacitated lot-sizing problem with setup times. *Computers and Operations Research*, 36:530–548, 02 2009.

C. Cervellera, A. Wen, and V. C. Chen. Neural network and regression spline value function approximations for stochastic dynamic programming. *Computers and Operations Research*, 34(1):70 – 90, 2007.

A. Costa, O. D. Jones, and D. Kroese. Convergence properties of the cross-entropy method for discrete optimization. *Operations Research Letters*, 35 (5):573 – 580, 2007.

G. B. Dantzig. *Linear programming and extensions*, volume 48. Princeton university press, 1965.

P.-T. De Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein. A tutorial on the cross-entropy method. *Annals of operations research*, 134(1):19–67, 2005.

C. Dillenberger, L. F. Escudero, A. Wollensak, and W. Zhang. On practical resource allocation for production planning and scheduling with period overlapping setups. *European Journal of Operational Research*, 75:275–286, 1994.

D.-Z. Du and K.-I. Ko. *Theory of computational complexity*, volume 58. John Wiley & Sons, 2011.

A. Federgruen and P. Zipkin. An efficient algorithm for computing optimal (s, s) policies. *Operations Research*, 32(6):1195–1384, 1984.

M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

H. Güner Gören and S. Tunali. Fix-and-optimize heuristics for capacitated lot sizing with setup carryover and backordering. *Journal of Enterprise Information Management*, 31(6):879–890, 2018.

R. B. Heady and Z. Zhu. An improved implementation of the wagner whitin algorithm. *Production and Operations Management*, 3(1):55–63, 1994.

E. Klotz and A. Newman. Practical guidelines for solving difficult mixed integer linear programs. *Surveys in Operations Research and Management Science*, 18:18–32, 2013.

C. Lemaréchal. Lagrangian relaxation. In *Computational Combinatorial Optimization*. Springer, Berlin, Heidelberg, 2013.

R. Marsten, G. Astfalk, I. Lustig, and D. Shanno. The interior-point method for linear programming. *IEEE Software*, 9:61–68, 1992.

C. Niculescu and L.-E. Persson. *Convex Functions and Their Applications*, page 133. Springer, Cham, 2004.

C. Roos, T. Terlaky, and J.-P. Vial. *Interior point methods for linear optimization*. Springer Science & Business Media, 2005.

R. Rubinstein and D. Kroese. *Simulation and the Monte Carlo Method: Third Edition*. Wiley, 11 2016.

R. Y. Rubinstein. Optimization of computer simulation models with rare events. *European Journal of Operational Research*, 99(1):89–112, 1997.

S. Sajadi, M. G. Arianezhad, and H. A. Sadeghi. An improved wagner-whitin algorithm. *International Journal of Industrial Engineering and Production Research*, 2009.

H. Süral, M. Denizel, and L. N. Van Wassenhove. Lagrangean relaxation based heuristics for lot sizing with setup times. *European Journal of Operational Research*, 194(1):51–63, 2009.

J. M. Thizy and L. N. V. Wassenhove. Lagrangean relaxation for the multi-item capacitated lot-sizing problem: A heuristic implementation. *IIE Transactions*, 17(4):308–313, 1985.

M. A. Trick and S. E. Zin. Spline approximations to value functions: linear programming approach. *Macroeconomic Dynamics*, 1:255–277, 1997.

W. W. Trigeiro, L. J. Thomas, and J. O. McClain. Capacitated lot sizing with setup times. *Management Science*, 35:353–366, 1989.

H. M. Wagner. A postscript to dynamic problems in the theory of the firm. Technical report, STANFORD UNIV CA, 1959.

H. M. Wagner and T. M. Whitin. Dynamic version of the economic lot size model. *Management Science*, 5:89–96, 1958.

W. I. Zangwill. Minimum concave cost flows in certain networks. *Management Science*, 14(7):429–450, 1968.

X. Zhao and P. B. Luh. New bundle methods for solving lagrangian relaxation dual problems. *Journal of optimization theory and applications*, 113: 373,397, 2003.

Y.-S. Zheng and A. Federgruen. Finding optimal (s, s) policies is about as simple as evaluating a single policy. *Operations Research*, 39(4):528–687, 1991.

L. Özdamar and M. A. Bozyel. The capacitated lot sizing problem with overtime decisions and setup times. *IIE Transactions*, 32(11):1043–1057, 2000.

List of Algorithms

List of Tables

List of Figures