POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria
Gestionale e della Produzione

TESI DI LAUREA MAGISTRALE

ROBUST MIN-MAX REGRET APPROACH FOR A SINGLE MACHINE SCHEDULING PROBLEM WITH COMMON DUE-DATE

Relatore:

Prof. Fabio Salassa

Candidato:

Alessandro Guerrazzi

Mat. 268014

Anno Accademico 2020/2021

POLITECNICO DI TORINO

Summary

Introduction

This thesis aims to give to firms a tool that supports short-term decisions about the processing order of different jobs on a single machine.
In particular, a lot of companies often face *scheduling problems* where the goal is to minimize the number of overdue jobs or, simply, delays. The importance to find a good solution has impact not only for customer loyalty but, most of all, in terms of production costs: usually delays, due to contractual constraints, involve the payment of penalties related to the number of late jobs or related to the overall amount of delay. In these situations, if the order scheduling is not computed by a robust tool, decisions follow, in most cases, the importance of jobs or the importance of the customer without having an overall view of the different effects that the entire schedule entails. Unfortunately, these *soft resolution methods* don't ensure an optimal solution but only a solution that may or may not be considered good enough.
The idea behind this thesis is to present and to explain an algorithm that, given a number *n* of jobs and parameters for each one, returns a schedule order with the objective to minimize the weighted number of jobs completed after a specific due-date. It's very important to understand that the presented algorithm doesn't ensure the calculation of the optimal solution but simply a solution that we can consider sufficiently close to the optimal one: we might be lucky and get the best schedule, but if that doesn't happen we are sure to obtain a sufficiently good solution. The difference between the proposed solution method and a simply subjective scheduling method, since both don't ensure the optimal schedule, is given by the reliability of the solution: behind the first one there is a defined scientific criterion described by the concept of

maximum regret, while in the second one there may not be a criterion that makes the scheduling completely reliable in terms of achieving the objective function.
Finally, we have to emphasize a strong assumption: the jobs, which will be processed in the same machine (*single machine scheduling problem*), have uncertain processing times described by intervals. However, there is also no information about the probability distribution of processing times within the intervals. So, we have to face a scheduling problem in which the timing scenario is not deterministic for each job but completely random between a lower and an upper bound.
In Chapter 1, we discuss the topic of Combinatorial Optimization that is the basis of our solution algorithm; in Chapter 2, we analyze the concept of robustness of an algorithm in Combinatorial Optimization by distinguish the deterministic case and uncertain variables case; Chapter 3 is dedicated to the concept of scheduling which will lead us, subsequently, to a complete description and formulation of the problem (Chapter 4); in Chapter 5, a solution approach and its execution are described from a logical and demonstrative point of view; in the Chapter 6, some tests are performed and explained to verify the robustness of the solution. Finally, in Chapter 7, the results obtained and the general conclusions are discussed.

Chapter 1
Combinatorial Optimization

1.1 What is Combinatorial Optimization

In everyday life and in particular industries, there are frequently a series of problems that require an effective solution methodology in supporting decisions. The central argument of this thesis is the theme of problems that can be solved through Combinatorial Optimization (CO) and to understand their meaning we try to break down the definition into its two terms: *optimization* means looking for the optimal solution within a set of feasible solutions, for example the shortest route from Rome to Florence or the reorder quantities which minimize overall transport and warehouse costs for a company; *combinatorial*, on the other hand, represent a subset of optimization problems which by characteristic possess a finite set of feasible solutions. To better understand the concepts, we make some examples:

- A steel plant, every week, have to program the quantity of steel (in kg) to be processed to minimize production costs in face of a known demand and some internal constraints;
- A transport company, every day, have to establish the shortest route to minimize delivery times.

Both problems, although coming from completely different realities, have a common factor: the need to look for a solution that better than any other satisfies the final objective of minimizing costs (in the first case) or minimizing time (in the second case). The substantial difference, however, is in the structure of the set of admissible solutions: the quantity in kg to be produced for a steel

plant is a continuous variable and, as such, makes the range of possible solutions infinite; on the other hand, the different routes that a transport company can evaluate are limited and, even if they were an exorbitant number, one of them is certainly cheaper than the others. As we can see, for definition given above, the second example is the one that falls within problems that can be solved through combinatorial optimization, while the first example, in which the solution is to be found within a continuous set, belongs to the world of Linear Programming (LP) and has a completely different solution approach.

In Operational Research, generic optimization problems are formulated through a mathematical modeling of the type:

$$min_x \ \ f(x) \qquad (1)$$
$$g \ \leq k \qquad (2)$$

The first row (1) represents the so-called objective function *f (x)*, that is the "entity" which is object of optimization of the type *min* or *max* - for example you want to maximize the profit deriving from an investment or create the secure portfolio with minimal risk. The second line (2) instead represents a generic constraint, for example the investment sum that must not be exceeded or the set of admissibility for some variable values. Obviously, numerous constraints can exist in more complex structures. In addition to the objective function and constraints, there are other fundamental logical constructs in the mathematical formulation of a problem such as *sets*, *decision variables* and *parameters*. Sets are a grouping of elements that share the same characteristic, for example the set W of the investments to be evaluated or the set R of the productive resources available in a company. The decision variables are those not immediately defined but object of definition in the search for an optimal solution, for example the number of

jobs to be processed daily in order to minimize production costs. Finally, the parameters represent known values such as the production capacity of a plant or the maximum number of daily transactions to be respected.

One of the most famous optimization problems is the *Knapsack Problem*, of which we provide a description and a mathematical model. In particular, we have N items and a knapsack. Each object i is represented by a volume w_i and an importance value v_i. The purpose is to maximize the total importance of the objects that we choose to place in the knapsack respecting a capacity constraint k.

$$max \qquad \sum_i v_i x_i \tag{1}$$

$$s.t.$$

$$\sum_i w_i x_i \leq k \qquad \forall i \in N \tag{2}$$

$$x_i \in \{0,1\} \qquad \forall i \in N \tag{3}$$

Of course, the input data is such that it is impossible to select all objects without exceeding the capacity limit k; therefore, it is necessary to decide which ones to insert and which not. We define individually all the elements and constructs present:

- N: set of available objects (each generic object is indicated with the subscript i);
- x_i: binary decision variables; they can only take the values $x_i = 0$ (object i is not inserted in the knapsack) and $x_i = 1$ (object i is inserted in the knapsack);
- w_i: volume of the object i;

- v_i: importance of the object i;
- k: knapsack capacity.

The objective function (1) is therefore a maximization of the total value of the objects that are inserted in the backpack, in which those with a term with $x_i = 0$ do not contribute to the total sum; the constraint (2) imposes an upper bound on the insertion of objects, or the capacity of the knapsack; the constraint (3) indicates the admissibility set of the decision variables which, in this case, is of the binary type.

As it is possible to notice, there is a finite set of solutions that coincides with all the possible combinations of objects that respect the capacity constraint; the aim is to select the grouping with the highest overall value. Therefore, given the discrete structure of the admissibility set of the solutions, the knapsack problem falls within the combinatorial optimization problems. Otherwise, if we had modeled the production problem of a steel plant, the constraint on the decision variable indicating the quantity to be produced would have been of the type $Q \geq 0$, that is any non-negative value which therefore represents a continuous variable. In general, from a mathematical point of view, if decision variables are binary, the problem is a combinatorial optimization problem.

The solution approaches of combinatorial optimization problems are divided into two categories: *exact methods* and *heuristic methods*.

1.2 Exact Methods

The **exact methods** are solution algorithms that investigate the entire set of solutions to ensure the search for an optimal solution

(or affirm its inadmissibility). The advantages of using an exact method are many, for example the possibility of having some indications on the lower bound and the upper bound of the optimal solution if the procedure stops prematurely. Among the disadvantages, the most significant is certainly the search time for a solution which in some problems may be acceptable while for others it is not reasonable. For this reason, over the years, more and more efficient exact methods have been developed to obtain the optimal solution, among which one of the most important is certainly the Branch & Bound.

1.2.1 Branch & Bound Algorithm

Branch & Bound is a solution technique for combinatorial optimization problems that uses an efficient exploration procedure of the solution tree. To understand how it works, let's imagine a scheduling problem where it is necessary to decide the processing sequence of *n* jobs in order to minimize delays. With *n* jobs, we have *n!* different sequences that form the set of solutions: obviously, one (or more) of them is the optimal solution and for this, as we will see later, scheduling problems can be categorized as combinatorial optimization problems. The solution tree is instead the logical composition structure of each feasible solution, and for $n = 3$ there is a representation in *Fig. 1*.
As we can see, at the roots of the tree there are all the admissible schedules, while the so-called “branches” represent a partial composition (or a path) that can be followed to reach one solution rather than another. The Branch & Bound algorithm starting from an initial solution that it deems good enough and, in this example, explores the entire tree in search of a schedule that can return, in terms of objective function, a better result than the one currently stored. The exploratory procedure is not random but only those

paths in which it is reasonable to go deeply to look for a solution are investigated; in other paths, which in the best case would return a worse solution than the current one, no analysis is carried out and the branches are therefore "cut". Whenever the algorithm identifies a better schedule, the optimal solution is updated and the exploration continues. The main advantage lies in computational savings, since not necessarily all possible solutions are questioned to establish which is the best one. Note that, although the algorithm seems to take shortcuts, no feasible solution is excluded ex-ante: once a parent-node does not possess the characteristics to generate a better solution, these characteristics are inherited by all the child-nodes and so on.

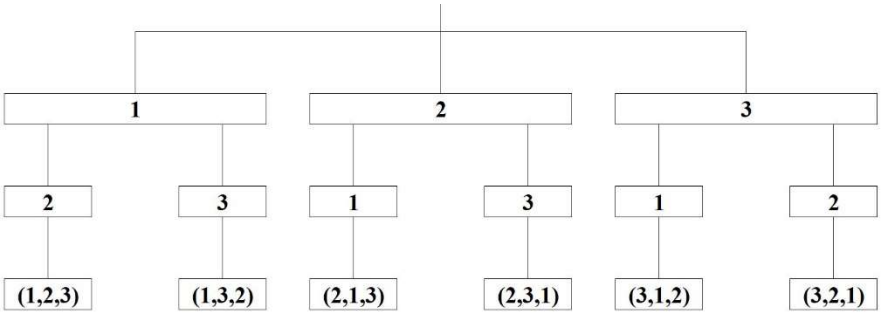

Fig. 1: Solution tree for a scheduling problem with n = 3 jobs

As mentioned above, one of the main criticalities of exact methods is the search time for a solution which, in some problems, is not sustainable. The advantage of obtaining an optimal solution is at the same time the disadvantage of patiently analyzing the entire solution tree: in scheduling problems, in particular, resorting to exact methods is almost impossible. To understand its complexity, let's imagine a likely situation with $n = 30$ jobs; in the worst case, if the algorithm took a second to analyze each of the $n!$ schedules (2.65E+32 different permutations), we would get an optimal

solution after 8.45E+24 years. The Branch & Bound method could reduce the search time which, in all probability, would remain equally unacceptable. To find a solution to this difficulty, let's analyze a second class of solving methods for combinatorial optimization problems, namely heuristic methods.

1.3 Heuristic Methods

The seen methodologies guarantee, at least in theory, to solve a combinatorial optimization problem in an exact way, that is, to find a feasible solution that corresponds to the optimum of the objective function among all admissible solutions. Heuristic methods (from the greek *heuriskein*, to discover), on the other hand, represent algorithms for obtaining a "good solution" in a reasonable time; in particular, the search for an optimal solution is given up in order to favor a sufficiently acceptable issue of resolving promptness. This approach can be corrected for several reasons: first of all, the search for an approximate solution is what is needed in reality when dealing with large problems where, moreover, the parameters involved represent estimates subject to error; moreover, the search for an optimal solution may have the simple purpose of evaluating, in general, which direction to converge in the decision-making field and, therefore, a sub-optimal solution is sufficient. In most combinatorial optimization problems it is possible to implement specific heuristics that exploit the characteristics of the considered problem and the experience deriving from the knowledge of the sector to obtain a high quality solution.

Let's take an example of heuristics:

Suppose a group of 8 jobs that can be worked indifferently on two identical machines. Each job has its own processing time and the common due-date is 9 min. The goal is to create a job schedule that minimizes the maximum overall delay, regardless of the number of delayed jobs.

Job-1(2 min); Job-2(3 min); Job-3(1 min); Job-4(3 min); Job-5(6 min); Job-6(2 min); Job-7(5 min); Job-8(3 min)

An heuristic method suggests that a quick, but still robust solution, is to sequence jobs in ascending order of processing time using both machines in parallel. Obviously, each job can be worked on only one machine and each machine must finish processing a job before starting a new one. In this way, the result is as shown in *Fig. 2*.

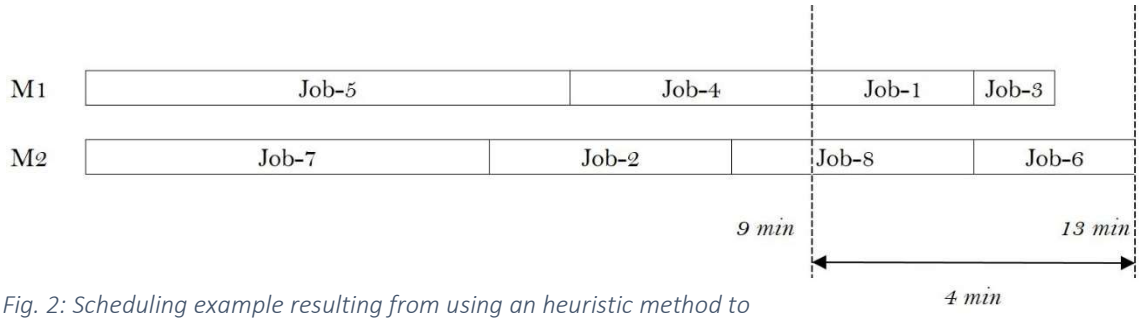

Fig. 2: Scheduling example resulting from using an heuristic method to solve the problem

Without using an exact method and, therefore, without investigating all possible combinations, there is no absolute certainty that a gap of 4 minutes on this schedule is the optimal solution; we can only say that a 4-minute result, using this heuristic method, is an admissible solution to the problem.

A possible classification of heuristic methods is the following:

- Constructive heuristics
- Metaheuristic methods
- Approximate algorithms
- Hyper-heuristics

In particular, we will analyze in more detail the constructive heuristics and the metauristic methods, while the approximate algorithms and the hyper-heuristics will not be mentioned. Moreover, before introducing metaheuristic methods, in sub-chapter *1.3.2* a famous class of solving methods in the heuristic field is described, namely the local search methods.

1.3.1 Constructive heuristics

Constructive heuristics represent a class of algorithms where, starting from an empty set, a good feasible solution is iteratively "created" by adding one element at a time. In scheduling problems, for example, if we had $n = 3$ jobs, a constructive heuristic has the task of filling a vector of three elements, indicating step-by-step the job i that entry in the sequence. The peculiarity of these solution methods is that the elements that form the solution are never re-called into question once inserted. Of course, the process ends when a complete solution is generated.

Among constructive heuristics we distinguish:

- Greedy algorithms;
- Algorithms that simplify potentially exact procedures;

In addition to them, there are obviously other types of constructive heuristics such as, for example, *algorithms based on exact optimization techniques*.
The idea behind the **greedy algorithms** is very simple: at each iteration, choices are made that seem optimal only at the moment of choice. Therefore, the correctness of the solution is not always guaranteed, but they are often very simple and efficient methods. In particular, at the generic iteration k, the algorithm is myopic with respect to the globality of the problem and has the task of obtaining an optimal local solution only for the subproblem k; after that, a local optimal solution will be found for the subproblem $k + 1$ and so on until the end of the procedure. To better understand how this algorithm works, let's take an example:

We resume the scheduling problem with $n = 3$ jobs. Above all, remember that the aim is to minimize the number of delayed jobs by deciding the optimal processing sequence. In this regard, we add a fundamental parameter: the processing time p_j for a generic job j. A step-by-step greedy algorithm could be the following:

1. Generate an empty set S of n elements and initialize the counter variable $k = 0$;
2. **If** $k = 3$, the algorithm ends (that is, the sequence is complete);
   Else Select the job with the shortest processing time and insert it in position k; after which set $k = k + 1$;
3. Go to step (2).

The general idea of this algorithm is to process jobs in increasing order of processing time p_j; in this way, the most quick jobs are promptly processed while the most durable jobs are queued. This solution approach is obviously subjective; as previously mentioned, an expert in this field could take advantage of his

experience and choose to implement a totally different algorithm to achieve the objective function more effectively. Let's make some observations:

- This procedure does not investigate the totality of solution set (*Fig. 1*), but uses a totally subjective approach to reach a good feasible solution: therefore, it is a heuristic method;

- The final sequence is built iteratively until $k = 3$ (complete sequence); therefore, it is a constructive heuristic method;

- At each iteration k, an optimal local solution is extracted on the basis of the jobs still to be inserted in sequence and according to a subjective criterion for selecting the jobs with shorter processing time; this means that the jobs already processed are no longer re-called and, therefore, it is a greedy algorithm.

Algorithms that simplify potentially exact procedures, instead, foresee the use of a solution algorithm that partially uses an exact method. The simplest variant is that obtained by terminating a Branch & Bound algorithm after a certain time-limit or after a predetermined number of nodes and exploiting the best feasible solution generated up to that moment. For example, in certain problems in which the search for an optimal solution is not essential, it is useful to obtain an indication of its lower or upper bound; hence, by executing an exact algorithm for a certain period of time, good quality solutions are certainly obtained.

1.3.2 Local Search Methods

The basic idea of the Local Search Methods is to define an initial solution (current solution) and try to improve it by exploring an its appropriately neighborhood. If the optimization around the current solution produces an improvement, the procedure is repeated starting, as current solution, from the solution just determined. The algorithm can terminate for several reasons:

- It is no longer possible to find better results around the current solution and this one represents an optimal solution;
- The current solution coincides with a limit value (upper bound or lower bound of the objective function) and therefore we have reached the optimal solution;
- The procedure provides for the insertion of a time-limit which returns the best solution obtained up to that moment.

A generic Local Search algorithm can be described as follows:

1. Determine an initial solution *k*;
2. **If** there exists a solution *k'* in the neighborhood of *k* for which f(*k'*) < f(*k*), then *k* = *k'*;
   If f(*k*) = LB, *k* is an optimal solution and the algorithm ends;
   Else Go to Step (2);
3. **Else** *k* is an optimal local point.

Note that in this case the aim is to minimize the objective function; a dual algorithm can be implemented for the maximization of f (*k*).

We reiterate that, in general, local search methods guarantees to find only optimal local points, that is, solutions in which there are no better solutions around. The previously described algorithm can be used in a large variety of problems; however, it remains to be defined how to extrapolate an initial solution, how to define and explore a neighborhood and how to evaluate obtained solutions. Answers to these questions are not univocal but depend on the type of problem, the industry and, sometimes, experience: it is possible that a procedure for generating an initial solution is effective in certain problems but completely ineffective in others.

Local search methods are useful for better understanding the next class of approximate solution methods, namely metaheuristics.

1.3.3 Metaheuristics Methods

In recent years, the study of a new type of algorithms has been introduced which, by combining heuristic methods, are able to explore the solution space more efficiently. These algorithms are called *metaheuristics*.
"A metaheuristics is formally defined as an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space, learning strategies are used to structure information in order to find efficiently near-optimal solutions" [Osman and Laporte, 1996].

In other words, metaheuristics are high-level solution strategies that, using different methods, guide the search process within the solution space so that a *rational* exploration is made in certain areas in which it is reasonable to perform a search; the goal is to reach sufficiently good solutions in a very short computing time.

They can be considered as a third category of solution approach, beyond exact methods and heuristics, to solve optimization problems. The exact methods, which explore the entire solution space through an exhaustive research, guarantee the achievement of a global optimal solution; the disadvantage, as already mentioned, is the great sensitivity to the extent of the problem which, in many cases, could lead to giving up the search for an optimal solution. Heuristics *ad-hoc*, instead, are approximate methods of searching for a sub-optimal solution in a reasonable time. Metaheuristics aim to have an intermediate resolution efficiency and, in particular, they are not exhaustive like the exact methods but use combinations of heuristics in a more intelligent way (*Fig. 3*). For example, in relation to the problem, I could exploit the effectiveness of a heuristic to find a good initial solution and the effectiveness of a second heuristic to explore its surroundings.

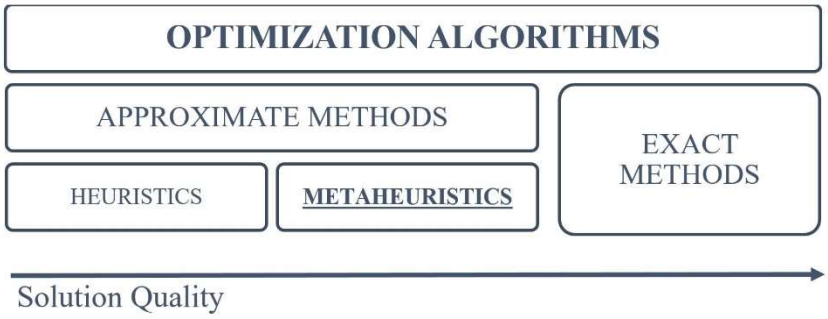

Fig. 3: Structure of Optimization Algorithms

There are different ways to classifying metaheuristics. The most important division is that between *Trajectory Methods* vs. *Population-based algorithms*.

Trajectory Methods

The great merit of local research lies in representing an excellent compromise between simplicity of implementation and results obtained. However, the stopping criterion used trapped the method within local minimum points. In some exceptional cases, the characteristics of the neighborhoods guarantee that a local minimum point is also a global minimum point, but this remains an exceptional circumstance.
In recent years, a number of evasion methods from local minimum points have been developed. Some of these, the trajectory methods, build a real trajectory in the search space by memorizing the best solution that is encountered along the "way" (*Fig. 4*). The basic mechanism is that seen for local search methods with the only big difference that, in this case, other possible places are explored in solution space where it is possible to study a neighborhood that could potentially yield a better solution than the current one. The main features of these methods are the randomization of the exploration and the memorization of the solutions already explored: this avoids possible cycles in the event that, in the exploration of a further neighborhood, worse solutions are accepted.

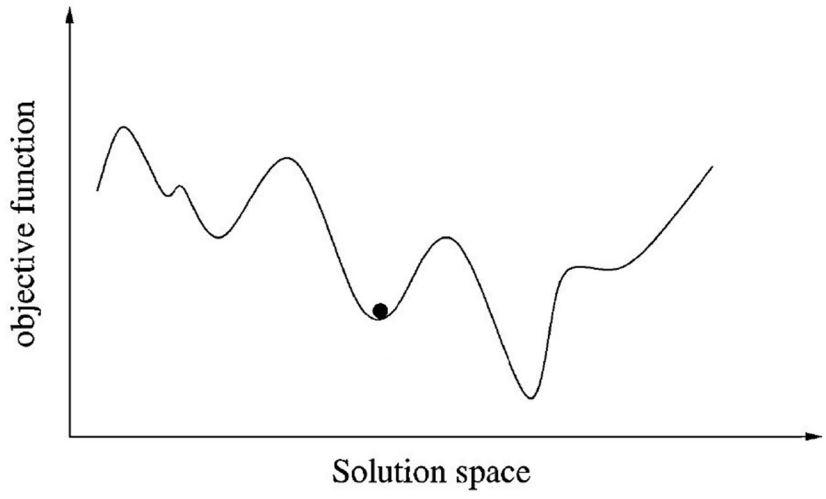

Fig. 4: Example of Trajectory concept

One of the most famous trajectory method is the **Tabu Search** algorithm, which is an excellent compromise between quality of performance and computational effort.
The Tabu Search algorithm is a local search method capable of evading from local minima exploiting *memory mechanisms* (mentioned above). In fact, the search without keeping track of the exploration history of the solution space only remembers the best solution explored up to that moment (which coincides with the current solution in the local search) and the corresponding value of the solution. In this way, if worsening moves are accepted to revisit a certain solution, there are no structural elements to avoid returning to solutions already visited and, therefore, the only way to avoid cycling is by chance. In the Tabu Search, we try to exploit the structure of the search space, memorizing some information on the solutions already visited in order to orient the search and escape from local minima avoiding cycling. Basically, this is achieved by changing the neighborhood of the solutions according to the history of the exploration, making some neighbors "taboo", that is, not explorable at that moment.

Population-based algorithms

The methods described in the previous section have the characteristic of constructing a punctual trajectory in the solution space, considering, at each iteration, only one solution. On the other hand, there are metaheuristics which maintain a population of solutions, i.e. a set of several solutions, and, at each iteration, combine these solutions together to obtain a new population. The idea is that, through appropriate recombination operators, better solutions can be obtained than the current ones.

A class of population-based algorithms is that of **Genetic Algorithms**, which are distinguished mainly by their simplicity of implementation and their adaptability to multiple types of problems.
The basic principle of genetic algorithms follows the evolutionary theory of individuals and their adaptation to the environment in which they live. In particular, an analogy is recreated between groups of admissible solutions and individuals in society, who share, in some way, the same characteristics: individuals combine with each other to generate new and different individuals who will be part of the "populations" following; those who participate in reproductive processes are the individuals most adapted to the environment in which they live.
Genetic Algorithms try to simulate the evolutionary process by creating a logical matching between individuals and solutions, each in its own environment; therefore, a *fitness* measure that describes the adaptation quality is established for each solution.
Very generally, genetic algorithms start from an initial population of solutions and generate iterative evolution. At each iteration, the solutions are evaluated according to their level of adaptation and, on the basis of this evaluation, some of them are selected, favoring the parent-solutions with greater fitness. The selected solutions are recombined together (as a real reproduction) to generate new solutions which tend to transmit the good characteristics of the parent-solutions to subsequent generations (child-solutions).

Chapter 2
Robust Optimization

In the previous chapter we have introduced the topic of Combinatorial Optimization, its limits and some hints on solution approaches. In particular, we have mentioned the existence of some resolution methods that unlike the so-called *exact methods*, i.e. algorithms that analyze the entire solution tree to find the optimal one, investigate only a subset of the solution set. The goal is to bypass some limits of Combinatorial Optimization, especially the search for a solution in an unreasonable time.

2.1 Concept of Robustness

In many cases, however, in the attempt to find an optimal result, solution algorithms are implemented in which parameters are considered *deterministic*, therefore not subject to any uncertainty. This is a very strong hypothesis since a wrong approach to the problem or an inability to predict the behaviour of some factors can lead to serious errors in solution evaluation and, in particular, in its quality, reliability and feasibility.
Problems already discussed or heuristic methods mentioned in the previous chapter, assume, for example, parameters without uncertainty (e.g. knapsack capacity, due-date, job processing time, etc.) even if, in reality, we don't have any conviction about the manifestation of those values. Obviously, there are factors whose variability has a low impact on solution performance while, on the other hand, there are factors that require an accurate forecasting of possible scenarios.

As result of this, in recent years, a new branch in optimization methods has emerged, called *Robust Optimization*.
The robustness of an algorithm defines its ability to respond promptly and effectively to the possible variations to which the system is subject. Therefore, in the deterministic case, the solution approach returns a solution that can be considered the optimal one only in the event that there is no variability of the parameters with respect to those provided in the model.
In the Knapsack Problem, for example, we have defined the weight of each items w_j as if they were known upstream. Under this assumption, however, the solver generates a solution that can be sensitive to any variation: if real weights are completely different from the hypothesized one, we would obtain a completely wrong and, probably, highly dangerous solution. Similarly, the knapsack capacity k may not be known upstream but, for example, is defined within a range of values. The damage resulting from an incorrect decision can be immediately visible if the problem is an operational problem that requires a solution for short-term decisions; on the other hand, if it's necessary to solve a strategic problem, the effects could only be visible in the long-term. The substantial difference is in the *correction costs* which, in the second case, risk being highly significant.

2.2 Types and Models of Robustness

In many cases, a solution can be defined *robust* if, under certain conditions, it behaves reasonably with respect to characteristics of quality, optimality and feasibility; in other cases, a solution is robust if, given the nature of the problem, it's the best choice even in a worst-case scenario. So, it's possibile to distinguish several types of uncertainty within an optimization problem:

Uncertainty in the feasibility of the solution
In this case, it's necessary to understand if the solution obtained can exist for each combination of the parameters considered (or for each possible scenario). Therefore, there is a trade-off between the search for a solution with respect to the established parameters and the quality of the solution obtained;

Uncertainty in the optimality of the solution
In relation to the set of uncertainty chosen, the optimality of the solution found can be altered. In this case, a robust optimization algorithm is the one that performs correctly in all different scenarios;

Uncertainty in the optimization problem
Uncertainty is mainly generated by errors in the evaluation of some parameters or by changes caused by external environment.

Depending on the problem nature, there are different models to approach a robust solution. A brief classification was made below:

- **Strict Robustness**
  Optimization problems with a *Strict Robustness* approach are those in which every possible scenario is of critical importance. Therefore, it's fundamental to consider all possible manifestation of uncertain parameters before extracting a robust solution. For example, in aerospace industry, it's not possible to ignore critical scenario which, although unlikely, could impact the aircraft stability; in scheduling problems, on the other hand, an unfavorable event has a lesser impact than the previous case.
  This is the strictest and most stringent robust optimization approach.

- **Cardinality Constrained Robustness**
  This approach is less stringent than the previous one. This is because it's considered only a subset of critical parameters from the entire set of uncertainty: this subset is able to autonomously create an unfavorable scenario regardless of the value assumed by the other parameters not considered. The latter can be modeled with their most frequent value.

- **Adjustable Robustness**
  In this case, the space of uncertainty is divided into two groups: in the first one, there are variables that can be evaluated before the occurrence of the generic scenario *k*; in the second group, instead, there are variables that can be determined after observing the scenario *k*. The purpose of this subdivision is to further relax the space of uncertainty.

- **Light Robustness**
  Another way to relax a *strict robustness* is to ease some problem constraints to accommodate a higher quality solution. The basic concept is that a good solution that respects the restrictions can be found close to a good solution calculated in the most likely scenario (the average case). For this reason, there is a trade-off between robustness and quality of the solution.

- **Regret Robustness**
  This relaxation method, as we will see later, calculates the difference in terms of objective function, in a generic scenario *k*, between a possible solution and the one with the best target value for that scenario.

Chapter 3
Overview on Scheduling Problems

3.1 Preliminary Scheduling Concepts

Scheduling is a decision-making process that concerns the allocation of limited resources over time. In general, the associated decision process requires determining the order in which the set of activities is performed, the time in which each activity is performed with the aim of pursuing a certain goal.
Resources, activities and objectives, depending on the context, can have different aspects. In particular:

- The **resources** can be the machines that make up the production plant of a company or the branches that a bank owns;
- The **activities** correspond to the elementary processes carried out on the resources, for example the processing of a material or a home delivery;
- The **goal**, on the other hand, may be the minimization of delivery delays or the average time to complete a given process.

The term "scheduling" refers to a vast class of problems, very different from each other in complexity and structure. However, unlike what happens for other areas of combinatorial optimization, for more complex scheduling problems it is not possible to indicate a single solution approach but it is more appropriate to use heuristic or approximate methods (as mentioned in Chapter 1). In most cases, the limited (and sometimes scarce) resource to be optimally allocated to certain activities is the *time* factor.

Let's take an example:

Suppose we have two perfectly identical production departments for the manufacturing (activities) of footwear and a list of orders to be fulfilled. Contracts stipulated with customers provide for the payment of a penalty for each day of delay with respect to that agreed for delivery. When problems of this type are faced, especially in cases where resources are so limited that they cannot be allocated to all activities, it is essential to establish a solution method that returns the optimal processing sequence in order to best meet the objective, which in this case is the minimization of the penalty costs. In other words, if the resources available (the two departments and the time) are not sufficient to fulfill all the orders before their *due-date*, then it is necessary to decide which jobs to process first and which jobs to queue even if paying a penalty.

Scheduling problems usually satisfy operational and consequently short-term decision-making processes. On the other hand, when decisions need to be made at a strategic or tactical level (eg. opening a warehouse or not or choosing the production capacity for a company) completely different solution approaches are used. In general, scheduling activities is a process that can be repeated periodically and that has an impact only within a limited time interval. For example, every day a workshop can decide the order of processing of some repairs or a student can organize the order of study of their subjects in order to have an optimal preparation for the exams.

The parameters associated with the job j to be processed can be of various kinds:

- Processing time p_j, which corresponds to the time that job j can request on machine i;

- Delivery time d_j, or the instant of time beyond which the job is considered late;
- Weight w_j, which represents the relative importance of job j;
- Release time r_j, which indicates the instant of time in which job j enters the scheduling process.

As regards the scheduling objectives, it is useful to introduce some variables that are important in the definition of an *objective function*:

- Completion time C_j, represents the instant of time in which job j ends its processing;
- Lateness L_j, indicates the difference between the completion time of a job and its delivery date, therefore $L_j = C_j - d_j$. We can note that if $L_j > 0$ then job j is late, while if $L_j < 0$ then job j is early;
- Tardiness T_j, coincides with lateness L_j when it is positive and assumes value 0 otherwise, that is $T_j = \max \{0, C_j - d_j\}$.

Starting from these elementary performance measures, it is possible to construct more complex and articulated variables that better represent the elements to be optimized in the scheduling problem.

In scheduling problems, a synthetic three-field notation $a|b|c$ is usually used to describe a specific scenario and the objective to be achieved:

- a: identifies the machine system (1 for single machine, P for identical parallel machines, F for Flow Shop and J for Jop Shop);
- b: represents particular characteristics of the jobs (eg. priority constraints);

- c: indicates the measure of performance that requires optimization.

3.2 Classification of Scheduling Problems

One of the most common methods of classifying scheduling problems is that relating to the characteristics of the system under consideration. The possible architectures of a production (or service) system are many, and in this chapter the most frequent cases will be analyzed.

3.2.1 Single Machine

Single machine problems are the simplest ones, as all jobs must be processed on the same resource.

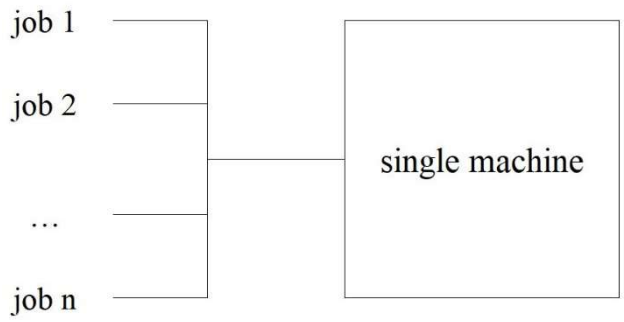

Fig. 5: Structure of a Single Machine case

Suppose a production process in which at time t = 0 there are $n = 5$ jobs to be processed, each with its own processing time p_j. The goal is to minimize the overall waiting times of the jobs, that is $min \sum_j C_j$. In particular, the problem notation is the following:

$$1 \,//\, \sum_j C_j$$

In this case, the solution algorithm *SPT* (Shortest Processing Time) suggests sequencing the jobs in ascending order of processing time p_j in order to promptly fulfill the most quickly jobs and queue those that require more lasting processing. Although the overall processing time ($\sum_j p_j$) is the same regardless of the sequence chosen, the order established according to SPT allows for faster disposal of orders and therefore, overall, a more efficient and optimal fulfillment process.
So if the vector p of the job processing times is composed of the elements *{8,16,10,7,2}*, the optimal processing sequence would become *{2,7,8,10,16}* where each element of the vector corresponds to a job j.
A second variant of this problem is the presence of a relative weight w_j for each job j. With the same objective function, *min* $\sum_j C_j$, the *WSPT* (Weighted Shortest Processing Time) algorithm suggests ordering the jobs in ascending order of the ratio p_j / w_j to obtain the optimal sequence.
In addition to these examples, there are numerous types of single machine problems where each one has its own characteristics and performance measures to be optimized.

3.2.2 Parallel Machines

Parallel machine scheduling problems are usually presented as situations where a group of n jobs can be processed on m machines in parallel. Therefore, unlike the previous case, there are m resources that can carry out the same activity at the same time.

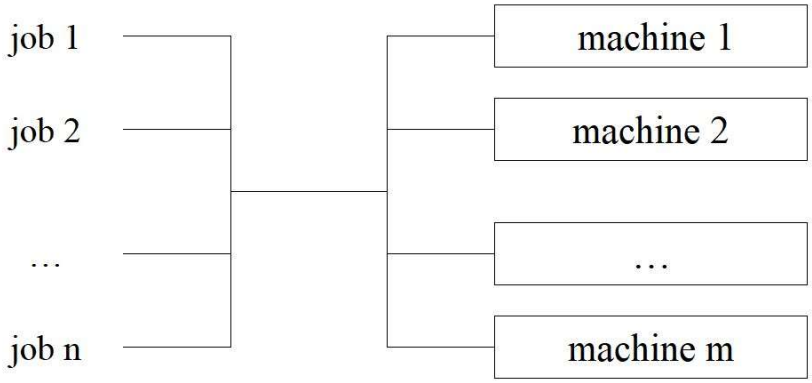

Fig. 6: Structure of a Parallel Machines case

Suppose a production process in which at time t = 0 there are $n = 5$ jobs to be processed, each with its own processing time p_j. The goal is to minimize the *maximum Completion Time*, that is *min* C_{max} having $m = 2$ identical machines available in parallel. The problem notation is as follows:

$$P \,//\, C_{max}, \qquad \text{or better} \qquad 2 \,//\, C_{max}$$

The *Longest Processing Time* (LPT) rule assigns the longest m jobs to the m machines. Later, when a machine is free, the unscheduled job with the largest processing time is assigned to it, and so on. This solution methodology tries to assign the shortest jobs at the end of the scheduling process where they can be used to balance loads between machines.

3.2.3 Flow Shop

The main feature of scheduling problems of the *Flow Shop* type lies in the production process of the jobs: in particular, in absence of priority constraints, jobs follow the same processing cycle

consisting of a certain number of phases according to a predefined sequence. Therefore, all jobs share production steps that can represent, for example, the departments of a company. If we think of a footwear company, each product goes through production phases with a rigid sequence, such as the cutting of raw materials, hemming and final assembly. Of course, there may be numerous variants of Flow Shop problems: in the simplest case each *task* can be carried out only on a single resource (only one cutting machine available), while in more complex cases there may be more resources for a single task. The goal is to generate the work order so that a certain objective is optimized. In problems of this type, the objectives can be multiple, for example the minimization of the overall completion time of all the processes or the minimization of the overall delay time. As in the previous cases, there may be different solution methodologies that return sufficiently good solutions, among the most important we find the *Johnson's algorithm*.

3.2.4 Job Shop

Finally, the *Job Shop* scheduling problems are the most complex to manage as numerous assumptions of the previous cases are relaxed. In fact, in the standard version there are n jobs and m operations $O_1, O_2, \ldots, O_m$ to be carried out on certain resources. Each job can have a different production process that follows a rigid and well-defined work order: therefore, for example, Job 1 needs to perform operations O_1 and O_2, while Job 2 follows the process O_3, O_2, O_3. In addition, there may be priority constraints between the n jobs and in the most common cases it is important to respect the order of arrival and consequently the processing order.

3.3 Resolution Methods in Scheduling Problems

The search for optimal solutions in scheduling problems is often the subject of reflection. As can be seen, the optimal sequence certainly belongs to a finite set of solutions of amplitude *n!* (with *n* number of jobs considered), i.e. all possible extractable permutations. Therefore, trivially, it would be enough to enumerate the entire set of solutions and select the sequence that best satisfies the objective function. However, this resolutive procedure has operational limits: in fact, with a sufficiently small number of jobs it is possible to obtain a solution in a reasonable time while with a modest number of jobs, however, the waiting times could be long or even unacceptable. To get an idea, with $n = 10$ jobs we have $10! = 3,628,800$ different permutations, while with $n = 30$ jobs there are instead $30! = 2.65E + 32$ candidate solutions: in the first case, assuming a processing time equal to 0.5 seconds per permutation, the optimal solution will be returned after about 40 days; in the second case after 1.33E + 32 years. As in Chapter 1, we can distinguish the solution methods in two different categories: the exact methods and the heuristic methods.

3.3.1 Exact Methods

The exact methods are distinguished by their completeness and effectiveness in the search for the optimal solution. In particular, they are solution methods that undoubtedly return the best solution (unlike heuristic methods that return an approximate solution instead). When the number *n* of jobs is quite small, the exact methods represent an ideal class of algorithms in solving scheduling problems. Among them, we can further distinguish the construction methods and the enumerative methods:

Constructive methods are able to generate an optimal solution by simply using strict priority rules in the sequencing of jobs based on the nature and structure of the problem. Among these we can find the *SPT* and *WSPT* algorithms seen above.

Enumerative methods, on the other hand, are algorithms that analyze the set of solutions and that can be made more efficient to obtain a lower computational complexity. For example, algorithms such as Branch and Bound, albeit analyzing the entire solution tree, are able to exclude families of solutions that would not guarantee higher performance than those found up to that moment: in this way, no analysis is carried out on the child-nodes. and the search time is reduced.

3.3.2 Heuristic Methods

Where the exact methods have computational limits and the waiting times for the search for a solution become unacceptable, a second macro-class of solution algorithms intervenes: heuristic methods. As explained in Chapter 1, heuristic methods do not guarantee the search for an optimal solution but a reasonably good solution with the sole purpose of reducing search times. The resolution strategies based on heuristic methods are manifold: for example, some of them favor speed of execution but produce low quality solutions; others, on the other hand, require longer times but generate sufficiently reliable solutions. Often we can recognize in a complex problem some substructures that allow the problem to be decomposed into simpler subproblems, which can be solved in optimality. It is then a question of aggregating the partial solutions in order to obtain the global solution. Often, in this operation sufficient information is obtained to review the partial solutions to improve the overall solution. This process ends when

it is believed that an acceptable solution has been obtained or when the solution cannot be further improved. Another widely used strategy, as it is virtually applicable to any problem, is to recursively generate a series of solutions obtained from each other through small improvements. The type of improvement that can be achieved obviously depends on the structure of the problem under consideration. The procedure ends when no further improvements are possible. This kind of strategy, in its simplest form, is called *local search*. More elaborate forms have also been proposed, two of which seem to give good results, one of the deterministic type, called *tabu-search*, and which could also be called search with memory, and the other of the stochastic type, called *simulated annealing*, which is based on an interesting physical analogy.

3.4 Robustness in Scheduling Problems

Generally, scheduling problems are studied and solved in a totally deterministic environment. This means that the parameters and variables involved are not subject to uncertainty but assume certain values for the entire duration of the scheduling process. This is a very strong assumption since the optimality of the solution found strongly depends on what happens in the real world and on the *noise* related to the unpredictability of the external environment. In most cases, the input data in scheduling problems are the structural ones (number of jobs, number of resources, number of tasks, etc.) and those related to the jobs to be processed (processing time, importance, etc.). While structural parameters are hardly subject to uncertainty, those relating to jobs are more sensitive to variability. The processing time, for example, is an unpredictable value: in many problems that assume deterministic data, the input data coincides with the most frequent occurrence values. For this reason, in contexts that require it, it is essential to implement a

solution strategy that addresses a probable scheduling problem and that consciously returns the best solution to the variability of certain parameters. A first observation is that the solution of a deterministic problem does not necessarily coincide with the robust solution, as both have different objectives: for example, the first aims to find a global optimal solution while the second can search for the best solution in the face of all unfavorable scenarios (therefore the best in the worst cases).

Chapter 4
Problem Description

Once the concept of Combinatorial Optimization and robustness of a solution algorithm is understood, a detailed description of the scheduling problem, object of this work, is presented below; subsequently, a solution approach is proposed and discussed.

We study a scheduling problem in which *n* jobs must be processed in sequence on the same machine. All jobs don't have an order of arrival, so it's not possible to define any priority constraints. Furthermore, once the job is in progress, it's forced to finish its working process before leaving the machine and therefore it's not possible to fragment the working time. Each job has an uncertain processing time defined only within an interval and there is no indication of its probability distribution: the real value can assume any value between a lower bound and an upper bound. Finally, all jobs can have different weights and share a common due-date, beyond which a job, still in progress, is considered late.
The objective is to find the schedule that minimizes the weighted number of delayed jobs (or, otherwise, that maximizes the weighted number of jobs on time).

A similar problem, with deterministic processing times and common due-date, could be solved by using heuristic methods capable of returning an optimal schedule. For example, the SPT (Shortest Processing Time) algorithm provides an optimal solution, in the unweighted case, by sorting jobs in ascending order of processing time. Therefore, the fastest jobs are processed immediately while the longer ones are processed last; intuitively, it's difficult to find a more effective algorithm to minimize the number of late jobs (*Fig. 7*).

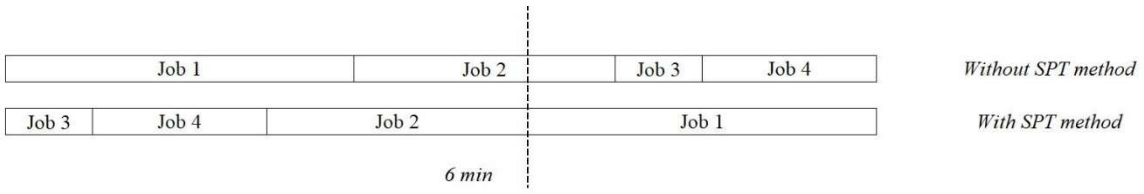

Fig. 7: In the first case, there are 3 delayed jobs after six minutes; in the second case, with SPT method, there is only one job delayed

However, the decision-maker may not know exactly some starting data and an algorithm such as SPT, which by hypothesis uses known values, risk being practically inefficient (or even dangerous).

When some input data are uncertain, the solution approach to the problem and the objective function change completely, and the algorithm to be implemented must be able to work optimally despite the possible external turbulence that can create unexpected situations: an algorithm that performs in this way, as already mentioned, is defined *robust*. Therefore, the robust approach will allow a *safe* scheduling even in the event of an unfavorable time scenario or, specifically, it will be the one that returns the solution that performs best despite the worst case scenario occurring.

4.1 Concept of Maximum Regret

When dealing with problems of this type, the concept of *maximum regret* is used in the search for a solution strategy in the presence of uncertain variables. Briefly, it represents the greatest distance, in terms of objective function, between one solution and one another that is defined as its *adversarial solution*.

To understand better, let's suppose we have to work $n=4$ jobs with common due-date on single machine and we consider the schedule

(1,2,3,4). All four jobs do not have deterministic processing times but will assume a completely random value between two definite extremes.
At this point, we can ask what will be considered as the *worst-time-scenario* that could happen. A first possibility is to imagine a situation in which each processing time for each job reaches its upper bound (an *extreme* scenario); in this way, apparently, the whole process will be performed with the greatest possible completion-time and this would mean maximizing the number of delayed jobs. In reality, the concept of worst-time-scenario, according to this point of view, assumes an absolute value: if the due-date is not too far away, then it's possible to notice that an extreme scenario may not guarantee a good solution regardless of the chosen scheduling, and therefore there is a good chance that our sequence *(1,2,3,4)* is no worse than the others; in other words, there would be *no regret* if all sequences (including ours), with an extreme scenario, would return all jobs late. A second possibility is to consider the worst case scenario as the one that creates the greatest gap between the number of jobs on-time using the current schedule and the number of jobs on-time using another schedule defined the *adversarial solution* which maximizes, precisely, this difference. If so, in the event that the worst case scenario occurs, the time-vector will be such that used by the adversarial schedule, it will create the greatest difference (compared to our schedule) in terms of jobs on-time: in this way, we will have *regret* to not have chosen the adversarial solution to the current one. It's important to note that every possible sequencing has its worst time scenario and its own adversarial solution. Therefore, is "someone" told us that the *maximum regret* choosing the sequence *(1,2,3,4)* is 12, it means that 12 is the maximum number (possibly weighted) of jobs that I could not have sent late by choosing the opponent scheduling; on the other hand, if sequencing *(1,2,3,4)* had 0 as *maximum regret*, it

means that *(1,2,3,4)* is an optimal solution even if its worst-time-scenario occurs.

Once the maximum regret for each schedule has been defined, the best schedule will be the one with the *smallest maximum regret*; this is because a low regret value indicates a low error that is committed in the event that an unfavorable scenario occurs. In other words, it would be the best solution being the most rational solution to choose.

4.2 Problem Formulation

At this point, it's possible to give a mathematical formulation of the problem.

Let $J = \{1,2,3,\ldots,n\}$ the set of n jobs to be processed. Each job j is described by a weight w_j and a processing time p_j included within a range of real values where p_j^{min} and p_j^{max} are respectively the lower bound and the upper bound of the interval; the common due-date is d; Moreover, let $C(\pi, j)$ the completion-time of job j in the generic scheduling π, defined as the sum of the processing times of the jobs preceding j in the sequence π and the processing time of j.

Subsequently, we define the boolean variable $U_j(\pi)$ which assume value = 1 if job j in schedule π is late and value = 0 otherwise, i.e.:

$$U_j(\pi) = \begin{cases} 0 & C(\pi, j) \leq d \\ 1 & \text{otherwise} \end{cases}$$

Remember that a job is defined as late when its completion-time exceeds the value of due-date, even if its work process has started before d.

Now, we can model the *objective function* by incorporating the concept of *maximum regret* previously explained.
Using a generic sequence π, the weighted number of delayed jobs is the following:

$$F(\pi) = \sum_j w_j U_j(\pi)$$

Let define the generic time-scenario **p** as a vector of *n* elements that contains the processing times of all the jobs; obviously, since p_j is defined on a set of real numbers, there will be infinite **p** vectors:

$$\mathbf{p} = (p_1, p_2, \ldots, p_n): \qquad p_j^{min} \leq p_j \leq p_j^{max} \qquad \forall \text{ j}$$

At this point, we need to find that scheduling which, using the **p** scenario, returns the greatest difference - in terms of weighted delayed jobs - compared to our schedule π, i.e. the *adversarial schedule* σ defined as:

$$\min_\sigma F(\sigma, \boldsymbol{p})$$

Therefore, using the **p** scenario and the π schedule, the *regret* is formulated as follows:

$$R(\pi, \boldsymbol{p}) = F(\pi, \boldsymbol{p}) - \min_\sigma F(\sigma, \boldsymbol{p})$$

The worst time-scenario for schedule π is obtained by finding the time vector **p** which, among the infinite ones, maximizes the regret value R(π,**p**): in this way, we found the *maximum regret* for the generic schedule π that we can call Z(π):

$$Z(\pi) = max_{\boldsymbol{p}}\, R(\pi, \boldsymbol{p})$$

As mentioned above, once we get the maximum regret for each possible *n!* schedule, we choose the one with the *lowest* maximum regret value:

$$Z(\pi^*) = min_{\pi}\, Z(\pi)$$

This formulation of the problem, if solved, allows to find an optimal schedule without giving, however, any indication on the computational complexity that derives from it. A sensible observation could be that if with a modest number of jobs, for example *n = 30* jobs, it was necessary to search for the maximum regret on about 2,65E+32 possible schedules, it would take light-years to obtain an optimal solution even if a hypothetical calculator should spend one second for each schedule π.
Therefore, it's important to underline that, although there is an exact formulation of the problem, nothing can be said about its resolutive complexity. Often, to obtain a solution within an acceptable time, a compromise must be created between the search time for a solution and the sub-optimality of the solution found. About that, a solution approach of the problem will be illustrated in Chapter 5.

To conclude the discussion, we observe that this is a *dual* problem, so it's possible to obtain the same result by coming from two different paths: either by trying to minimize the number of late jobs or by trying to maximize the number of jobs on-time. Obviously, each of these paths must be followed by a strict consistency in the modelling of the problem.

Chapter 5
Solution Approach

In Chapter 4, a mathematical approach and a logical structure of the starting problem have been provided. In this chapter, instead, we will introduce a solution approach by describing the theoretical and practical tools used.

5.1 Maximum Regret Subproblem

Summarizing the structure of the starting problem, we said that (in order to find an exact solution) it's necessary to identify the *maximum regret* for each of the possible schedules π and finally choose the sequence with the *lowest* maximum regret.
In this way, we can ideally divide our problem into two problems nested one inside the other: the first, given a schedule π, deals with finding its maximum regret value; the second (the *nominal problem* - because it doesn't present uncertain data) will have to return, in some way, the sequence with the smallest maximum regret value.

As mentioned in Chapter 1, there are particular problems in which the space of solutions is composed of infinite elements and, therefore, it's not possible to solve them through Combinatorial Optimization; in fact, in Maximum Regret Subproblem, the time-vector $\mathbf{p}$ which maximizes the distance between the weighted number of jobs on-time of a generic schedule π and the weighted number of jobs on-time in its adversarial solution contains real values and thus generates an infinite set of solutions; in other words, it's impossible to investigate all time-vectors, each of which identifies a possible scenario.

To solve this subproblem, we will use a solution approach capable of returning optimal solutions even in case of continuous decision variables: the *Mixed-Integer Programming* (MIP).
In this way, we will formulate a mixed-integer programming model that outputs the maximum regret value (with relative time-scenario) for a generic schedule π given in input the number n of jobs to be processed, their weights w_j, their ranges of values for processing times $[p_j^{min}; p_j^{max}]$ and, above all, the π schedule considered. This is a model that, in theory, must be solved for each of the possible schedules:

$$max \quad \sum_j w_j(z_j - y_j) \tag{1}$$

$$s.t.$$

$$\sum_j v_j \leq d \qquad \forall j \tag{2}$$
$$p_j + p_j^{max} z_j - v_j \leq p_j^{max} \qquad \forall j \tag{3}$$

$$p_j^{min} \leq p_j \leq p_j^{max} \qquad \forall j \tag{4}$$
$$z_j, y_j \in \{0,1\} \qquad \forall j \tag{5}$$

$$\sum_{i=1}^{k} p_{\pi(i)} \geq d\left(1 - y_{\pi(k)}\right) + \varepsilon \qquad \forall k = 1, \dots, n \tag{6}$$

The objective function, as already mentioned, is to maximize the maximum regret value for a schedule π (1). In this model, two binary variables are provided: z_j assumes value 1 if job j is on-time in the adversarial schedule and value 0 otherwise, while y_j assumes value 1 if job j is on-time in schedule π and value 0 otherwise.
The constraints (2) and (3) are a linearization of a fundamental constraint that explain the "rules of the game" to the adversarial

schedule: in particular, a consecutive sequence of jobs is considered *on-time* when its completion-time is lower or equal to due-date; subsequent jobs, in any order, are defined *late*. The constraints (4) and (5) explain the admissibility set of values for each decisional variable – p_j, z_j, y_j for each job j. Finally, the constraint (6) allows to establish if the job j at the position k in the schedule π, given a certain time scenario, is late; it's important to note that the value of the objective function increases whenever it's possible to send a job k late in the schedule π, and this happens whether the sum of the processing times of the first k jobs exceeds the due-date d.

For solution development, this mathematical model has been translated into *Python* programming language and subsequently solved by using a commercial solver for MILP (mixed-integer linear programming) called *The Gurobi Optimizer*, while the user interface that manages the code in Python is that of *JupyterLab* produced by *Project Jupyter*.

The complete code for this subproblem is available in Appendix at the end of this document (**Exhibit 1**), where $n = 30$ jobs with unit weights $w_j = 1$ were used for simplicity; the values of p_j^{min} and p_j^{max} for each job and the sequence π (vector *x1*) have been inserted randomly; the due-date is equal to *500*.
The results of this model, with exemplary input data, are visible in *Fig. 8*: the first column represents the time-scenario calculated by the solver with processing times for each job; the second column indicates which jobs, with that time-scenario, can be completed before d and which ones are delayed; the third column, similarly, indicates which jobs, the adversarial schedule, is able to complete on-time and which ones are late.

Process Times Scenario	Current Schedule	Adversarial Schedule
Job 30 : 60.0 min	Job 30 : 1	Job 30 : -0
Job 28 : 58.0 min	Job 28 : 1	Job 28 : -0
Job 26 : 56.0 min	Job 26 : 1	Job 26 : -0
Job 24 : 54.0 min	Job 24 : 1	Job 24 : -0
Job 22 : 52.0 min	Job 22 : 1	Job 22 : -0
Job 20 : 50.0 min	Job 20 : 1	Job 20 : 1
Job 1 : 31.0 min	Job 1 : 1	Job 1 : 1
Job 3 : 29.0 min	Job 3 : 1	Job 3 : 1
Job 5 : 35.0 min	Job 5 : 1	Job 5 : 1
Job 7 : 37.0 min	Job 7 : 1	Job 7 : 1
Job 9 : 39.0 min	Job 9 : -0	Job 9 : 1
Job 21 : 22.0 min	Job 21 : -0	Job 21 : 1
Job 23 : 24.0 min	Job 23 : -0	Job 23 : 1
Job 25 : 26.0 min	Job 25 : -0	Job 25 : 1
Job 27 : 28.0 min	Job 27 : -0	Job 27 : 1
Job 29 : 30.0 min	Job 29 : -0	Job 29 : -0
Job 10 : 11.0 min	Job 10 : -0	Job 10 : 1
Job 12 : 13.0 min	Job 12 : -0	Job 12 : 1
Job 14 : 15.0 min	Job 14 : -0	Job 14 : 1
Job 16 : 17.0 min	Job 16 : -0	Job 16 : 1
Job 18 : 19.0 min	Job 18 : -0	Job 18 : 1
Job 17 : 18.0 min	Job 17 : -0	Job 17 : 1
Job 15 : 16.0 min	Job 15 : -0	Job 15 : 1
Job 13 : 14.0 min	Job 13 : -0	Job 13 : 1
Job 11 : 12.0 min	Job 11 : -0	Job 11 : 1
Job 2 : 3.0 min	Job 2 : -0	Job 2 : 1
Job 6 : 7.0 min	Job 6 : -0	Job 6 : 1
Job 4 : 5.0 min	Job 4 : -0	Job 4 : 1
Job 8 : 9.0 min	Job 8 : -0	Job 8 : 1
Job 19 : 20.0 min	Job 19 : -0	Job 19 : 1

LEGEND
1 On Time
0 Late

Maximum Regret: 14.0

Fig. 8: Results from the Maximum Regret Subproblem

Therefore, we can say that, with the worst-time-scenario (first column), there is a difference of *14*, in terms of job on-time, between the schedule π and its adversarial schedule, and this represents the greatest regret for π. Sure, you have to be unlucky for those processing times to occur, but as we can see the maximum regret value is an indicator of the sensitivity of the schedule π to the variability of the time-scenario: if with another sequence the

maximum regret had been *5*, obviously we would have concluded that, between the two schedules, the second is more reliable since a smaller error is made even if its worst-time-scenario occurs.
For this reason, and as previously mentioned, it would be interesting to calculate the maximum regret for each possible schedule π and select the one (π*) with the minimum maximum regret value: it would represent the solution to our problem, the most robust sequence and the "less scared" schedule (than the others) by a possible worst-time-scenario.

5.2 The Nominal Problem

After providing a precise solution for the *maximum regret* subproblem, let's face the nominal problem of finding the optimal schedule among the *n!* available.

As we can see, it's possible to adopt a solution strategy through Combinatorial Optimization since the set of solutions is composed of a discrete number of elements. This resolutive approach allows to investigate the whole set of feasible solutions to find the optimal sequence that solves our problem, but it's necessary to understand how long this research would take. To get an idea of the computational complexity of this problem let's look at the following table:

Jobs	**Feasible Solutions**	**Time-to-search for an exact solution (0.5 sec/sol.)**
5	120	60 seconds
10	3,628,800	21 days
20	2,43E+18	45,805,922,353 years
30	2.65E+32	…
50	3.04E+64	…

The magnitude of the solution set is very sensitive to the growth of the number *n* of jobs, so the exact search for a solution makes sense for a limited number of jobs. For this reason, it's necessary to find an alternative method to solve the problem.

In order to not abandon the idea of analyzing the entire set of solutions, it's right to think of an efficient algorithm like Branch and Bound which, in a smart way, investigates only the branches of the solution tree that could contain the optimal schedule. Unfortunately, on a higher number of jobs, even the B&B may not guarantee a solution within a reasonable time but only allows to save a not significant time.
As mentioned in Chapter 1, there are solution methods in Combinatorial Optimization which return sub-optimal solutions in sufficiently good times. The idea is to find the right criterion to investigate only on a finite subset of elements to obtain a result which, without scientific evidence being the optimal solution, can still be considered an excellent compromise between computational complexity and time search for a solution.

In our problem, since each schedule is totally independent and there is no reason to privilege some sequences rather than others, we could think of considering a random sample of *k* schedules and applying the *maximum regret subproblem* on each of them; finally, the schedule π^* with the **min-max regret** is extracted.
In order to test the goodness of this solution strategy, some tests on the algorithm's performance are carried out in Chapter 6.

In the Appendix (**Exhibit 2**), the Python code for the implementation of the Nominal Problem is available. The input data are identical to that in Exhibit 1 while a variable *iterations*, which counts the number of random schedules to extract, has been added; note that, for simplicity, *iterations* is equal to *5*.

The results, instead, are visible in *Fig. 9*. As we can see, 5 random schedules have been generated by the solver in which each of them presents its maximum regret value; in this case, *sequence 3* is the one with the lowest maximum regret value and, therefore, is identified as the most robust schedule among those considered.
The reliability of this solution, of course, is directly proportional to the size of the sample chosen: the greater the number of schedules extracted, the greater the probability of approaching the optimal solution; at the same time, as the computational complexity increases, the search time for a solution increases.

```
Iteration 1
Max Regret: 14
[24, 11, 21, 30, 20, 3, 13, 12, 15, 23, 28, 19, 1, 10, 6, 29, 4, 16, 18, 22, 14, 26, 5, 8, 9, 27, 2, 7, 17, 25]


Iteration 2
Max Regret: 14
[19, 23, 11, 15, 22, 14, 10, 25, 12, 29, 13, 2, 18, 24, 1, 28, 16, 21, 27, 5, 9, 3, 20, 30, 7, 17, 4, 6, 8, 26]


Iteration 3
Max Regret: 12
[8, 1, 27, 9, 28, 19, 18, 3, 13, 12, 17, 11, 30, 5, 25, 22, 24, 6, 15, 7, 21, 20, 4, 29, 16, 2, 10, 26, 23, 14]


Iteration 4
Max Regret: 14
[25, 18, 13, 28, 12, 11, 29, 23, 14, 10, 7, 3, 30, 6, 24, 15, 16, 4, 5, 27, 21, 2, 8, 20, 26, 17, 9, 1, 19, 22]


Iteration 5
Max Regret: 14
[11, 13, 20, 24, 9, 17, 30, 18, 25, 2, 10, 19, 6, 3, 4, 22, 27, 26, 7, 23, 8, 21, 29, 16, 5, 15, 12, 28, 14, 1]


-----------------------------------------------------------------------------------------------------------------

Min Max Regret: 12

Best Schedule:
[8, 1, 27, 9, 28, 19, 18, 3, 13, 12, 17, 11, 30, 5, 25, 22, 24, 6, 15, 7, 21, 20, 4, 29, 16, 2, 10, 26, 23, 14]
```

Fig. 9: Results from the Nominal Problem (# of iterations)

A second variant of the solution of the nominal problem is that which provides for the insertion of an execution *time-limit*: in particular, if we cannot get an indication of how many extractions are needed to reach a reliable solution, it's possible to indicate how long the solver will generate random schedules. On the latter, the program will update each time the min-max regret value found among the sequences analyzed up to that moment.
In **Exhibit 3**, the Python code for this second variant is available and, for simplicity, a time-limit equal to *0.2 seconds* has been inserted. Similarly to the previous case, from a logical point of view, the greater the time for extracting the schedules, the greater the probability that the returned solution approaches the optimal one.

The relative results are present in *Fig. 10*, and they can be read as the results in Exhibit 3, since only the solution approach has changed but not the final goal. In this case, in 0.2 seconds, the solver was able to extrapolate *8 different schedules* with the respective maximum regret value; the "best schedule" is the last iteration that present a min-max regret equal to *12*.

In order to test the goodness of this solution strategy, some tests on the algorithm's performance are carried out in Chapter 6.

```
Iteration 1
Max Regret: 15
[29, 18, 28, 21, 20, 27, 4, 12, 1, 2, 5, 24, 15, 23, 19, 11, 8, 10, 26, 13, 6, 17, 16, 14, 30, 3, 9, 22, 7, 25]


Iteration 2
Max Regret: 15
[9, 5, 6, 23, 19, 26, 20, 30, 4, 28, 22, 12, 7, 29, 10, 17, 15, 3, 18, 14, 13, 8, 2, 21, 1, 27, 16, 25, 11, 24]


Iteration 3
Max Regret: 13
[2, 19, 13, 15, 22, 24, 8, 11, 10, 23, 21, 29, 5, 20, 6, 1, 27, 25, 7, 17, 4, 28, 18, 16, 14, 3, 26, 30, 12, 9]


Iteration 4
Max Regret: 17
[16, 22, 29, 27, 21, 24, 6, 26, 30, 2, 1, 12, 9, 23, 8, 13, 15, 11, 19, 14, 28, 10, 3, 17, 5, 4, 18, 7, 20, 25]


Iteration 5
Max Regret: 14
[5, 20, 4, 1, 28, 19, 24, 22, 15, 17, 16, 6, 27, 18, 26, 3, 30, 13, 11, 25, 8, 7, 29, 14, 10, 21, 12, 2, 23, 9]


Iteration 6
Max Regret: 15
[5, 21, 22, 2, 18, 8, 27, 20, 19, 26, 10, 7, 12, 30, 3, 9, 17, 4, 15, 13, 29, 24, 11, 25, 1, 16, 6, 28, 14, 23]


Iteration 7
Max Regret: 15
[20, 11, 26, 15, 17, 10, 12, 22, 25, 9, 2, 5, 16, 4, 28, 3, 30, 24, 23, 13, 7, 14, 21, 8, 1, 6, 18, 19, 27, 29]


Iteration 8
Max Regret: 12
[19, 3, 2, 12, 18, 5, 29, 16, 8, 7, 17, 24, 20, 14, 28, 4, 22, 1, 11, 23, 21, 9, 15, 27, 10, 6, 26, 25, 13, 30]


----------------------------------------------------------------------------------------------------------------------

Min Max Regret: 12

Best Schedule:
[19, 3, 2, 12, 18, 5, 29, 16, 8, 7, 17, 24, 20, 14, 28, 4, 22, 1, 11, 23, 21, 9, 15, 27, 10, 6, 26, 25, 13, 30]
```

Fig. 10: Results from the Nominal Problem (Time-limit)

Chapter 6
Performance Tests

Once the algorithm that aims to solve the initial problem has been implemented, it's necessary to carry out a testing phase to understand if this solution approach can be considered sufficiently robust. In fact, until now, a *smart* method to quickly reach a sub-optimal solution has been hypothesized, but only theoretically: there is still no tool that indicates or does not indicate the reliability of this solution strategy. A testing phase is essential to provide a yardstick on the algorithm performance and, in particular, it will indicate how much the program is able to return a sufficiently robust solution despite the variables that represent the jobs processing time p_j are uncertain.

6.1 Test Structure and Results

The structure envisaged for testing phase must be able to measure the algorithm robustness by changing the value of some fundamental variables. In particular, it's important to observe how the solution approach, within a generic instance, responds when the number n of jobs increases but, above all, to observe its behavior as iterations increase, that is the number of random schedules extracted from the entire set of solutions. Performance results deriving from *time-limit* executions have also been included in the test structure.

Before explaining the testing phase model and its basic logic, tables with the results obtained are shown below (*Fig. 11, 12, 13*) :

n = 50	w = 1	Inst. 1	10	14
			100	13
			1000	13
			T.L.	13
		Inst. 2	10	14
			100	13
			1000	13
			T.L.	13
		Inst. 3	10	13
			100	12
			1000	13
			T.L.	12
		Inst. 4	10	14
			100	13
			1000	13
			T.L.	13
		Inst. 5	10	13
			100	12
			1000	12
			T.L.	13
	w != 1	Inst. 1	10	37
			100	35,5
			1000	34,8
			T.L.	35,1
		Inst. 2	10	46,6
			100	42,3
			1000	43,7
			T.L.	44,4
		Inst. 3	10	34,9
			100	34,5
			1000	33,9
			T.L.	34,3
		Inst. 4	10	42,3
			100	38,6
			1000	37,6
			T.L.	37
		Inst. 5	10	42,6
			100	40,9
			1000	40,9
			T.L.	40,3

TIME LIMIT = 1 min
d = 80

n = 100	w = 1	Inst. 1	10	16
			100	15
			1000	15
			T.L.	15
		Inst. 2	10	15
			100	14
			1000	14
			T.L.	14
		Inst. 3	10	17
			100	16
			1000	16
			T.L.	16
		Inst. 4	10	16
			100	15
			1000	15
			T.L.	15
		Inst. 5	10	14
			100	14
			1000	14
			T.L.	14
	w != 1	Inst. 1	10	50,6
			100	49,7
			1000	47,9
			T.L.	48,1
		Inst. 2	10	57,2
			100	51,1
			1000	53
			T.L.	54,2
		Inst. 3	10	51,7
			100	49,6
			1000	48,7
			T.L.	50
		Inst. 4	10	51,1
			100	49,3
			1000	47,5
			T.L.	48,7
		Inst. 5	10	51,3
			100	50
			1000	47,3
			T.L.	48,1

TIME LIMIT = 2 min
d = 80

Fig. 11: Test structure with n = 50 jobs and n = 100 jobs

n = 150	w = 1	Inst. 1	10	17
			100	16
			1000	16
			T.L.	15
		Inst. 2	10	16
			100	16
			1000	16
			T.L.	16
		Inst. 3	10	16
			100	16
			1000	16
			T.L.	16
		Inst. 4	10	16
			100	16
			1000	16
			T.L.	16
		Inst. 5	10	17
			100	16
			1000	16
			T.L.	16
	w != 1	Inst. 1	10	52,6
			100	51,9
			1000	48,8
			T.L.	51,4
		Inst. 2	10	58,8
			100	56,7
			1000	55,6
			T.L.	55,8
		Inst. 3	10	56,4
			100	52,9
			1000	51,9
			T.L.	53,5
		Inst. 4	10	54,5
			100	52,7
			1000	51,8
			T.L.	50,6
		Inst. 5	10	58,7
			100	54,5
			1000	52,4
			T.L.	52,7

TIME LIMIT = 3 min
d = 80

n = 200	w = 1	Inst. 1	10	17
			100	17
			1000	17
			T.L.	17
		Inst. 2	10	17
			100	16
			1000	16
			T.L.	16
		Inst. 3	10	17
			100	17
			1000	16
			T.L.	17
		Inst. 4	10	17
			100	17
			1000	17
			T.L.	17
		Inst. 5	10	17
			100	17
			1000	16
			T.L.	17
	w != 1	Inst. 1	10	62,7
			100	60,5
			1000	57,9
			T.L.	60,6
		Inst. 2	10	55,1
			100	52,7
			1000	52,4
			T.L.	53
		Inst. 3	10	55
			100	54,6
			1000	53
			T.L.	53,6
		Inst. 4	10	53
			100	51,8
			1000	51,7
			T.L.	51,2
		Inst. 5	10	59
			100	56,1
			1000	53,6
			T.L.	53,1

TIME LIMIT = 4 min
d = 80

Fig. 12 : Test structure with n = 150 jobs and n = 200 jobs

n = 300	w = 1	Inst. 1	10	18
			100	17
			1000	17
			T.L.	17
		Inst. 2	10	18
			100	17
			1000	17
			T.L.	17
		Inst. 3	10	17
			100	17
			1000	17
			T.L.	17
		Inst. 4	10	18
			100	17
			1000	17
			T.L.	17
		Inst. 5	10	18
			100	17
			1000	17
			T.L.	17
	w != 1	Ist. 1	10	66.8
			100	63,2
			1000	62,3
			T.L.	60,3
		Inst. 2	10	62,8
			100	59,6
			1000	60,1
			T.L.	60,6
		Inst. 3	10	62,3
			100	63,2
			1000	60,2
			T.L.	58,5
		Inst. 4	10	63,4
			100	62,6
			1000	60,3
			T.L.	62,2
		Inst. 5	10	58,9
			100	58,6
			1000	56,9
			T.L.	57,8

TIME LIMIT = 5 min
d = 80

n = 500	w = 1	Inst. 1	10	18
			100	17
			1000	17
			T.L.	17
		Inst. 2	10	18
			100	18
			1000	18
			T.L.	18
		Inst. 3	10	18
			100	17
			1000	17
			T.L.	17
		Inst. 4	10	18
			100	17
			1000	17
			T.L.	17
		Inst. 5	10	18
			100	18
			1000	18
			T.L.	18
	w != 1	Inst. 1	10	67,96
			100	65,42
			1000	65,44
			T.L.	65,29
		Inst. 2	10	68,67
			100	67,24
			1000	63,67
			T.L.	66,82
		Inst. 3	10	62,9
			100	61,98
			1000	60,27
			T.L.	62,56
		Inst. 4	10	61,28
			100	61,93
			1000	61,07
			T.L.	60,88
		Inst. 5	10	77,73
			100	74,69
			1000	74,65
			T.L.	75,4

TIME LIMIT = 6 min
d = 80

Fig. 13: Test structure with n = 300 jobs and n = 500 jobs

In particular, a test was performed on six different values of the number of jobs : $n = 50$, $n = 100$, $n = 150$, $n = 200$, $n = 300$ and $n = 500$. For each of them, the *unweighted* case and the *weighted* case are considered : in the first one, $w = 1$, all jobs have unit weight and, therefore, have the same level of importance ; in the second one, instead, $w != 1$, all jobs can assume different weights and priorities. For both the unweighted case and the weighted case, five different instances have been generated, which individually include four different tests : extraction of 10, 100, 1,000 random schedules and a time-limit execution (T.L.). Within the single instance (e.g. n = 50, w = 1, Inst. 1), all four tests are performed on the same input data ; therefore, it's possible to have an evidence of performance results based on the number of iterations and on the extraction method used. Input data (i.e. values of p_j^{min}, p_j^{max} and w_j of each job), for each of the 60 instances, were generated randomly respecting only some generic compliance constraints; for this reason, the input structure represents a database that contains starting data to perform each type of test. As we can see, there is a different time-limit for each value of the number of jobs: this is because as n increases, the computional complexity of the problem increases and the solver requires more time to generate an acceptable solution. The due-date, for semplicity, is always equal to 80 and the results, in terms of *min-max regret* returned, are shown in red.

The single instances, while sharing some structural characteristics, are considered indipedent of each other; so, it's possible to test the algorithm performance based on a comparison between the results of the four tests within the same instance (10, 100, 1,000 iterations and the time-limit execution T.L.) and between instances with same n and same case ($w = 1$ or $w != 1$).

6.1 Result Analysis

As regards the **unweighted case**, results are summarized in three graphs in *Fig. 14*. For semplicity, only cases with $n = 50$, $n = 200$ and $n = 500$ were considered.

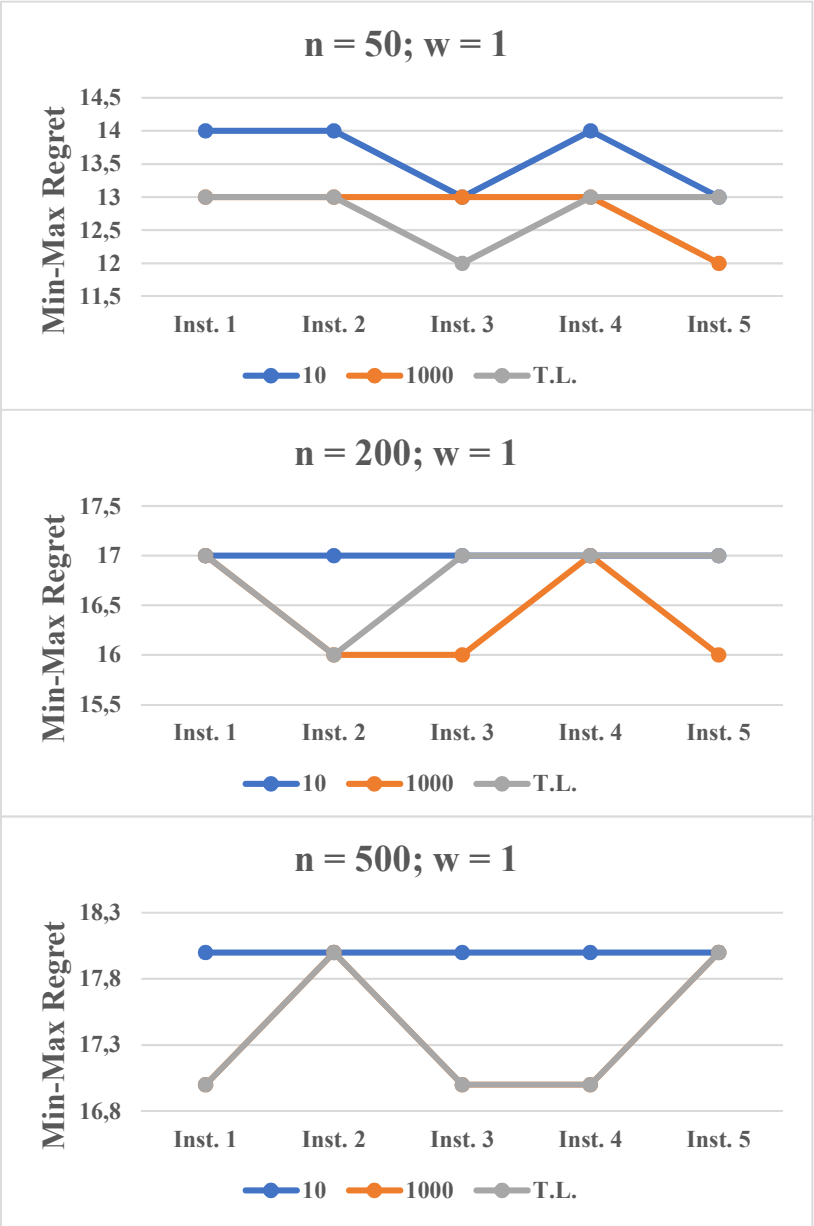

Fig. 14: Results from the Unweighted Case

In order to have a better view of differences in performance, the intermediate case with 100 extractions was excluded from the graphs: therefore, three broken lines represent the point values of *min-max regret* respectively with 10 iterations (blue), 1,000 iterations (orange) and with a time-limit based on the number of jobs (grey). As previously mentioned, for each category of *n*, performances can be compared within a single instance (vertical difference in terms of min-max regret) and between the five instances of the same "group" (comparison of the height of the lines). Naturally, given the problem nature, the best line is the one that is lower than the others; then, in general, the one that returns best solutions. In fact, increasing the number of extractions improves, on average, the solution quality (comparison between blue lines and orange lines). In all cases, however, time-limit extractions return solutions that sometimes coincide with those deriving from an extraction of 1,000 schedules (i.e. third graph).
Therefore, in the unweighted case, no excessive large differences are obtained in terms of objective function by incresing the number of iterations or even by increasing the number of jobs *n*; so, it's likely that a solution obtained with 50 extractions is very close to a solution obtained with 2,000 extractions and much more time is saved. In this way, it's possible (on average) to obtain a good solution without investigating an excessive number of schedules. It's important to underline that by increasing the number of iterations, a higher quality solution is still more likely to be obtained and that using fewer extractions doesn't ensure the same results; only a sub-optimal solution that is very close to the best one is guaranteed in less time. As regarding the distance between the best solution obtained and the real optimal solution, it's possible to see that within the single instance all values seem to approach a lower bound which however is unkown. Since the variability in min-max regret values remains very low increasing the number of iterations, it's probably that the optimal solution has

been reached or that it's very close. Obviously, there can always be a single and isolate optimality case that is distant from the other sub-optimal values.

Even in the **weighted case**, results are summarized in three graphs (*Fig. 15*).

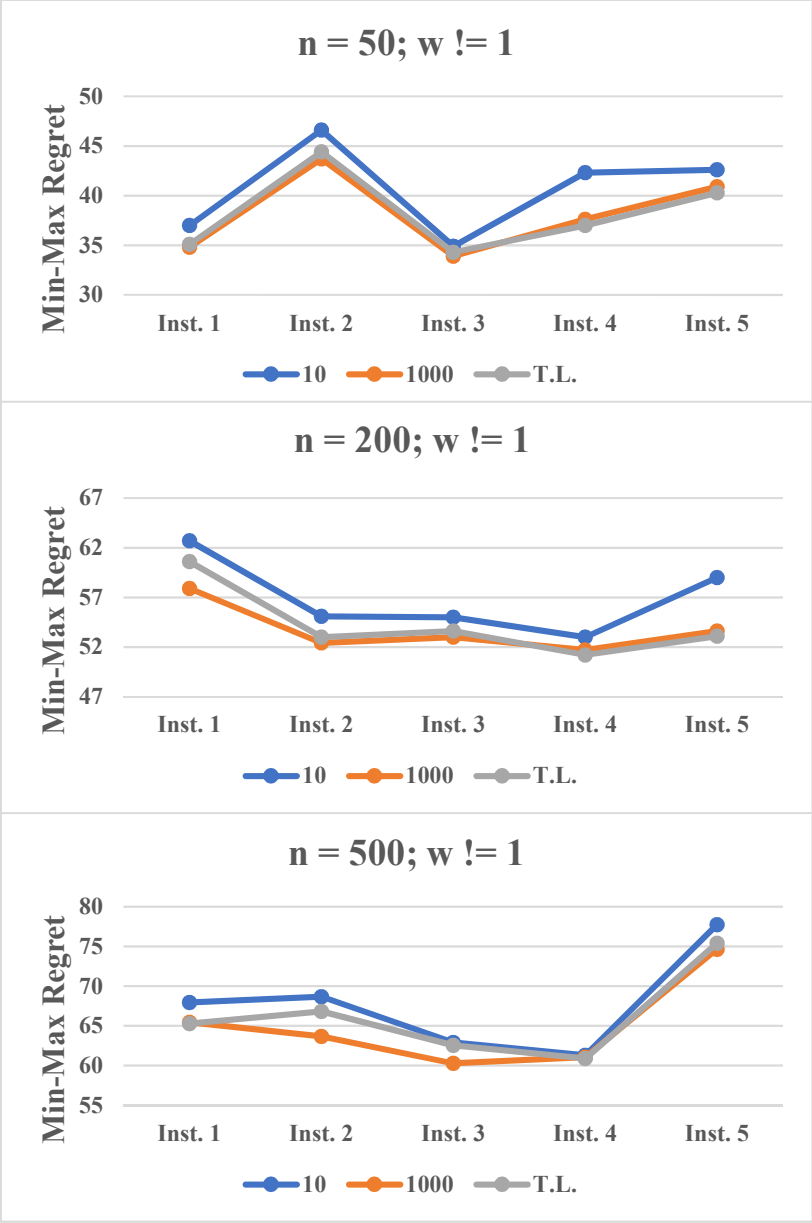

Fig. 15: Results from the Weighted Case

As in the previous case, best solutions seem to be those extracted with a number of iterations equal to 1,000 (orange line). The time-limit approach (grey line) returns excellent results comparable to the best ones and, for this reason, it's established as a very reliable solution method.
In the weighted case, jobs have a value of importance w_j which can also assume decimal numbers (e.g. *2.15*). In this way, the difference in terms of min-max regret is more sensitive and more visible graphically within the single instance and between instances of the same type. Again, by increasing the number of jobs n or the number of iterations, there isn't a significant difference in results between extractions with 10 random schedules and extractions with 1,000 random schedules; even time-limit solutions do not seem to differ excessively from those obtained with the other methods. Obviously, as in the unweighted case, as the number of iterations increases, on average, the solution quality increases and moreover, by inserting *decimal weights* into the objective function, it's possible to see how a solution methodology is significantly better (or less) than the others.
In conclusion, the solution proposal shows good robustness. In fact, by extracting a defined number of iterations (in the specific cases) or an indefinite number of iterations (extractions with time-limit), results are very *close* to each other and, in the unweighted case, sometimes they coincide between different solution methods. Naturally, not knowing the real optimal solution, there is no measure of the quality of the solution obtained but, as previously mentioned, the different approaches in testing phase seem to converge towards a lower bound that we can hypothesize as a "minimum point". In all instances, a min-max regret value extracted from 10 schedules generally has a performance similar to that obtained with 1,000 schedules, and for this reason it's not necessary to carry out an exorbitant number of iterations to achieve

a sufficiently acceptable result; therefore, the trade-off between search time for a solution and solution quality is very limited.

Chapter 7
Conclusions

Combinatorial optimization problems, in general, are often treated superficially and, in some cases, even incorrectly. The issue of *uncertainty* plays a fundamental role in the modeling of some structural elements, such as parameters, objective functions and decision-making variables. The idea behind this thesis was to provide a slightly different treatment of a problem that, in general, is defined and solved assuming that all factors involved are deterministic. Forcing uncertain processing times for each job is a strong assumption, which, as seen in the previous paragraphs, leads to a completely different solution approach, but at the same time describes a situation that, in everyday life, is almost likely. Obviously, uncertainty is an omnipresent factor and it is essential to understand where it is possible to live with it and where, instead, it is not possible to ignore: in this case, the experience of someone that lives every day with certain types of problems is fundamental. The robustness issue is introduced in those problems where, in fact, relevant uncertain factors are present. The proposed solution approach do not represent an exact solution method but an approximate method which, in some way, are able to return a solution that is sufficiently good in a reasonable time. In this way, we were able to create a robust and useful tool to support certain decisions, demonstrating at the same time its effectiveness, its timeliness and, above all, its robustness.

Bibliography

Drwal e Józefczyk (2019). Robust min–max regret scheduling to minimize the weighted number of late jobs with interval processing times - Faculty of Computer Science and Management, Wroclaw University of Science and Technology

C. Blum, A. Roli (2003). Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison – Université Libre de Bruxelles, Università degli studi di Bologna

J. García, A. Peña (2018) - Robust Optimization Concepts and Applications

Paolo Serafini (2009). Algoritmi euristici – Springer, Milano

L. De Giovanni. Metodi e Modelli per l'Ottimizzazione Combinatoria – Università degli Studi di Padova

E. Aarts and J.K. Lenstra (1997). Local Search in Combinatorial Optimization - J. Wiley & Sons

Appendix

Exhibit 1 – Python Code for Maximum Regret Subproblem

```
from gurobipy import *
from mip import *
import random

pmin=[2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31]
pmax=[31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60]
w=[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]

d=500
eps=0.01

x1=[30,28,26,24,22,20,1,3,5,7,9,21,23,25,27,29,10,12,14,16,18,17,15,13,11,2,6,4,8,19]
n=len(x1)

p=list(range(0,n))
z=list(range(0,n))
y=list(range(0,n))
v=list(range(0,n))

pmin2=list(range(0,n))
pmax2=list(range(0,n))
w2=list(range(0,n))
x2=list(range(0,n))

for i in x2:
    x2[i]=x1[i]-1

for i in pmin2:
    pmin2[i]=pmin[x2[i]]

for i in pmax2:
    pmax2[i]=pmax[x2[i]]

for i in w2:
    w2[i]=w[x2[i]]

m = Model()

i=0
for elements in p:
    p[i]=m.add_var(lb=pmin2[i],ub=pmax2[i],var_type=CONTINUOUS)
    i+=1

i=0
for elements in z:
    z[i]=m.add_var(var_type=BINARY)
    i+=1

i=0
for elements in y:
    y[i]=m.add_var(var_type=BINARY)
    i+=1

i=0
for elements in v:
    v[i]=m.add_var(lb=0,var_type=CONTINUOUS)
    i+=1
```

```
sum_z2=list(range(0,n))
sum_y2=list(range(0,n))

i=0
for elements in w2:
    sum_z2[i]=z[i]*w[i]
    i+=1

i=0
for elements in w2:
    sum_y2[i]=y[i]*w[i]
    i+=1

sum_z=sum(sum_z2)
sum_y=sum(sum_y2)
fo=sum_z-sum_y

m.objective = maximize(fo)

m += sum(v)<=d

i=0
for elements in x2:
    m += (p[i]+pmax2[i]*z[i]-v[i])<=pmax2[i]
    i+=1

k1=list(range(0,n))
k1[0]=p[0]

i=1
while i<=n-1:
    k1[i]=k1[i-1]+p[i]
    i+=1

i=0
for elements in k1:
    m += k1[i]>=((d*(1-y[i]))+eps)
    i+=1

m.optimize()

print('Maximum Regret: {}\n'.format(m.objective_value))

print('Process Times Scenario')

i=0
for v in p:
    print('Job {} : {} min'.format(x1[i],v.x))
    i+=1

print('\nLEGEND\n1 On Time\n0 Late')
print('\nAdversarial Scenario')

i=0
k=n
for elements in p:
    print('Job %g : %g' % (x1[i],m.vars[k]))
    i+=1
    k+=1

print('\nCurrent Scenario')

i=0
k=2*n
for elements in p:
    print('Job %g : %g' % (x1[i],m.vars[k]))
    i+=1
    k+=1
```

Exhibit 2 – Python Code for The Nominal Problem (# of iterations)

```
from gurobipy import *
from mip import *
import random
import time

n=30

pmin=[2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31]
pmax=[31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60]
w=[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]

d=500
eps=0.01

iterations=5
cont=0

min_maxregret=1000000
best_schedule=list(range(0,n))

process_time=time.time()

m = Model()

while cont<iterations:

    x0=list(range(1,n+1))
    x1=list(range(0,n))
    for elements in x1:
        x1[elements]=0

    i=0

    for elements in x1:
        x1[i]=random.choice(x0)
        u=x1[i]
        x0.remove(u)
        i+=1

    p=list(range(0,n))
    z=list(range(0,n))
    y=list(range(0,n))
    v=list(range(0,n))

    pmin2=list(range(0,n))
    pmax2=list(range(0,n))
    w2=list(range(0,n))
    x2=list(range(0,n))

    for i in x2:
        x2[i]=x1[i]-1

    for i in pmin2:
        pmin2[i]=pmin[x2[i]]

    for i in pmax2:
        pmax2[i]=pmax[x2[i]]
```

```
    for i in w2:
        w2[i]=w[x2[i]]

    i=0
    for elements in p:
        p[i]=m.add_var(lb=pmin2[i],ub=pmax2[i],var_type=CONTINUOUS)
        i+=1

    i=0
    for elements in z:
        z[i]=m.add_var(var_type=BINARY)
        i+=1

    i=0
    for elements in y:
        y[i]=m.add_var(var_type=BINARY)
        i+=1

    i=0
    for elements in v:
        v[i]=m.add_var(lb=0,var_type=CONTINUOUS)
        i+=1

    sum_z2=list(range(0,n))
    sum_y2=list(range(0,n))

    i=0
    for elements in w2:
        sum_z2[i]=z[i]*w[i]
        i+=1

    i=0
    for elements in w2:
        sum_y2[i]=y[i]*w[i]
        i+=1

    sum_z=sum(sum_z2)
    sum_y=sum(sum_y2)
    fo=sum_z-sum_y

    m.objective = maximize(fo)

    m += sum(v)<=d

    i=0
    for elements in x2:
        m += (p[i]+pmax2[i]*z[i]-v[i])<=pmax2[i]
        i+=1

    k1=list(range(0,n))
    k1[0]=p[0]

    i=1
    while i<=n-1:
        k1[i]=k1[i-1]+p[i]
        i+=1
    i=0
    for elements in k1:
        m += k1[i]>=((d*(1-y[i]))+eps)
        i+=1

    m.optimize()

    u=m.objective_value
    print("Iteration %g" % (cont+1))
    print("Max Regret: %g" % u)
    print(x1)
    print("\n")

    if u<min_maxregret:
        min_maxregret=u
        best_schedule=x1

    cont+=1
```

Exhibit 3 – Python Code for The Nominal Problem (Time-limit)

```
from gurobipy import *
from mip import *
import random
import time
from threading import Thread, Event

stop_event=Event()

def minmaxregret_problem():

    n=30

    pmin=[2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31]
    pmax=[31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60]
    w=[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]

    d=400
    eps=0.01

    iterations=0
    cont=10

    min_maxregret=1000000
    best_schedule=list(range(0,n))

    process_time=time.time()

    m = Model()
    while cont>iterations:

        x0=list(range(1,n+1))
        x1=list(range(0,n))

        for elements in x1:
            x1[elements]=0

        i=0

        for elements in x1:
            x1[i]=random.choice(x0)
            u=x1[i]
            x0.remove(u)
            i+=1

        p=list(range(0,n))
        z=list(range(0,n))
        y=list(range(0,n))
        v=list(range(0,n))

        pmin2=list(range(0,n))
        pmax2=list(range(0,n))
        w2=list(range(0,n))
        x2=list(range(0,n))

        for i in x2:
            x2[i]=x1[i]-1
```

```
for i in pmin2:
    pmin2[i]=pmin[x2[i]]

for i in pmax2:
    pmax2[i]=pmax[x2[i]]

for i in w2:
    w2[i]=w[x2[i]]

i=0
for elements in p:
    p[i]=m.add_var(lb=pmin2[i],ub=pmax2[i],var_type=CONTINUOUS)
    i+=1

i=0
for elements in z:
    z[i]=m.add_var(var_type=BINARY)
    i+=1

i=0
for elements in y:
    y[i]=m.add_var(var_type=BINARY)
    i+=1

i=0
for elements in v:
    v[i]=m.add_var(lb=0,var_type=CONTINUOUS)
    i+=1

sum_z2=list(range(0,n))
sum_y2=list(range(0,n))

i=0
for elements in w2:
    sum_z2[i]=z[i]*w[i]
    i+=1

i=0
for elements in w2:
    sum_y2[i]=y[i]*w[i]
    i+=1

sum_z=sum(sum_z2)
sum_y=sum(sum_y2)
fo=sum_z-sum_y

m.objective = maximize(fo)

m += sum(v)<=d

i=0
for elements in x2:
    m += (p[i]+pmax2[i]*z[i]-v[i])<=pmax2[i]
    i+=1

k1=list(range(0,n))
k1[0]=p[0]

i=1
while i<=n-1:
    k1[i]=k1[i-1]+p[i]
    i+=1

i=0
for elements in k1:
    m += k1[i]>=((d*(1-y[i]))+eps)
    i+=1
```

```
        m.optimize()

        u=m.objective_value
        print("Iteration %g" % (cont-9))
        print("Max Regret: %g" % u)
        print(x1)
        print("\n")

        if u<min_maxregret:
            min_maxregret=u
            best_schedule=x1

        cont+=1
        if stop_event.is_set():
            break

    print("-------------------------------------------------------------------------------------------------------")
    print('Min Max Regret: %g' % min_maxregret)
    print('\nBest Schedule:')
    print(best_schedule)
    print("\n%g seconds" % (time.time()-process_time))

if __name__ == '__main__':
    action_thread = Thread(target=minmaxregret_problem)
    action_thread.start()
    action_thread.join(timeout=0.2)
    stop_event.set()
```