Part III
Scheduling Methods

The Part III of the book consists of Chaps. 7–10 and is devoted to scheduling methods. Here we will review the main approaches related to solution procedures of scheduling models. We emphasize general approaches rather than specific solutions, and provide a general scheme to build and validate new procedures for scheduling problems. We pay attention to the integration of heuristic and exact approaches and devote a chapter to multiobjective procedures.

Chapter 7
Overview of Scheduling Methods

7.1 Introduction

In the previous part of the book, we have presented the concept of a scheduling model as a way to formalise the decision-making scheduling problem. This part of the book is devoted to present the methods to provide (good or even optimal) solutions for these scheduling models. In this chapter, we give an overview of scheduling methods, leaving for the next chapters the detailed discussion of specialised methods.

After a few basic definitions related to scheduling methods (most notably, the concept of algorithm), we discuss, in a rather intuitive way, why scheduling problems are tremendously hard to solve to optimality, thus giving a rationale for the traditional use of scheduling policies, which try to ease these difficulties at the expense of an unimpressive performance, at least as compared with 'true' scheduling algorithms. We also give an overview on the main types of scheduling algorithms, and discuss how the adequacy of a scheduling method for solving a manufacturing scheduling decision problem can be assessed.

More specifically in this chapter, we

- provide the main definitions regarding scheduling methods, together with a dicussion on the main assumptions behind these (Sect. 7.2),
- discuss the concept of computational complexity and its consequences when designing manufacturing scheduling methods (Sect. 7.3),
- review the main scheduling policies along with their rationale (Sect. 7.4),
- present an overview of the main types of scheduling algorithms (Sect. 7.5) and
- introduce the principal issues related to the (formal and real-world) assessment of manufacturing scheduling methods (Sect. 7.6).

J. M. Framinan et al., *Manufacturing Scheduling Systems*,
DOI: 10.1007/978-1-4471-6272-8_7, © Springer-Verlag London 2014

7.2 Basic Concepts

In this section, we provide the standard definitions of scheduling methods and related terms. It is important to note that these terms refer exclusively to formal/computational aspects for solving mathematical models, and cannot be, in general, translated into the real-world decision problem in a straightforward manner. Therefore, along with these definitions, we provide a list of the main implicit and explicit assumptions behind, and leave for a later section (Sect. 7.6) the discussion on assessing the adequacy of a scheduling method for solving a real-world scheduling decision problem.

7.2.1 Formal Definitions

In the context of this book, a scheduling method is a formal procedure that can be applied to any instance of a scheduling model in order to obtain a feasible schedule that (presumably) obtains good scores with respect to a sought objective. Note that, according to Chap. 3, a scheduling model is assumed to be a representation of a well-defined manufacturing scheduling decision problem. Therefore, since the procedure can be applied to any instance of a scheduling model, it can be said (rather informally, as it has been noted in several places throughout the book) that a scheduling method *solves* a scheduling problem.

A scheduling method can thus be viewed as a procedure taking an instance of a scheduling model as an input in order to produce (at least) one schedule as an output. Since the procedure has to be formal, it can be described using a finite number of steps, and it is thus amenable to be coded and eventually executed by a computer. The technical name for such finite procedure is *algorithm*.

It is customary to classify algorithms into exact and approximate. Exact algorithms guarantee that no other schedule performs better than the one obtained with respect to the objective sought. The so-obtained solution is named *optimum* or *optimal solution*. In contrast, approximate algorithms do not guarantee that the solution is optimal although, in some cases, we will see that it is possible to estimate the maximum deviation from the optimum.

7.2.2 Assumptions

As mentioned above, there are several explicit and implicit assumptions in the previous definition of scheduling methods that are worth to be discussed:

1. Well-defined scheduling model. As already mentioned in Chap. 2, sometimes real-world manufacturing scheduling problems lack of precise description. Machine availability considerations, rework and many other aspects are excluded from the

model formulation. So it might be that some of the constraints of the problem are not included into the definition of the model or are even not known to the decision maker although they might be relevant for the real-world problem's solution. Relative to this exclusion/ignorance, the terminus 'solution' may become inadequate.

2. Formal model. It is supposed that the problem is given as a mathematically strictly formulated formal model. However, as already mentioned in Chap. 2 as well, these formal models always represent a more or less significant simplification of the real-world problem. Therefore, deriving a solution for this formal model, even if it is an optimal one, means nothing more than deriving an approximate solution for the respective real-world problem. Hence, with respect to the real-world problem, there will be neither an exact problem formulation nor an exact solution procedure or an exact solution in the strict sense. For the moment, we will confine our perspective to the formal sphere of the problem under consideration and later, in Sect. 7.6 we will address the assessment of the solutions provided by the formal model with respect to the real-world problem.
3. Objectives. Usually, the formal model intends to derive the best feasible solution of a manufacturing scheduling problem. However, there are many other plausible types of objective functions, which could represent the goals of the formal problem, e.g.

   - Generation of a feasible solution. This is especially relevant if the set of constraints does not obviously imply feasible solutions.
   - Generation of a good solution. That is, e.g., a solution being at most a certain percentage away from the optimal objective function value or being not worse than some predefined value, etc.
   - 'Dualising' constraints and/or objective functions. Constraints of an 'original' formal model might be dualised to the objective function and vice versa. With respect to the 'exactness' of this dualisation, it is usually supposed that these dualisation steps have been performed before exactness is an issue and are input into the formulation of the formal optimisation model, or these dualisations are part of the (maybe exact) solution procedure itself.
   - Number of solutions. In manufacturing scheduling models, often more than one optimal solution occurs, i.e. the same best (or satisfactory) objective function value is achieved by more than one solution. Most (also exact) solution procedures obtain only one of these solutions. However sometimes, it is required to determine more than one or even all best solutions.

   Note that defining the objectives in the previous manner would mean that an exact procedure would just fulfil these types of requirements or regards the additional model modifications as input, respectively, sometimes even not necessarily looking for an extreme (maximum or minimum) solution.
4. Single objective function. As will be discussed in Chap. 10, and is well known from multi-objective optimisation in general, the 'optimality' in the multi-objective context will often be expressed by some efficiency calculus, usually

by the concept of Pareto set which requires, for an 'exact' procedure, to determine the whole efficient boundary of solutions.

5. (Deterministic) exact procedure. Finally, to be called 'exact', a method for fulfilling the requirements of the formal optimisation problem (i.e. here usually to derive the optimal solution of every possible problem instance for a formal manufacturing scheduling problem) must deliver the solution within a finite (but possibly large) number of computation steps and in a completely deterministic manner. Completely deterministic here means that the result should be identically replicable in another run of the procedure for the same problem instance.
6. Stochastics setting. It should be mentioned that exact procedures might also be defined for stochastic decision problems, including stochastic manufacturing scheduling models. The exact nature of the algorithm is not in conflict with possible stochastic components of the model formulation as long as the criterion of the model is not a random variable. For instance, one may speak of an exact procedure for the $\alpha|\beta|C_{\max}$ model as well as for the $\alpha|\beta|E[C_{\max}]$ model where processing times are random variables.

So, in the sequel of this chapter, we will suppose that we have a precisely formulated (i.e. also completely quantified) manufacturing scheduling problem (e.g. a flow shop problem) with a unique objective function (e.g. makespan) which is intended to be minimised, i.e. one (not all) optimal solution is to be determined by a well-defined deterministic procedure, i.e. an exact procedure. These assumptions will allow us to go forth until Chap. 10, where several objectives would be discussed, and Chap. 14, in which the relaxation of these hypotheses will be discussed in the context of a manufacturing scheduling system.

Once these assumptions have been established, a natural question is why employing approximate algorithms (without performance guarantee) if there are exact methods at hand. The answer lies in the computational complexity of most scheduling models, which will be discussed in the next section, so we will see that approximate algorithms are rather often the only suitable alternative given the long times that some exact procedures take to complete. Indeed, the complexity of most real-world scheduling models is so high that in practice, a particular case of scheduling methods are the so-called scheduling policies (which will be studied in Sect. 7.4) which are widely employed.

7.3 Computational Complexity of Scheduling

Computational complexity theory is a field inside the theory of computation that focuses on classifying *computational problems* according to their underlying structure and difficulty. In our context, the computational problem is the scheduling model at hand, for which we intend to build formal procedures (i.e. algorithms) to find the best solution.

Rather informally, we can define two types of computational complexity:

- Algorithm complexity. Algorithm complexity informs us on the performance of a specific algorithm for solving a model by assigning to it an estimation of its cost (in time). This estimation is known as the computational or time complexity (also referred to as running time). Intuitively, the actual time to run the algorithm will depend, among other factors, on the specific instance of the model for which the algorithm is applied.
- Model complexity. Model complexity informs us about the complexity of the 'best algorithm' able to find the optimal solution this model by classifying the models into *complexity classes*.

It is interesting to note that the complexity of an algorithm can be calculated for any algorithm, either exact or approximate. However, model complexity refers to the complexity of the best exact algorithms to solve the model. In the next section, we will discuss these rather abstract concepts first by introducing an example (Sect. 7.3.1) and then by deepening into the two types of computational complexity (Sects. 7.3.2 and 7.3.3). Note that the treatment we are going to give to all these disciplines is going to be necessarily succinct, as a precise discussion is completely out of the intended scope of this book.

7.3.1 An Example

In this section, we illustrate the computational complexity of scheduling with a very simple model, i.e. a single machine, for which we try to find out a schedule that minimises the total tardiness of the jobs. This model is denoted as $1||\sum T_j$. For simplicity, we will focus on finding the optimal semi-active schedule; therefore, each possible semi-active schedule can be represented by specifying the sequence of the jobs for each machine, i.e. the order in which jobs are to be processed on the machines. Then, in our setting, $n!$ different sequences exist, which represent job permutations. Let us define by Π the set of those $n!$ sequences and by π any of those permutations.

Note that $1||\sum T_j$ is not trivial. Depending on the order in which the jobs are processed, some of them will finish before their due dates and some others will finish late, adding to the total tardiness value. Unless there is enough time to finish all jobs before the earliest due date, i.e. $d_j \geq \sum_{j=1}^{n} p_j$, there might be a possibility that some jobs will finish late.

Now, let us assume for the moment that the only method we have at our disposal is the complete enumeration of all the possible $n!$ job sequences in order to determine the minimum total tardiness. Let us also suppose that we want to solve different instances of the model with $1, 2, \ldots, 30$ jobs each one. Notice that 30 jobs is a small problem size for most real-life settings. Therefore, there would be $1!, 2!, \ldots, 30!$ possible solutions as a function of the number of jobs. We are going to assume that we have an 'old' computer capable of doing one billion (in short

scale, i.e. 1,000,000,000) basic operations per second. This would be the equivalent of a CPU running at 1.0 GHz. For simplicity, we are also assuming that one full schedule can be calculated in just one basic operation. Normally, for calculating total tardiness, we would need at least n basic operations as the individual tardiness of all jobs has to be added. We can then calculate the time it would take for this 'old' computer to calculate all possible sequences. In order to put these CPU times into perspective, we compare these CPU times against those of a theoretical computer that is 5 million times faster, i.e. this computer would be running at 5 PHz or 5,000,000 GHz. A computer running at such incredible speeds is way beyond current computers that, at the time of the writing of this book, rarely operate at frequencies higher than 6 GHz. All this data is given in Table 7.1, where AU stands for 'Age of the Universe' which has been estimated to be 14 billion years.

As we can see, for trivial problems with about 12 jobs, both computers would give the optimal solution in less than half a second in the worst case. However, increasing just 3 jobs to 15 jobs (a 25 % increase in the size of the problem) results in a CPU time of 21.79 min for the slow computer (2,730 times slower). From 15 jobs onwards, the CPU times of the slow computer quickly turn unfeasible, as anything longer than 12 h or so is unpractical for daily use. It can be observed that applying these complete (or further elaborated implicite) enumeration approaches to combinatorial optimization problems (such as scheduling problems) often results in quickly finding the optimal solution but in long term proving that this solution is indeed an optimal one.

Conversely, a computer no less than 5 million times faster would be expected to solve problems of much larger size than the slow computer. However, if $n = 17$ is already unpractical for the slow computer, for the fast computer we would be able to solve problems of up to $n = 21$. In other words, a 5 million times faster computer allows us to solve problems that are less than 24 % larger.

This single machine scheduling problem is an example of a problem with an exponential number of solutions. Needless to say, realistically sized problems of $n = 150$ are completely out of the question. Saying that enumerating all possible sequences of a single machine problem with 150 jobs is impossible is actually an underestimation. Consider that 150! results in $5.71 \cdot 10^{262}$ sequences. This number is extremely huge. To have an idea, the number of atoms in the observable universe is estimated to be between 10^{78} and 10^{80}. Therefore, a simplistic problem with one machine and some 150 jobs is impossible to enumerate. The permutation flow shop problem also has a solution space of $n!$ However, other more complex problems, like the regular flow shop or job shop have an even much larger number of solutions with $(n!)^m$ or the parallel machine layout with m^n. A job shop with, let us say, 50 jobs and 10 machines, will have a solution space of $(50!)^{10} = 6.77 \cdot 10^{644}$ sequences.

The fact that a problem has an extraordinarily huge number of possible sequences does not necessary imply that no efficient methods exist for obtaining a good solution or even an optimal solution. For example, picture again the $1||C_{\max}$ problem which also has $n!$ possible sequences but any of them will return the minimum makespan. The example given in Table 7.1 has an obvious caveat: evaluating all possible sequences in the search for the optimum is actually the worst possible method.

Table 7.1 Total CPU times for two types of computers for evaluating all solutions in single machine problems as a function of the number of jobs n

n	$n!$	Computer time (slow)		Computer time (fast)	
1	1	1	ns	0.20	fs
2	2	2	ns	0.40	fs
3	6	6	ns	1.20	fs
4	24	24	ns	4.80	fs
5	120	120	ns	24	fs
6	720	720	ns	144	fs
7	5,040	5.04	μs	1.01	ps
8	40,320	40.32	μs	8.06	ps
9	362,880	0.36	ms	72.58	ps
10	3,628,800	3.63	ms	726	ps
11	39,916,800	39.92	ms	7.98	ns
12	479,001,600	479	ms	95.80	ns
13	6,227,020,800	6.23	s	1.25	μs
14	87,178,291,200	87.18	s	17.44	μs
15	1,307,674,368,000	21.79	min	262	μs
16	20,922,789,888,000	349	min	4.18	ms
17	355,687,428,096,000	98.80	h	71.14	ms
18	6,402,373,705,728,000	74.10	days	1.28	s
19	121,645,100,408,832,000	3.85	years	24.33	s
20	2,432,902,008,176,640,000	77.09	years	8.11	min
21	51,090,942,171,709,400,000	16.19	centuries	170.30	min
22	1,124,000,727,777,610,000,000	356	centuries	62.44	h
23	25,852,016,738,885,000,000,000	819	millennia	59.84	days
24	620,448,401,733,239,000,000,000	19.66	million years	3.93	years
25	15,511,210,043,331,000,000,000,000	492	million years	98.30	years
26	403,291,461,126,606,000,000,000,000	12.78	billion years	25.56	centuries
27	10,888,869,450,418,400,000,000,000,000	345	billion years	690.09	centuries
28	304,888,344,611,714,000,000,000,000,000	690	AU	1.93	million years
29	8,841,761,993,739,700,000,000,000,000,000	20,013	AU	56.04	million years
30	265,252,859,812,191,000,000,000,000,000,000	600,383	AU	1.68	billion years

Slow computer running at 1 GHz. Fast computer running at 5 PHz

The previous example serves to illustrate two aspects earlier mentioned in this section: on the one hand, not all algorithms are equally efficient with respect to solving a scheduling model, so it is of interest analysing its performance. On the other hand, giving these incredible computer times, it seems that there are models for which it is difficult to obtain algorithms with a reasonable (not to mention good) performance.

7.3.2 Algorithm Complexity

Let us start with the running time of an algorithm. The running time is an estimation of the time needed by an algorithm (usually coded into a computer) to complete as a function of the input size. Measuring crude time is a tricky business since each computer architecture runs at different speeds, has different frequencies and many other specificities that make the comparison of running times of the same algorithm in two different architectures a daunting task. To overcome this problem, the running time is not actually given in time units but as a number of elementary operations performed by the algorithm instead. These elementary operations or 'steps' are an independent measure since each elementary operation takes a fixed amount of time to complete. As a result, the amount of time needed and the number of elementary operations are different by at most a constant factor.

For example, if we have an unsorted list of n elements, what is the time needed to locate a single item in the list (assuming that all items are different and that they are unsorted)? A basic search algorithm will require, in the worst case, to analyse each element and would require n steps to complete. These n steps are n steps with independence of the computing platform used, but obviously, will translate into different CPU times in the end. Note that in the best case, the first element in the unsorted list could be the element searched for. Therefore, only one step is needed. On average, and given a completely random input, $n/2$ steps will be needed to locate an element.

In complexity theory, the so-called 'Big-O' notation or, more specifically, the Bachmann-Landau notation is used. This notation describes the limiting behaviour of a function when the argument tends towards a particular value or infinity, usually in terms of simpler functions. Big-O notation simplifies the running times in order to concentrate on their growth rates. This requires the suppression of multiplicative constants and lower order terms. Therefore, running time expressed with the Big-O notation is said to be the asymptotical running time, i.e. as the input size goes to infinity.

Continuing with the search algorithm example, we would say that the best case running time is $O(1)$ or constant time, the average running time would be $O(n/2)$ or linear time and the worst case running time $O(n)$ or linear time also. Note that, for very large input sizes (big value of n), the large time needed is only divided by a constant.

Normally, running times are measured for the average running time, and for simplicity the O notation is used. As a result, our basic searching algorithm has a running time complexity of $O(n)$ or linear running time. Notice that this means that n basic operations are needed. However, at least three several basic operations are performed at each step j, $j = \{1, 2, \ldots, n\}$: reading the j-th element from the unsorted list and comparing this element with the desired element. If the comparison is positive, end the search; if not, increase the counter and repeat for the next element in the list. However, for large values of n, $O(3n)$ is equal to $O(n)$.

Let us picture now a different algorithm. The algorithm just adds a constant value to each element of a square matrix of size $n \times n$. This algorithm will have one outer

Table 7.2 Total CPU times for two types of computers for applying the Quicksort algorithm of $O(n \log n)$ computational complexity

n	$n \log n$	Computer time (slow)	Computer time (fast)
1,000	3,000	3.00 µs	0.60 ps
10,000	40,000	40.00 µs	8.00 ps
100,000	500,000	0.50 ms	100.00 ps
1,000,000	6,000,000	6.00 ms	1.20 ns
10,000,000	70,000,000	70.00 ms	14.00 ns
100,000,000	800,000,000	0.80 s	160.00 ns
1,000,000,000	9,000,000,000	9.00 s	1.80 µs

Slow computer running at 1 GHz. Fast computer running at 5 PHz

loop to traverse all rows (or columns) and an inner loop to traverse columns (or rows) with one basic addition operation. The running time of the algorithm will be $O(n^2)$ or quadratic time. In general, many algorithms have polynomial running times like for example $O(n^3)$ or $O(n^4)$. Note that something like $O(10n^4 + 3n^3 + n^2 + n)$ is not correct under the Big-O notation and must be written as $O(n^4)$. Recall that for large n values, the dominating term is the n^4, not the remaining polynomial.

There are many different categories of running times. For example, the most efficient comparison sort algorithm known, Quicksort, has an average running time of $O(n \log n)$ or linearithmic time, in particular, or polynomial time, in general. In the previous single machine problem example, the brute-force approach needed $O(n!)$ steps, something referred to as factorial time, in particular, or exponential time, in general.

Not all polynomial running time algorithms are equal. After all, an algorithm with a polynomial running time of $O(n^{10})$ is surely going to be slower than another with $O(n^3)$. Well, this is obviously true, but recall the previous single machine example depicted in Table 7.1. Exponential running times of $O(n!)$ quickly run out of hand for extremely small values of n. However, $O(n^{10})$ with $n = 20$ needs 248 h on the 'slow' computer of the example, whereas the brute-force approach needed 77.09 years. Furthermore, the same algorithm would need 2.05 ms on the fast computer of the example (compared to 8.11 min of the other algorithm). In other words, in a polynomial running time algorithm, a computer that is 5,000,000 times faster reduces the real CPU times linearly, i.e. 2.05 ms is 5,000,000 times faster than 248 h. This is not the case for exponential running time algorithms. Table 7.2 shows the CPU times that the two types of computers would need for very big n values of the Quicksort algorithm.

As can be seen, sorting one billion items with this algorithm would require in the worst case barely 9 s in the slow computer and an almost instantaneous 1.8 µs in the fast computer.

7.3.3 Model Complexity

In the previous subsection, we have discussed the complexity of algorithms when applied to a specific scheduling model. Informally speaking, scheduling models for which a polynomial running time algorithm is known to exist are 'easy', since such algorithms are able to produce, on average, an optimal solution in a computational time that at least does not grow 'too much' with the instance size. In contrast, scheduling models that do not have a polynomial running time algorithms are 'hard'. These scheduling models are said to belong to the NP-hard class of computational problems. Determining whether a model belongs to the NP-hard class requires a mathematical proof and it is based on a conjecture (not demonstrated until today) that the NP-hard class exists on their own.

Although very approximate, the previous definitions serve to classify scheduling models according to their complexity. This is useful as, quite often, one algorithm for a scheduling model can be applied (in a straightforward manner of with some modifications) to another scheduling model. The simplest case are the weighted and unweighted version of many criteria. For instance, it is clear that $\alpha|\beta|\sum C_j$ is a special case of $\alpha|\beta|\sum w_j C_j$ and therefore, *any* algorithm for the latter model could be used for the former. We can then say that $\alpha|\beta|\sum C_j$ *reduces* to $\alpha|\beta|\sum w_j C_j$, and it is denoted as:

$$\alpha|\beta|\sum C_j \propto \alpha|\beta|\sum w_j C_j$$

Using these reduction ideas, a hierarchy of complexity of (some) scheduling models can be established depending on the layout, on the criterion and on the constraints. Some of these reductions are rather straighforward. For instance, it is clear that $1||\sum C_j$ reduces to $F_m||\sum C_j$.

Other reductions may require some (moderate) explanation. For instance, if we have an optimal algorithm for the $\alpha|\beta|\max L_j$ model, then we can construct a special case of this model for which all $d_j = 0$. For this instance, $L_j = C_j$; therefore $\max L_j = \max C_j$. Consequently, having an algorithm for solving the $\alpha|\beta|\max L_j$ model implies that it can solve (with the same computational complexity) the $\alpha|\beta|\max C_j$ model. Thus, $\alpha|\beta|\max C_j \propto \alpha|\beta|\max L_j$.

Finally, some other reductions require more thorough proofs. In Figs. 7.1, 7.2 and 7.3, the complexity hierarchy (also named *reduction tree*) is given for most models with respect to different layouts, constraints and criteria, respectively. It is perhaps interesting to note that adding constraints to a model does not necessarily turn it into less or more complex.

The reductions also work the other way round, i.e. if a model lower in the complexity hierarchy is of a given complexity, then clearly the models to which this former model is reduced are at least of the same complexity. In a sort of tautology, if a scheduling model is difficult, then more complex models are more difficult.

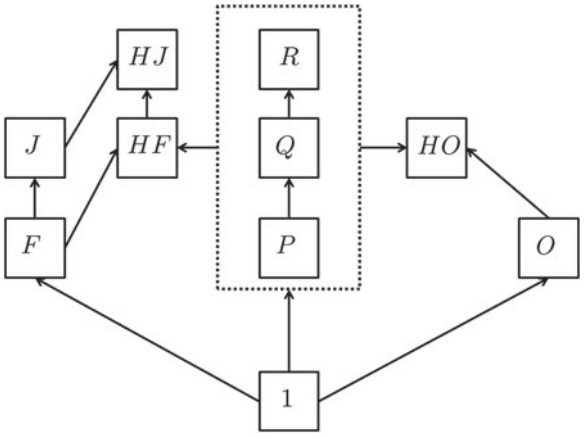

Fig. 7.1 Complexity hierarchy with respect to the layouts

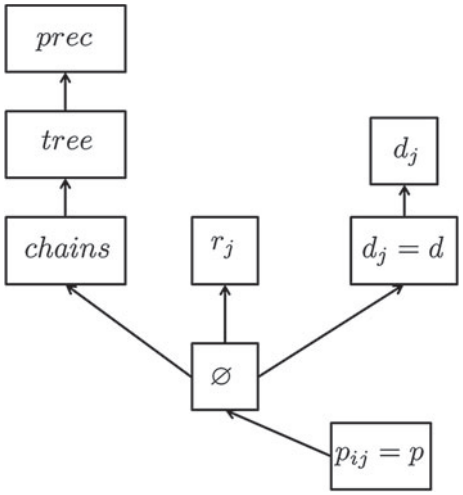

Fig. 7.2 Complexity hierarchy with respect to the constraints

Unfortunately, most scheduling problems belong to the NP-hard class. Only the simplest cases, and most of them are single machine layouts, do not belong to the NP-hard class. The question is: What to do then? The answer is in the very heart of all the scientific efforts that the scientific scheduling community has been doing over the past decades. If the problem belongs to the NP-hard class all that is known is that an optimum solution for large-sized instances is unlikely to be viable in acceptable, i.e. polynomial time. However, for practical scheduling there are several alternatives to handle this computational complexity. These are discussed in the next section.

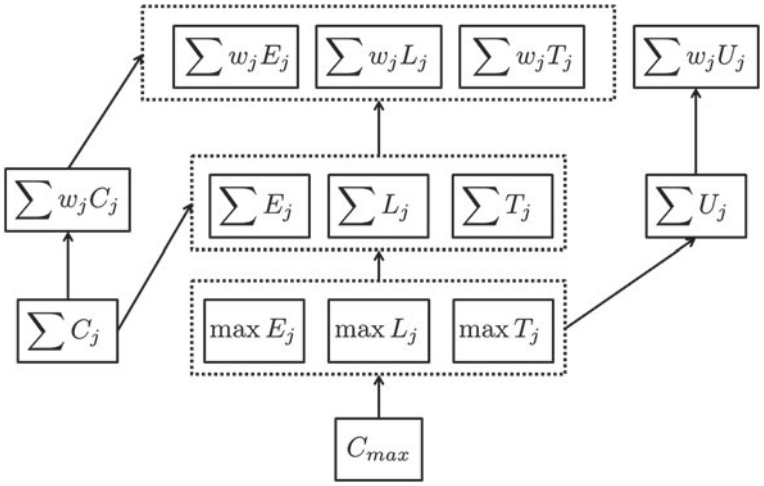

Fig. 7.3 Complexity hierarchy with respect to the criteria

7.3.4 Handling Computational Complexity

When faced with a computationally complex model (which happens most of the cases), several alternatives are open, depending on whether the model complexity is acceptable in practical terms, or not. Note that the running time is instance dependent, and that the Big-O notation neglects the size of constant factors or exponents in the running time and these may have practical importance. Therefore, an exact exponential running time algorithm might still return an optimal solution in an acceptable time for moderately sized problems. If these problems are of interest to the practical application, then the computational complexity is (at least from a practical viewpoint) acceptable.

If the computational complexity is not acceptable, then again two options (not necessarily alternative) are available:

1. Try to reduce the complexity of the model and/or that of the associate algorithms to an acceptable level, or
2. Avoid the use of algorithms to solve the model.

With respect to the first option, recall that two types of complexity in manufacturing scheduling were introduced in Sect. 2.3.2, according to the discussion in Sect. 1.5: A computational complexity reduced to a formal sphere, and a real-world complexity. While the reduction of the latter was discussed in Sect. 6.3, we will address here the reduction of the computational complexity. A general framework for complexity reduction with respect to model and/or method can be seen from Fig. 7.4.

Our point of origin is situation 1 in Fig. 7.4, in which a (original) model is shown in combination with a (original) exact algorithm. If the complexity of this combination of model and algorithm is not acceptable, the required complexity reduction

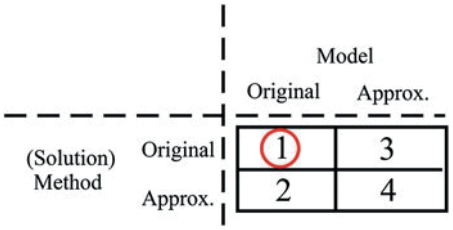

Fig. 7.4 Complexity reduction and approximate methods

might take place into two directions. On one hand, the same original model might be approximately solved with an approximate method. This case is represented by situation 2 in Fig. 7.4. For example, a given model can be solved by an algorithm returning a solution without guarantee of optimality. On the other hand, the model might be simplified and solved with the same algorithm which is available for the original model (situation 3 in Fig. 7.4). For example, jobs of the original model might be clustered into job groups which are to be executed consecutively within their group and the exact algorithm for the original model is applied only to the job group model which is hopefully, significantly less complex than the original one. Finally, both approaches can be combined (situation 4 in Fig. 7.4).[1]

Note that, after having solved the model- and/or method-simplified setting, the solution derived for this simplification has to be adjusted to the requirements of a solution for the original model. Despite of possible ambiguities, we will follow this intuitive line of thinking in our discussion of selected ideas for approximate methods which, accordingly, are seen as a simplification with respect to model and/or method in combination with a subsequent transfer of the solution of the simplified setting to the original model.

With respect to the second option—avoiding the use of algorithms to solve the model—it may seem a bit unrealistic, but yet it has been (and still is) one of the favourite options for the scheduler. While algorithms are designed to obtain a schedule, it may be that one can schedule jobs without a 'true' schedule. Instead, some rules can be given to assign on-the-fly jobs to machines as soon as they enter the shop floor. Such a rule is called a *dispatching rule*, and the set of rules that one can use to avoid deriving a schedule is called *scheduling policy*. More formally, a scheduling policy consists of a set of principles usually based in 'common sense' or 'rule of thumb' that are used to generate a schedule. However, note that a schedule itself is not explicitly given, although it can be obtained from this set of principles. For instance, a reasonable scheduling policy may be to process first those jobs whose due date is closest, in order to honour the commitment with the customer. Clearly, given a known set of jobs with known due dates, this policy can be translated into a

[1] It should be noted that this separation of model-oriented and of method-oriented simplification approaches might appear rather intuitive. In fact, it is somewhat artificial and the unique separation of simplification approaches into these two categories might be problematic. Ambiguities with respect to this separation might occur.

schedule. Quite clearly as well, this policy can lead to disastrous results, as it does not take into account the different processing times of the jobs, and therefore jobs with short processing times but relatively close due dates would be prioritised in front of jobs with further due dates, but also high processing times and therefore less slack.

The two aspects mentioned before summarise the main features of using scheduling policies as a scheduling method, i.e.: their simplicity and their myopic nature. The first feature is the reason why scheduling policies were and still are so popular in practice, since we will see in this chapter that developing optimal or even good schedules is, in general, far from being an easy task. In contrast, the second feature exposes their great disadvantage in front of other methods. There is no point in trying to make the principles more sophisticated so they can take into account most possible situations. Every rule has an exception and except for few cases, the performance of the dispatching rules is rather poor. Even worse, for most models the performance of the dispatching rule heavily depends on the problem instance, so they are not very reliable.

The most employed scheduling policies are described in detail in Sect. 7.4.

7.4 Scheduling Policies

Industrial activity dates back to the dawn of the industrial revolution. The need for scheduling is also very old and can be traced back to the onset of the production line at the beginning of the twentieth century. Obviously, at that time there were no computers and things were done manually. Even today, a large percentage of companies rely on human schedulers or on simple spreadsheets and rules of thumb to schedule jobs in the shop.

Manual schedulers do lots of iterations over production plans that evolve over time from production wishlists to more detailed production needs. Schedules are continually revised, changed, adapted and patched on a daily, even hourly basis. Most manual methods work in a forward fashion, i.e. jobs are launched to the shop and sequenced at machines to be started as soon as those machines are ready. However, there is seldom any accuracy when determining when jobs will be completed. Priority choices are made at some decision points, subject to many judgmental exceptions and special cases. While this might sound chaotic, we have to remind that it has worked and still works for countless firms. Human schedulers have a number of advantages like the ability to react quickly to changes in the production floor, common sense, experience, etc. However, complex production systems are often out of the possible reach of human beings. The intricacies and dynamics of several hundred jobs moving inside a shop is beyond the cognitive capability of even the best scheduling expert.

There are several ways for dealing with this situation. Sometimes, changing the whole production system to eliminate the need for scheduling is the best approach. If a production line has become too complex to handle since it is processing many different products, a possibility is to reduce the number of products or to break down that line into several different lines. Ideally, one line for each product and the scheduling problem is mostly gone or at least greatly reduced.

Manual scheduling is often based on the so-called dispatching rules. As the name implies, these rules basically launch or dispatch tasks to machines according to usually a simple calculation that returns an index or relative importance for pending tasks. Dispatching rules are also known as priority rules.

Dispatching rules are classical methods and therefore cannot be easily attributed to a single author and their origins are hard to trace. Basically, a dispatching rule maintains a list of eligible tasks or pending tasks. These tasks are entire jobs in a single machine layout or the tasks that make up a full job in a flow shop or job shop layout. Tasks enter the eligible set whenever it is actually possible to start processing them, i.e. when the preceding task is finished, when the release date has elapsed, etc. Let us denote the list of eligible tasks as $\wp$. Similarly, $|\wp|$ denotes the number of tasks in the eligible list.

In a nutshell, general dispatching rules carry out a simple calculation for all tasks in $\wp$ and dispatch the tasks according to the result of this calculation, sometimes referred to as priority. Dispatching rules can be generally classified in two dimensions:

1. Local/Global rules. This refers to the data considered in the priority index calculation. Some rules only consider the data of each task in $\wp$ separately. These are called local dispatching rules. Rules that consider the data of more than one task in $\wp$ simultaneously or other additional information not just related to the task itself are referred to as global dispatching rules.
2. Static/Dynamic rules. The result of the dispatching rule depends on the time at which it is applied. Static dispatching rules always return the same priority index, regardless of the state of the schedule or the list $\wp$. On the contrary, dynamic dispatching rules depend on the instant of time t at which they are calculated and hence on the information resulting from the partial schedule derived up to time t.

7.4.1 Basic Dispatching Rules

- First Come First Served (FCFS). The oldest pending task is scheduled. It is equivalent as ordering the tasks in $\wp$ in a First In First Out (FIFO) fashion. This rule is the simplest one and also seems the fairest when dealing with (human) customers as jobs that arrived first are also processed first. Obviously, this rule is too simple and does not work well in complex settings. FCFS is a local and static rule. In the presence of release dates r_j, this rule can be easily modified to Earliest Release Date first (ERD) which sorts jobs in ascending order of their release dates.
- Last Come First Served (LCFS). It is the opposite of FCFS. The most recent task is scheduled. It is equivalent to Last In First Out (LIFO). LCFS is a local and static rule.
- Random Order. Pending tasks are sequenced in a random order. While this might seem naive, we have a guarantee that no specific objective will be either favoured or penalised in a consistent way. Random Order is often used as a reference solution in simulation studies when comparing different dispatching rules.

- Shortest Processing Time first (SPT). The task in the pending list $\wp$ with the shortest processing time has the highest priority. More specifically, among m machines, indexed by i, the task k that satisfies the following expression has the highest priority:

$$p_{ik} = \min_{j=1}^{|\wp|} \{p_{ij}\}$$

  SPT is a local and static rule. Most interestingly, SPT results in an optimal solution for the problem $1||\sum C_j$. SPT just requires the pending job list to be ordered. Furthermore, this can be done once as it is a static rule. This results in a computational complexity for SPT of $O(n \log n)$.
- Longest Processing Time first (LPT). Contrary to SPT, LPT gives higher priority to the pending task with the longest processing time, i.e. the task k satisfying the following expression is given the highest priority:

$$p_{ik} = \max_{j=1}^{|\wp|} \{p_{ij}\}$$

  LPT is a local and static rule. It also has a computational complexity of $O(n \log n)$.
- Earliest Due Date first (EDD). This rule considers the due dates of the pending tasks in $\wp$. The highest priority is given to the task with the smallest due date or deadline, i.e.:

$$d_k = \min_{j=1}^{|\wp|} \{d_j\} \vee \bar{d}_k = \min_{j=1}^{|\wp|} \{\bar{d}_j\}$$

  EDD is also a local and static rule. EDD gives the optimum solution for the problems $1||T_{\max}$ and $1||L_{\max}$. It has a computational complexity of $O(n \log n)$. EDD is sometimes referred to as Jackson's rule. EDD has some important drawbacks for other objectives different than $L_{\max}$. Take for example, the $\sum T_j$ or $\sum_{j=1}^{n} w_j T_j$ criteria. It is very easy to get a very bad result if there is a job with a large p_j and an early d_j. That job would be sequenced first by EDD, even if doing so will delay the start of many other jobs, possibly forcing them to finish later than their due dates even if the first job is impossible to finish on time if $d_j < p_j$. In order to solve this issue, a dynamic version of the EDD is commonly used, this is described in the following.
- Minimum Slack first (MS). The slack with respect to the due date is the amount of time that a task still has before it would be tardy. This slack decreases as more and more jobs are scheduled and the current time t increases. Therefore, from the pending jobs list $\wp$, and at any given time t, a job k with the minimum following calculation is selected:

$$k = \operatorname*{argmin}_{j=1}^{|\wp|} \Big\{ \max \{d_j - p_j - t, 0\} \Big\}$$

Therefore, a job with the earliest due date but short processing time is not necessarily sequenced first. MS is an example of a dynamic dispatching rule, as it has to be recalculated each time a job is ready to be dispatched. Assuming that there are n jobs and that after completing a job, the pending job list $\wp$ is examined again, the computational complexity of MS can be stated as $O(n^2 \log n)$. While considerably slower than the more simple EDD, with modern computers there is no actual difference in running times between EDD and MS even for thousands of jobs.

- Largest Number of Successors (LNS). In problems with precedence relationships among jobs, it is interesting to process early jobs that 'unlock' several other jobs once finished. These jobs correspond to those that have many successors. In order to work correctly, this rule has to be combined with others that further give priority to some specific objective, such as due dates or completion times. LNS is a local and static rule. However, this last classification is a bit blurry depending on how LNS is implemented. If we only consider the immediate successors, then LNS is a local rule. However, if all successors are considered (i.e. the successors of the successors and so on), then LNS could be considered as a global rule.
- Shortest Setup Time first (SST). This rule helps in problems where setup times are a concern. Basically, from all the pending jobs in the $\wp$ list, the job with the highest priority is the one with the shortest setup time. SST is a local and static rule.
- Least Flexible Job first (LFJ). In problems with machine eligibility, it might be of use to sequence jobs that have very specific machine needs, in order to avoid scheduling them later where those specific machines might be already busy with previously scheduled jobs. LFJ is a local and static rule.
- Shortest Remaining Work Next Machine first (SRWNM). This is another example of a global rule. In flow or shop layouts, when assigning jobs to a specific machine, we might be interested to assign a job to a current machine looking ahead in the information of that job for subsequent machines. For example, two jobs to be processed on a given machine might have similar requirements in that machine and similar due dates so one could myopically think that it does not matter which job to assign to that specific machine. However, the next machine to be used for those jobs (picture that the next machine for those two jobs is not the same) could give us more information. From those two machines, we could look into the one that has the least remaining work left. Giving more work to that machine is crucial so to avoid imminent idleness. Note that there are many possibilities as regards global dispatching rules and SRWNM is just an example.

Let us give an example of the EDD rule for the $1||\max L_j$ problem with six jobs. The processing times and due dates of these jobs are given in Table 7.3.

EDD just needs to sort the jobs in ascending order of d_j, this requires the application of the Quicksort algorithm which has a running time complexity of $O(n \log n)$. For the example, the sorted list, and EDD solution is $\pi_{EDD} = (1, 2, 4, 3, 5, 6)$. As we can see, the completion times are $C_1 = p_1 = 6$, $C_2 = C_1 + p_2 = 6 + 3 = 9$,

Table 7.3 Processing times (p_j) for the single machine maximum lateness minimisation problem example

j	1	2	3	4	5	6
p_j	6	3	9	7	8	2
d_j	3	7	12	10	20	29

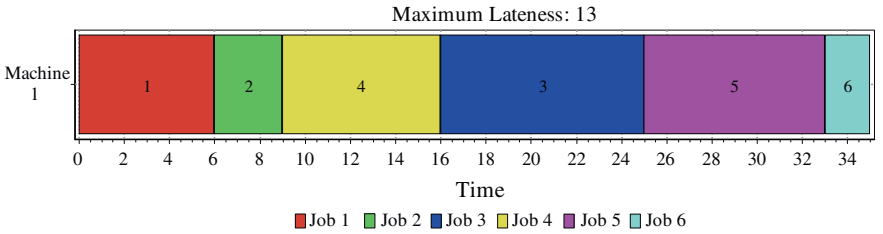

Fig. 7.5 Gantt chart with the EDD solution or the single machine maximum lateness minimisation problem example

$C_4 = C_2 + p_4 = 9 + 7 = 16$, $C_3 = C_4 + p_3 = 16 + 9 = 25$, $C_5 = C_3 + p_5 = 25 + 8 = 33$ and finally, $C_6 = C_5 + p_6 = 33 + 2 = 35$. As a result, the lateness of the jobs are: $L_1 = C_1 - d_1 = 6 - 3 = 3$, $L_2 = 9 - 7 = 2$, $L_3 = 25 - 12 = 13$, $L_4 = 16 - 10 = 6$, $L_5 = 33 - 20 = 13$ and $L_6 = 35 - 29 = 6$. Therefore, the maximum lateness $L_{\max} = L_3 = L_5 = 13$ which is optimal for this example. The resulting Gantt chart is pictured in Fig. 7.5.

There are variants for dispatching rules that consider weights. For example, the Weighted Shortest Processing Time first (WSPT) is the weighted variant of the SPT where the task k that satisfies the following expression is given the highest priority:

$$k = \underset{j=1}{\overset{|\wp|}{\operatorname{argmax}}} \left\{ \frac{w_j}{p_{ij}} \right\}$$

Basically, all previous dispatching rules can be modified in order to consider weights in a straightforward way.

7.4.2 Advanced Dispatching Rules

The dispatching rules in the previous section are useful mainly for sequencing jobs. In problems with parallel machines or where there are hybrid layouts as hybrid flow or job shops, there is an additional decision to be made about to which machine each job must be assigned to. There are some basic dispatching rules for this as well:

- First Available Machine (FAM). Assigns the job to the first eligible machine available. This is the machine with the minimum liberation time from its last scheduled

job, or lowest availability date for the machine if no job is scheduled at the machine yet.
- Earliest Starting Time (EST). Chooses the machine that is able to start job j at the earliest time. Note that only in simple layouts with no constraints this EST rule coincides with FAM. For example, availability of the job, lags, overlaps and a possible setup time can be considered. EST rule measures when the job will be able to start at the machine, not only when the machine is free from previous jobs.
- Earliest Completion Time (ECT). Takes the eligible machine capable of completing job j at the earliest possible time. Thus the difference with the previous rule is that this rule includes processing (possibly machine-dependent) times and other constraints.

Note than the EST and ECT rules need more calculations than the FAM as all pending jobs in $\wp$ have to be tested for earliest starting time and earliest completion times in all possible eligible machines in the hybrid environments. However, in some cases, this added CPU time might prove crucial for a good machine assignment decision.

Similarly, in recent years, some other more advanced machine assignment rules have been proposed. Some of them are discussed below:

- Earliest Preparation Next Stage (EPNS). The machine able to prepare the job at the earliest time for the next stage to be visited is chosen. It is a hybrid layout adaptation of the SRWNM rule. Many possible special situations can be considered like time lags between the current and the next stage, transportation constraints, etc. The rule uses more information about the continuation of the job, without directly focusing on the machines in the next stage. This is a global machine assignment rule.
- Earliest Completion Next Stage (ECNS). The availability of machines in the next stage to be visited and the corresponding processing times are considered as well. This rule also considers all the eligible machines at the next stage to make a decision. ECNS is also a global machine assignment rule.
- Forbidden Machine (FM). Excludes machine l^* that is able to finish job k at the earliest possible time, where job k is the next job in the sequence immediately following job j. ECT is applied to the remaining eligible machines for job j. This rule tries to minimise the excessive greediness of the ECT and similar rules. The fastest machine for a given stage might create problems in downstream stages. This rule is expected to obtain better results for later jobs, as it reserves some capacity in the machine able to finish the next job earliest. As with the previous rules, this is also a global or a 'look ahead' rule.
- Next Job Same Machine (NJSM). The assumption is made that two consecutive jobs in the sequence j and k are assigned to the same machine. The machine chosen for assignment is the one also able to finish jobs j and k at the earliest time although only job j is assigned. This is a look-ahead rule that also considers the next job in the sequence. This rule is especially useful if setups are relatively large, as not the setup for job j is considered, but also the setup between jobs j and k.

- Sum Completion Times (SCT). Completion times of job j and job k are calculated for all eligible machine combinations. The machine chosen is the one where the sum of both completion times is the smallest. This rule is similar to NJSM, but without the assumption that job k is assigned to the same machine.

Note that extremely complex dispatching rules or machine assignment rules can be devised. However, whether these more complicated rules perform better or not is even today a matter of research. For example, Urlings et al. (2010) demonstrated that the more complex machine assignment rules are often not the best ones. The added CPU time needed for the more complex assignments did not pay off when compared to faster and simpler rules. Let us not forget that the main benefit of dispatching rules is their simplicity and speed; therefore, going in the opposite direction might cancel out these benefits.

7.4.3 Combination of Dispatching Rules

Sometimes dispatching rules are combined. For example, in hybrid environments we could have a EDD-FAM rule that would possibly solve the job scheduling and machine assignment interrelated problems. In some other scenarios, specially when the optimisation of more advanced objectives is desired, composite dispatching rules are of interest. One well-known example is the Apparent Tardiness Cost (ATC) dispatching rule, proposed by Vepsalainen and Morton (1987), also explained in detail in Pinedo (2012). This rule was proposed for the job shop with total weighted tardiness objective but has been widely applied also to the single machine total weighted tardiness problem or $1||\sum w_j T_j$. Recall that this problem was shown to be NP-hard by Du and Leung (1990). Note that in this single machine problem, some of the earlier studied dispatching rules provide excellent results. For example, the rule WSPT will be optimal if all due dates are zero or if release dates are all equal to a large value greater than $\sum_{j=1}^{n} p_j$. EDD and MS rules will also result in optimal or close to optimal schedules if the due dates are easy to satisfy. By easy we refer to well spread out due dates (not clustered around a point in time) and loose (due dates much larger than processing times). ATC tries to combine the good characteristics of both WSPT and MS. ATC is a dynamic rule that is applied each time the single machine is freed from the last assigned job. Therefore, if there are n jobs, ATC is firstly applied for the n jobs and one job, the one with the highest priority according to ATC is scheduled. Then, when the machine is free, the rule is calculated again for all remaining $n-1$ jobs, then $n-2$ and so on until only one job remains, which is directly appended to the schedule. In total, ATC is calculated $n \cdot (n+1)/2 - 1$ times, which results in a running time of $O(n^2)$.
The priority index is calculated as follows:

$$I_j(t) = \frac{w_j}{p_j} \cdot e^{\left(\frac{\max\{d_j - p_j - t, 0\}}{K\overline{p}}\right)}$$

Note that $I_j(t)$ has two main components. The first one corresponds to the WSPT rule and the second is the MS rule. $\overline{p}$ is just the average of the processing times on the single machine or $\overline{p} = \dfrac{\sum_{j=1}^{n} p_j}{n}$. K is referred to as the look-ahead parameter and has to be set in a particular way. For the simple machine problems, the due date tightness factor τ and the range of due dates R are employed as follows:

$$\tau = 1 - \frac{\sum_{j=1}^{n} d_j}{n \cdot C_{\max}}$$

$$R = \frac{d_{\max} - d_{\min}}{C_{\max}}$$

where $d_{\max}$ and $d_{\min}$ are the minimum and maximum due dates, respectively. Note that in the single machine case, the makespan is trivially calculated as $C_{\max} = \sum_{j=1}^{n} p_j$.

From τ and R there are several ways of calculating K. In any case, it can be seen that the larger the K value, the lower the quotient of the exponent of the e in the $I_j(t)$ priority index, which translates in a lower participation of the 'MS part' in the ATC rule. In the extreme, if that quotient is 0, and knowing that $e^0 = 1$, the $I_j(t)$ priority index is reduced to the WSPT rule.

The main drawback of dispatching rules is their intrinsic myopic nature. Decisions are too static and too near-sighted for a good overall performance. Furthermore, dispatching rules tend to be tailored for specific objectives and favour some performance measures over others. However, they are general in the sense that they can be applied with little modifications to most machine layouts and constraints. Considering that advanced approaches are very slowly gaining terrain in real factories and are still not widespread, it is easy to understand that dispatching rules are still nowadays used intensively at companies.

7.5 Scheduling Algorithms

As already discussed in Sect. 7.2.1, it is customary to classify algorithms into exact and approximate. Regarding exact algorithms, there is a guarantee that no other schedule performs better than the one obtained with respect to the objective sought. Such guarantee does not exist for approximate algorithms and their performance is established upon experience. Nevertheless, in Sect. 7.6.2.1 we will discuss some cases where the maximum deviation of the schedule obtained by an approximate algorithm with respect to the optimal solution can be obtained.

Exact and approximate algorithms are discussed in the next subsections.

7.5.1 Exact Algorithms

Broadly speaking, there are two types of exact algorithms for scheduling problems:

- Exact constructive algorithms. These algorithms exploit some properties of the specific model in order to construct a solution which is guaranteed to be optimal. Constructive exact procedures will be discussed in detail in Sect. 8.2, but a hint on its logic is given here using the problem of scheduling a number of jobs on one machine with the goal of minimising the total completion time or flowtime. According to the notation introduced in Chap. 3, this problem can be modelled as $1|d_j|\sum C_j$. For a given sequence $\pi = (\pi_1, \ldots, \pi_n)$, the value of the objective function can be expressed as $\sum_j C_j(\pi) = n \cdot p_{\pi_1} + (n-1) \cdot p_{\pi_2} + \cdots + 2p_{\pi_{(n-1)}} + p_{\pi_n}$, where p_{π_k} is the processing time of job π_k in the sequence π. It is easy to see then (and can be formally proved) that the best value of the objective function is obtained scheduling first these jobs with lowest processing times. Therefore, it is possible to construct a solution which is optimal. Since the procedure requires to sort the jobs in ascending order of their processing times, its computational complexity is given by $O(n \log n)$.[2]
  Although such constructive procedures cover only a very limited spectrum of rather simple problems if they are computational manageable, they can be used as sub-modules of more complex approaches for the approximate solution of more complex scheduling problems. For example, if one machine or stage can be identified as a distinctive bottleneck of the real-world problem under consideration, an adequate solution procedure for a single machine might yield an initial solution for this bottleneck machine. Afterwards, this solution is complemented and possibly adjusted by a schedule for the operations before and behind this bottleneck machine/level.
- Enumerative algorithms. These algorithms implicitly or explicitly guarantee to evaluate all possible solutions of the model. It includes complete enumeration as well as branch and bound, cutting plane, or branch and cut approaches (as well as other implicit enumeration approaches, often based on some decision tree consideration)—if these procedures are executed comprehensively (which might be problematic because of time reasons). The main types of enumerative algorithms for manufacturing scheduling are discussed in Sects. 8.3.1–8.3.3.
  As already mentioned, enumeration approaches offer the comfort and safety of exact optimal solutions at the expense of computational effort that, in many cases, grows exponentially with the problem size making large problem sizes almost intractable for solution.

[2] The thorough reader should have recognised that such procedure is identical to the SPT rule discussed in Sect. 7.4.

7.5.2 Approximate Algorithms

Approximate methods in optimisation can be defined as procedures yielding a solution to the optimisation problem under consideration which is supposed to be of adequate or good quality. However, this assumed quality cannot be formally proven in advance but is simply based on experience. Of course, such methods will only be applied if either no exact procedure is available or the application of the latter is too expensive and/or time-consuming.

There is a wide spectrum of approximate methods for almost every NP-hard manufacturing scheduling problem. And as wide are also the ideas and concepts behind them, so even a classification of these methods can be done under different approaches. In this introduction, we will distinguish between *heuristics*, which are approximate methods specifically tailored for a particular decision problem, and more generic procedures known as *metaheuristics*. Obviously, this classification is not unambiguous, as many heuristics originally designed for a specific problem can be successfully applied outside the original scope. Additionally, we find it useful to differentiate between *constructive heuristics* and *improvement heuristics*. These types are discussed in the next sections.

7.5.2.1 Constructive Heuristics

Constructive heuristics in manufacturing scheduling generate a hopefully feasible and good sequence/schedule from scratch, i.e. exclusively based on the input data of the problem, without referring to another solution to the problem under consideration. These approaches are sometimes used as stand-alone solution approaches, i.e. their solution is implemented as is—or at least taken as the result from the formal optimisation procedure and as basis for the real-world solution to be implemented. Within more sophisticated (heuristic) optimisation procedures, constructive heuristics are often used as the starting point for improvement approaches (see next section).

Constructive approaches use rules for schedule construction which have been shown or are at least supposed to return good or acceptable schedules. This rating of their performance may simply arise from the thumb or the experience of the decision makers, may these experiences stem from other (probably similar) scheduling problems and/or from more or less sophisticated results from optimisation in general.

Many constructive approaches are of (iterative) *greedy* or *myopic* type, i.e. out of the remaining operations/tasks to be included to a so far generated partial schedule, the next operation is chosen as the one which is contributing best to the objective function under consideration. Interaction with the other previously scheduled operations is exclusively regarded with respect to the partial schedule derived so far. Interaction with other non-scheduled operations is ignored. Myopic here refers to the fact that only the very next operation to be scheduled is considered with respect to its interaction with the schedule derived so far. (Of course, the expression ‘myopic’ might also be used for a limited scope of the manufacturing setting considered; e.g.

the restriction of the consideration to a single machine or job. However, this refers more to the decomposition and coordination aspects mentioned in Sect. 6.3 and is basically not addressed here.) It should be mentioned that the myopia in this type of approaches is, as a start, addressing operations as scheduling objects. Since operations are combining jobs and machines/stages, the prioritisation of operations might be with respect to both jobs (on a given machine or group of machines) or machines (for given jobs or group of jobs) or even a combination of both.

The strictness of pure greedy approaches can by reduced by simultaneously considering the inclusion of more than one operation into a sequence/schedule and/or by not only appending an operation to a given partial sequence but to consider more than one position within this partial schedule for insertion of one or more new operations.

Summarising, constructive heuristics in manufacturing scheduling usually work iteratively: First, they sort the objects (operations, jobs etc.) to be scheduled statically or dynamically and, second, they choose one or more of these up to now not scheduled objects and insert them at one or several potential slots in the so far determined sequences. Third, one or more of the best insertions are kept and represent the sequence/schedule of the so far considered objects for the next iteration. This procedure is continued until all objects have been scheduled.

Finally, a short remark is given with respect to so-called ties in the iterative process of generating a solution: Both, in constructive as well as in improvement approaches referred to in the next section, it might happen that the criterion for the choice of one or more new scheduling objects to be inserted in the partial schedule derived so far does not yield a unique selection. For example, if one out of several remaining operations should be chosen to be scheduled next, the criterion might give more than one proposal for the next best insertion. (Also more than one 'best' position in the sequence could appear for only one object to be inserted.) This ambiguity is called a tie. The scheduling literature does not provide many hints on how to deal with such a situation. Apart from the simple suggestion to consider all of these alternatives separately (which might result in exponential computational effort), only few hints are given. For example, as an adaptation from multi-criteria optimisation, it is proposed to use a secondary criterion which then has to be determined carefully in addition.

7.5.2.2 Improvement Heuristics

In contrast to the constructive approaches, improvement heuristics try to derive a better solution based on one or more solutions found so far. 'Better' here in the wide majority of scientific contributions refers to the improvement of the objective function value of solutions but might also refer to the reduction of infeasibility, or both. (In many problem settings for manufacturing scheduling (in-) feasibility is not seen as the main field of study. However, especially in more complex real-world settings, reaching feasibility might be a challenge as well.)

To set up improvement approaches, a single modification rule or a set of modification rules which is intended to yield better solutions has to be provided as well as a

stopping criterion to finish the improvement process. This holds for both, exact and heuristic approaches. However, the stopping criterion for exact approaches is implicitly or explicitly the achievement of the optimal solution while the stopping criterion for a heuristic improvement approach will not and cannot include this optimality condition.

A core problem in enumerative improvement of most combinatorial optimisation problems—including most in manufacturing scheduling—is their non-convexity. Consequently, especially looking for improvements in the surrounding of a given solution (so-called local search procedures) will often lead to local optima while the intended determination of global optima will not be possible by these heuristic approaches (see, e.g. Groetschel and Lovasz 1993). Here we distinguish two different approaches, which are shown below.

Limited enumeration
Limited enumeration as a heuristic for combinatorial optimisation problems here basically comprises enumerative approaches where the limitation results from a direct or indirect limitation of the number of solutions considered directly or indirectly to a number below the overall number of solutions. Consequently, these approaches do not evaluate (explicitly or implicitly) all solutions and therefore cannot guarantee the achievement of an optimal solution.

The limitation might result from:

- a direct upper bound on the number of solutions considered,
- an indirect upper bound on the number of solutions considered, e.g. given by a bound on the computation time or other resources,
- a limitation to the quality of the solution to be derived, e.g. by some upper bound on the deviation from the optimal objective function value which has to be fulfilled by the final solution from the approach (this requires an additional procedure for determination of tight upper bounds),
- another type of limitation of the number of solutions evaluated, e.g. by some 'distance' constraint which means that only solutions in the surrounding of a given solution are explored as in local search procedures.

Of course, these limitation procedures can also be applied in combination.

Local Search
Local search approaches try to explore the neighbourhood of one or more given solutions. Defining local search procedures requires:

- the definition of a neighbourhood of a solution,
- a rule for the sequence of the solutions in the neighbourhood to be explored
- a rule which of the solutions examined should be kept, i.e. for the next iteration of the neighbourhood exploration,
- a rule whether the whole neighbourhood should be explored or only a limited part of it, e.g. only until a first better solution has been detected, as well as
- a stopping criterion to finish the search procedure (see also Domschke et al. 1997).

Neighbourhoods of sequences and schedules can be determined in many different ways: Exchanging two or more consecutive or non-consecutive jobs or operations of a given solution are standard ways to define neighbourhoods of this given solution. More sophisticated and more problem-specific definitions of neighbourhoods are available for many problem settings.

Within the local search process it has to be guaranteed as far as possible that a solution already examined is not considered again (so-called cycling) to avoid superfluous computational effort.

There are two main kinds of accepting solutions from a neighbourhood. In the 'permanent improvement' approach, new solutions are only accepted if the objective function value of the new solutions is better or at least not worse than the best one so far. Obviously, this procedure might quickly run into a suboptimal solution. 'Allowance of intermediate worsening' approaches temporarily also accept solutions for further consideration which might even be (slightly) worse than the current best solution. These approaches might avoid the process to get stuck too quickly in a suboptimum and have been shown to work rather well when applied to scheduling problems or combinatorial optimisation problems in general. These latter class of approaches may be further subdivided into those with deterministic (but maybe varying, i.e. usually iteratively reduced) maximum degree of worsening or with stochastic limitation of this degree of worsening.

Local search approaches usually stop after a certain number of solutions explored, after a pre-determined time limit and/or when no improvement has been generated for a certain amount of time or a certain number of solutions explored.

7.5.2.3 Metaheuristics

Because of the non-convexity of many combinatorial optimisation problems and its consequence of easily getting stuck in local suboptima when applying constructive heuristics in combination with deterministic 'permanent improvement' neighbourhood search approaches, during the last decades several approaches have been developed which try to overcome or at least reduce this problem. These approaches—collectively known as metaheuristics—systematically try to expand the search space by allowing temporal and stochastically worsening of solutions to be considered for the next neighbourhood search and/or to adapt principles from phenomena of nature to explore also rather different solutions which are excluded by pure neighbourhood search.

Because manufacturing scheduling problems represent a prominent group within the class of combinatorial optimisation problems, scheduling problems have been intensively analysed with respect to the performance of metaheuristic approaches. The basic principles for controlling of search procedures from metaheuristics, however, are applicable for a multiplicity of problems and neighbourhood definitions. There are many classes of metaheuristics, including simulated annealing, tabu search, genetic algorithms, ant colony algorithms, particle swarm algorithms, etc. Metaheuristics for manufacturing scheduling are discussed in Sect. 9.4.

7.6 Assessing Scheduling Methods

To give some hints for a performance evaluation of a scheduling method, let us first recall the decision-making process already discussed in Chap. 2 (Sect. 1.5), and particularly Fig. 1.2 where the flow of the decision-making process was presented. Evaluating the performance of the process sketched out above may be accomplished (at least) in two ways:

- The 'formal adequacy' can be checked. Then, the performance of the method on the formal level is restricted to the link between the formal model and the formal solution. This task is exclusively dedicated to the formal sphere and it will be discussed in Sect. 7.6.2.
- The 'real-world adequacy' can be evaluated. Evaluating the contribution of the process to the solution of the real-world problem needs to have a broader perspective, since it requires to evaluate the overall process from Fig. 1.2. This type of adequacy is discussed in Sect. 7.6.1.

Evaluating real-world adequacy is usually case specific to a high degree (i.e. strongly dependent on the particular manufacturing scheduling decision problem to be addressed) and will include many soft aspects. Therefore, classifying this type of evaluation will probably remain on a rather abstract level. However, the evaluation of formal adequacy, as is or as a sub-component of real-world adequacy is rather advanced, since restricting to the formal sphere offers a perspective which can be seen more or less neutral with respect to and independent of the real-world background of the problem. In this sequel, we will address both types.

7.6.1 Real-World Adequacy

From a real-world application point of view, the overall process of manufacturing scheduling is successful if the solution finally to be implemented fulfils the requirements of the decision maker and/or of the real-world problem for what it is intended for. Therefore, the evaluation refers to the overall process from identifying the real-world problem via its simplification to a formal model (discussed in Chap. 6) and the solution of this model to the already mentioned transfer and implementation of the solution in the real-world setting. This evaluation requirement is obvious on one hand and depends in main parts more or less completely on the specific situation on the other hand.

Given the already mentioned case-specific nature of this type of adequacy, only few non case-specific aspects can be addressed. These are more or less valid for all real-world decision problems which are solved in a way as sketched out in Fig. 1.2. More specific statements could be possible by referring to the specific case under consideration.

The aspects addressed here are the following:

- Designing of the decision process on one hand and executing the process on the other may be seen separately but interrelated and maybe iteratively connected with each other. In particular, time budgets for the overall process as well as individually for the design and the execution phase of this process have to be determined and accepted.
- Related to the first aspect, as part of the decision-making process the time to set-up and to execute the process (not only of the formal part) might be decisive for the success of the decision, i.e. most probably there will be a trade-off between accuracy of the process and of the solution on one hand and the time needed to execute the process on the other hand.
- The decision maker or the real-world participants will often evaluate the quality of the solution itself—but not the decision process as a whole. Retracing their (dis-) satisfaction and deriving modifications of the process is task of those who are responsible for the design and the execution of the decision-making process itself.
- As mentioned in many references, robustness, flexibility, stability and nervousness of solutions are relevant aspects for analysing the solution quality, maybe limited to the formal sphere alone but of course more important with respect to the real-world problem and its solution. With respect to scheduling, these expressions include sensitivity of the solution to changes in face of parameter uncertainty and disruptive events. Also the consequences caused by upcoming new information over time, maybe before the plan is implemented or while it has already partly been executed, have to be anticipated in terms of robustness etc. Schedule robustness gives a measure for the effect of how much disruptions would degrade the performance of the system as it executes the schedule. Flexibility covers the ability of a system or a solution to adapt itself in the best way possible to unforeseen changes. Scheduling stability measures the number of revisions or changes that a schedule undergoes during execution, may it be caused by internal or external reasons. Scheduling nervousness (already introduced in Sect. 1.2) is defined as the amount or the share of significant changes in the execution phase of the schedule. All four expressions are closely related to each other and, depending on the problem under consideration might be precisely defined and/or measured in many different ways.
- Partly already belonging to the aspects to be discussed next with respect to formal adequacy, it is part of the analysis of real-world adequacy whether and how good the interface between the formal sphere and the real-world sphere fits to the solution of the underlying (real-world) decision problem.

7.6.2 Formal Adequacy

As already mentioned, the check of formal adequacy refers exclusively to the formal sphere of a decision problem as sketched out in Fig. 1.2. If we suppose that a given formal problem in connection with one or several algorithms yields one or more

solutions, formal adequacy then refers exclusively to the evaluation of the quality of the solution(s) within this formal framework. The issue of formally evaluating algorithms becomes particularly acute in the case of approximate algorithms, since we already know that, for most scheduling decision-making problems, we will obtain models that cannot be solved in an exact manner, at least not in reasonable time.

Since some algorithms—and in particular metaheuristics—depend on some parameters, the term 'different algorithms' also include comparing two versions (i.e. with different parameters) of the same algorithm.

The evaluation should refer to (at least) two indicators, e.g. objective function value and computation time, which results in a multi-dimensional evaluation of the algorithms. Usually, there will be a trade-off between these two dimensions which has to be interpreted as in standard multi-criteria evaluations. Without loss of generality, we will usually refer to the first indicator, i.e: the objective function value.

With these two performance indicators in mind, two possibilities are usually employed for the evaluation of algorithms:

- Benchmark performance. This often means to give an upper bound for the deviation of the algorithm's solution from the optimal objective function value. For example, the algorithm can be shown to yield a solution that is at most x % worse than the optimal solution's objective function value. However, results of this type are not as often to be derived. With respect to the computation time, an upper bound for the effort related to the accomplishment of the algorithm and depending on the problem instance length (e.g. measured by the number of jobs or operations to be scheduled) might be given. This upper bound analysis for the computational effort is provided by the computational complexity, which has been addressed in Sect. 7.3.
- Comparing two or more algorithms among each other. In this case, two different approaches (which will be discussed in the next sections) can be employed:
  1. A formal proof can be given showing one algorithm to perform better than another one, always or for some characteristics of the instances of the model.
  2. A simulation study (also many times referred in the literature as to *computational experiment*) tries to derive some dominance results between the algorithms.

7.6.2.1 Formal Proofs

A formal proof could be given, e.g. for the trivial situation that an algorithm is a refinement of another one, in the sense that it examines explicitly or implicitly all solutions of the non-refined algorithm. Then, obviously, the refined algorithm will never be worse than the non-refined one. However, conclusions of above type mostly are either trivial or non-available. An alternative is to compare the performance of several algorithms by means of *worst-case analysis*.

Let us formalise the main concepts behind worst-case analysis: Assume that, for a given problem instance l, the objective function value for the algorithm under

consideration is $OFV_a(l)$ and the respective objective function value of the reference algorithm is $OFV_{\text{ref}}(l)$. A worst-case analysis results usually expresses that $OFV_a(l)$ and $OFV_{\text{ref}}(l)$ (or the respective optimal objective function values) are linked by some function as

$OFV_a(l) \leq f(OFV_{\text{ref}}(l))$ or $OFV_a(l) \leq f(OFV_{\text{ref}}(l), m, n, ...)$ for instance l
or
$OFV_a^* \leq f(OFV_{\text{ref}}^*)$ or $OFV_a^* \leq f(OFV_{\text{ref}}^*, m, n, ...)$ for the respective optimal objective function values.

An example for the latter is given for the 2-machine flow shop problem with makespan objective and limited intermediate storage of b jobs between the two machines (Papadimitriou and Kanellakis 1980). These authors compare the (polynomially solvable) approach for the respective no wait problem (obviously not needing any buffer between the two machines) yielding makespan $C^*_{\max,NW}$ with the (optimal) solution to the buffer-constrained problem $C^*_{\max,b}$, i.e. they apply the generation of the no-wait solution as an approximate method to the buffer-constrained problem and use an exact enumerative approach for the buffer-constrained problem as reference approach. They prove that

$$C^*_{\max,NW} \leq \frac{2b+1}{b+1} C^*_{\max,b}$$

holds. Obviously, therefore the optimal solution to the no-wait problem, used as heuristic solution to the buffer-constrained problem, is not worse than twice the optimal solution for the buffer-constrained problem.

Although these analytical results are smart, their contribution to the evaluation of an approximate approach might be regarded restrainedly, due to the following reasons:

- The results are only valid for the worst case. To what extent this worst case is relevant for the problem solution has to be evaluated separately.
- The quality of worst-case conclusions might be somewhat disillusioning. In the above example, from a mathematical point of view, the result is definitely interesting. To what extent a limitation of the worst case deviation from the optimal solution by 100 % provides valuable information for a possibly underlying real-world situation or even only for the formal approximate solution approach via the no-wait problem might be discussed controversially.
- Analytical results of the above nature are only available for a few and rather simple problem settings.

7.6.2.2 Simulation Study

Given the limitations of formal proofs, simulation studies using a set of sample instances for the evaluation of an approximate method are the most widespread method for the evaluation of the performance of several algorithms. There is no

universally accepted approach for conducting such simulation studies, but it can be summarised into two steps:

1. Testbed generation. A certain number of instances of the model are either generated specifically for the model, or adapted from those employed for other models. This set of sample instances are usually denoted as *testbed*. Note that this adaptation might take place twofold: Either the adaptation refers only to the method of generating test instances while the instances themselves are generated anew, or the instances are taken directly from the references or from some public databases.
2. Algorithms analysis. The algorithms to be compared are run with every single problem instance, and some metrics/analyses are employed to establish the relative performance of each algorithm on the testbed. If, according to the metrics/analysis, one algorithm is found to outperform the other on the testbed, then it is assumed that the latter would, in general, perform better than the former.

It is clear that the correctness of each step greatly affects to the goal, which is to establish, in general, the relative performance of different algorithms. Next we give some remarks on each one of these steps.

Regarding testbed generation, the following consideration should be followed:

- The size of the problem instances (e.g. indicated by the number of jobs and/or machines) should be in line with real-world problem dimensions and/or the dimensions of problem instances used in comparable studies. This includes also the different combinations of number of jobs and number of machines, number of operations, etc.
- The number of problem instances should be sufficient to derive useful results, i.e. this number should reflect the real-world problem setting and/or the statistical significance of the conclusions to be obtained in the next step. Note that, if the algorithms under comparison are deterministic, a single run of each algorithm on every instance will suffice in step 2. However, if the algorithms under comparison are of stochastic nature, then one option is to design a testbed large enough to average the performance of these algorithms over all the testbed.
- The main parameters of the model (such as processing times, set up times, release dates, etc.) should be chosen adequately with respect to the intended goal. For instance, highly diverse processing times (e.g. randomly drawn from a uniform distribution between 1 and 99, as is often proposed) might yield complicate problem instances and therefore might give indications for some kind of robustness of an algorithm relative to the heterogeneity of processing times. However, as compared with real-world problem settings, more or less significant correlations of processing times with respect to different jobs on a specific machine/stage and/or with respect to different operations of a specific job on different machines might be related closer to real-world applications—and might yield different conclusions with respect to the (relative) performance of the algorithms considered. Note that a hypothesis often addressed with respect to correlations in processing times is that the relative performance differences of algorithms decrease if the correlation of processing times increases. However, this is by no means a conclusion valid for

all scheduling models. A loophole from this problem with respect to real-world applications is to base the generation of test instances on observations from the specific real-world case.

- When extending/adapting an existing testbed to different models, the additional features of the model under consideration, e.g. due dates, availability assumptions, precedence constraints, have to be carefully determined and embedded into the test instances. For instance, taking literature-based standard testbeds for permutation flow shop models with makespan objective and using these instances for problems with due dates and tardiness objective requires a careful determination of the due dates. However, setting the due dates too tight on one hand, results in not reachable due dates with the consequence that approaches which intend to maximise throughput will be adequate while very loose due dates on the other hand will always enable to reach these due dates and eases the optimisation problem from the opposite side. Although this observation may seem trivial, it is sometimes ignored or at least not explicitly regarded in some references, i.e. the discussion of the adequacy of adapted testbeds is often neglected or at least not dealt with satisfactorily.

Regarding testbed analysis, once the objective function values for each algorithm on each instance have been obtained, some metrics can be derived for analysis. Although one option would be to employ the objective function values as a performance indicator, it is usually preferred to employ the *Relative Percentage Deviation* (RPD) metric for each algorithm and instance. RPD of an instance l ($l = 1, \ldots, L$) is usually defined as:

$$RPD(l) = RPD_{a,\mathrm{ref}}(l) = \frac{OFV_a(l) - OFV_{\mathrm{ref}}(l)}{OFV_{\mathrm{ref}}(l)}.$$

Recall from Sect. 7.6.2.1 that $OFV_a(l)$ is the objective function value obtained by algorithm l when applied to instance l, and $OFV_{\mathrm{ref}}(l)$ the respective objective function value of the reference algorithm.

Considering the whole set of L test instances the *Average Relative Percentage Deviation* of a algorithm l is defined as

$$ARPD = ARPD_{a,\mathrm{ref}} = \frac{1}{L}\sum_{l=1}^{L} \frac{OFV_a(l) - OFV_{\mathrm{ref}}(l)}{OFV_{\mathrm{ref}}(l)}.$$

In many references, particularly older ones, the algorithms under consideration are exclusively evaluated by just comparing the *ARPD* values in a more or less 'folkloristic' manner, i.e. by verbally interpreting similarities and/or differences. In addition, but only sometimes standard deviation or variance values of *ARPD* are addressed in the verbal discussion of the performance of the algorithms. Note that *ARPD* refers to an average performance, while usually scheduling algorithms do not have the same 'performance profile' over the whole benchmark, i.e. one algorithm might be better for some type of instances than another algorithm and vice versa.

Recently, more and more statistical test procedures become standard to evaluate the relative performance of approximate methods. We explicitly approve such statistical approaches because they give at least some statistical reasoning for the relative performance of the approaches under consideration.

To apply statistical test procedures, first a hypothesis has to be formulated. Afterwards, an adequate test procedure has to be chosen and the number of problem instances for a certain level of significance of the evaluation of the hypothesis has to be determined or, vice versa, for a given number of instances a level of significance can be derived.

For small sizes of problem instances, especially with respect to the number of jobs, $OFV_{\text{ref}}(l)$ might be the value of the optimal solution (obtained e.g. by means of an enumeration approach). Then *ARPD* gives the average deviation from the optimal objective function value. For larger problem sizes, an algorithm cannot be compared to the optimal solution but only to another algorithm or to (the best value of) a set of algorithms. An alternative is to use some bound value for the optimal objective function value as a reference point.

For those instances for which the optimal solution can be determined, a hypothesis concerning the mean values may be formulated as follows:

$$\text{Hypothesis: } ARPD_{am,\text{ref}} \leq \alpha \text{ or equivalently}$$

i.e. subject to a certain level of significance, the solution value of the approximate method will not deviate more than α % from the optimal objective function value.

For larger problem sizes, a similar hypothesis can be stated as well. However, since in this case the optimal objective function value is not available for comparison, the deviation from the best approach among the ones considered can be covered by respective hypotheses.

In addition and more common for larger problem sizes, comparing two different algorithms $a1$ and $a2$, an appropriate hypothesis can be stated as

$$\begin{aligned} \text{Hypothesis: } & ARPD_{a1,\text{ref}} \leq ARPD_{a2,\text{ref}} \text{ or equivalently} \\ & DIST_{a2,a1} := ARPD_{a2,\text{ref}} - ARPD_{a1,\text{ref}} \geq 0, \end{aligned}$$

The reference then can be defined as the best solution of $a1$ and $a2$ for every instance, i.e. $OFV_{\text{ref}} = \min(OFV_{a1}, OFV_{a2})$. The confirmation of this hypothesis (or the rejection of its opposite) would state that $a1$ performs better than $a2$, relative to the reference approach and subject to a certain level of significance. (Remark: Sometimes choosing random solutions as one approximate method and doing a respective statistical comparison analysis proves or refutes the assertion that a proposed method is at least competitive with a random choice of a solution. However, in most—but not all!—cases of methods proposed, the result of this comparison of hypotheses should be obvious.)

Having precisely defined the hypothesis to be tested, the toolbox of statistical test procedures yields adequate approaches to decide whether to accept or to reject the hypothesis. We will not describe these test procedures in detail but refer the reader to numerous textbooks on statistical testing (see Sect. 7.7 on further readings).

Instead, we give some hints for the application of test procedures when evaluating (approximate) methods in manufacturing scheduling.

If simply one approximate method is compared with another one, paired t-test procedures can be applied. t-test approaches can be used if $DIST_{a2,a1}$ follows a normal distribution. This requirement is often supposed to be fulfilled without explicit proof. If normal distribution of $DIST_{a2,a1}$ is questionable, an approximate Gauss test approach (parametric test) or Wilcoxon's signed ranks test (non-parametric test) are more appropriate since they do not require any assumption with respect to the type of the distribution of $DIST_{a2,a1}$.

If more than two heuristics are to be compared with respect to their performance, ANalysis Of VAriance (ANOVA) can be used to accept or to reject the hypothesis that all algorithms l under consideration perform equally with respect to their $ARPD_{a,l,\mathrm{ref}}$-values. If this hypothesis is rejected (what is desired to determine a best method or at least significant differences between (some of) the methods under consideration) it has to be regarded that the identification of better or best approaches among the approaches under consideration is based on the (pairwise) comparison of these approaches on the same testbed. This, however, requires statistical corrections when applying test procedures to more than one pair of approaches. Among others, the most popular of these correction approaches are by Tukey, Tukey-Kramer, Bonferroni and Holm-Bonferroni. Again, we will not discuss this aspect in detail, but just mention that the correction procedure refers to some adjusted interpretation of the p-values when determining the level of significance of the evaluation of the respective hypotheses.

Additionally, we point out that applying ANOVA demands for the fulfilment of three requirements, i.e.

1. Randomness and independence
2. Normality: sample values are from a normal distribution
3. Homogeneity of variance/homoscedasticity: variances of every indicator are the same.

Before applying ANOVA to hypotheses testing, these requirements have to be checked. If these requirements are not fulfilled, the Kruskal-Wallis rank test might be applied.

Finally, we point out that most of the statistical test procedures mentioned here and the respective discussions concerning the relative performance of the approximate scheduling approaches explicitly or implicitly refer to the so-called α error (type 1 error), i.e. how large is the risk to reject a hypothesis although it is true. The β error (type 2 error), i.e. the risk to accept a hypothesis although it is false, is usually not considered when statistically evaluating the performance of scheduling approaches.

Concluding this section, we would like to point out that also other more or less different statistical approaches can yield additional insight in the performance of approximate methods. For example, referring to the complete distribution of objective function values for a problem and comparing this distribution with the solutions generated by an approximate method may yield additional insight into the approximate method's performance.

7.6.2.3 Further Remarks on the Evaluation of Algorithms

Summarising the previous section, partly also from a real-world perspective, we point to the following aspects:

1. We recall once again that a properly defined and executed statistical analysis of the performance of algorithms for manufacturing scheduling models should be standard when evaluating these methods. However, we have the impression that this has not been accomplished so far.
2. Another aspect, also relevant from a real-world application point of view is the (non-) necessity of a certain level of solution quality. Especially because of arguments referring to the uncertainty of many problem data, a medium-size (whatever that means) deviation from a best solution might be accepted within the formal problem since the real-world implementation will differ from the solution derived for the formal problem more or less significantly anyway. Therefore, pre-determination of non-zero levels of acceptance (with respect to optimisation error and/or degree of infeasibility) might be a setting even in the formal sphere of the optimisation process. However, an inevitable but also rather trivial requirement for the solution of the formal problem is its ability to be transformed to an adequate solution of the real-world problem or at least to deliver some valuable additional information to such a solution.
3. For an externally (by the decision maker) given level of significance for the acceptance or the rejection of a hypothesis of the above type, the test procedures will give a minimum number of problem instances to be generated and calculated (or vice versa: for a given number of instances a level of significance can be derived).
4. If several (i.e. more than one) hypotheses shall be accepted or rejected using the same testbed, i.e. more than two approaches are to be compared, the respective comparisons are not statistically independent anymore. Therefore, in this case either the level of significance decreases (if the number of instances is not increased) or the number of instances has to be increased (to maintain a pre-specified level of significance). In statistical analyses of the performance of approximate methods for manufacturing scheduling (where often more than two methods are compared), this aspect is ignored rather often.
5. If a thorough statistical analysis of the performance of scheduling approaches is performed, both types of errors in statistical test theory, i.e. α error (type 1 error) as well as β error (type 2 error), should be addressed.
6. Finally, not only the quality of a given algorithm for a given formal manufacturing scheduling problem has to be considered. Also the effort for the process of setting up the model itself as well as of the solution algorithm might be limited.

7.7 Conclusions and Further Readings

This chapter has presented a brief overview on many possible scheduling methods. We have initially discussed the complexity of scheduling problems from a simple approach avoiding as much as possible the technicalities of computational complexity. The conclusion from this is that most scheduling problems are computationally complex and have an astronomical number of possible solutions. We have also pointed out the relative importance of obtaining the exact optimum solution for a scheduling problem, given that what we are actually solving is a model from the studied reality and the approximations and simplifications made to that model might perfectly offset the gains of a precisely obtained optimal solution.

The main scheduling methods have been introduced in this chapter as well. Each method has its own advantages and disadvantages. In a nutshell, dispatching rules are easy to understand, to implement and to calculate. However, their results are often poor except for some simplistic problems and are easily surpassed by most scheduling algorithms. Exact approaches provide the optimum solution but are only viable for simple problems and for small instances as well. Unless some very specific problem is being studied for which a very effective exact optimisation approach exists, they are not advisable to be used in complex production shops where hundreds, potentially thousands of jobs are to be scheduled. Heuristics are often tailored for some specific problems and objectives but they usually provide much better solutions when compared to dispatching rules. With today's computing power, heuristics are practically as fast as dispatching rules.

The final part of the chapter is devoted to discussing how to assess scheduling methods, both from formal and real-world points of view. From a formal viewpoint, the predominant approach for approximate methods are the simulation studies, for which no standard procedure yet exists, although we give some insight into the basics of these analyses.

Interested readers wishing to delve into the world of computational complexity should start with the books of Garey and Johnson (1979), Papadimitriou (1979) or Arora and Barak (2009), just among the many more specialised texts. Studies about computational complexity for scheduling problems appeared in Garey et al. (1976), Rinnooy Kan (1976), Błazewicz et al. (1983), Lawler et al. (1993) and many others. The book of Brucker (2007) and the one of Brucker and Knust (2006) contain a large deal of complexity results for scheduling problems. The same authors run a website with complexity results for scheduling problems at http://www.mathematik.uni-osnabrueck.de/research/OR/class/. This website is very detailed and contains lots of up-to-date information.

Many dispatching rules and heuristics are covered by the works of Panwalkar and Iskander (1977), Blackstone et al. (1982), Haupt (1998), Rajendran and Holthaus (1999) or Jayamohan and Rajendran (2000). Some advanced dispatching rules (including many global machine assignment rules) can be found in Urlings et al. (2010).

A classical text for heuristics is the book of Morton and Pentico (1993). For a much more detailed overview on exact algorithms, dispatching rules and heuristic algorithms, the reader may consult the general books on scheduling: Conway et al. (1967), Baker (1974), French (1982), Błazewicz et al. (2002), Brucker (2007), Pinedo (2012), Baker and Trietsch (2009) and Pinedo (2009). Regarding the discussion of the simulation studies, the reader might want to have a look at Bamberg and Baur (1998), Berenson et al. (2006), Montgomery (2012), particularly for covering the technical/statistical issues. Finally, an interesting simulation study regarding correlations in processing times and the relative performance of algorithms is Watson et al. (2002).

References

Arora, S. and Barak, B. (2009). *Computational Complexity: A Modern Approach.* Cambridge University Press, Cambridge.

Baker, K. R. (1974). *Introduction to Sequencing and Scheduling.* John Wiley & Sons, New York.

Baker, K. R. and Trietsch, D. (2009). *Principles of Sequencing and Scheduling.* Wiley, New York.

Bamberg, G. and Baur, F. (1998). *Statistik.* Oldenbourg, Muenchen, Wien.

Berenson, M. L., Levine, D. M., and Krehbiel, T. C. (2006). *Basic business statistics: Concepts and applications.* Pearson Education, Upper Saddle River, NJ.

Blackstone, Jr, J. H., Phillips, D. T., and Hogg, G. L. (1982). A state-of-the-art survey of dispatching rules for manufactuing job shop operations. *International Journal of Production Research,* 20(1):27–45.

Błazewicz, J., Ecker, K. H., Pesch, E., Schmidt, G., and Węglarz, J. (2002). *Scheduling Computer and Manufacturing Processes.* Springer-Verlag, Berlin, second edition.

Błazewicz, J., Lenstra, J. K., and Rinnooy Kan, A. H. G. (1983). Scheduling Subject to Constraints: Classification and Complexity. *Discrete Applied Mathematics,* 5:11–24.

Brucker, P. (2007). *Scheduling Algorithms.* Springer, New York, fifth edition.

Brucker, P. and Knust, S., editors (2006). *Complex Scheduling.* Springer-Verlag, Berlin.

Conway, R. W., Maxwell, W. L., and Miller, L. W. (1967). *Theory of Scheduling.* Dover Publications, New York. Unabridged publication from the 1967 original edition published by Addison-Wesley.

Domschke, W., Scholl, A., and Voss, S. (1997). *Produktionsplanung: Ablauforganisatorische Aspekte.* Springer, Berlin. 2nd, revised and upgraded edition.

Du, J. and Leung, J. Y. T. (1990). Minimising total tardiness on one machine is NP-hard. *Mathematics of Operations Research,* 15(3):483–495.

French, S. (1982). *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop.* Ellis Horwood Limited, Chichester.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability. A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York.

Garey, M. R., Johnson, D. S., and Sethi, R. (1976). The Complexity of Flowshop and Jobshop Scheduling. *Mathematics of Operations Research,* 1(2):117–129.

Groetschel, M. and Lovasz, L. (1993). *Geometric algorithms and combinatorial optimization.* Springer, Berlin [a.o.]. 2th, corrected edition.

Haupt, R. (1989). A survey or priority rule-based scheduling. *OR Spectrum,* 11(1):3–16.

Jayamohan, M. S. and Rajendran, C. (2000). New dispatching rules for shop scheduling: a step forward. *International Journal of Production Research,* 38(3):563–586.

Lawler, E. L., Lenstra, J. K., and Rinnooy Kan, A. H. G. (1993). Sequencing and Scheduling: Algorithms and Complexity. In Graves, S. C., Rinnooy Kan, A. H. G., and Zipkin, P. H., editors,

Logistics of Production and Inventory, volume 4 of Handbooks in Operations Research and Management Science, Amsterdam. Elsevier Science Publishers, B. V.

Montgomery, D. C. (2012). *Design and Analysis of Experiments.* Wiley; 8 edition.

Morton, T. E. and Pentico, D. W. (1993). *Heuristic Scheduling Systems With Applications to Production Systems and Project Management.* Wiley Series in Engineering & Technology Management. John Wiley & Sons, Hoboken.

Panwalkar, S. and Iskander, W. (1977). A survey of scheduling rules. *Operations Research,* 25(1):47–55.

Papadimitriou, C. H. (1993). *Computational Complexity.* Addison-Wesley, Reading.

Papadimitriou, C. H. and Kanellakis, P. C. (1980). Flowshop scheduling with limited temporary-storage. *Journal of the ACM,* 27(3):533–549.

Pinedo, M. (2009). *Planning and Scheduling in Manufacturing and Services.* Springer, New York, second edition.

Pinedo, M. L. (2012). *Scheduling: Theory, Algorithms, and Systems.* Springer, New York, fourth edition.

Rajendran, C. and Holthaus, O. (1999). A comparative study of dispatching rules in dynamic flowshops and jobshops. *European Journal of Operational Research,* 116(1):156–170.

Rinnooy Kan, A. H. G. (1976). *Machine Scheduling Problems: Classification, Complexity and Computations.* Martinus Nijhoff, The Hague.

Urlings, T., Ruiz, R., and Sivrikaya-Şerifoğlu, F. (2010). Genetic algorithms with different representation schemes for complex hybrid flexible flow line problems. *International Journal of Metaheuristics,* 1(1):30–54.

Vepsalainen, A. P. J. and Morton, T. E. (1987). Priority rules and lead time estimation for job shop scheduling with weighted tardiness costs. *Management Science,* 33(8):1036–1047.

Watson, J.-P., Barbulescu, L., Whitley, L., and Howe, A. (2002). Contrasting structured and random permutation flow-shop scheduling problems: Search-space topology and algorithm performance. *INFORMS Journal on Computing,* 14:98–123.

Chapter 8
Exact Algorithms

8.1 Introduction

In the previous chapter we discussed that most scheduling models are hard to be solved optimally and that there is little hope in expecting the optimal solution for complex problems arising in real production settings. However, some simple problems might arise as subproblems or surrogate settings in complex shops. Additionally, exact algorithms help in closely and deeply understanding scheduling models and therefore, they have an intrinsic utility. Moreover, some exact algorithms do exist for some special cases of scheduling problems and are worth of study. Finally, we have already mentioned that some of the exact procedures can be employed as approximate methods (e.g. by terminating them before optimality is reached), and can serve to inspire the construction of approximate methods for more complex problems.

After the introduction to exact methods for scheduling models carried out in Sect. 7.5.1, this chapter deepens in these methods, and we discuss both exact and constructive algorithms, and some widely employed exact algorithms, i.e. branch and bound approaches as well as dynamic programming procedures.

More specifically, in this chapter we

- discuss the most important exact constructive algorithms and their limitations (Sect. 8.2),
- present exact algorithms based on integer programming (Sect. 8.3.1) and on problem-specific branch and bound approaches (Sect. 8.3.2) and
- introduce the main concepts of dynamic programming employed in manufacturing scheduling (Sect. 8.3.3).

J. M. Framinan et al., *Manufacturing Scheduling Systems*,
DOI: 10.1007/978-1-4471-6272-8_8, © Springer-Verlag London 2014

8.2 Exact Constructive Algorithms

Recall from Sect. 7.5.1 that exact constructive algorithms attempt to exploit specific properties of the scheduling model in order to construct a solution which is guaranteed to be optimal.

There are some cases for which finding exact constructive algorithms is, if not trivial, rather straightforward. For instance, for the single-machine model with makespan objective (model $1||C_{\max}$) every schedule yields the same makespan if the operations are assigned left shifted to the timescale. Therefore every solution is optimal. For the 1-machine model with total completion time objective (model $1||\sum C_j$), we have already discussed in Sect. 7.5.1 that sorting the jobs according to the shortest processing time-rule (*SPT*-rule) yields the optimal solution. This sorting can be performed with computational effort which is limited by $O(n \cdot \log n)$. Similarly, the 1-machine problem with due dates and the objective of minimising maximum lateness (model $1||\max L_j$) can be solved easily by sorting the jobs according to the earliest due date rule (*EDD*-rule).

For other models, deriving exact constructive algorithms is not trivial or straightforward. Indeed, apart from some single machine layouts and some rather simplistic two-machine settings, there are not many other known exact algorithms. However, exact constructive algorithms may play an important role in the development of approximate algorithms, as we will discuss in Chap. 9. Therefore, it is worth studying in the next subsections some of the most famous.

8.2.1 Johnson's Algorithm

Probably the most famous case of an exact constructive algorithm is the well-known *Johnson's rule* (Johnson 1954) for solving the 2-machine permutation flow shop model with makespan criterion or $F2|prmu|C_{\max}$. Note than in Conway et al. (1967), as well as in many other texts, it is actually demonstrated that the permutation schedules suffice for obtaining the optimum makespan in 2-machine flow shops. Johnson's rule determines that a given job j should precede in the sequence another job k if the following expression is satisfied:

$$\min\{p_{1j}, p_{2k}\} < \min\{p_{1k}, p_{2j}\}$$

In this way, jobs with a short processing time on the first machine go first in the sequence (this in turn minimises the idle time in the second machine). Jobs with a short processing time in the second machine are processed last also to avoid idle times in the second machine. Note that in a 2-machine flow shop without constraints, there are no idle times on the first machine. Therefore, minimising makespan is equivalent to minimising idle times on the second machine. Johnson's algorithm is further described Algorithm 1.

Table 8.1 Processing times for a six job and two machine flow shop example to be solved with Johnson's rule

Machine (i)	Job (j)					
	1	2	3	4	5	6
1	8	4	10	5	3	12
2	12	1	6	5	2	8

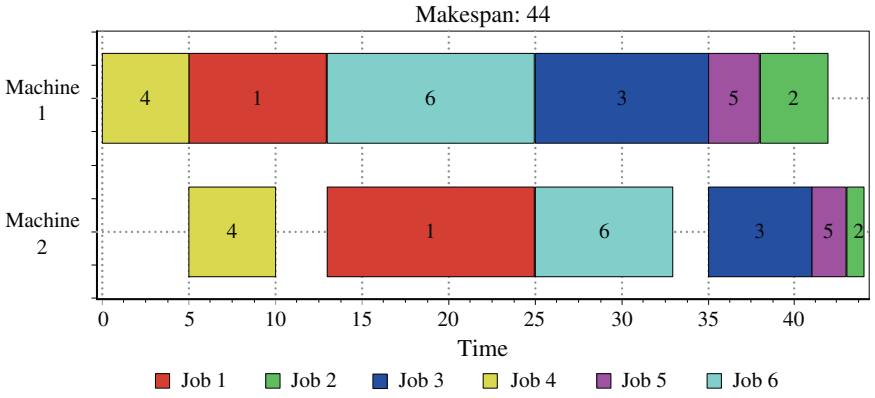

Fig. 8.1 Gantt chart with the resulting sequence in the Johnson's rule example

Algorithm 1: Johnson's Algorithm

Input: Instance data
Output: Optimal sequence Π
begin
  Let $\Pi = \varnothing$, $J = \{1, \ldots, n\}$;
  Let $J_1 = \{j \in J / p_{1,j} \leq p_{2,j}\}$;
  Let $J_2 = \{j \in J / p_{1,j} > p_{2,j}\}$;
  Construct Π_1 sorting jobs in J_1 by increasing values of $p_{1,j}$;
  Construct Π_2 sorting jobs in J_2 by decreasing values of $p_{2,j}$;
  $\Pi = (\Pi_1 | \Pi_2)$;
  return Π
end

Let us apply Johnson's algorithm to an instance comprised of six jobs whose processing times on the two machines are given in Table 8.1.

According to Johnson's rule, $J_1 = 1, 4$ and $J_2 = 2, 3, 5, 6$. Consequently, $\Pi_1 = (4, 1)$ and $\Pi_2 = (6, 3, 5, 2)$. Therefore, $\Pi = (4, 1, 6, 3, 5, 2)$ with $C^*_{\max} = 44$. This optimal schedule is represented by a Gantt chart in Fig. 8.1.

Johnson's rule can be implemented with a computational complexity of $O(n \log n)$. This means that even for thousands of jobs, Johnson's rule can obtain the optimum solution in a very short amount of time for 2-machine flow shop layouts and makespan objective. Most regrettably though, adding almost any constraint and/or changing the objective or increasing the number of machines, results in Johnson's rule is not being able to return the optimum solution.

8.2.2 Lawler's Algorithm

This algorithm was shown optimal by Lawler (1973) for the model $1|prec|\gamma$, being γ any regular objective function with max-form. Let $\max g_j$ be a regular max-form function. The algorithm is shown in Algorithm 2.

Algorithm 2: Lawler's Algorithm for $1|prec|\max g_j$ Lawler (1973)

Input: Instance data
Output: Optimal sequence Π
begin
  Let $\Pi = \varnothing$, $J = \{1, \ldots, n\}$ and J' the set of all jobs with no successors.;
  while J *is* not $\varnothing$ **do**
    Let j^* be the job for which

$$g_{j^*}\left(\sum_{k\in J} p_k\right) = min_{j\in J'}\left(g_j\left(\sum_{k\in J} p_k\right)\right)$$

    Add j^* to Π;
    Delete j^* from J;
    Modify J' to represent the new set of schedulable jobs;
  return Π
end

8.2.3 Moore's Algorithm

The problem $1||\sum U_j$ is solved optimally by the algorithm presented in Algorithm 3 by Moore (1968).

Algorithm 3: Moore's Algorithm for $1||\sum U_j$ Moore (1968)

Input: Instance data
Output: Optimal sequence Π
begin
  Let $\Pi = \varnothing$, $J = \{1, \ldots, n\}$ verifying $d_1 \leq d_2 \leq \ldots d_n$;
  Let $\Pi = (1, \ldots, n)$;
  Let $Tardy = \varnothing$;
  while $\exists l \in \Pi$ *such as* $C_l > d_l$ **do**
    L
  et k be such that $C_k > d_k$ and $\forall i < k,\ C_i \leq d_i$;
  Let j be such that $j \leq k$ and $p_j = \max 1 \leq i \leq k\, p_i$;
  $\Pi = \Pi - \{j\}$; $Tardy = Tardy \cup \{j\}$; **return** Π
end

8.3 Enumerative Algorithms

Enumerative algorithms implicitly or explicitly guarantee to evaluate all possible solutions of the model. These algorithms include a large number of different approaches, and discussing all of them will be outside the scope of this book. Instead we will focus in the next subsections on the three types of solution methods that are more popular in manufacturing scheduling namely Integer Programming (Sect. 8.3.1), Branch and Bound (Sect. 8.3.2), and Dynamic Programming (Sect. 8.3.3). Note that, from a pure technical viewpoint, Integer Programming could be seen as a particular type of (generic) branch and bound. However, given the specific characteristics of Integer Programming, we prefer to discuss it in a separate section.

8.3.1 Integer Programming (IP)

Many manufacturing scheduling models can be formulated as integer or binary (linear or nonlinear) programs, therefore the whole variety of branch and bound methods in binary or integer programming can be applied. This method would consist basically of two steps:

1. Formulation of the manufacturing scheduling model under consideration in terms of MILP (Mixed Integer Linear Programming). This formulation will usually either directly derive a solution or refer to some standard intermediate model which can be transferred to an integer program in a standard way, e.g. by using models from graph theory (see, e.g. Pinedo 2012; Tseng et al. 2004 for examples of such formulations).
2. Applying IP software to the MILP model generated in the previous step. Here, smart parameter adjustments within the IP software, e.g. error bounds, branching strategies, etc., have to be executed. Examples of software dealing with MILP models are the solver of Microsoft Excel, LINGO from Lindo Systems (http://www.lindo.com/), CPLEX from IBM (http://www-01.ibm.com/software/integration/optimisation/cplex/) and many others, including the Gurobi solver (http://www.gurobi.com/).

Since the last step is more or less exclusively referring to the standardised formal problem and to the software used, we will not refer to this here. Instead, we will sketch out some model formulations which cover the first step. More specifically, we will present in the next sections MILP formulation of several models in order to illustrate how to capture the processing constraints and formulate the objectives. It is to note that a smart formulation of the MILP model might reduce the computational effort significantly.

8.3.1.1 Parallel Machine Scheduling Models with Makespan Objective

In this section we consider two parallel machines scheduling models. Recall from Sect. 3.2.3.2 that in this layout there is a set N of n independent jobs that have to be processed on a set M of m machines arranged in parallel. The following (rather standard) assumptions are adopted:

- Each job j, $j = 1, \ldots, n$ has to be processed by exactly one out of the m parallel machines.
- No machine can process more than one job at the same time. Furthermore, once the processing of a job by a given machine has started, it has to continue until completion (e.g. no preemption is allowed).
- The processing time of a job is a known, finite and fixed positive number. In general, machine i ($i = 1, \ldots, m$) has a different speed p_{ij} when processing each job j, i.e. machines i will be occupied by p_{ij} units of time when processing job j.

For now, we will consider that there are no further restrictions (later we will introduce sequence-dependent set-up times in this model), so this model can be denoted as $R||C_{\max}$. Note that this is, in reality, an assignment problem since the processing order of the jobs assigned to a given machine do not alter the maximum completion time at that machine. Therefore, there are m^n possible solutions to the problem after all possible assignments. The $R||C_{\max}$ has been shown to be NP-Hard, since the special case with identical machines (referred to as $P||C_{\max}$) was already demonstrated by Garey and Johnson (1979) to belong to that class. Even the 2-machine version ($P2||C_{\max}$) is already NP-Hard according to Lenstra et al. (1977).

As we can see, even the two identical parallel machines case is a hard problem to solve. Furthermore, Lenstra et al. (1990) showed that no polynomial time algorithm exists for the general $R||C_{\max}$ problem with a better worst-case ratio approximation than 3/2.

A MILP model for $R||C_{\max}$ would contain the following variables:

$$x_{ij} = \begin{cases} 1, & \text{if job } j \text{ is assigned to machine } i \\ 0, & \text{otherwise} \end{cases}$$

$$C_{\max} = \text{Maximum completion time or makespan}$$

The objective function is the makespan minimisation:

$$\min \ C_{\max} \tag{8.1}$$

And the constraints:

$$\sum_{i=1}^{m} x_{ij} = 1 \quad \forall j \in N \tag{8.2}$$

$$\sum_{j=1}^{n} p_{ij} \cdot x_{ij} \leq C_{\max} \quad \forall i \in M \tag{8.3}$$

$$x_{ij} \in \{0, 1\} \quad \forall j \in N, \forall i \in M \tag{8.4}$$

Note that the constraint of type (8.2) ensures that each job is assigned to exactly one machine. The second set of constraints just states that the makespan is equal or greater than the sum of the processing times of the jobs assigned to each machine. This is repeated for all machines since in this problem, the makespan is given by the machine that finishes all its assigned jobs at the latest time. The last set of constraints just defines the nature of the variables.

The previous model was thoroughly tested by Fanjul-Peyró and Ruiz (2010) using IBM CPLEX 11.0 and the results were very encouraging. From a complete set of 1,400 instances, and with a maximum CPU time limit of 2 h, CPLEX 11.0 was able to solve instances of up to 1000 jobs and 50 machines in many cases, with optimality rates reaching 88 % in some cases and maximum-observed optimality gaps for the unsolved instances of 8.66 % and average-observed gaps usually below 2 %. As one can see, although the MILP model does not result in optimal solutions in all cases, the results are very good in most scenarios. This enforces the already commented idea that although most scheduling problems are indeed NP-hard, sometimes usable solutions for sizeable instances are possible.

Let us add sequence-dependent set-up times to the previous setting. As the parallel machines are unrelated, we assume that there is a set-up time matrix per machine. Therefore, S_{ijk} denotes the machine-based sequence-dependent set-up time on machine $i, i \in M$, when processing job $k, k \in N$, after having processed job $j, j \in N$.

The so-resulting model can be denoted as $R|S_{sd}|C_{\max}$. One could think that adding sequence-dependent setup times to the problem is not a big deal. However, the implications of the setups are far reaching as far as the model goes. Notice that when adding setups, this is no longer just an assignment problem. Depending on how the different jobs are sequenced among those assigned to a machine, the completion time of that machine will differ. As a result, the sequence has to be determined by the model as well, not just the assignment. The new MILP model involves the following decision variables:

$$\begin{aligned} X_{ijk} &= \begin{cases} 1, & \text{if job } j \text{ precedes job } k \text{ on machine } i \\ 0, & \text{otherwise} \end{cases} \\ C_{ij} &= \text{Completion time of job } j \text{ at machine } i \\ C_{\max} &= \text{Maximum completion time} \end{aligned}$$

As one can see, there are $n \cdot (n-1) \cdot m$ binary variables, which quickly grow to unacceptable numbers in medium-sized problems. Note that X_{ijj} is not defined. However, the total number grows because some dummy jobs are needed as well. Below are the details.

The objective function is:

$$\min C_{\max} \tag{8.5}$$

And the constraints are:

$$\sum_{i \in M} \sum_{\substack{j \in \{0\} \cup \{N\} \\ j \neq k}} X_{ijk} = 1, \qquad \forall k \in N \tag{8.6}$$

$$\sum_{i \in M} \sum_{\substack{k \in N \\ j \neq k}} X_{ijk} \leq 1, \qquad \forall j \in N \tag{8.7}$$

$$\sum_{k \in N} X_{i0k} \leq 1, \qquad \forall i \in M \tag{8.8}$$

$$\sum_{\substack{h \in \{0\} \cup \{N\} \\ h \neq k, h \neq j}} X_{ihj} \geq X_{ijk}, \qquad \forall j, k \in N, j \neq k, \forall i \in M \tag{8.9}$$

$$C_{ik} + V(1 - X_{ijk}) \geq C_{ij} + S_{ijk} + p_{ik}, \quad \forall j \in \{0\} \cup \{N\}, \forall k \in N, j \neq k, \forall i \in M \tag{8.10}$$

$$C_{i0} = 0, \qquad \forall i \in M \tag{8.11}$$

$$C_{ij} \geq 0, \qquad \forall j \in N, \ \forall i \in M \tag{8.12}$$

$$C_{ij} \leq C_{\max}, \qquad j \in N, \ i \in M \tag{8.13}$$

$$X_{ijk} \in \{0, 1\}, \qquad \forall j \in \{0\} \cup \{N\}, \forall k \in N, j \neq k, \forall i \in M \tag{8.14}$$

Constraint set (8.6) ensures that every job is assigned to exactly one machine and has exactly one predecessor. Notice the usage of dummy jobs 0 as $X_{i0k}, i \in M, k \in N$. With constraint set (8.7) we set the number of successors of every job to a maximum of one (a job could be the last one on a machine). Set (8.8) limits the number of successors of the dummy jobs to a maximum of one on each machine. With set (8.9), we ensure that jobs are properly linked on their machine. if a given job j is processed on a given machine i, a predecessor h must exist on the same machine. Constraint set (8.10) is to control the completion times of the jobs at the machines. Basically, if a job k is assigned to machine i after job j (i.e. $X_{ijk} = 1$), its completion time C_{ik} must be greater than the completion time of j, C_{ij} plus the setup time between j and k and the processing time of k. If $X_{ijk} = 0$, then the big constant V renders the constraint redundant. Sets (8.11) and (8.12) define completion times as 0 for dummy jobs and non-negative for regular jobs, respectively. Set (8.13) defines the makespan. Finally, set (8.14) defines the binary variables.

We can see how the sequence-dependent set-up times version of the unrelated parallel machines problem is significantly larger and more complex. As a matter of fact, and as shown in Vallada and Ruiz (2010), this model is quite unsolvable. The previous model, when tested also with CPLEX 11.0 and under comparable settings, resulted in optimum solutions for problems of up to 8 jobs and 5 machines in all cases. However, only some problems of 10 jobs and 12 jobs could be solved to optimality. We can see that these are extremely small problems.

8.3.1.2 Flow shop Scheduling Models with Makespan Objective

Several MILP formulations of flow shop or job shop problems (especially with makespan objective) are due to Liao and You (1992); Manne (1960); Wagner (1959). In this section, we will sketch out three different MILP formulations for elementary models for the permutation flow shop with makespan objective, i.e. the models by Manne (1960); Wagner (1959) and by Wilson (1989). We will then analyse the different formulations in order to highlight their implications in terms of their applicability beyond small-sized problems.

Manne's model (adapted for permutation flow shop problems as in Tseng et al. (2004):

The original version of the model by Manne is suited to solve the general job shop problem. However, for reasons of simplification and comparability to other approaches, we present the model in its permutation flow shop version which can be found (Table 8.2), e.g. in Tseng et al. (2004).

Suppose a standard permutation flow shop problem setting with makespan objective. The model is formulated as follows:

Table 8.2 Manne model: Notation

Indices	
i	Machine index, $i = 1, \ldots, m$
j	Job index, $j = 1, \ldots, n$
Variables	
$D_{j,j'}$	Binary variable with $D_{j,j'} = 1$ if job j is scheduled before (not necessarily immediately before) job j', $D_{j,j'} = 0$ otherwise, $j < j'$
C_{ij}	Completion time of job's j operation on machine i
$C_{\max}$	Finishing time of the last operation on machine m
Parameters	
p_{ij}	Processing time of job j on machine i
M	Large number

Model:

$$\min \quad C_{\max} \tag{8.15}$$

s. t.

$$C_{1j} \geq p_{1j} \qquad j = 1, \ldots, n \tag{8.16a}$$

$$C_{ij} - C_{i-1,j} \geq p_{ij} \qquad i = 2, \ldots, m; \quad j = 1, \ldots, n \tag{8.16b}$$

$$C_{ij} - C_{ij'} + M \cdot D_{j,j'} \geq p_{ij} \qquad i = 1, \ldots, m; \quad j, j' = 1, \ldots, n, \quad j < j' \tag{8.17a}$$

$$C_{ij} - C_{ij'} + M(D_{j,j'} - 1) \leq -p_{ij'} \qquad i = 1, \ldots, m; \quad j, j' = 1, \ldots, n, \quad j < j' \tag{8.17b}$$

$$C_{\max} \geq C_{mj} \qquad j = 1, \ldots, n \tag{8.18}$$

This model is more or less straight forward. Equation (8.15) represents the objective function of makespan minimisation. Equations (8.16a, 8.16b) give the job availability constraints, i.e. they guarantee that an operation (i, j) of a job j on machine i cannot be finished before its preceding operation $(i - 1, j)$ of the same job j on the preceding machine $i - 1$ is finished (no such operation on machine 1 in (8.16a)) and its processing time p_{ij} has passed. Equations (8.17a, 8.17b) give the machine availability constraints. In case of job j precedes job j' in a solution/permutation, constraint (8.17a) becomes a 'true' constraint while if job j' precedes job j, constraint (8.17b) is the relevant constraint while the respective other constraint becomes irrelevant in the respective situation. Constraints (8.18) give the lower limit of the makespan as the completion time of every single job j.

This model includes $0.5n(n - 1)$ binary variables $D_{j,j'}$ and $n \cdot m$ continuous variables C_{ij} plus one continuous variable $C_{\max}$, i.e. in total $n \cdot m + 1$ continuous variables. The number of constraints is nm (8.16a, 8.16b) plus $2mn(n-1)/2$ (8.17a, 8.17b) plus n (8.18), i.e. in total $n(mn + 1)$ constraints.

One characteristic of this adapted Manne model is that it basically only implicitly constructs the permutation by the jobs' precedence relation imposed by the binary variables $D_{j,j'}$. In contrast, the following models of Wilson and Wagner explicitly define permutations as solutions by an assignment problem structure.

Wilson model:

By including an assignment problem the Wilson model avoids the dichotomic constraints of the Manne model which cover both cases of job precedence for all pairs of jobs, i.e. job j_1 preceding job j_2 and vice versa (Table 8.3).

Model:

$$\min \quad C_{\max} = S_{m[n]} + \sum_{j=1}^{n} p_{mj} \cdot Z_{j[n]} \tag{8.19}$$

s. t.

$$\sum_{j=1}^{n} Z_{j[j']} = 1 \qquad j' = 1, \ldots, n \tag{8.20a}$$

Table 8.3 Wilson model: Notation

Indices	
i	Machine index, $i = 1, \ldots, m$
j	Job index, $j = 1, \ldots, n$
$[j']$	Position index, $j' = 1, \ldots, n$. This index indicates the j'-th position of a job permutation
Variables	
$Z_{j,[j']}$	Binary variable with $Z_{j,[j']} = 1$ if job j is assigned to position j' in the sequence/permutation, $Z_{j,[j']} = 0$ otherwise
$S_{i,[j']}$	Starting time of the operation of the job in position $[j']$ on machine i
$C_{\max}$	Finishing time of the last operation on machine m
Parameters	
p_{ij}	Processing time of job j on machine i

$$\sum_{j'=1}^{n} Z_{j[j']} = 1 \qquad j = 1, \ldots, n \tag{8.20b}$$

$$S_{1[1]} = 0 \tag{8.21a}$$

$$S_{1[j']} + \sum_{j=1}^{n} p_{1j} Z_{j[j']} = S_{1,[j'+1]} \qquad j' = 1, \ldots, n-1 \tag{8.21b}$$

$$S_{i[1]} + \sum_{j=1}^{n} p_{ij} Z_{j[1]} = S_{i+1,[1]} \qquad i = 1, \ldots, m-1 \tag{8.21c}$$

$$S_{i[j']} + \sum_{j=1}^{n} p_{ij} Z_{j[j']} \leq S_{i+1,[j']} \qquad i = 1, \ldots, m-1; \quad j' = 2, \ldots, n \tag{8.22}$$

$$S_{i[j']} + \sum_{j=1}^{n} p_{ij} Z_{j[j']} \leq S_{i,[j'+1]} \qquad i = 2, \ldots, m; \quad j' = 1, \ldots, n-1 \tag{8.23}$$

The objective function, i.e. the makespan, is given by (8.19) as the sum of the starting time of the last operation on machine m plus its processing time. Observe that the sum contains only one 'true' summand since only one job is assigned to the last position of the permutation and therefore all other binary variables in this sum are 0. Constraints (8.20a, 8.20b) represent the classical assignment constraints, i.e. every position of the permutation $[j']$ has to be assigned to exactly one job j (8.20a) and every job has to be assigned to exactly one position in the permutation (8.20b). Equations (8.21a, 8.21b, 8.21c) give the jobs' initialisation constraints on machine 1 (8.21b) and the machine initialisation constraints for the job in position [1]

Table 8.4 Wagner model: Notation

Indices	
i	Machine index, $i = 1, \ldots, m$
j	Job index, $j = 1, \ldots, n$
$[j']$	Position index, $j' = 1, \ldots, n$. This index indicates the j'-th position of a job permutation
Variables	
$Z_{j,[j']}$	Binary variable with $Z_{j,[j']} = 1$ if job j is assigned to position j' in the sequence/permutation, $Z_{j,[j']} = 0$ otherwise
$X_{i,[j']}$	Idle time on machine i before the operation of the job in position $[j']$ on machine i starts
$Y_{i,[j']}$	Idle time on machine i after the operation of the job in position $[j']$ on machine i is finished
Parameters	
p_{ij}	Processing time of job j on machine i

(8.21c). Equation (8.22) represent the job availability constraints while (8.23) give the machine availability constraints.

This model includes n^2 binary variables $Z_{j,[j']}$ (being n of them non-zero in every solution) and nm continuous variables $S_{i,[j']}$ plus one continuous variable $C_{\max}$, i.e. in total $nm + 1$ continuous variables. The number of constraints is $2n$ (8.20a, 8.20b) plus $(n + m - 1)$ (8.21a, 8.21b and 8.21c) plus $2(m - 1)(n - 1)$ (8.22 and 8.23), i.e. in total $2mn + n - m + 1$ constraints.

Wagner model (modified as in Tseng et al. 2004):
The Wagner model is probably the oldest MILP model for permutation flow shop scheduling. It combines the job availability constraints and the machine availability constraints of the Wilson model into one type of constraints (Table 8.4).

Model:

$$\min \quad C_{\max} = \sum_{j=1}^{n} p_{mj} + \sum_{j'=1}^{n} X_{m[j']} \tag{8.24}$$

s. t.

$$\sum_{j=1}^{n} Z_{j[j']} = 1 \qquad j' = 1, \ldots, n \tag{8.25a}$$

$$\sum_{j'=1}^{n} Z_{j[j']} = 1 \qquad j = 1, \ldots, n \tag{8.25b}$$

$$\sum_{j=1}^{n} p_{ij} Z_{j[j'+1]} - \sum_{j=1}^{n} p_{i+1,j} Z_{j[j']} + X_{i[j'+1]} - X_{i+1,[j'+1]} + Y_{i[j'+1]} - Y_{i[j']} = 0$$

$$i = 1, \ldots, m-1; \quad j' = 1, \ldots, n-1 \tag{8.26a}$$

$$\sum_{j=1}^{n} p_{ij} Z_{j[1]} + X_{i[1]} - X_{i+1,[1]} + Y_{i[1]} = 0 \qquad i = 1, \ldots, m-1 \tag{8.26b}$$

The objective function, i.e. the makespan, is given by (8.24) as the sum of the processing times of all jobs on the last machine (which is constant and decision irrelevant) and the idle time on the last machine. (This means nothing but that the minimisation of makespan is equivalent to minimising idle time on the last machine m.) Constraints (8.25a, 8.25b) represent the classical assignment constraints, as in the Wilson model above. Equations (8.26a, 8.26b) links the jobs on positions $\left[j'\right]$ and $\left[j'+1\right]$ on all machines, (8.26a) couples jobs 1 through n (or jobs [1] through $[n]$, respectively) while (8.26a) gives the initialisation for job 1. Equation (8.26b) indicates that

- the waiting time on machine i before the first job in the sequence [1] will start on i plus
- the processing time of [1] on i, i.e. $p_{i[1]}$, plus
- this job's waiting time after it is finished on i

are equal to this first job's waiting time on the next machine $i+1$. This can be seen easily by rearranging (8.26b) so that $X_{i+1,[1]}$ is isolated on one side of the equation.

Rearranging (8.26a) accordingly yields the expanded expression for the finishing time of job's $\left[j'+1\right]$ operation on machine $i+1$ as follows:

$$Y_{i[j']} + \sum_{j=1}^{n} p_{i+1,j} Z_{j[j']} + X_{i+1,[j']} = X_{i[j'+1]} + \sum_{j=1}^{n} p_{ij} Z_{j[j'+1]} + Y_{i[j'+1]}$$

This formula can be interpreted by means of Fig. 8.2. The figure, for reasons of clarity, supposes that the underlying permutation is the identical permutation, i.e. $j = [j]$ for all $j = 1, \ldots, n$. The equation can be deducted from the time interval which is marked by the curly brace in Fig. 8.2, i.e. the situation after job j's operation on machine i has been finished. (It should be mentioned that in many cases several of the X and Y variables sketched out in Fig. 8.2 will be zero for an efficient (with respect to makespan) schedule).

This modification of Wagner's model includes n^2 binary variables $Z_{j,[j']}$ (being n of them non-zero in every solution) and $2nm-(n-1)$ continuous variables $X_{i,[j']}$ and $Y_{i,[j']}$. The number of constraints is $2n$ (8.25a, 8.25b) plus $n(m-1)$ (8.26a, 8.26b), i.e. in total $n(m+1)$ constraints.

As already mentioned, we will not discuss the B&B optimisation approaches with respect to these 'pure' MILP formulations of the problem under consideration. However, we give a table that compares the different dimensions of these three models since these model dimensions might be one crucial aspect for the solvability of the models.

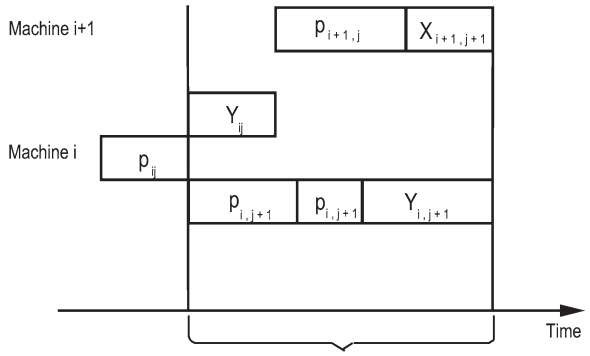

Fig. 8.2 Time calculation for the adapted Wagner MILP model for permutation flow shops relative to operation of job j on machine i

As can be seen from Table 8.5, the number of binary variables is the smallest in the Manne model. However, since the binary variables' influence in the Wilson model and the Wagner model is of assignment model type, additional features of this problem type might be advantageous for the solution of the problem. Therefore, the lower number of binary variables in the Manne model may not pay off in the solution process. The number of continuous variables can be neglected for the evaluation of the expected computational effort. However, the number of constraints might influence the performance of the model solution. With respect to this aspect, the Wagner formulation is best as compared with the other formulations. For more detailed numerical comparison of these models see Tseng et al. (2004). Their numerical results indicate that the Wagner formulation performs best, the Wilson formulation is slightly worse and the Manne formulation performs worst—at least for exactly solvable small problem instances. This demonstrates that the same (formal) problem might be solved by different (formal) models and the performance of the model in terms of computational effort to derive the same optimal solution might differ significantly from model to model.

8.3.2 Problem-Oriented B&B Approaches

Branch and bound approaches are one of the most wide-spread types of approaches to solve combinatorial optimisation problems exactly or, if prematurely stopped, in an approximate manner. We will not discuss their mathematical, formal aspects in detail but refer the interested reader to the relevant literature cited as further readings in Sect. 8.4.

The problem-oriented approaches to branch and bound procedures refer to the underlying manufacturing scheduling problem itself. For these approaches we present some aspects of how the design of the four components that could be

Table 8.5 Model sizes with respect to number of machines m and number of jobs n (see also Tseng et al. 2004)

	Manne	Wilson	Wagner
Continuous variables	$nm + 1$	nm	$2nm - (n - 1)$
Binary variables	$0.5n(n - 1)$	n^2	n^2
Constraints	$n(mn + 1)$	$2mn + n - m + 1$	$n(m + 1)$

Table 8.6 Processing times (p_{ij}) for a four jobs and three machines flow shop scheduling problem

Machine (i)	Job (j)			
	1	2	3	4
1	6	9	2	6
2	8	4	2	7
3	7	4	3	8

customised, i.e. solution/node definition, branching definition, bound computation and branching strategy.

To exemplify some basic ideas, we will refer to the simple permutation flow shop scheduling problem with makespan objective and a given instance with four jobs and three machines. The processing times can be seen from Table 8.6.

A complete decision tree for this problem is given by Fig. 8.3. (We will refer to the numbers in the nodes of this figure further down.)

(1) *Node definition*: The nodes coincide with a subset of the overall solution set. In Fig. 8.3 these subsets are defined by a sequence S^+ of jobs that have already been scheduled, i.e. the subset is defined by all sequences which have the same initial subsequence S^+. Example, the node 23** indicates all sequences/permutations of the $F|prmu|C_{\max}$ problem instance under consideration which start with subsequence 23 while jobs one and four have not yet been scheduled. These jobs constitute the set of non-scheduled jobs in node 23** called S^-. (It should be mentioned that S^+ is a

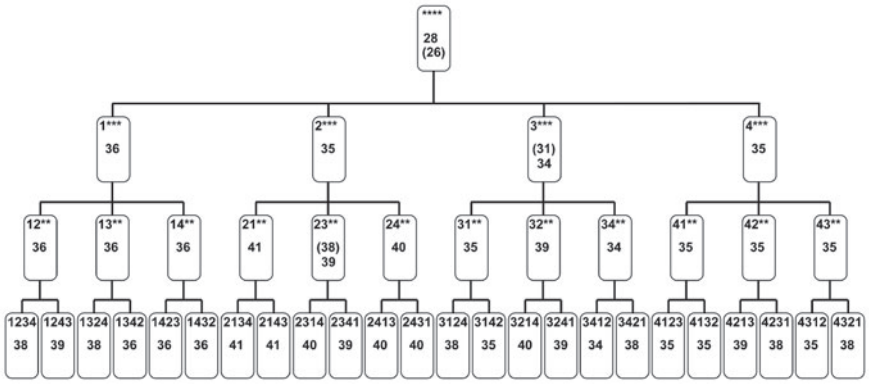

Fig. 8.3 Complete decision tree of the $F|prmu|C_{\max}$ example instance

(well-ordered) vector of jobs while S^- is an unsorted set of remaining jobs which have not yet been scheduled. The final nodes of the decision tree in Fig. 8.3 represent the complete sequences. Looking only at the bottom level and evaluating all solutions is the well-known complete enumeration).

However, even in this very basic example, definition of nodes can be done different than it is done in the standard way depicted in Fig. 8.3. For example, the sequences must not be constructed forward from the first job to the last one but can also be generated backward from the last job to the first one. Especially in those types of problems where it does not make a difference whether a sequence is constructed from the first operation of the first job on its first machine to the last operation of the last job on its last machine or vice versa (which is obviously the case for the simple flow shop problem in our example), exact as well as approximate/heuristic procedures can be applied 'in both directions'. Since the (approximate) procedures will often give two different sequences by the application of both, the forward and the backward approach, the best of them can be chosen while the effort is only doubled. Exact procedures, such as the branch and bound procedure discussed here, might yield different optimal solutions (if more than one exists) and/or may result in different computational effort for the same solution if a forward or a backward approach is used. This aspect of reversibility is, e.g., discussed by Pinedo (2012). It was applied to buffer-constrained flow shop problems by Leisten (1990). However, although the effort of this combined forward/backward approach is (coarsely) only twice as high as the application of one of the single direction approaches and it might result in some improvements of the single direction approaches, this double direction approach is not very popular so far.

Many other approaches can be considered here as well. Potts (1980) describes a simultaneous forward/backward approach in a branch and bound context and Leisten (1990) applies this idea to heuristics for buffer-constrained flow shops. Every intermediate approach could be used as well. However, forward, backward, forward/backward or a combination of these approaches are those being discussed in the literature. From an application point of view, scheduling problems with a distinct bottleneck stage could focus on planning this bottleneck first and then apply backward and forward strategies based on the schedule for this bottleneck stage.

(2) *Branching definition*: Within the branching definition, the transition from one (predecessor) node to its successor(s) has to be defined. Having already scheduled a subset of jobs (or operations respectively) which results in a solution subvector S^+, one (or more) of the remaining jobs/operations from the currently unscheduled objects in the set S^- have to be selected to generate the new solution subvector(s) S^+ on the next level of the decision tree. In our permutation flow shop example with forward construction of solutions, a currently unscheduled job $j^* \in S^-$ is selected, appended to the sequence of S^+ and deleted from S^-. This defines a node on the next level. Doing this for all jobs $j^* \in S^-$ in a given node S^+ defines the whole set of successor nodes of the node under consideration. In our example, e.g. the successor nodes of 23** are 231* and 234*. However, since on the third level only one job remains to be scheduled, 231* can be identified with 2314 and 234* with 2341.

(3) *Bound computation*: Bound computation is one of the most crucial items of a branch and bound approach. It substantially determines the quality of the approach, i.e. the ability to cut off subtrees of the overall decision tree before they are refined to the final leaves of complete solutions. In manufacturing scheduling (with a minimisation objective function), bounds are calculated to give a lower limit on the best solution/schedule which can be derived from this node (and all its successors). If this lower bound indicates that the best solution from this node and its successors is worse (or at least not better) than the objective function of another, already known solution, this node can be 'cut off' and must not be considered further. And obviously: The earlier a subtree can be cut off (i.e. the better the bounds are), the less is the computational effort.

(If only one optimal solution is to be determined, then 'not better' suffices as criterion. If all optimal solutions are to be determined, then 'worse' is required to cut off a subtree. In manufacturing scheduling problems, especially those with processing times with moderate variance or variability, this distinction might result in significant different computational effort since the number of optimal solutions might increase significantly with a decrease in variance/variability of processing times).

In manufacturing scheduling problems in every node, lower bounds in principle based on the objective function contribution of the subsolution in S^+ try to estimate the contribution to the objective function of the remaining jobs/operations in S^- to the final solution without computing all complete solutions (which can be generated departing from S^+).

Sophisticated bounds for many branch and bound approaches are available in the literature. To structure or even only to describe this variety of bounds would go far beyond the scope of this book. Therefore, we will only present two common bounds to the $F|prmu|C_{\max}$ problem, i.e. the above example, to demonstrate a basic approach to bound computation in manufacturing scheduling. These bounds focus on the remaining operations on the different machines (machine-based bound) or on the jobs still to be scheduled (job-based bound). See Ignall and Schrage (1965) for an early description of these bounds.

1. Machine based bound.
   If $C(S^+, i)$ is defined as the completion time of the subsequence S^+ on machine i, i.e. the time when the last operation scheduled so far on machine i is completed, then a bound for the makespan of the overall schedule can be determined as follows:

$$\mathrm{MBB}(S^+) = \max_{i=1,\ldots,m} \left\{ C(S^+, i) + \sum_{j \in S^-} p_{ij} + \min_{j \in S^-} \sum_{i'=i+1}^{m} p_{i'j} \right\}$$

   The second summand of the formula guarantees that every job not scheduled so far is processed on machine i (without idle time on i) while the third summand indicates that, after finishing all jobs on machine i, at least the job with the minimum remaining processing times has to be processed on all subsequent machines

$i + 1$ through m (while ignoring potential waiting time). Since this is a lower bound for the makespan of the overall schedule for all machines, the maximum of this indicator over all machines gives a valid lower bound for all schedules having the same initial subsequence S^+.

2. Job based bound. A job-based bound can be determined as follows:

$$\mathrm{JBB}(S^+) = \min_{j \in S^-} \left\{ \max_{i=1,\ldots,m} \left\{ C(S^+, i) + \sum_{i'=i}^{m} p_{i'j} + \sum_{j' \in S^-;\, j' \neq j} \min\left\{ p_{ij'}, p_{mj'} \right\} \right\} \right\}$$

The second summand of the formula indicates that job $j \in S^-$ is to be processed on machines i through m and the third summand says that every other job $j' \in S^-$ not scheduled so far is either to be processed before job j, i.e. its processing time on machine i takes place before job j starts on machine i, or it is processed after job j, i.e. its processing time on machine m takes place after job j is finished on machine m. The minimum of both processing times (of every job $j' \neq j$) on machines i and m therefore can be included in a lower bound for the overall makespan. This can be summarised over all remaining jobs $j' \neq j$ since this holds for all other jobs j'. And since this conclusion holds also for every single machine i, the maximum over all machines can be taken. However, this conclusion holds only, if job j is scheduled immediately after S^+. Since all jobs $j \in S^-$ might immediately follow S^+, the minimum over all remaining jobs gives a valid lower bound for the makespan of all schedules starting with S^+.

The first summand in the job-based bound derived above can be improved as follows: Since the calculation of the summands refers to the assumption that job j is scheduled immediately after S^+, job's j 'true' starting time on machine i can be determined by temporarily scheduling this job after S^+, i.e. by determining $C\big((S^+, j), i\big)$ which includes possible idle time of machine i if job j has not been finished on machine $i-1$ yet. The earliest starting time of job j on machine i is then $min\Big(C(S^+, i), C\big((S^+, j), i-1\big)\Big)$ where $C\big((S^+, j), i-1\big)$ can be determined recursively from machine 1 through m by placing job j immediately after S^+. (It should be mentioned that the above considerations require the permutation property of the solution of the problem under consideration).

In Fig. 8.3, for every node the bound values are shown within the node. If there is only one bound value, both values are the same. If two values are shown (only in ****, 3***, 23**), the first value gives the machine-based bound, the second the job-based bound while the value in brackets is the worse of both.

Summarising with respect to bound calculation approaches it should be mentioned that the basic ideas of bound computation schemes refer to two aspects.

1. What is the minimum additional contribution to the objective function value of the subsequence identified with the node under consideration when considering all remaining jobs/operations to be scheduled?
2. Use a calculation scheme that considers basically the manufacturing stages (which might be advantageous especially if a clear bottleneck stage can be identified) or use a job wise calculation scheme (especially if clear 'bottleneck jobs'/'bottleneck operations' occur)—or combine both approaches.

(4) *Branching strategy*: Finally, the branching procedure has to be determined. As in all branch and bound approaches, also in those exclusively determined by the formal IP problem, depth-first or breadth-first strategies or combinations of both are known and used. It is well known that depth-first search procedures generate a feasible solution rather quickly while the number of nodes to be explored until one optimal solution has been determined might be rather large while in breadth-first search approaches the opposite holds. We will not discuss these approaches further here.

Referring to Fig. 8.3, the breadth-first search approach requires the analysis of 9 nodes while the depth-first search approach analyses 20 nodes before the optimal solution 3412 with makespan of 34 is determined (and its optimality is confirmed). We mention that the same approaches applied to the reverse problem as described above requires the analysis of 14 nodes for the first-depth search approach while 12 are required for the breadth-first search approach. (The interested reader might calculate the bound values to confirm these numbers).

Here, we referred more or less exclusively to the simple permutation flow shop setting. Every other (combinatorial) manufacturing scheduling problem may be formulated and solved by branch and bound techniques as well. The more general flow shop problem (without permutation constraint), job shop problems and/or other objective functions as well as many more settings have been intensively discussed. Nevertheless, the four components of B&B approaches, i.e. node definition, branching definition, bound computation and branching strategy, in their problem-specific setting have to be executed explicitly.

The advantage of branch and bound approaches as compared with simple, complete enumeration approaches arises if a larger subset (possibly on an early solution set separation level) can be identified to not contain an optimal solution. In this case, the further consideration of this subset including the complete calculation of every single solution in this subset can be omitted. However, to cover the worst case, for every problem setting and every definition of a branch and bound approach, usually a problem instance can be constructed so that the respective branch and bound approach has to finally calculate every single discrete solution. Therefore, the intended advantage of reducing the calculation effort cannot be guaranteed for any branch and bound approach in general. However, computational experience shows that usually a significant reduction of this computational effort can be expected.

There are numerous alternative and modified branch and bound approaches to combinatorial optimisation problems in general and to scheduling problems as (formal) applications of these approaches. Branch and cut (Crowder and Padberg

1980) or branch and price methods are widely discussed in more mathematically oriented references on combinatorial optimisation. We will not discuss these approaches here since they more or less exclusively refer to the above mentioned IP-based B&B approaches and emphasize on formal aspects of these problems. However, since these approaches have proven to be numerically rather efficient, they might become relevant also for the problem-oriented B&B approaches: If these newer formal approaches would significantly outperform numerically the problem-oriented approaches, one might consider to emphasize on the mapping of manufacturing scheduling problems into formal IP problems (and solve them with the advanced formal IP techniques) instead of developing and refining more problem-specific approaches to manufacturing scheduling.

Other approaches try to identify good feasible solutions before a B&B procedure is started, e.g. by sophisticated constructive and/or improvement heuristics. This usually reduces the error bound in a B&B procedure in the very beginning (as compared with the simple upper bound for the objective function value which might be derived from a first-depth search approach), giving reason for stopping the B&B procedure rather quickly if the remaining maximum error is below a pre-specified limit.

Another, heuristic approach based on B&B is beam search where, within a node under consideration only a limited number of subsequent nodes, i.e. the most promising ones, are taken into account for the B&B procedure while the other nodes are excluded from further consideration. Fixing the number of nodes to be explored further and determination of the respective selection criterion are decisive for the quality of these approaches. Beam search approaches are usually combined with breadth-first B&B approaches to limit the number of open nodes to be stored simultaneously.

8.3.3 Dynamic Programming

Roughly spoken, dynamic programming in mathematical optimisation is a classical technique applicable if

- the overall decision can be separated into partial decisions which (including their objective function contribution) can be assigned to points/stages of a continuous or a discrete scale
- while the optimal (partial) decision sequence *up to* a stage depends completely on the unvarying optimal (partial) decision sequence *up to* the stage in front of this stage under consideration *and* the decision *in* this stage under consideration.

Under these conditions, dynamic programming approaches can be applied which use the recursive optimisation approach induced by the well-known Bellman principle/the Bellman recursion equation. For a discrete scale, this setting allows the application of the discrete backward or forward recursive optimisation scheme of dynamic programming stage by stage. It means that an optimal solution up to a stage consists of the optimal combination of a decision *on* this stage in combination with the best decision *up to* the previous stage. In fact, this limits the effort, e.g. as compared with complete enumeration, by avoiding every solution on every

stage to be computed from the beginning stage (may it be the first stage if forward calculation is used or the last stage if backward calculation is executed) onwards. The (optimal sub-) decisions (or subdecision sequences) up to the previous stage remain unchanged and only the decision of the current stage has to be linked with exactly these previous (sub-) decisions to get the best decisions up to the stage under consideration.

While in many dynamic programming applications the scale is representing time and the decisions might be taken at certain points in time (discrete dynamic programming approach) or continuously (continuous dynamic programming approach), in manufacturing scheduling mostly discrete approaches are used and the discrete stages often represent the number of jobs/operations scheduled so far. On every stage, the different settings which might occur on this stage (i.e. the states on the stage) have to be identified. For a simple 1-machine scheduling problem with n jobs and forward recursion scheme, a stage might be defined as the number of jobs already scheduled so far (e.g. the stage index $j' - 1$ indicates that $j' - 1$ jobs have been scheduled already) while the maximum set of states on this stage consists of the $(j' - 1)!$ permutations of $j' - 1$ jobs. Moving to stage j' then means taking one by one every remaining job out of the $\big(n - (j' - 1)\big)$ so far non-scheduled jobs, constructing the expanded (sub-) sequences of length j' and looking for the best combinations for the specific job added.

We explain the approach by a most simple example problem which tries to find the optimal solution for a one-machine problem with four jobs and flowtime objective as well as no further 'specialities'. (As is well known, this simple problem can be easily solved to optimum by using the shortest processing time rule. However, for the sake of simplicity of the demonstration of the approach, we stick to this trivial example and give hints on extensions later on.)

Suppose four jobs to be scheduled having processing times 6, 2, 4, and 3, respectively. The objective function is to minimise flowtime, i.e. the sum of completion times of every job, $\sum_j C_j$. The stages are defined by the number of jobs j' being scheduled up to this stage while the possible states on the stage are characterised by all subsequences of length j' having a certain job k on the last position j'. We use a forward recursive approach. Then, the state transition from a sequence $S_{j'-1}$ on stage $j' - 1$ to a sequence including the so far non-scheduled job k in position j' of $S_{j'}$ is $S_{j'-1} \rightarrow S_{j'} = \big(S_{j'-1}, k\big)$. The Bellman recursion equation per state on stage j' is

$$C_{\sum}\left(S_{j'} \text{ s.t. job } k \text{ on position } j' \text{ in } S_{j'}\right)$$

$$= \begin{cases} p_k & \text{for } j = 1 \\ \min_{S_{j'-1},\, k \notin S_{j'-1}} \left(C_{\sum}(S_{j'-1}) + \left(\sum_{j' \epsilon s_{j'-1}} p_{j'} + p_k\right)\right) & \text{for } j = 2, \ldots, n \end{cases}$$

Figure 8.4 shows the computational steps. In Fig. 8.4a, the states as the partial job sequences with a certain last job on each stage are mentioned first. Then the possible realisations of these subsequences are mentioned. In italics letters, the flowtime values of every single possible subsequence are given while a '*' gives $C_{\sum}(S_{j'})$,

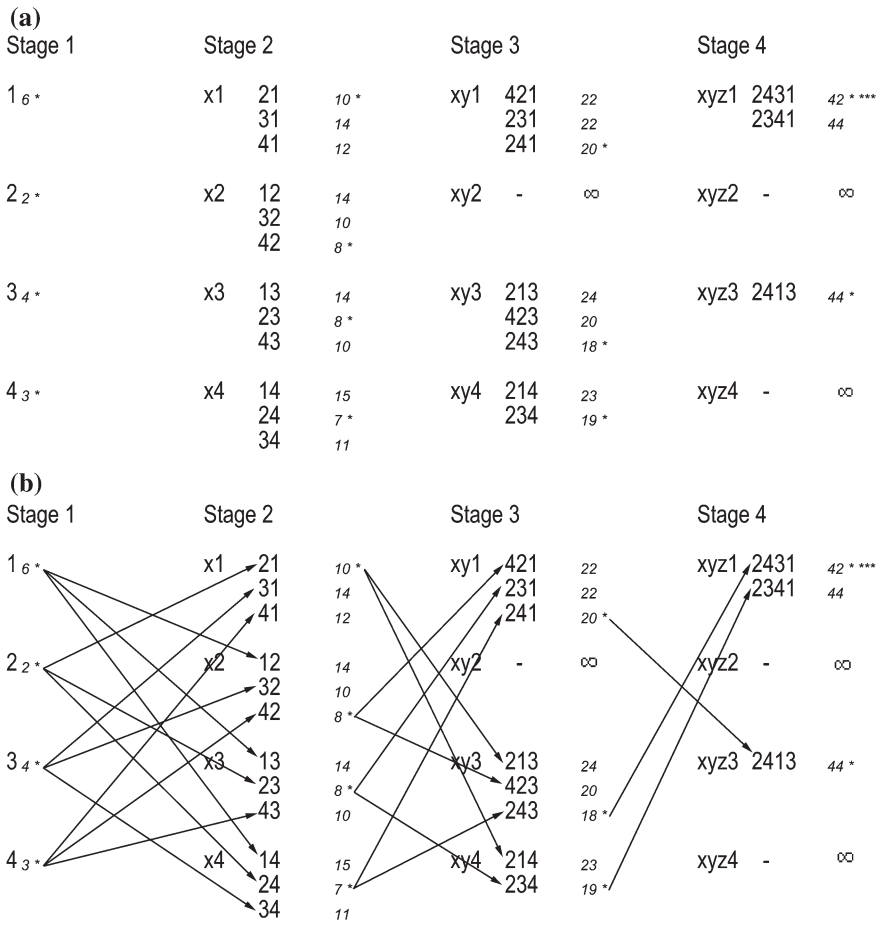

Fig. 8.4 Dynamic programming approach for the example instance of the one-machine flowtime minimisation problem, **a** Results per stage, **b** Additionally indicating the calculation paths

i.e. the optimal subsequence with minimum recursion value and an additional '***' indicates the overall optimal solution (on stage 4). Figure 8.4b shows the same values but additionally gives the relevant state transitions considered in every step. Note that an infinite value (e.g. in xy2) indicates that there is no feasible sequence left to place job 2 in stage/position 3 since job 2 is already scheduled in positions 1 or 2 in all (!) optimal subsequences up to stage 2. Analogously, the same holds for xyz2 and xyz4.

As already mentioned, the example problem presented above can be easily solved by just sorting the jobs according to the shortest processing time rule. For this type of problem, a dynamic programming approach therefore is superfluous. However, if the objective function is more complicate, the solution is not as easy and a dynamic programming approach might become adequate. Early contributions concerning this

Stage 1		Stage 2			Stage 3			Stage 4		
1	*21 **	x1	21	*47*	xy1	321	*58 **	xyz1	3421	*95 **
			31	*30 **		231	*69*		3241	*97*
			41	*49*		341	*60*			
2	*17 **	x2	12	*46*	xy2	312	*57 **	xyz2	-	∞
			32	*26 **		342	*57 **			
			42	*46*						
3	*7 **	x3	13	*45*	xy3	-	∞	xyz3	-	∞
			23	*37 **						
			43	*45*						
4	*21 **	x4	14	*50*	xy4	314	*61*	xyz4	3214	*98*
			24	*37*		324	*58 **		3124	*95 **
			34	*30 **		234	*69*			

Fig. 8.5 Dynamic programming calculation for the three machine, four jobs problem instance from Sect. 8.3.2 for the permutation flow shop problem with flowtime objective

matter are by Held and Karp (1962); Rinnooy Kan et al. (1975) or Schrage and Baker (1978). Instead of the pure linearity of the influence of jobs' completion times with respect to the flowtime objective, initially constant and later linear influence on the objective function as, e.g. in tardiness problems result in problems which cannot be solved by polynomial approaches. Therefore, the above-mentioned papers and the dynamic programming approaches proposed therein allow rather general objective functions, being monotonicity (not necessarily strict monotonicity) in time of jobs objective function contribution the more or less only relevant requirement for the application of these approaches. Also other specific constraints, e.g. precedence relations between the operations, can be advantageously handled by dynamic programming approaches.

We will not deepen the discussion on dynamic programming in manufacturing scheduling but give only some additional remarks on the application of these approaches in manufacturing scheduling.

As in all dynamic programming approaches, the adequate definition of stages and their states as well as the stage separability as mentioned above are the crucial aspects to setup a dynamic programming procedure for a scheduling problem under consideration.

State separability may be guaranteed by an adequate separation of the set of jobs or operations into operations/jobs scheduled so far and not scheduled so far.

The separability with respect to objective functions is often fulfilled for additive (or averaging) objective functions as (mean) flowtime or (mean) tardiness. In contrast, separability for maximin objective functions cannot be expected in general. Therefore, e.g. makespan or maximum tardiness problems are often not suited to be solved by means of dynamic programming approaches.

However, to demonstrate problems which might occur with respect to separability, we refer to the standard permutation flow shop with flowtime objective. We

suppose the three machine, four jobs problem instance given as a B&B example in Sect. 8.3.2. But now we suppose flowtime to be minimised. Performing the dynamic programming approach as in the above example yields the results as in Fig. 8.5.

As can be seen, the dynamic programming approach yields two 'optimal' permutations, i.e. 3421 and 3124, with a flowtime of 95. However, the true optimal solution for this problem instance is 3412 with a flowtime of 94. This solution is not generated by the dynamic programming approach because the initial partial sequence 341 of the overall optimal solution is excluded from further consideration on stage 3 since 321 gives a better solution for the state xy1. Therefore, even in this simple example, the separability requirement of dynamic programming is not fulfilled and dynamic programming cannot be applied to this kind of problem—at least not with this definition of stages.

Summarising, dynamic programming (in manufacturing scheduling) can be classified as a limited enumeration approach such as B&B approaches. In contrast to B&B approaches, dynamic programming requires certain additional and specific structural features of the manufacturing scheduling problem considered, i.e. separability. If these requirements are fulfilled, dynamic programming will be usually competitive to respective B&B methods. However, on one hand dynamic programming is also a non-polynomial enumerative approach which is, in general, only applicable for small sizes of problem instances if the optimal solution is to be determined. On the other hand, as in B&B, dynamic programming can be stopped prematurely and/or combined with other heuristic approaches.

8.4 Conclusions and Further Readings

As discussed in the previous chapter, exactness in solving manufacturing scheduling problems is closely related to the formal definition and simplification of these scheduling problems and can only be interpreted relative to these formal problems. With respect to the underlying real-world problems, because of their complexity, 'exact' (i.e. optimal) solutions can never be expected.

Referring to this notion of exact approaches to (formal) manufacturing scheduling problems, only few and rather simple problems can be solved exactly by constructive methods, i.e. by polynomial effort. However, since manufacturing scheduling problems represent a subset of combinatorial optimisation problems the more complex formal problems, in principle, can be solved by enumerative approaches such as complete enumeration, branch and bound or dynamic programming. In the worst case, these approaches require non-polynomial, exponential effort relative to the problem size. Therefore, these procedures, on one hand have to be designed carefully and sophisticated. On the other hand, since the 'pure' exact procedures will usually have to be stopped prematurely, at least for larger problem instances, these approaches should and can successfully be combined with advanced heuristic approaches. It should be mentioned that there are numerous practical cases where the problem under

consideration is of non-polynomial type. However, since the problem size of these cases often remains moderate, these instances can be solved to optimality anyway.

Regarding further readings, early scheduling books such as Conway et al. (1967); French (1982) cover exact procedures for many scheduling problems, while a key book on integer programming and branch and bound procedures is the one by Nemhauser and Wolsey (1988). The paper of Johnson is one of the first that appeared in the scheduling literature and some authors, like the already mentioned Conway et al. (1967), attribute the extensive use of the makespan criterion (probably this is overrated) to this initial seminal paper. This has been criticised by other authors, like Gupta and Dudek (1971) who stated that makespan in not realistic. A detailed description of Lawler's and Moore's algorithms (among many others) can be found in several scheduling textbook, such as for instance T'Kindt and Billaut (2006).

References

Conway, R. W., Maxwell, W. L., and Miller, L. W. (1967). *Theory of Scheduling*. Dover Publications, New York. Unabridged publication from the 1967 original edition published by Addison-Wesley.

Crowder, H. and Padberg, M. W. (1980). Solving large-scale symmetric travelling salesman problems to optimality. *Management Science*, 26(5):495–509.

Fanjul-Peyró, L. and Ruiz, R. (2010). Iterated greedy local search methods for unrelated parallel machine scheduling. *European Journal of Operational Research*, 207(1):55–69.

French, S. (1982). *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*. Ellis Horwood Limited, Chichester.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York.

Gupta, J. N. D. and Dudek, R. A. (1971). Optimality Criteria for Flowshop Schedules. *IIE Transactions*, 3(3):199–205.

Held, M. and Karp, R. M. (1962). A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210.

Ignall, E. and Schrage, L. E. (1965). Application of branch- and bound technique to some flow shop problems. *Operations research*, 13(3):400–412.

Johnson, S. M. (1954). Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1:61–68.

Lawler, E. L. (1973). Optimal Sequencing of a Single Machine Subject to Precedence Constraints. *Management Science*, 19(5):544–546.

Leisten, R. (1990). Flowshop sequencing problems with limited buffer storage. *International Journal of Production Research*, 28(11):2085.

Lenstra, J. K., Rinnooy Kan, A. H. G., and Brucker, P. (1977). Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362.

Lenstra, J. K., Shmoys, D. B., and Tardos, E. (1990). Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(3):259–271.

Liao, C.-J. and You, C.-T. (1992). An improved formulation for the job-shop scheduling problem. *Journal of the Operational Research Society*, 43(11):1047–1054.

Manne, A. S. (1960). On the job-shop scheduling problem. *Operations Research*, 8(2):219–223.

Moore J. (1968). An *n* job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Science*, 15 (1):102–109.

Nemhauser, G. L. and Wolsey, L. A. (1988). *Integer and combinatorial optimization*. Wiley, New York.

Pinedo, M. L. (2012). *Scheduling: Theory, Algorithms, and Systems*. Springer, New York, fourth edition.

Potts, C. N. (1980). *An adaptive branching rule for the permutation flow-shop problem*. Elsevier, Amsterdam.

Rinnooy Kan, A., Lageweg, B., and Lenstra, J. (1975). Minimizing total costs in one-machine scheduling. *Operations Research*, 23(5):908–927.

Schrage, L. E. and Baker, K. R. (1978). Dynamic programming solution of sequencing problems with precedence constraints. *Operations Research*, 26(3):444–449.

T'Kindt, V. and Billaut, J.-C. (2006). *Multicriteria Scheduling: Theory, Models and Algorithms*. Springer, New York, second edition.

Tseng, F. T., Stafford, Jr, E. F., and Gupta, J. N. D. (2004). An empirical analysis of integer programming formulations for the permutation flowshopn. *OMEGA, The International Journal of Management Science*, 32(4):285–293.

Vallada, E. and Ruiz, R. (2010). Genetic algorithms for the unrelated parallel machine scheduling problem with sequence dependent setup times. *European Journal of Operational Research*. In review.

Wagner, H. M. (1959). An integer linear-programming model for machine scheduling. *Naval Research Logistics Quarterly*, 6(2):131–140.

Wilson, J. M. (1989). Alternative formulations of a flowshopn scheduling problem. *Journal of the Operational Research Society*, 40(4):395–399.

Chapter 9
Approximate Algorithms

9.1 Introduction

In this chapter we deal with approximate algorithms. In principle, one may think that approximate algorithms are not a good option if exact approaches exist and indeed we already discussed that their widespread use is basically justified by the computational complexity inherent for most manufacturing scheduling models. As mentioned in Sect. 7.5.2, approximate algorithms are usually divided into heuristics and metaheuristics, being the main difference among them that the first are specifically tailored for a particular model, while the second constitute more generic procedures. This difference—although not entirely unambiguous—is very important when it comes to describing both approaches, as we intend to do in this chapter: While it is clear that it is possible to discuss the templates of the different metaheuristics that can be employed for manufacturing scheduling models, such thing is not possible regarding heuristics, as they are heavily problem-dependent. Instead we will first illustrate how heuristic algorithms work by presenting perhaps the two most popular heuristics for manufacturing scheduling, i.e. the NEH heuristic (Nawaz et al. 1983) for flowshop scheduling with makespan objective, and the Shifting Bottleneck Heuristic (Adams et al. 1988) for job shop scheduling. Then, in a sort of 'reverse engineering' attempt, we will discuss the main ideas and concepts behind (these and other) heuristics. We believe that this attempt to structure the behaviour of the heuristic may be useful for the reader when faced to the task of designing new heuristics for existing or new scheduling models.

More specifically, in this chapter we

- present two well-known heuristics in order to grasp the general mechanics of heuristic construction (Sect. 9.2),
- give some description of (some) underlying ideas of the approximate methods under consideration (Sect. 9.3),
- present the main concepts behind metaheuristics, and discuss the most widely employed in the manufacturing scheduling context (Sect. 9.4), and
- briefly introduce further approximation techniques (Sect. 9.5).

J. M. Framinan et al., *Manufacturing Scheduling Systems*,
DOI: 10.1007/978-1-4471-6272-8_9, © Springer-Verlag London 2014

9.2 Two Sample Heuristics

In this section, two heuristics are presented which have turned out to be rather efficient for the scheduling problems they are created for on one hand. On the other hand, the basic ideas of these heuristics have been the starting point for many adaptations to other heuristics and/or other problem classes.

First, in Sect. 9.2.1, we consider the heuristic by Nawaz et al. (1983) which has been basically the result of a master thesis in the early 1980s and since then is something like a benchmark for the respective type of problems. It deals with the standard permutation flow shop problem with makespan objective. Afterwards, in Sect. 9.2.2, we describe a basic version of the Shifting Bottleneck Heuristic (SBH) which has been published first by Adams et al. (1988). This heuristic is designed for the standard job shop problem with makespan objective.

9.2.1 The NEH Heuristic

By far and large, one the best known and highly cited heuristic for the flow shop problem and makespan criterion is the NEH heuristic. This method has been widely recognised as being very high performing, flexible, and efficient. Furthermore, NEH is actively used as a seed sequence for metaheuristic techniques (these will be discussed in the next section). Some examples where NEH is used as a seed sequence are Tabu Search, Simulated Annealing, Genetic Algorithms, Iterated Local Search, Ant Colony Optimisation and many other metaheuristic algorithms.

The NEH heuristic is dedicated to solve the basic permutation flow shop scheduling problem with minimizing makespan as objective function, i.e. $Fm|prmu|C_{max}$. It reduces the exponential effort of complete enumeration of all permutations of job sequences to a polynomial effort by sorting the jobs according to some importance index (see below) and then determines job by job its best position while keeping the relative sequence of the already scheduled jobs unchanged.

Due to the importance of the NEH, we describe it in detail. The NEH procedure is based on the idea that jobs with high sum of processing times on all the machines should be scheduled as early as possible. NEH is explained in Algorithm 1.

Computing P_j has a complexity of $O(nm)$, and sorting n jobs of $O(n \log n)$. Most CPU time is needed in the main loop of the heuristic, where we have a loop of $n-2$ steps and, at each step k, we carry out k insertions of job k in a partial sequence that contains $k-1$ jobs and for each insertion we have to calculate the $C_{\max}$ value of k jobs (including the inserted one). In total, there are $\frac{n(n+1)}{2} - 3$ insertions. Among these, only in the last iteration we have to evaluate n complete sequences. As a result, the computational complexity of the whole method goes to $O(n^3 m)$. This method can be slow for large instances. However, when inserting the k-th job in position, let us say, j, all $C_{i,h}$, $h = j-1, j-2, \ldots, 1$ were already calculated in the previous insertion and we do not need to recalculate these values. A similar approach was

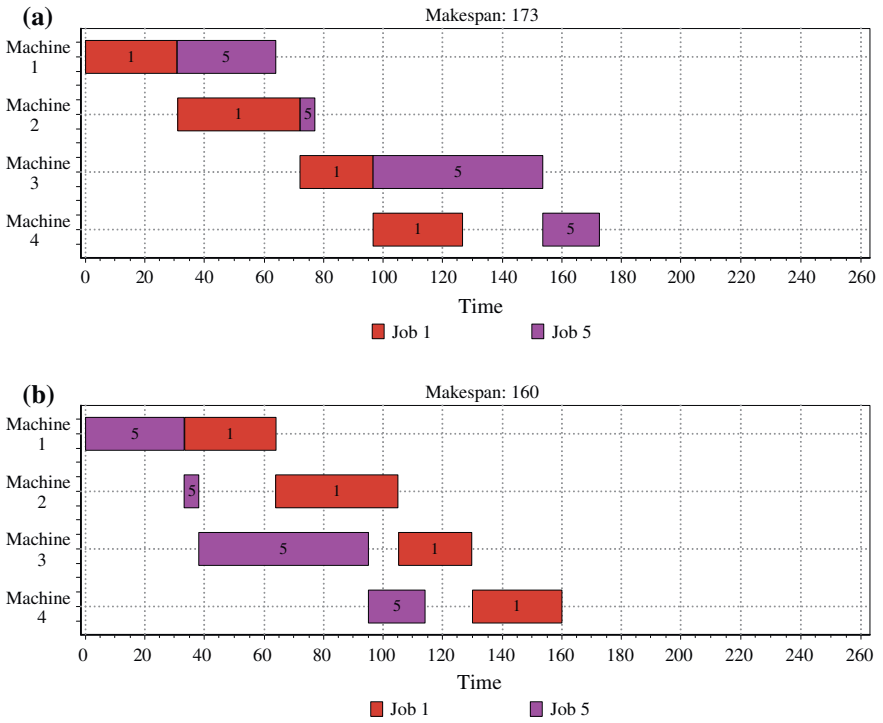

Fig. 9.1 First iteration of the NEH heuristic for the example of Table 9.1. **a** Sequence (1, 5) with a $C_{\max}$ value of 173. **b** Sequence (5, 1) with a $C_{\max}$ value of 160

Algorithm 1: NEH's Algorithm (Nawaz et al. 1983)

Input: Instance data
Output: Sequence Π, Makespan C_{max}
begin
  Let $P_j = \sum_{i=1}^{m} p_{ij}$;
  Let $\Pi = \varnothing$, $J = \{1, \ldots, n\}$ verifying $P_1 \geq P_2 \geq \ldots P_n$;
  Let Π the best permutation between (1, 2) and (2, 1);
  for $k = 3$ *to* n **do**
    Insert job k in the position of Π yielding the lowest makespan value;
  return Π, C_{max}
end

followed by Taillard (1990) and as a result the complexity of all insertions in a given step can be calculated in $O(nm)$ by a series of matrix calculations. This reduces the total complexity of NEH to $O(n^2m)$. Such an improved method is many times referred to as NEHT or NEH with Taillard's accelerations.

Table 9.1 Processing times (p_{ij}) for a five jobs and four machines flow shop scheduling problem

Machine (i)	Job (j)				
	1	2	3	4	5
1	31	19	23	13	33
2	41	55	42	22	5
3	25	3	27	14	57
4	30	34	6	13	19

Let us work with an example with five jobs and four machines. Actually, this example was already given in Chap. 3, Table 3.3. However, in order to avoid flipping pages back and forth, the table is replicated in Table 9.1.

We calculate in the first step the total processing times P_j, that result in: $P_1 = 31 + 41 + 25 + 30 = 127$, $P_2 = 111$, $P_3 = 98$, $P_4 = 62$ y $P_5 = 114$. Then we arrange the jobs in decreasing order of P_j and the resulting list is $J = (1, 5, 2, 3, 4)$. The first two jobs are extracted. Let us see how the two possible schedules end up (1, 5) and (5, 1). The corresponding Gantt charts appear in the two subfigures of Fig. 9.1. We can see that $C_{\max}((5, 1)) < C_{\max}((1, 5))$. Therefore, the partial (relative) sequence (5, 1) is fixed for the next iteration. There are three positions where to insert the next job 2 and therefore, three possible partial sequences are derived, which are depicted in the subfigures of Fig. 9.2. As we can see, the partial sequence with least $C_{\max}$ value is (2, 5, 1). It is selected for the next iteration. There are four possible partial sequences resulting from inserting job 3 in the four possible positions. The resulting schedules, along with their makespan values are shown in Fig. 9.3. The last from the previous partial sequences is also the one with the least makespan value, (2, 5, 1, 3), it is selected for the next iteration where job 4 remains to be inserted. There are $n = k$ possible positions where to place this last job 4. These sequences are now full sequences since each one contains n jobs. All of these are shown in Fig. 9.4. In this last step, the first sequence is the one with the minimum $C_{\max}$ value and therefore, it is the result of the application of the NEH procedure to the example: (4, 2, 5, 1, 3). This sequence has a $C_{\max}$ value of 213.

The NEH has been applied to other objectives and even to other layouts. While it has been demonstrated that in other problems, it is no as effective and efficient, it is still competitive.

9.2.2 The Shifting Bottleneck Heuristic

In its original version, the SBH is dedicated to solve the basic job shop scheduling problem with minimizing makespan as objective function, i.e. $Jm|.|C_{max}$. This problem is known to be of the NP-hard class.

The basic idea is as follows: Fix one machine as the bottleneck of the manufacturing system under consideration. Then, for this machine and for every job, it is considered what has to be accomplished *before* the job is processed on this machine (i.e. previous operations of this job on other machines). This results in release times for all jobs on the machine under consideration. Additionally, for this machine and

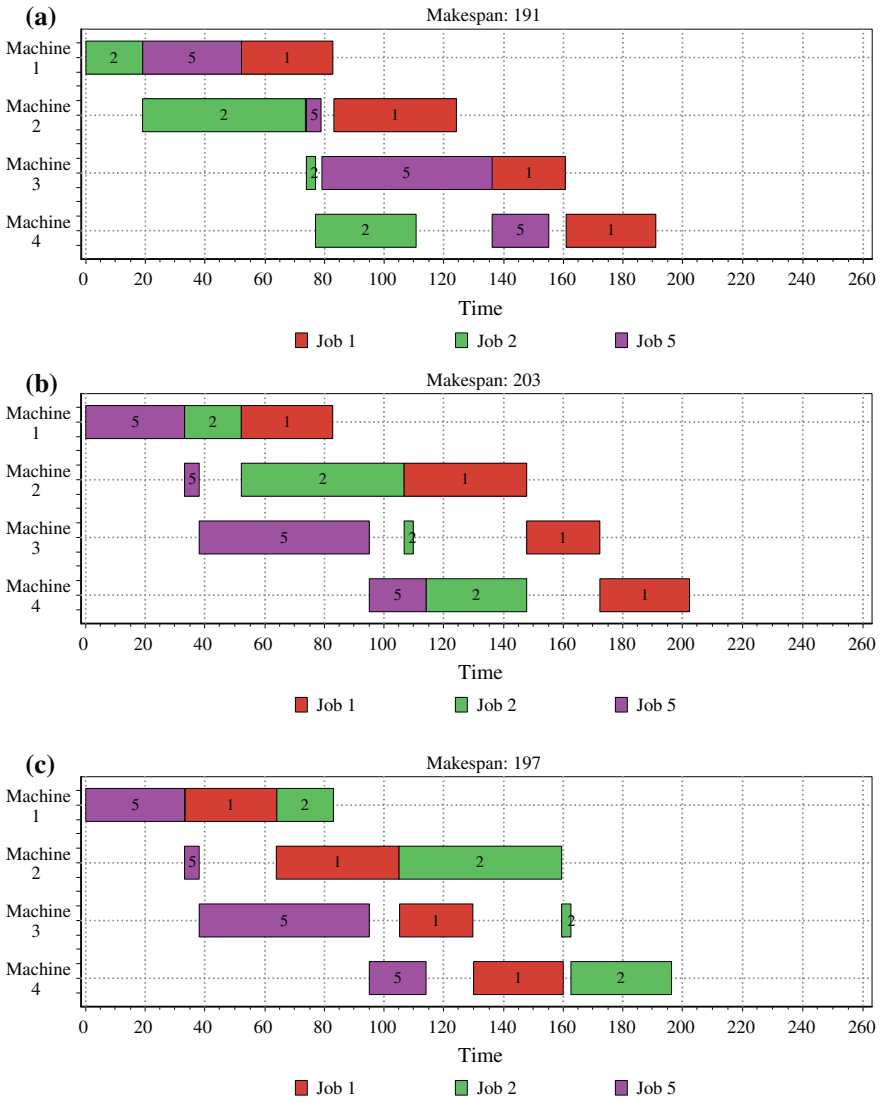

Fig. 9.2 Iteration for $k = 3$ of the NEH heuristic for the example of Table 9.1. **a** Sequence (2, 5, 1) with a $C_{\max}$ value of 191. **b** Sequence (5, 2, 1) with a $C_{\max}$ value of 203. **c** Sequence (5, 1, 2) with a $C_{\max}$ value of 197

for every job, it is considered what has to be accomplished *after* the job is processed on this machine (i.e. subsequent operations of this job on other machines). This results in due dates for all jobs on the machine under consideration. Since the due dates might be exceeded by a current solution, the corresponding $1|r_j, d_j|\max L_j$ problem is (heuristically) solved for the machine under consideration. The resulting

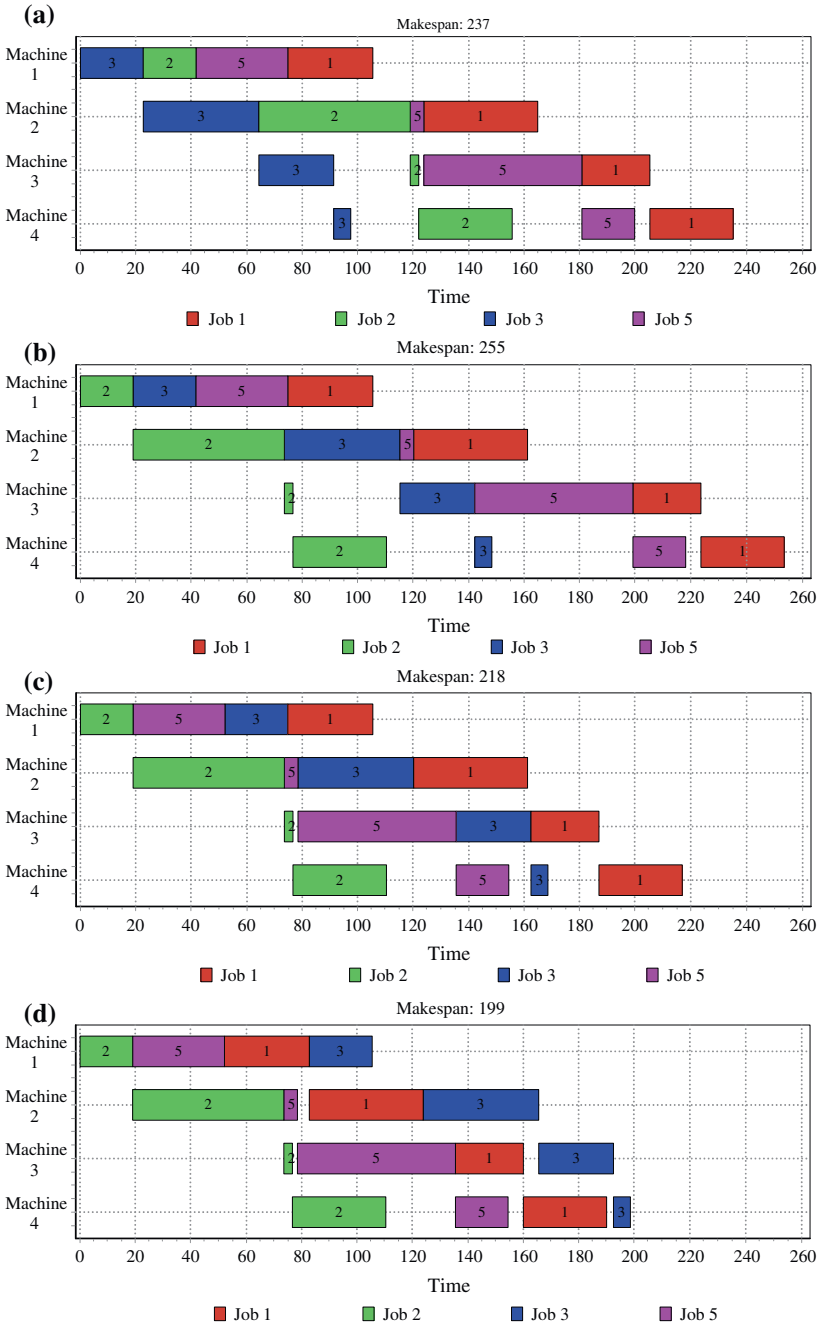

Fig. 9.3 Iteration for $k = 4$ of the NEH heuristic for the example of Table 9.1. **a** Sequence (3, 2, 5, 1) with a $C_{\max}$ value of 237. **b** Sequence (2, 3, 5, 1) with a $C_{\max}$ value of 255. **c** Sequence (2, 5, 3, 1) with a $C_{\max}$ value of 218. **d** Sequence (2, 5, 1, 3) with a $C_{\max}$ value of 199

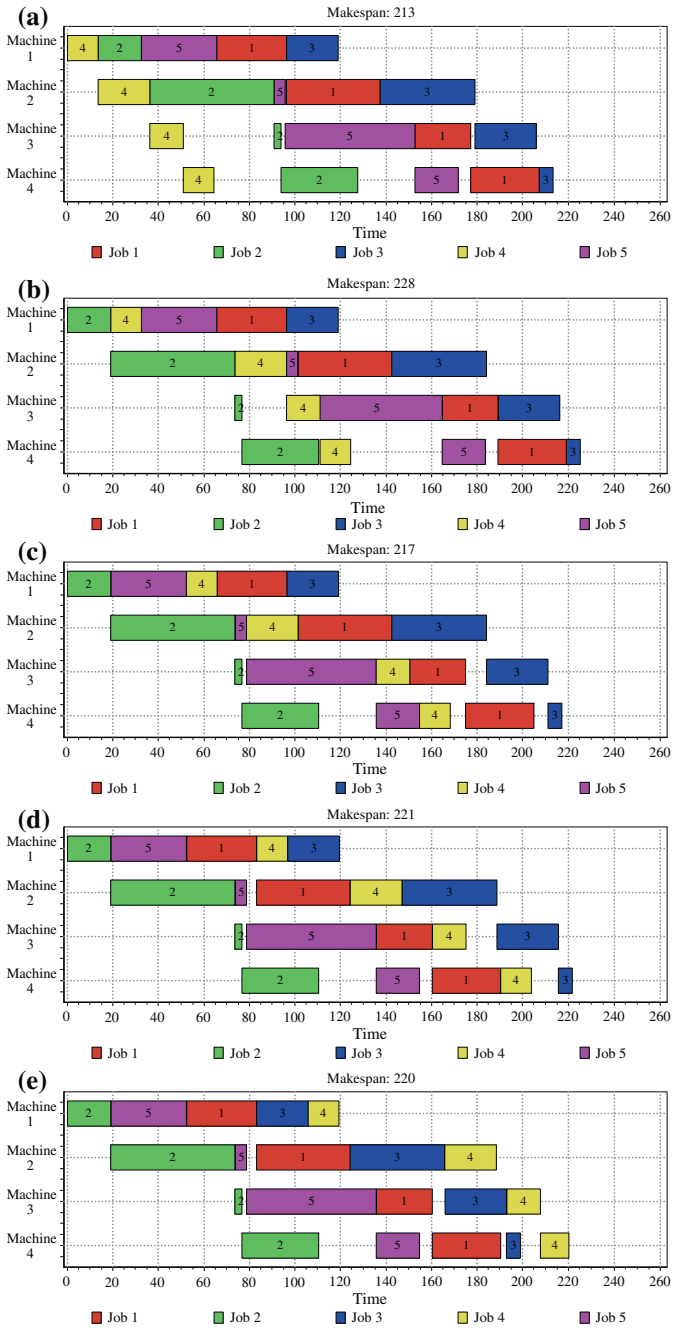

Fig. 9.4 Final iteration ($k = 5 = n$) of the NEH heuristic for the example of Table 9.1. **a** Sequence $(4, 2, 5, 1, 3)$ with a $C_{\max}$ value of 213. **b** Sequence $(2, 4, 5, 1, 3)$ con un $C_{\max}$ value of 228. **c** Sequence $(2, 5, 4, 1, 3)$ con un $C_{\max}$ value of 217. **d** Sequence $(2, 5, 1, 4, 3)$ with a $C_{\max}$ value of 221. **e** Sequence $(2, 5, 1, 3, 4)$ con un $C_{\max}$ value of 220

Table 9.2 Machine sequences (per job)

Job machine	J_1	J_2	J_3	J_4
$M_{[1]}$	1	2	2	3
$M_{[2]}$	2	3	1	1
$M_{[3]}$	3	1	3	–

Table 9.3 Processing times (p_{ij})

Job machine	J_1	J_2	J_3	J_4
M_1	3	3	3	2
M_2	3	2	4	–
M_3	2	3	1	3

job sequence is kept as the SBH's solution job sequence for the machine under consideration. $1|r_j, d_j|\max L_j$ itself is NP-hard. However, very good heuristics exist for this problem.

Since the bottleneck machine of a system is usually not identifiable in advance and might even change from solution to solution, the approach described above is applied successively to all machines while fixing the job sequences on those machines which have already been considered. The sequence of machine consideration is chosen appropriately (see the following example). The procedure of the SBH is described in Algorithm 2.

We exemplify the SBH by means of a simple example instance with three machines and four jobs. Suppose the machine sequences per job and the processing times are given by Tables 9.2 and 9.3.

We demonstrate the approach by means of the disjunctive graph representation of the problem (see also Sect. 1.5.1).

The disjunctive graph for this problem is given by Fig. 9.5.

The nodes in Fig. 9.5 represent the respective operation of J_j on M_i while being the processing times noted in the respective cycle. Initial node A and final node Ω are added. The black conjunctive (uni-directional) arcs represent a job's machine sequence which is given and fixed. The purple disjunctive (bi-directional) arcs represent the possible relative sequences of each two jobs on a specific machine. The dark blue arcs represent the initial conditions (starting of jobs) and the turquoise arcs represent the finishing conditions (finishing of jobs). Minimizing the makespan in the corresponding job shop problem means to transfer every disjunctive arc into a conjunctive one by giving it a uni-directional orientation (and maintaining logic feasibility of the resulting sequence, e.g. no cycles are allowed) so that the longest path from node A to the final node Ω in the network (whose length corresponds to the makespan of the solution derived) is of minimum length.

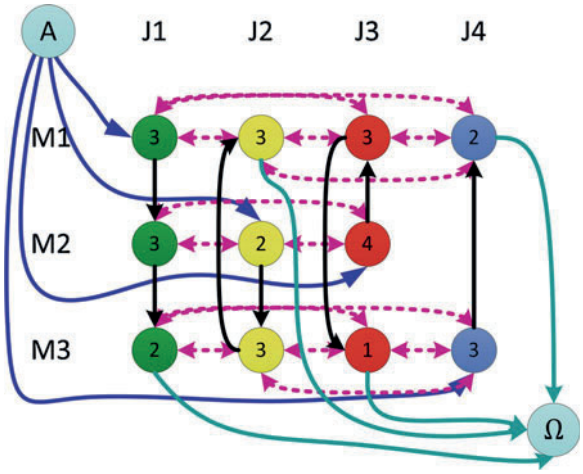

Fig. 9.5 Disjunctive graph of the example for the SBH

Algorithm 2: SBH (Adams et al. 1988)

Input: Instance data
Output: Sequence Π, Makespan C_{max}
begin
Let $UM = \{1, \ldots, m\}$ set of unscheduled machines;
Let $SM = \varnothing$ set of scheduled machines;
Let $\Pi_i = \varnothing, \forall i \in M$;
while *UM is* ***not*** $\varnothing$ **do**
  for *all machines i in UM* **do**
    Compute (regarding the fixing of operations' sequence on the machines in SM and the jobs' machine sequences) r_j and d_j with respect to already scheduled machines *SM* as follows:
      r_j := lower bound for earliest starting time of operation j on machine i;
      d_j := upper bound for the latest finishing time of operation j on machine i;
    Let Π_i the solution of the respective $1|r_j, d_j|\max L_j(i)$ problem
  Let $k \in UM$ be the machine such that max $L_j(k)$ = max $i \in UM$_max $L_j(i)$_(i.e the bottleneck among the remaining unscheduled machines, indicated by the largest maximum lateness);
  Keep Π_k as definitive schedule for machine k (i.e. the job sequence on this machine);
  Insert k in *SM*;
  Remove k from *UM*;
return Π, C_{max}
end

Applying the SBH to this problem instance results in the following.

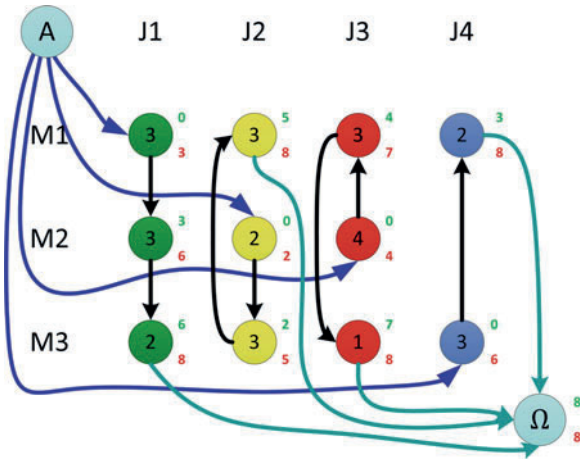

Fig. 9.6 Reduced network in the SBH heuristic (initialization)

Initialization:

Initialize the set of already scheduled machines: $SM = \emptyset$, i.e. the set of unscheduled machines is $UM = (1, 2, 3)$.

Eliminate all disjunctive arcs from the overall problem's disjunctive graph, i.e. the constraints with respect to job sequences on each machine. (In a real-world interpretation, this means that all jobs could be processed in parallel on every machine, assigning every machine (for the time being) an unlimited capacity.)

Determine the longest path in the resulting network (see Fig. 9.6). In Fig. 9.6, the green numbers assigned to every node indicate the earliest starting time of the respective operation while the red numbers represent the latest finishing time of this operation. These time values can be easily computed by the standard forward and backward calculation scheme well-known from time planning in networks.

The C_{max} value of this solution is the maximum of every job's last operation finishing time, i.e.

$C_{max}(SM) = max\ (8, 8, 8, 5) = 8$, as well as the time value at Ω.

The (green) earliest starting time of each operation is now interpreted as the release time r_j of the operation when it is considered on its respective machine. The (red) latest finishing time of each operation accordingly is interpreted as the due date d_j of the operation when it is considered on its respective machine.

Iteration 1: $SM = \emptyset$, $UM = (1, 2, 3)$

Determine min max L_j for all single machine problems incl. release and due dates, i.e. solve $1|r_j, d_j|\max L_j$ for all so far unscheduled machines. As already

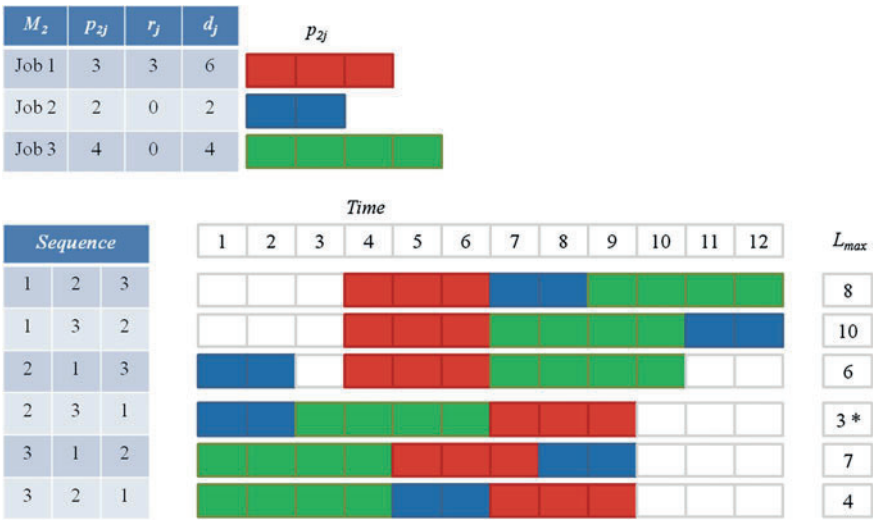

Fig. 9.7 Solving $1|r_j, d_j|\max L_j$ for M_2 in first iteration by complete enumeration

mentioned this problem is NP-hard, but good heuristics exist for it. Here, since there are only $3! = 6$ (on M_2) or $4! = 24$ (on M_1 and M_3) possible solutions for each of these problems, we use complete enumeration to solve the problems exactly:

M_1: min max $L_j = 3$ for job sequence 1432
M_2: min max $L_j = 3$ for job sequence 231
M_3: min max $L_j = 1$ for job sequence 4213

We do not list all possible permutations for all unscheduled machines here. This is left to the reader. Instead, exemplarily we give only the results for M_2 here. Since there are only three operations to be considered on M_2, there are only $3! = 6$ possible sequences to be calculated. The calculation is seen from Fig. 9.7.

We choose the machine with the worst minimum max L_j value and include the disjunctive arcs for the respective solution in Fig. 9.6. The result can be seen from Fig. 9.8. Since there are two machines with the worst max L_j value (namely M_1 and M_2), in the simplest version of the SBH we can choose arbitrarily among those machines. Here we choose M_1 and update $SM = (1)$, $UM = (2, 3)$.

Performing forward and backward calculation of earliest starting times and latest finishing times of every operation yields again the (updated) green and red numbers at every node in Fig. 9.8.

The longest path gives the current 'makespan' (for this relaxed problem) which is the maximum of every job's last operation finishing time in Fig. 9.8, i.e.

$C_{max}(SM) = max\ (11, 11, 11, 5) = 11$, as well as the time value at Ω.

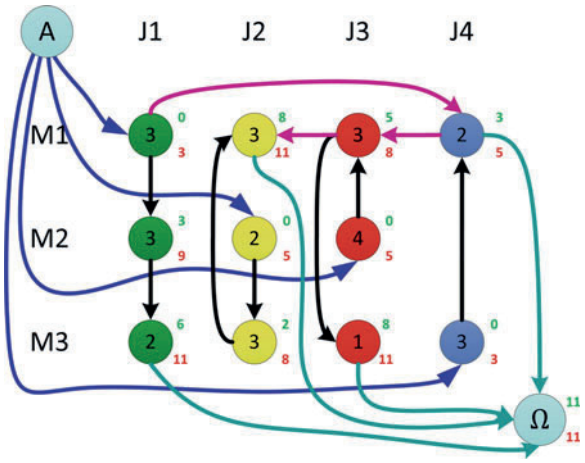

Fig. 9.8 Reduced network in the SBH heuristic ($SM = (1)$, $UM = (2, 3)$)

Iteration 2: $SM = (1)$, $UM = (2, 3)$

Now the procedure is repeated for all remaining unscheduled machines:

Determine min max L_j for all single machine problems incl. release dates and due dates, i.e. solve $1|r_j, d_j|L_{max}$ for all so far unscheduled machines.

M_2: min $L_{max} = 1$ for job sequences 231 and 321
M_3: min $L_{max} = 0$ for job sequences 4213 and 4231.

We choose the machine with the worst minimum max L_j value (i.e. M_2 here) and include for the respective solution the disjunctive arcs in Fig. 9.8. The result can be seen from Fig. 9.9. Here, the machine is uniquely determined. But the respective sequence is not unique (231 and 321 could be chosen). In the simplest version of the SBH we can choose arbitrarily among these sequences. Here we choose 231 on M_2 and update $SM = (1, 2)$, $UM = (3)$.

Performing forward and backward calculation of earliest starting times and latest finishing times of every operation yields the green and red numbers at every node in Fig. 9.9.

The longest path gives the current 'makespan' (for this still relaxed problem) which is the maximum of every job's last operation finishing time in Fig. 9.9, i.e.

$C_{max}(SM) = max\ (12, 12, 12, 6) = 12$, as well as the time value at Ω.

Iteration 3: $SM = (1, 2)$, $UM = (3)$

Determine min max L_j for all single machine problems incl. release dates and due dates, i.e. solve $1|r_j, d_j|L_{max}$ for all so far unscheduled machines.

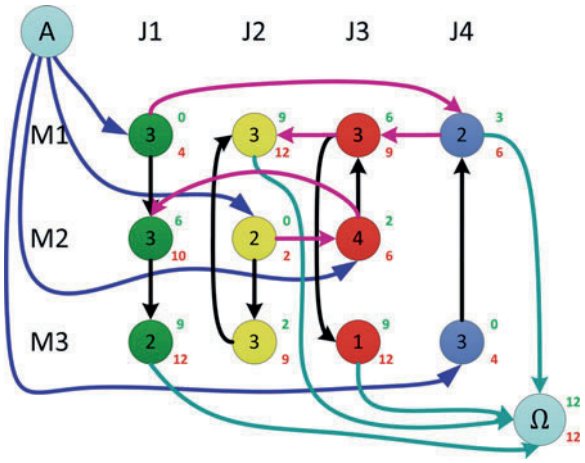

Fig. 9.9 Reduced network in the SBH heuristic ($SM = (1, 2)$, $UM = (3)$)

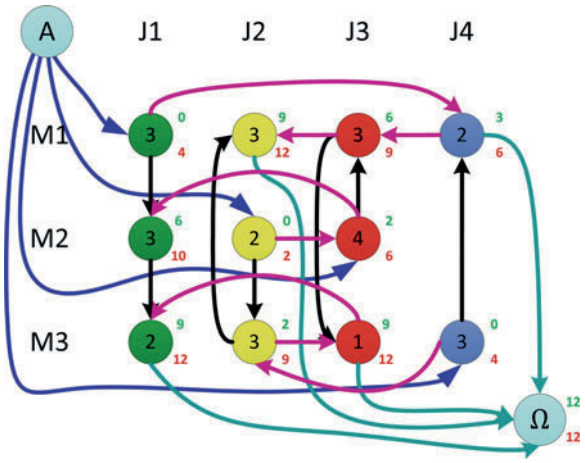

Fig. 9.10 Final network in the SBH heuristic ($SM = (1, 2, 3)$, $UM = \emptyset$)

M_3: min $L_{max} = 0$ for job sequences 4213 and 4231.

The result can be seen from Fig. 9.10. The machine is uniquely determined. But the respective sequence is not unique (4213 and 4231). Here we choose 4231 on M_3 and update $SM = (1, 2, 3)$, $UM = \emptyset$.

Performing forward and backward calculation of earliest starting times and latest finishing times of every operation yields the green and red numbers at every node in Fig. 9.10.

The longest path gives the makespan of the SBH application to the given job shop problem which is the maximum of every job's last operation finishing time in

Fig. 9.10, i.e.

$C_{max}(SM) = max\ (12, 12, 12, 6) = 12$, as well as the time value at Ω.

Since no unscheduled machines remain ($UM = \emptyset$), the SBH procedure is finished. (Note that only by chance the sequence of considering the machines was 1-2-3 in this instance. Usually, the sequence of machine indexes in the iterations must not need to be continuously increasing.)

Finally, we received a complete solution to the original job shop problem instance with makespan minimization which is—in our example instance—also the true optimum solution to this instance.

The solution data from the SBH can be easily deduced from Fig. 9.10.

Job sequences on every machine:

M_1: 1432
M_2: 231
M_3: 4231.

Finishing times of every operation:

$T(i, j)$	J_1	J_2	J_3	J_4
M_1	3	12	9	5
M_2	9	2	6	
M_3	12	6	10	3

Makespan value of this solution: $C_{max} = 12$.

9.3 Some Ideas Behind Heuristics for Manufacturing Scheduling

Once the detailed description of two representative heuristics has been presented, in this section we will outline some of the main ideas behind many heuristics for manufacturing scheduling. We first give some general ideas which are then developed in several subsections. These general ideas are:

- **Relaxation**. A first idea of scheduling procedures is to *relax the model* under consideration until the remaining formal problem becomes tractable. Afterwards the solution of the formal problem is adjusted and modified somehow to a solution of the real-world problem. This approach, as is well-known, is common to almost every model building, not even optimisation-oriented, in almost every real-world based setting.
- **Decomposition**. A second type is to *decompose the problem* under consideration until the remaining partial problems become tractable, to solve these partial

problems afterwards and to *coordinate* the partial solutions finally to receive an integrated solution. This approach is very common to manufacturing scheduling problems, may it be e.g. that scheduling tasks in different plants have to be performed simultaneously, for different manufacturing stages in one shop or for different machines on one stage. These examples imply a machine- or resource-oriented decomposition approach. However, job- or operation-oriented and/or time-oriented decomposition approaches can be developed as well. A main aspect for this approach is, if possible, to identify and treat the core or bottleneck problem (may it be given by a machine, a stage, a job etc.) as a whole and to coordinate the solution of this core problem with its environment.

Decomposition of a scheduling problem might be even pushed so far that on the most detailed levels only *'atoms'* of the problem are scheduled, e.g. jobs waiting in front of a machine are sequenced (and scheduled) by simple or more sophisticated *priority or dispatching rules*. Afterwards, these sequences have to be coordinated to an integrated schedule for the overall scheduling problem under consideration.

- **Adaptation**. Another, very often adapted way for generation of heuristics consists of identifying a smaller or simpler subproblem where a certain approach performs good or even optimal, this *approach is modified according to the more complex situation* under consideration and supposed to yield good results here as well. A prominent and well-known approach belonging to this group is Johnson's algorithm for the 2-machine flow shop problem with makespan objective which has been adapted in many different ways to more general problems, such as the m-machine flow shop problem with makespan objective by the heuristic of Campbell et al. (1970).

  With respect to the inclusion of additional resources into a scheduling problem and their temporal exclusion from the respective formal problem, time or capacity *buffers* for jobs/operations and/or resources might give some tolerances to include the limited resources afterwards. (This approach is less concerned with the solution procedure but more with buffers for the input data. In addition, these buffers might not only be included in the model input data but also be implemented after deriving an implementable solution from the formal problem's solution.)
- **Reversibility**. The structure of a schedule as result of some manufacturing scheduling procedure is often *forward-oriented*: Starting from the beginning (e.g. the assignment of the first operation to the first machine) the schedule ends with the final assignment (e.g. the last operation on the last machine). However, using just the opposite, *backward-oriented* perspective (i.e. first assigning the last operation on the last machine and then going backward until the first operation on the first machine has been scheduled) in many settings might give another solution to the decision maker which might even be more in line with the requirement of minimising tied-up capital or Just in Time-delivery.

We want to point out that the ideas and approaches mentioned in this section are by far neither comprehensive nor is the way of their presentation used here mandatory. Many other structuring and other aspects will be found definitely. We restricted ourselves to some main aspects employed in scheduling heuristics. On the

other hand, these ideas are not neccessarily confined to the manufacturing scheduling field. Finally, some of these ideas could be incorporated into metaheuristics.

Some specific guidelines can be constructed upon the general ideas outlined above. They start with the consideration of the single-operation as the basic unit of a scheduling model, thus ignoring the possibility of interrupting operations and generating sub-operations thereby. More specifically, we will assume that an operation

- belongs to a job and requires a machine, i.e. it combines a job and a machine,
- possibly requires additional resources,
- cannot be started until certain predecessing operations (usually of the same job) have been finished (or must be finished before its successors can start),
- has to be assigned to one or more (consecutive) time periods,
- ...

Model-based complexity reduction according to the framework discussed in Sect. 7.3.4 therefore implies, implicitly or explicitly,

- reduction of the number of operations to be considered in the reduced model,
- complexity-related modification of some or all operations themselves,
- omission or simplification/relaxation of (resource) constraints,
- ...

With these concepts in view some ways to reduce this complexity appear. Some of them are discussed in the next subsections.

9.3.1 Exclusion of Specific Operations

An obvious but at a first glance not very manageable guideline for the determination of operations which can be excluded in the simplified model is that these operations should not contribute significantly to the quality of the solution derived finally for the original model. However, there are at least three interpretations of this guideline which might be effective and which are used in model simplification.

First, a *machine-oriented interpretation* of this guideline proposes to completely exclude non-relevant machines in the simplified model, i.e. to simplify the model by omitting all operations of all jobs on these machines—and only consider the remaining model with significantly less machines. The choice of the machines to be excluded will often be based on their (non-) bottleneck property in the manufacturing system considered. Bottleneck machines will be preserved in the simplified model while non-bottleneck machines are deleted. Especially, if the number of machines in the simplified model becomes very low (e.g. one or two), the remaining simplified model might be solved with specialised algorithms or heuristics. A typical example of this idea is the SBH already discussed in Sect. 9.2.2. In our context, one of this approach's basic ideas is to identify the bottleneck machine in advance and then confine the simplified model to a one-machine problem which only considers exactly this bottleneck machine.

However, to include this machine's interaction with other machines, on one hand for every job a release time is included which incorporates the time needed for all previous operations of every job or better: the job's expected earliest availability for the bottleneck machine under consideration. On the other hand, a follow-up time is included for every job which represents the time needed to finish all its following operations. (As we have already seen, SBH is much more elaborate. Especially its 'shifting' property is used to consecutively apply the idea iteratively to different machines which, from an application point of view, is very well relevant if there are several bottlenecks or a hierarchy of importance of machines in the system. Several bottlenecks are not as uncommon if the capacities of machines in the manufacturing system under consideration are balanced at least from a mid-term or long-term perspective.)

Second, a *job-oriented interpretation* of the above guideline induces to exclude whole jobs in a simplified model, i.e. to omit all operations belonging to these prespecified jobs. If we remember that the number of jobs in manufacturing scheduling is the main driver of complexity in many solution procedures, reducing the number of jobs in a simplification approach turns out to be a rather attractive simplification strategy from a computational point of view. From an application point of view, job selection might be executed in two ways, i.e. positive selection and negative selection. Positive selection in our context means to choose 'important' jobs to remain in the scheduling model. Importance of jobs might result from their size, from customers' and/or products' importance, from their (expected) processing time variance, etc. Negative selection means to first consider all jobs and successively eliminate unimportant jobs in the simplification process until the number of jobs becomes manageable.

Third, these two approaches to simplification, i.e. job selection and machine selection, might be combined into a *machine/job-oriented interpretation*. And the approach might be even more specific with respect to operations when single operations are included or excluded in the simplification process.

Without going into more details here, it has also to be mentioned that the above outlined simplification approaches might also require some adjustment of the objective function in the simplified model. E.g., using bottleneck ideas might imply to include opportunity cost considerations into the simplified model.

9.3.2 Modification/Aggregation of (Specific) Operations

Somewhat like a generalisation of the approaches presented above in Sect. 9.3.1 is the modification and/or aggregation of operations themselves, i.e. of their processing times. Here as well, machine-oriented and job-oriented approaches can be identified.

Machine-oriented approaches are widespread in operations aggregation. Often processing times of consecutive operations of a job are simply added up straight or a weighted sum of processing times is used, e.g. as follows for a simple flow shop problem:

$$p_{kj}^{\text{agg}} = \sum_{i=\text{lowind}(k)}^{\text{uppind}(k)} g_i \, p_{ij} \, .$$

Here the processing time p_{ij} of job j on machine i is weighted using the machine-based weight g_i and added up from machine lowind(k) through machine uppind(k) while these machines define the k-th 'macro' machine in the simplified model by using machine aggregation. Dannenbring (1977) or Park et al. (1984) approximate approaches to multi-machine permutation flow shop problems with makespan objective give examples for a 'true' weighting approach while the procedure of Campbell et al. (1970) applies machine aggregation for the same type of problem by using equal weights $g_i = 1$ for all machines. These approaches simplify, i.e. aggregate the model into a two machine model which can be solved easily to optimum by Johnson's algorithm. This solution, as an approximate approach, is then applied to the original multi-machine problem under consideration.

Job-oriented approaches construct clusters of jobs which are usually interpreted as one 'macro' job in the simplified model. It is supposed that jobs included in one of these macro jobs are then processed consecutively. The processing times of a macro job are usually constructed as

$$p_{il}^{\text{agg}} = \sum_{j=\text{lowind}(l)}^{\text{uppind}(l)} g_j \, p_{ij} \, .$$

Here the processing time p_{ij} of job j on machine i is weighted using the job-based weight g_j and added up from job lowind(l) through job uppind(l) while these jobs define the l-th 'macro' job in the simplified model by using job aggregation. It has to be mentioned that this approach requires a prior assignment of jobs to macro jobs. This is usually much less obvious as compared with the machine-aggregation approach since actually the jobs might occur in (almost) every possible sequence (per machine). Therefore, machine-aggregation approaches are often used when the standard assumption of cluster construction is fulfilled, i.e. the objects (jobs) in a cluster are rather homogeneous while objects (jobs) from different clusters differ more or less significantly. This requirement, e.g., often is supposed to be fulfilled if changeover/setup times between jobs of one cluster are negligible while changeover times between jobs of different clusters are of significant size.

Clearly, machine-oriented and job-oriented aggregation approaches might be applied simultaneously. A further generalisation of this machine-based and/or job-based view on aggregation in scheduling problems can be derived from considering the disjunctive graph presentation of the original model and looking for network partitioning and network aggregation based on this problem representation. We will not discuss this further here.

9.3.3 Exclusion of Relations Between Operations

Ignoring constraints of the original model in the simplified one represents another group of simplification approaches. E.g. excluding resource constraints, precedence constraints, the sequence dependency of changeover times, etc. usually simplifies the model significantly. However, to receive a feasible solution to the original model might require sophisticated adjustment procedures with respect to the solution of the simplified model. We will not address these approaches in more detail here.

9.3.4 Modification of Time Scale or Adjustment Policy

Another rather simple but somewhat ambivalent way to reduce the complexity of a model is to reduce the time horizon of the jobs to be considered in the model. By this approach, on one hand probably the number of jobs to be considered is reduced. This reduces the effects of a main complexity driver. On the other hand, excluding later periods from consideration might yield adjustment problems at the end of the scheduling horizon and cause additional feasibility problems.

Instead of reducing the number of jobs to be considered by reducing the time horizon in the model, this number of jobs can also be reduced by performing an add-on scheduling approach instead of starting the procedure from scratch every time a (re-) scheduling is executed. That is instead of scheduling all jobs anew only new jobs are scheduled in an optimisation run while the former jobs' sequence (or only their relative sequence on every machine) remains constant. Basically these approaches adapt the difference known from regenerative and net-change approaches in MRP to manufacturing scheduling. This approach is rather common in dynamic scheduling settings.

9.3.5 Identification of Symmetry Properties

Another (real-world) based idea of constructing an approximate solution is applying an existing approach/method to the inverted or *symmetric* model. The symmetric model inverts the machine sequence of a job's operations, i.e. the last operation (on the job's last machine) is supposed to be the job's first operation in the reverse problem, job's last but one operation is its second operation in the reverse problem and so on. For several scheduling problems it is straightforward that for every problem instance the objective function value of the original model is equal to the objective function value of the symmetric model. The idea behind this 'concept' originates from the consideration that in a network (or in a disjunctive graph) for many problem settings the longest path (e.g., the project duration or the makespan) is the same no matter whether it is calculated from the beginning to the end or vice versa. Therefore, applying a certain approximate approach to the original model or to the symmetric

model probably yields two different solutions to the model (where, of course, the solution of the symmetric model has to be reverted again to yield a solution to the original model). However, these two solutions probably will both be of adequate quality and the best one of these two solutions could be finally chosen. Additionally, it should be mentioned that very often when testing approaches using randomly generated test-beds, both, the original instance as well as its symmetric counterpart represent valid problem instances. Therefore, from a numerical testing point of view, there is no difference between the application of an approach to the original or to the symmetric model when using this approach for each of these two instances. Furthermore, the increase in effort by applying an approach also to the symmetric model is nothing but duplication. And by experience, in several problem settings nice improvements might be achieved by using this combined type of approach, i.e. applying a given approach to both, the original as well as the symmetric model.

Symmetry ideas have been described, for instance, in the book of Pinedo (2012). One paper where these ideas are employed for designing a heuristic is Ribas et al. (2010).

9.3.6 Combination of Enumeration and Construction of Solutions

Enumerative constructive methods try to employ enumerative components into the generation of a solution to the scheduling problem. According to Fig. 7.4 in Sect. 7.3.4, these methods therefore refer to a given model and try to simplify the search for a solution, i.e. they usually can be assigned to situation 2 in Fig. 7.4. Simple approximate enumerative constructive methods include

- sequences and accordingly schedules resulting from the simple application of static or dynamic dispatching rules as well as
- every premature stopping of an (exact) explicit or implicit complete enumeration approach.

There is almost no limit to where imagination can take the design of enumerative constructive methods. However, several rather successful methods might be classified according to the following iterative scheme which refers to job-based iteration approaches. Per iteration, these schemes select exactly one job from the set of so far non-sequenced jobs, include this job into the (sub-) sequence(s) resulting from the previous iteration and adjust the sets of sequenced and non-sequenced jobs (as well as the respective (sub-) sequences) as follows:

Suppose that in the beginning of iteration k of the method under consideration the set J of all jobs is partitioned into the set $S(k-1)$ of those jobs which have already been sequenced in iterations 1 through $k-1$ and the remaining jobs are included in set $R(k-1)$. The set of (sub-) sequences sequenced up to iteration $k-1$, called $T(k-1)$, might be of cardinality 1 or higher.

Then one or more or even all jobs from $R(k-1)$ are tentatively and one by one selected according to some selection criterion and integrated into the (sub-)

sequence(s) of $T(k-1)$. The job k^* which yields the best (sub-) sequences is transferred from $R(k-1)$ to $S(k-1)$, i.e.

$$R(k) = R(k-1) - (k^*) \text{ and } S(k) = S(k-1) \cup (k^*).$$

$T(k)$ consists of those (sub-) sequences which have been accepted in iteration k.

Of course, this approach can be expanded in many directions. E.g.,

- some already sequenced jobs can be withdrawn from $S(k)$ and be put back into to $R(k)$,
- several jobs might be assigned to $S(k)$ simultaneously,
- the above approach might be applied to some tree structure, i.e. several jobs can be alternatively assigned to $S(k)$ and each of the resulting alternatives is considered in the following iterations separately,
- ...

All these expansions have to be defined such that the resulting effort for the method remains acceptable, i.e. the computational effort does not grow exponentially but remains polynomially bounded.

9.3.7 Generation of Variants of Efficient Heuristics

Another possibility is to analise the behaviour of existing heuristics and trying to generate new variants by performing a 'reverse engineering' of the heuristic, i.e. decomposing the original heuristic into blocks and generating variants for the blocks. Obviously, the development of this idea is very heuristic-specific, and in order to illustrate it, we take the NEH heuristic already discussed in Sect. 9.2.1. Recall that this heuristic (a) starts from an initial sequence, and (b) construct and evaluates partial sequences in an iterative manner. It has been discussed whether the initial sequence of step (a) or the numerous (partial) sequences considered in step (b) contribute more to the very good performance of NEH. Framinan et al. (2003) show that depending on the objective function other sorting criteria in (a) may be advantageous. However, the numerous sequences considered in (b) contribute as well a lot to the quality of the heuristic.

With respect to generalisations of the NEH approach, several extending links can be identified:

- The sequence of involvement of jobs may be modified. Woo and Yim (1998), e.g. use an ascending ordering of jobs for the permutation flow shop problem with flowtime objective. Also, several different sequences of involvement may be used to generate several schedules and the best of these schedules is then selected as solution to the scheduling problem (see Framinan et al. 2003).
- More than one job can be selected in the iterative loop (b) to be included into the sequence. Again, Woo and Yim (1998) choose every remaining job to be

inserted in every iteration of (b). This approach remains with polynomial complexity and yields good results for the flowtime flow shop problem but is rather time-consuming.
- As already mentioned, ties may occur both in (a) and (b). A sophisticated treatment of ties may improve the approach's performance noticeably. Nevertheless, keeping the computational effort polynomially bounded might be an issue in this case.

9.3.8 Vector Modification

Since manufacturing scheduling models are specific combinatorial optimisation problems and the solutions of these problems can usually be represented as vectors, many improvement approaches are related to vector modification.

First of all, it has to be mentioned that for some models, different schemes of vector representation for the same schedule may exist. Applying the same improvement method to different vector representation schemes might result in different solutions to be checked for improvement and, consequently, also in different improved solutions. We only allude to this aspect here.

Supposing a solution vector, i.e. a feasible schedule is given, many vector manipulation approaches can be designed. These approaches normally withdraw one or more operations (or even whole jobs) from the vector and place it/them at other position(s) in this vector. The approach might be repeated until a certain stopping criterion has been fulfilled. A most simple and sometimes rather efficient vector manipulation scheme is to exchange each two consecutive positions and to (a) look whether the new vector maintains feasibility and (b) yields a better objective function value. This procedure is repeated, e.g., until no further improvement is possible.

More general and supposing that a solution vector is given, vector manipulation approaches consist of the following components:

1. Determine the position(s) of the current vector from where operation(s)/job(s) are withdrawn.
2. Determine the position(s) where this/these withdrawn operation(s)/job(s) are re-inserted.
3. Modify the vector under consideration according to (1) and (2). (Maybe, more than one vector results from this procedure. Step (2) defines the so-called neighbourhood of the vector(s) from (1) which is/are examined by the vector modification approach under consideration.)
4. Check acceptance of the new vector(s) with respect to
   (a) Feasibility
   (b) Acceptance of objective function value.
5. If a pre-specified termination criterion (e.g. maximum number of iterations has been reached, no adequate improvement has been reached,...), then stop.
6. Use the accepted vector(s) from (4) as input for the next iteration and go to (1).

Three additional remarks have to made with respect to this scheme:

1. With respect to 4(b) it has to be mentioned that in its simplest version usually a new vector will be accepted if its objective function value improves the so far best solution. However, because of the pure greedy character of this approach and because of the non-convexity of the underlying combinatorial optimisation problem, this procedure might result in a suboptimal dead end very quickly as has been experienced many times. Therefore, there are approaches which explicitly allow for temporary worsening of the objective function value of intermediate solutions or vectors respectively, just to overcome this quick ending in a suboptimal solution. The extent to which such a temporary worsening of the objective function value is accepted represents another control variable of the enumerative improvement method. Usually, this extent of allowed worsening decreases with the number of iterations of the improvement process. Additionally, the extent of worsening might be determined stochastically, e.g. in combination with a decreasing expected value of accepting this worsening and in combination with a decreasing variance.
2. With respect to (1) and (2) the position(s) of change might be selected deterministically or according to some probabilistic scheme. Random determination of deletion and insertion positions is obviously the most simple of these schemes. However, more sophisticated determination of deletion and insertion locations, e.g. including improvement experience from the last iterations, might be appropriate, may it be deterministically or stochastically.
3. Another aspect which we only allude to here is the requirement that the effort for executing a vector modification approach must remain limited. Therefore, usually a time limit and/or some iteration limit will be set for the procedure. However, even if these two limitation parameters have been determined, (1) and (2) from the above scheme easily might suggest enumeration schemes which can become rather quick rather complex with respect to their computational effort because of the numerous solutions to be considered. Therefore, the limitation of the computational effort has to be kept track of when designing the above procedure, especially with respect to steps (1) and (2).

9.3.9 Modification Approaches Based on Real-World Features

Numerous approaches try to improve a given solution to a manufacturing scheduling problem by analyzing the real-world structure of a given solution. E.g. reducing idle time of machines by bringing forward operations into idle slots of machines might positively contribute to flowtime and makespan while postponing to later idle slots can contribute to reduced earliness. Every plausible consideration with respect to the real-world problem might be checked regarding the modification of solutions. These modifications might be considered myopically, i.e. sequentially for different

operations, or jointly for several operations. Note that the diversity of this type of approaches is again enormously.

Another type of approach is to include the human planners' expertise for a given schedule, i.e. to explicitly address the men-machine-interface. Corresponding approaches have turned out to be more or less successful empirically in many settings. Especially, if formerly manually solved manufacturing scheduling problems in practice are automated, on one hand they usually have to be competitive with the former manual processing which then serves as a benchmark with respect to the relevant indicators or objective function(s) for the automated solution. On the other hand, allowing the human planner 'to have a look' at the automated solution might improve even this solution and/or will probably increase acceptance of the automated solution within the real-world manufacturing setting. However, note that these third-type approaches refer to planners' intuition which might be mostly non-quantifiable, including the application of specific methods 'just in time'.

9.4 Metaheuristics

A metaheuristic can be seen as a general algorithmic framework which can be applied to different optimisation problems with relatively few modifications to make them adapted to a specific problem. Metaheuristics are commonly grouped into large classes of methods that need instantiation for a given problem. Although metaheuristics can be seen as a specical type of improvement heuristic, they tend to be more robust and flexible, in the sense that there is a common metaheuristic template for each class of metaheuristic methods that, after instantiation, is able to give reasonable solutions to mostly any scheduling problem. Furthermore, metaheuristics are usually able to obtain much more refined solutions. However, this usually comes at a higher computational cost and complexity than heuristics, although this later fact depends on the level and grade of the instantiation.

Metaheuristics are grouped, as has been stated, in different classes. The earliest methods are population and/or evolutionary based algorithms like genetic algorithms and evolutionary programming. Other classes include simulated annealing, tabu search, ant colony optimisation, particle swarm, estimation of distribution, differential evolution, scatter search, harmony search, honey bees, artificial immune systems and a really long list of other methods. In this section, we first discuss the main concepts in the general framework of metaheuristics, and then present some of the main types of metaheuristics, i.e. simple descent search methods (Sect. 9.4.2), simulated annealing (Sect. 9.4.3), tabu search (Sect. 9.4.4), and genetic algorithms (Sect. 9.4.5). Some other metaheuristics are briefly mentioned in Sect. 9.4.6.

9.4.1 Main Concepts

As a general algorithmic framework, a metaheuristic is composed of a set of concepts that can be used to define or to guide specific heuristic methods that can be applied to a wide set of different problems. These concepts are mainly the following:

- Problem or solution representation.
- Initialisation.
- Neighbourhood definition.
- Neighbourhood search process.
- Acceptance criterion.
- Termination criterion.

These concepts are discussed in detail in the next subsections.

9.4.1.1 Solution Representation

Metaheuristics commonly work with an abstract representation of solutions. Particularly, in scheduling settings, a metaheuristic would often work with a sequence of jobs, rather than with a full schedule. We have already seen that this is also the case with regular construction heuristics. This solution representation abstraction is key to most metaheuristic classes. Picture for example a metaheuristic working over a permutation of jobs as a representation. This permutation could be the sequence for a single machine or permutation flow shop layout, regardless of the objective and/or constraints present in the problem.

The choice of the representation is crucial to metaheuristic performance. An indirect representation might be more compact, but large parts of the solution space might not be possible to represent. On the contrary, an overdetailed representation might capture the complete solution space but might result in a cumbersome and complex metaheuristic.

An example of a permutation representation of a 1-machine, 20-job problem is given in Fig. 9.11. We can see that the permutation representation contains 20 positions. The solution representation space is formed by all possible permutations in Π of 20 jobs. The solution space has a cardinality of 20!. One possible permutation $\pi \in \Pi$ could be the one shown in the same Fig. 9.11 where the job in the first position of the permutation is $\pi_{(1)} = 7$, the job in the second position is $\pi_{(2)} = 12$, and so on until the job in the 20-th position: $\pi_{(20)} = 8$.

Note that a permutation solution representation could be used for many different scheduling problems, including single machine, flow shop or even hybrid flow shop layouts if some machine assignment rules are employed, as those explained before in Sect. 7.4. However, this representation is not valid for all problems. In parallel machine layouts, it is not easy to separate the machine assignment decision from the sequencing problem. Therefore, other representations are needed. The most common is the 'multi-permutation' where each machine has a permutation of jobs which

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
7	12	2	5	13	9	3	1	17	19	10	6	15	20	18	11	16	4	14	8

Fig. 9.11 Example of a permutation encoding with 20 jobs

Fig. 9.12 Example of a multi-permutation encoding with 20 jobs and three machines

	1	2	3	4	5	6	7
Machine 1	7	13	20	14	1	3	6

	1	2	3	4	5	6
Machine 2	12	11	5	19	10	8

	1	2	3	4	5	6	7
Machine 3	2	9	18	17	15	16	4

indicate the order in which they are processed on that machine. Notice that the term is not very accurate as each job has to be assigned to exactly one machine. As a result, each machine does not really contain a permutation of all the jobs, but rather a list of the assigned jobs. An example for a three machine, 20 job representation is shown in Fig. 9.12.

As we can see, every job appears once, but not all machines have the same number of jobs assigned. In the example, we can see that $\pi_{(2,4)} = 19$ i.e. the job in the fourth position of the second machine is the job 19. Notice that for complex problems, like for example, a hybrid job shop lay out with stage skipping and recirculation, much more complex representations are needed. Figure 9.13 shows a possible three stage hybrid flow shop with two, three and two parallel machines at each stage, respectively.

9.4.1.2 Initialisation

In general, a metaheuristic needs at least one starting solution. This initial solution is usually referred to as 'seed'. For some problems, this solution might just be generated at random. However, a random solution is expected to be of a very low quality. In some other cases, a random solution might even be unfeasible. For these reasons, it is usually common in the scheduling methods literature to initialise metaheuristics with good known constructive heuristics. In the very least, there is nothing wrong in using a good dispatching rule to seed a metaheuristic algorithm.

Some metaheuristic methods however, rely on a completely random initialisation as a heuristic initialisation might bias the search process to a specific region of the solution space, possibly hindering to find a global optimum solution in some way or another. Additionally, for some problems, there are no good known heuristics and/or dispatching rules do not offer sufficiently good quality solutions. In all these cases, it is common to randomly initialise metaheuristic algorithms.

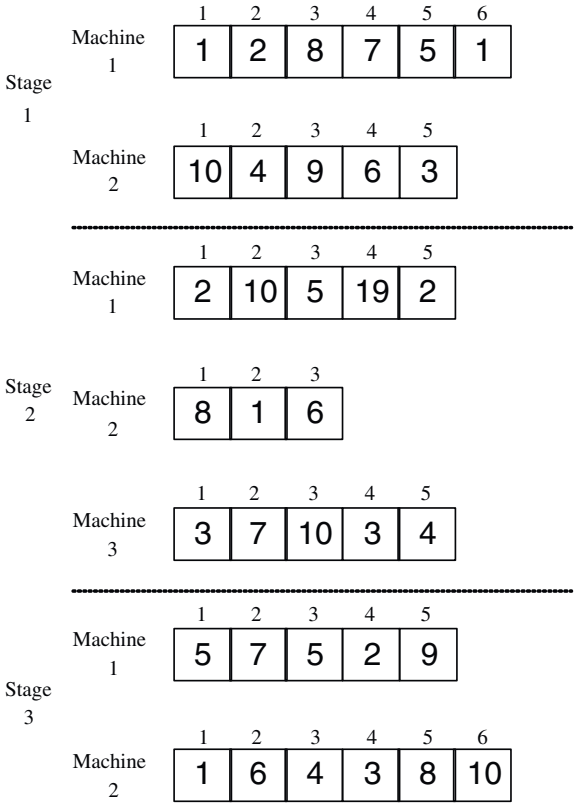

Fig. 9.13 Example of a more complex representation for hybrid layouts

9.4.1.3 Neighbourhood Definition and Search Process

Given the solution representation and also given the impossibility of examining the whole cardinality of the solution representation space, which is often of exponential size, the next concept that is needed in metaheuristics is the neighbourhood of a solution. The neighbourhood of a solution is a set of all possible solutions in the representation space that can be reached by carrying over a small local movement in the solution. For example, given a permutation, a simple neighbourhood is the one in which two adjacent jobs exchange their positions. This neighbourhood is called the adjacent swap. Let us define this adjacent swap neighbourhood in a more formal way. Let us suppose that we have a sequence of jobs expressed as a permutation $\pi = (\pi_{(1)}, \pi_{(2)}, \ldots, \pi_{(n)})$, one adjacent swap neighbourhood is obtained by selecting each possible job in position j and its immediately adjacent job in position k where $j = 1, \ldots, n-1, k = j+1$ and swapping the two jobs or also, the jobs in these two positions of the permutation. Given these two positions j and k, and the corresponding jobs in these two positions, namely, $\pi_{(j)}$ and $\pi_{(k)}$ the resulting sequence after the swap will be $\pi' = (\pi_{(1)}, \ldots, \pi_{(j-1)}, \pi_{(k)}, \pi_{(j)}, \pi_{(j+1)}, \ldots, \pi_{(n)})$.

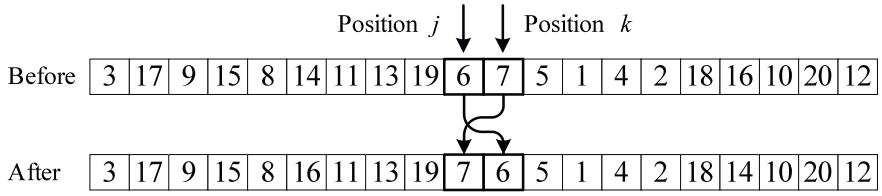

Fig. 9.14 Adjacent swap neighbourhood (A) example

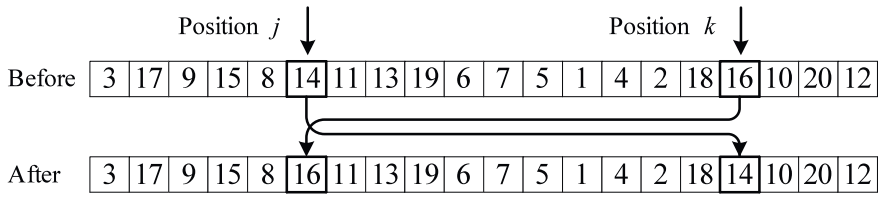

Fig. 9.15 General swap neighbourhood (S) example

More specifically, we can define an adjacent swap 'move' as a duple (j, k) where $j = 1, \ldots, n-1, k = j+1$. The whole set of adjacent swap moves can be defined as $A = ((j,k) : j = 1, \ldots, n-1, k = j+1)$. As we can see, the number of possible moves or cardinality of the adjacent swap neighbourhood is $|A| = n-1$. This means that any possible permutation or sequence of jobs π has $n-1$ adjacent swap neighbours. The adjacent swap neighbourhood of a permutation sequence π is finally and formally defined as: $N(A, \pi) = (\pi_v : v \in A)$. A graphical representation of this move is shown in Fig. 9.14.

An immediate generalisation of the adjacent swap neighbourhood is the general swap neighbourhood. In this case, the two positions j and k need not be adjacent. We can then define the general swap neighbourhood as a duple (j, k) where $j = 1, \ldots, n-1, k = j+1, \ldots, n$. The general swap movements are defined as $S = ((j,k) : j = 1, \ldots, n-1, k = j+1, \ldots, n))$. The cardinality of this neighbourhood is greater than that of A since $|S| = n \cdot (n-1)/2$. Formally, the general swap neighbourhood of a sequence π is defined as: $N(S, \pi) = (\pi_v : v \in S)$. Given two jobs in positions j and k where $j = 1, \ldots, n-1, k = j+1, \ldots, n$, the resulting sequence after swapping the jobs in these positions will be: $\pi' = (\pi_{(1)}, \ldots, \pi_{(j-1)}, \pi_{(k)}, \pi_{(j+1)}, \ldots, \pi_{(k-1)}, \pi_{(j)}, \pi_{(k+1)}, \ldots, \pi_{(n)})$. This is graphically depicted in Fig. 9.15.

Another very common neighbourhood employed in permutation sequences is the insertion neighbourhood. This is induced by all local movements where there are two jobs at positions j and k, where $j \neq k$, and the job at position j is extracted and reinserted at position k. More formally, the set of insert movements I is defined by all duples (j, k) of positions where $j \neq k,\ \ j, k = (1, \ldots, n)$ that, starting from any sequence π result in a sequence

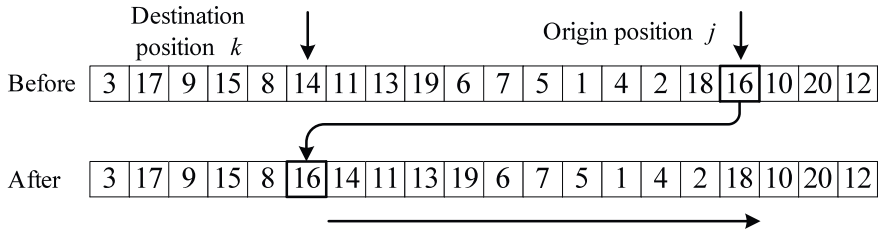

Fig. 9.16 Insert neighbourhood (I) example

$$\pi' = \pi_{(1)}, \ldots, \pi_{(j-1)}, \pi_{(j+1)}, \ldots, \pi_{(k)}, \pi_{(j)}, \pi_{(k+1)}, \ldots, \pi_{(n)}$$

if $j < k$, or in a sequence:

$$\pi' = \pi_{(1)}, \ldots, \pi_{(k-1)}, \pi_{(j)}, \pi_{(k+1)}, \ldots, \pi_{(j-1)}, \pi_{(j+1)}, \ldots, \pi_{(n)}$$

if $j > k$.

The set of insertion moves I is then defined as $I = ((j,k) : j,k = (1,\ldots,n), j <> k)$ and the insertion neighbourhood as $N(I,\pi) = (\pi_v : v \in I)$. Note that the cardinality of I is greater than that of A or S as $|I| = (n-1)^2$. A graphical example of the insertion neighbourhood is given in Fig. 9.16.

Of course, the previous three neighbourhoods are not the only possible ones. There are many other neighbourhoods, like reverse, block swap, block insert, three job insert, etc.

Once the neighbourhood has been defined, a metaheuristic method usually explores neighbours of a given solution in a given way. The general idea is that good solutions are clustered close to each other, i.e. a very good solution will have a hopefully even better one in the neighbourhood. Let us picture an example in the single machine layout with a permutation representation. The objective is the total tardiness minimisation. A very good starting solution has been found by applying the Earliest Due Date (EDD) dispatching rule. It is straightforward to see that an even better solution can be found by selecting a job that is being completed early and swapping it with another job that is being completed late. Normally, the late job will be processed earlier in the sequence and the early job later in the sequence. This is an example of a local movement in the general swap neighbourhood. Hopefully, after this swap, the early job is not tardy and the tardy job is now early. As a result, the total tardiness will have been reduced even further.

With the previous example, neighbours of a solution can be examined sequentially or randomly. Also, given a solution, all neighbours might be examined, and the 'best' selected or the first 'better' neighbour might be selected. This is known as 'best improvement' or 'first improvement', respectively.

There is a known trade-off between the cardinality of the neighbourhood, the quality of the expected solutions and the time it takes to traverse the neighbourhood. Obviously, larger cardinality neighbourhoods are more expensive to examine but

higher quality solutions are expected. What to do with new solutions is discussed next in the acceptance criterion.

9.4.1.4 Acceptance Criterion

Usually, neighbouring solutions will result in different objective function values. As such, metaheuristic methods have to implement an acceptance criterion to decide if the incumbent or current solution has to be replaced by any of its neighbours. Again, the underlying idea is to move to the best neighbour and then, from that best neighbour, move again to the best neighbour and continue until no more improvements are found. Accepting only better solutions is not the only possible approach. More elaborated methods allow to move to worse solutions temporarily, in the hope of finding better solutions later on. When accepting a worse solution, there might be probabilistic decisions as how much deterioration in the objective function is allowed. As a result, the acceptance criterion can be highly sophisticated.

9.4.1.5 Termination Criterion

Contrary to constructive heuristics, that commonly finish when a full sequence has been built, metaheuristics iterate until a specific, usually user given, termination criterion is met. The choice of stopping criterion is varied:

- Stop after a predefined number of iterations and/or elapsed time.
- Stop after a predefined number of iterations and/or elapsed time without improvement in the objective function value.
- Stop after a predefined number of iterations and/or elapsed time where no more than an X % improvement in the objective function value has been observed.

Some scheduling problems are known to have tight lower bounds that can be calculated, i.e. best case objective function values that are known a priori. Therefore, it is possible to run a metaheuristic method and to stop it when a given threshold deviation from the best possible optimal solution has been surpassed. However, tight lower bounds are only known for some simple layouts and rather unconstrained scheduling settings.

9.4.2 Simple Descent Search Methods

With all previous elements of metaheuristics, it is possible to define a simple method. Simple descent or iterative descent, also referred to as hill climbing local search, basically starts from an initial feasible solution and searches through a given neighbourhood while improvements are found. The algorithm template is given in Algorithm 3.

It has to be noted that the symbol '$\triangleleft$' is used as 'better than' in the listing of Algorithm 3. Note that basically, the simple descent local search method starts from an initial job sequence and iterates in a given neighbourhood, looking for better neighbours until all the given neighbours of the best solution found so far are not better, in which case the algorithm stops. The previous listing of Algorithm 3 is actually referred to as a first improvement type of descent local search since given a

Algorithm 3: Simple descent local search

```
Input: Instance data
Output: Best solution π* and best objective function value obj*
begin
    Calculate initial solution π_i;
    Set best known solution π* := π_i;
    Calculate best objective function value so far obj* := Obj(π_i);
    improving:=True;
    while improving=True do
        improving:=False;
        foreach neighbour π' of π* in V do
            Calculate objective function value of neighbour obj' := Obj(π');
            if obj' ◁ obj* then
                π* := π';
                obj* := obj';
                improving:=True;
    return Best solution π*, best objective function value obj*
end
```

solution, the first improving neighbour replaces the current solution. An alternative would be to examine all neighbours of a given solution sequentially, and replacing the current solution only with the best neighbour.

With the computing power available today, it is fairly straightforward to add a simple descent local search method as a post-processing stage after a solution has been obtained with a dispatching rule or heuristic algorithm. In the worst case scenario, for example in big instances where there might be thousands of improving iterations, the descent local search procedure can be stopped after a given predefined time has elapsed. However, there are many scheduling problems where neighbourhoods are hard to define, and/or each candidate solution is complex to calculate. In such scenarios, iterating through the neighbourhood can be a costly procedure.

9.4.3 Simulated Annealing

One big drawback of the simple descent local search methods is that they might get easily trapped in local optimum solutions. This is expected as they are specifically designed to stop at a solution that has no better neighbours. The problem is that in many cases, after just a few iterations, the algorithm will stop in a fairly mediocre local optimum. The problem of escaping from local optima is approached from many different perspectives. The simplest method could be an iterated simple descent local search where the simple descent local search is iteratively applied to a different random initial solution each time. Although this method will, after some iterations, provide a better solution than the simple descent local search, the probability of finding the optimum solution is slim. Recall that the search spaces in scheduling problems tend to be exponential in size and randomly performing local search is like searching for a needle in a haystack.

One of the earlier methods to cope with the local optimality trapping problems is the simulated annealing metaheuristic. Simulated annealing, as the same implies, emulates the physical annealing process where materials of crystalline structure are slowly cooled off and re-heated in order to alleviate tensions and to foster a perfect crystalline structure. The analogy to local search is the following: During the search, neighbours are always accepted if they are better than the current solution. However, if the first or best neighbour is worse, it will be probabilistically accepted with a diminishing probability that depends on the time that has elapsed (cooling schedule: the more time the algorithm has been running, the lower the probability to accept worse solutions) and also on the difference between the current solution and the best non-improving neighbour (the higher the difference, the lower the probability to accept a worse solution). The template of Simulated Annealing is given in Algorithm 4.

Note that *Random* returns a random uniformly distributed real number between 0 and 1. The probabilistic expression $e^{-\frac{D}{T}}$ returns a diminishing number as D increases and/or as T decreases. Furthermore, $0 < r < 1$ is referred as the cooling rate and it determines the cooling schedule.

As one can see, a high initial temperature and an r value close to 1 will ensure that a large number of iterations will be carried out. Some more advanced designs add a number of iterations L at each temperature level and other complex settings.

Simulated annealing was proposed for optimisation problems by Kirkpatrick et al. (1983) and by Černý (1985). Scheduling applications appeared in the late eighties and early nineties with the works of Osman and Potts (1989) and Ogbu and Smith (1990) to name just a few.

Simulated Annealing, as many other metaheuristics, employ different parameters that might affect the performance of the algorithm. The listing of Algorithm 4 shows some parameters as the initial temperature T_{ini} and the cooling rate r, apart from the initial solution. These parameters have to be tuned for performance and efficiency and there is a whole area in experimental algorithms devoted to the tuning of metaheuristics. Some relevant texts are those of Birattari (2005) and Hoos et al. (2005).

9.4.4 Tabu Search

Tabu Search or Taboo Search is actually a family of methods that, similarly to simulated annealing, include some mechanisms to escape from local optima. A key difference with simulated annealing is that, in tabu search, the incumbent solution is always replaced with the best neighbourhood, regardless of whether this neighbour is actually better or worse than the current solution. Obviously, this approach can lead to infinite loops in which the search moves from one solution π_1 to another neighbouring solution π_2 and from that one again back to π_1. In order to avoid this, tabu search methods keep a list of recently visited solutions. This list is what gives

Algorithm 4: Simulated Annealing

Input: Instance data
Output: Best solution π^* and best objective function value obj^*
begin
  Calculate initial solution π_i;
  Set best known solution $\pi^* := \pi_i$;
  Calculate best objective function value so far $obj^* := Obj(\pi_i)$;
  Set current solution $\pi_c := \pi^*$;
  Set initial temperature $T := T_{ini}$;
  while *Stopping criterion is* ***not*** *satisfied* ***or not*** *frozen* **do**
    foreach *neighbour* π' *of* π_c *in* V **do**
      Calculate objective function value of neighbour $obj' := Obj(\pi')$;
      if $obj' \triangleleft obj^*$ **then**
        $\pi^* := \pi'$;
        $obj^* := obj'$;
      else
        Calculate difference in objective function $D := obj' - obj^*$;
        Accept worse solution probabilistically;
        if $Random \leq e^{-\frac{D}{T}}$ **then**
          $\pi_c := \pi'$;
    $T := r \cdot T$;
  return *Best solution* π^**, best objective function value* obj^*
end

the name to this family of metaheuristics as it is called the Tabu list. The tabu list usually does not contain full visited solutions, but rather elements of the visited solutions or the movements that were carried out in the solution elements at the most recent iterations. Therefore, when moving from one solution to the best neighbour, the movement is not accepted if it leads to a tabu solution. The result is a methodology that has a short term memory that avoids the formation of cycles in the search.

A comprehensive reference on the mechanisms of Tabu Search is Glover and Laguna (1997), while some examples of its application to manufacturing scheduling are Widmer and Hertz (1989); Nowicki and Smutnicki (1996) and Nowicki and Smutnicki (1998) to name just a small set.

9.4.5 Genetic Algorithms

Genetic algorithms were initially proposed by Holland (1975) and later by Goldberg (1989). More recent books are those of Michalewicz (1996) and Davis (1996). The underlying idea greatly departs from the previous simulated annealing or tabu search methods. Genetic algorithms mimic evolutionary biological processes such as inheritance, mutation, selection, crossover, natural selection and survival of the fittest.

Genetic algorithms are also different from earlier metaheuristics in that they are population based. The search is not carried out over one single solution but rather over a set of solutions. This set of solutions is referred to as 'population' and each member of the 'population' is called an individual. Individuals are actual solutions to the problem, or rather, solution representations. In the genetic algorithm jargon, each individual is formed by a 'chromosome' in which the solution elements are encoded. For scheduling problems, chromosomes can just be the solution representations, for example, job permutations that form sequences.

The population is evaluated and all individuals are assigned a 'fitness value'. This fitness value is directly related with the objective function value, in such a way that better objective function values are assigned higher fitness values. A selection operator is introduced. This operator randomly chooses individuals from the population with a strong bias towards fitter solutions. As a result, the fittest individuals have greater probabilities of being selected. Selected individuals undergo a crossover procedure in which new individuals are generated. The crossover operator tries to generate new solutions by combining the best characteristics of the fitter selected individuals. A further mutation operator is introduced to randomly alter some characteristics of the newly created individuals. Lastly, the new solutions are evaluated and introduced in the population. The previous process is repeated generation after generation.

Algorithm 5: Genetic Algorithm

Input: Instance data
Output: Population *Pop* of solutions, including Best solution π^* and best objective function value obj^*
begin
  Initialise a population *Pop* of P_{size} individuals;
  Evaluate individuals;
  while *Stopping criterion is* ***not*** *satisfied* **do**
    Select individuals from *Pop*;
    Cross individuals;
    Mutate individuals;
    Evaluate individuals;
    Include individuals in the population;
  return *Pop, Best solution* π^**, best objective function value* obj^*
end

Genetic algorithms are exotic as far as metaheuristic methods go. When first exposed to them, the natural reaction is to wonder how this seemingly complex mechanism could work. Actually, once individuals are seen as simple representations of actual job schedules, the process is much better understood.

For example, a simple genetic algorithm using job permutations, for example, for a single machine layout scheduling problem could be based on the template shown in Algorithm 5.

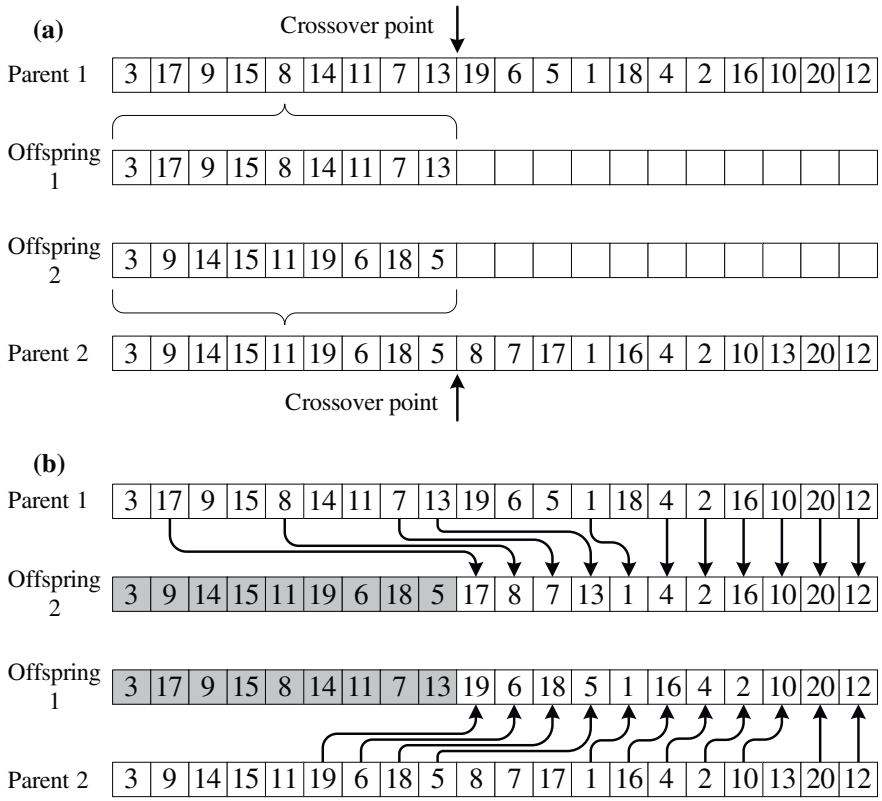

Fig. 9.17 One point order crossover in genetic algorithms. **a** Offspring inherit the jobs prior to the random cut point from the direct parent. **b** Remaining jobs are inherited from the other parent in their relative order

A population of, lets say, 50 individuals could be randomly generated as simple random job permutations. Fitness evaluation could simply be the objective function value. The most simple selection scheme is called the binary tournament selection, where two random individuals are selected from the population and the one with the best objective function value is selected as the first parent. The process is repeated to select a second parent. These two parents are crossed over to generate two new solutions as for example shown in Fig. 9.17 for what is called the one point order crossover.

As we can see, a random crossover point is generated and the jobs prior to that point are copied to one of the new solutions from a direct parent. Then, the jobs not already present in the permutation, are copied from the other parent in the order in which they appear. It is interesting to see how the new solutions contain characteristics of both previous solutions. This can be seen as a complex form of local search.

The mutation operator can be, for example as simple as a single adjacent job swap, or a single job insertion. After crossover and mutation, the new solutions are

evaluated and usually, replace the old solutions that were selected. In some more advanced genetic algorithms, an elitism operator is used in order to preserve the best individuals from the population to be replaced by new solutions of worse quality.

Applications of genetic algorithms to scheduling problems appeared as early as in the middle 1980s in Davis (1985) and also at Mattfeld (1996); Syswerda (1996); Biegel and Davern (1990); Chen et al. (1995); Reeves (1995) and more recently, Ruiz et al. (2006). Note that this is just a small list as genetic algorithms are among the most prolific metaheuristics from the literature. The interested reader is redirected to such texts for more information.

9.4.6 Other Metaheuristics

There are countless metaheuristic methods or classes of metaheuristics. We have previously commented briefly over simulated annealing, tabu search and genetic algorithms. However, there are much more classes and the number continues to grow. The following list contains some other classes of methods along with some references.

- Ant Colony Optimisation (Dorigo and Stützle 2004).
- Artificial Immune Systems (Dasgupta 1998; de Castro and Timmis 2002).
- Differential Evolution (Price et al. 2005; Feoktistov 2006).
- Estimation of Distribution Algorithms (Larrañaga and Lozano 2002; Lozano et al. 2006).
- Greedy randomised adaptive search procedure (Feo and Resende 1989, 1995).
- Harmony Search (Geem et al. 2001).
- Particle Swarm Optimisation (Kennedy et al. 2001; Clerc 2006; Engelbrecht 2006).
- Scatter Search (Laguna and Martí 2003).
- Variable Neighbourhood Search (Mladenovic and Hansen 1997; Hansen and Mladenovic 2001).

The previous list is not comprehensive, as there are other not as well known methods like Iterated Local Search, Threshold Accepting, Constructive Genetic Algorithms, Iterated Greedy, Dynamic Local Search, Bees Algorithm, as well as a whole myriad of hybrid methods. Additionally, there is a host of general texts on metaheuristics like Glover and Kochenberger (2003); Resende et al. (2004); Hoos et al. (2005); Ibaraki et al. (2005) and Doerner et al. (2007). As of late, the field of metaheuristics is evolving to a more holistic paradigm which is being known as 'computational intelligence' with some books already published, like those of Konar (2005); Eberhart and Shi (2007) or Engelbrecht (2007).

9.5 Other Techniques

This chapter presented some main ideas to design and to evaluate approximate algorithms for manufacturing scheduling problems. However, the large variety of such algorithms prohibits their comprehensive overview or even a comprehensive overview of the respective ideas. For instance, it should be mentioned here again that simply stopping a (possibly, but not necessarily exact) optimisation approach before its *pre-defined* termination, e.g. for time or other resource reasons, obviously defines a class of heuristics as well.

In addition, apart from the different methods reviewed in this chapter, it should be mentioned that additional heuristic solution procedures might result from automated integrated or non-integrated decision support systems, such as solution modules provided by approaches from artificial intelligence or agent-based systems. These methods will not be addressed here, but instead some further remarks on a general technique for generating approximate solutions (Lagrangean relaxation), and the combination of exact approaches with metaheuristics) are presented.

9.5.1 Lagrangean Relaxation

As already mentioned, a well-known (sometimes exact, sometimes only approximate) approach to the solution of optimisation problems is Lagrangean relaxation, i.e. the dualisation of constraints into the objective function by penalizing violations of these constraints in the objective function value using Lagrangean multipliers as unit costs of violation. Since manufacturing scheduling models usually are of combinatorial, i.e. binary or integer type, a non-zero duality gap will make Lagrangean relaxation almost ever an approximate and not an exact method in scheduling.

Apart from the fact that this approach basically can be applied in a formal way to more or less every constraint of every optimisation problem (at least as an approximate approach), in manufacturing scheduling there are settings where this approach is/can be used implicitly. Next we sketch this approach for the specific case of flowtime minimisation with due dates/deadlines of jobs. This problem could be solved in (at least) three manners:

1. The problem can be solved as a pure flowtime problem (without considering the due dates). The output would maybe yield that there is no feasible or acceptable solution because of the tight and strict deadlines.
2. The problem can be solved as a 2-objective problem including flowtime and tardiness as objectives. This approach would disclose the trade-off between flowtime and tardiness, probably by some kind of Pareto analysis (see Chap. 10).
3. The violation of due dates can also be added to flowtime in the objective function including adequate Lagrangrean multipliers which penalise the violation of due dates.

From a formal point of view, Lagrangean relaxations often are able to provide bounds on the optimal objective function value which enables the decision maker to estimate the maximum deviation of a current solution to the optimum. (However, the integrality requirements with respect to the bound interpretation have to be considered carefully, at least if mathematically valid bounds should be derived.)

9.5.2 Combination of Exact Approaches with (Meta-) Heuristics

At a first glance, it seems to be absurd to consider the combination of exact approaches with heuristics since exact approaches solve the problem under consideration exactly and no further processing is needed. However, the second glance offers at least two general perspectives of how exact approaches and heuristics can be combined in manufacturing scheduling.

On one hand, as already mentioned in Chap. 7, exactness of exact approaches have to be interpreted in a rather strict and narrow context relative to a formal problem under consideration. Almost every deviation and/or generalisation from this formal problem, usually pointing into the direction of the real-world problem to be solved and including more complexity will give reason for (heuristic) modifications and extensions of the 'exact' solution deduced from the formal problem which is to be solved by the exact procedure. We point to this perspective but will not discuss it further here since the respective heuristic approaches are usually rather specific with respect to the real-world problem under consideration.

On the other hand, while staying completely in the 'formal' sphere of manufacturing scheduling problems, the limited 'exact' solvability of many formal scheduling problems for complexity reasons implies the use of heuristics. Apart from the two 'pure' heuristic approaches, i.e. premature truncation of an exact approach, e.g. after reaching an accepted error level and/or a time limit, or exclusively referring to (constructive and/or meta-) heuristics, both pure approaches can be combined in several ways. We describe some basic types of these combined approaches.

1. The performance of a non-polynomial exact procedure will usually be increased if a good (and feasible) initial solution is available which might be obtained by a heuristic approach. On one hand, this solution gives a valid and (hopefully) good upper bound for the optimal objective function value. Obviously, the better this bound is the better the remaining potential for further optimisation can be estimated, e.g. in branch and bound approaches. On the other hand, the solution itself might be a good initial solution for starting a branch and bound approach where a feasible initial solution is required, especially in its first-depth version.
2. If an exact approach has to be stopped prematurely, some 'post optimisation' might take place. E.g., having determined one or several good solutions by a prematurely stopped exact procedure, these solutions can be used as starting point for improvement heuristics. This approach will often include deterministic or non-deterministic neighbourhood search and/or will use the best solutions determined

so far as initial population for a genetic algorithm or some other meta-heuristic approach.

3. Decomposing the overall problem into several exactly solvable subproblems, solving these subproblems exactly and finally heuristically composing the subproblems' solution into an overall solution characterises a third way of combining exact and heuristic approaches. Usually, the determination of exactly solvable subproblems is based on the determination of polynomially solvable subproblems, e.g. specific 1-machine/1-stage problems. However, also the number of jobs can be reduced to a number which allows also more complicate problems to be solved exactly. E.g., a standard 50 jobs flow shop problem (with an arbitrary objective function) might be separated into five 10-job problems. These smaller problems can be solved to optimum. Afterwards the five optimal solutions of the subproblems are combined by looking for the best among the $5! = 120$ (macro-) sequences of these 10-job subsequences. Obviously, the crucial component of this approach is the assignment of jobs to 10-job subproblems (as well as the determination of the number 10 as size of the subproblems). This job assignment might be induced by 'similar' jobs, e.g. by similarity in the processing time profile, by similar setups required etc.
4. A subset of decomposition/composition approaches represent those approaches who include aggregation/disaggregation components. In aggregation and disaggregation approaches the number of objects to be considered is reduced by aggregating objects into aggregated objects solving a problem on the aggregate level and then disaggregating the solution to a solution of the detailed formal problem under consideration. Objects to be aggregated could be jobs (into macro-jobs), machines/stages or a combination of both. The resulting aggregate problem (with less machines/stages and/or jobs) should be solvable exactly and the disaggregation step then consists of refining the aggregate result into disaggregate information for the detailed problem, maybe combined with some post-optimisation heuristic on the disaggregate level.

   A famous representative of this approach, considering machine aggregation, is the shifting bottleneck job shop heuristic, Sect. 9.2.2. Here scheduling on one machine is considered in detail while scheduling on the other machines is included only by release times (coarsely representing the lead time of a job's operations to be processed before the operation of this job on the machine under consideration may take place) and follow-up times (coarsely representing the time needed for the remaining operations of a job). The shifting bottleneck heuristic considers iteratively all machines/stages in the job shop. However, if a clear bottleneck can be identified in the system, the above consideration also can be limited only to this bottleneck machine/stage. An aggregation/disaggregation approach for flow shop scheduling is the well-known heuristic by Campbell et al. (1970). Here, the machines are aggregated into two machines, the two machine problem is solved to optimality by Johnson's algorithm and the resulting solution is adapted as an approximation for the original flow shop problem.

   Similar approaches can be derived for job aggregation/disaggregation where similar jobs are aggregated into one job. The similarity of jobs can be defined in many

ways which are usually problem-dependent. E.g., similar setup requirements or similar due dates might be approaches to substantiate the similarity of jobs.
More generally, in aggregation/disaggregation approaches, a common way of determining aggregation objects is to aggregate non-bottleneck objects (jobs and/or machines/stages) and to consider the bottleneck(s) (jobs and/or machines/stages) in detail.

5. Most of the approaches mentioned before might be applied as well iteratively, e.g. by using the solution/the schedule of the first phase as input for the second phase and vice versa.

9.6 Conclusions and Further Readings

In this chapter we have dealt with approximate algorithms. Approximate algorithms intend to reduce the complexity of a scheduling model and/or solution method. Regarding taylored approximate algorithms (i.e. heuristics), there is no template that can be used to design heuristics, so the main ideas and concepts behind heuristics have been discussed. Nevertheless, given the large variety of these methods and underlying ideas, such an exposition is far from being comprehensive.

In the last decades there has been a tremendous interest in metaheuristic techniques. While the templates in which metaheuristic methods are based are general and can be applied to most optimisation problems, the instantiation process to any scheduling problem is often not straightforward. A big advantage is that effective metaheuristics often obtain solutions that are very close to optimal sequences in most settings, which comes at a high computational cost when compared to other methodologies albeit they are often faster than exact methods.

One basic reference on approximate methods is the book of Morton and Pentico (1993). Although this book is fairly old meanwhile, it gives a structured overview of many approximate methods according to the state-of-the-art by the beginning of the 1990s. For more recent methods the interested reader is referred to the numerous references in scientific journals, may they present detailed approaches themselves or surveys on specific classes of scheduling problems. The performance of the NEH has been acknowledged by many authors, including Turner and Booth (1987); Taillard (1990); Ruiz and Maroto (2005). Furthermore, NEH is actively used as a seed sequence for metaheuristic, such as in Reeves (1995); Chen et al. (1995); Nowicki and Smutnicki (1996); Reeves and Yamada (1998); Wang and Zheng (2003); Rajendran and Ziegler (2004); Grabowski and Wodecki (2004); Ruiz and Stützle (2007), among others.

The literature on metaheuristics is overwhelming. Only listing some of the most interesting books is already a hard task. Our recommendation goes to the general texts of Glover and Kochenberger (2003); Resende et al. (2004); Hoos et al. (2005); Ibaraki et al. (2005) and Doerner et al. (2007). Basically, each type of metaheuristic has its own classical books and recent references and previous sections contain some interesting books and further reading pointers.

References

Adams, J., Balas, E., and Zawack, D. (1988). The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34(3):391–401.

Biegel, J. E. and Davern, J. J. (1990). Genetic Algorithms and Job Shop Scheduling. *Computers and Industrial Engineering*, 19(1):81–91.

Birattari, M. (2005). *The Problem of Tuning Metaheuristics as seen from a machine learning perspective*. Intelligence-Infix, Berlin.

Campbell, H. G., Dudek, R. A., and Smith, M. L. (1970). A Heuristic Algorithm for the *n* Job, *m* Machine Sequencing Problem. *Management Science*, 16(10):B-630–B-637.

Chen, C.-L., Vempati, V. S., and Aljaber, N. (1995). An application of genetic algorithms for flow shop problems. *European Journal of Operational Research*, 80(2):389–396.

Clerc, M. (2006). *Particle Swarm Optimization*. Wiley-ISTE, New York.

Dannenbring, D. G. (1977). An evaluation of flowshop sequencing heuristics. *Management Science*, 23:1174–1182.

Dasgupta, D., editor (1998). *Artificial Immune Systems and Their Applications*. Springer, New York.

Davis, L. (1985). Job Shop Scheduling with Genetic Algorithms. In Grefenstette, J. J., editor, *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, pages 136–140, Hillsdale. Lawrence Erlbaum Associates.

Davis, L., editor (1996). *Handbook of Genetic Algorithms*. International Thomson Computer Press, London.

de Castro, L. N. and Timmis, J. (2002). *Artificial Immune Systems: A New Computational Intelligence Approach*. Springer, London.

Doerner, K. F., Gendreau, M., Greistorfer, P., Gurjahr, W. J., Hartl, R. F., and Reinmann, M., editors (2007). *Metaheuristics: progress in complex systems optimization*. Operations Research/Computer Science Interfaces. Springer, New York.

Dorigo, M. and Stützle, T. (2004). *Ant Colony Optimization*. Bradford Books, USA.

Eberhart, R. C. and Shi, Y. (2007). *Computational Intelligence: Concepts to Implementations*. Morgan Kaufmann, San Francisco.

Engelbrecht, A. P. (2006). *Fundamentals of Computational Swarm Intelligence.* John Wiley & Sons, New York.

Engelbrecht, A. P. (2007). *Computational Intelligence: An Introduction.* John Wiley & Sons, New York, second edition.

Feo, T. A. and Resende, M. G. C. (1989). A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters,* 8(2):67–71.

Feo, T. A. and Resende, M. G. C. (1995). Greedy randomized adaptive search procedures. *Journal of Global Optimization,* 6:109–133.

Feoktistov, V. (2006). *Differential Evolution. In Search of Solutions.* Springer, New York.

Framinan, J. M., Leisten, R., and Rajendran, C. (2003). Different initial sequences for the heuristic of Nawaz, Enscore and Ham to minimize makespan, idletime or flowtime in the static permutation flowshop sequencing problem. *International Journal of Production Research,* 41(1):121–148.

Geem, Z. W., Kim, J. H., and Loganathan, G. V. (2001). A new heuristic optimization algorithm: Harmony search. *Simulation,* 76(2):60–68.

Glover, F. and Kochenberger, G. A., editors (2003). *Handbook of Metaheuristics.* Kluwer Academic Publishers, Dordrecht.

Glover, F. and Laguna, M. (1997). *Tabu Search.* Kluwer Academic Publishers, Dordrecht.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading.

Grabowski, J. and Wodecki, M. (2004). A very fast tabu search algorithm for the permutation flow shop problem with makespan criterion. *Computers & Operations Research,* 31(11):1891–1909.

Hansen, P. and Mladenovic, N. (2001). Variable neighborhood search: Principles and applications. *European Journal of Operational Research,* 130(3):449–467.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor.

Hoos, Holger, H. and Stützle, T. (2005). *Stochastic Local Search: Foundations and Applications.* Morgan Kaufmann, San Francisco.

Ibaraki, T., Nonobe, K., and Yagiura, M., editors (2005). *Metaheuristics: progress as real problem solvers.* Operations Research/Computer Science Interfaces. Springer, New York.

Kennedy, J., Eberhart, R. C., and Shi, Y. (2001). *Swarm Intelligence.* Academic Press, San Diego.

Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science,* 220(4598):671–680.

Konar, A. (2005). *Computational Intelligence: Principles, Techniques and Applications.* Springer, New York.

Laguna, M. and Martí, R. (2003). *Scatter search: methodology and implementations in C.* Operations Research/Computer Science Interfaces. Kluwer Academic Publishers, New York.

Larrañaga, P. and Lozano, J. A., editors (2002). *Estimation of distribution algorithms: A new tool for evolutionary computation*. Kluwer Academic Publishers, Dordrecht.

Lozano, J. A., Larrañaga, P., Inza, I. n., and Bengoetxea, E., editors (2006). *Towards a New Evolutionary Computation: Advances on Estimation of Distribution Algorithms.* Springer, New York.

Mattfeld, D. C. (1996). *Evolutionary Search and the Job Shop; Investigations on Genetic Algorithms for Production Scheduling.* Production and Logistics. Springer/Physica Verlag, Berlin.

Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, Berlin, tercera edition.

Mladenovic, N. and Hansen, P. (1997). Variable neighborhood search. *Computers & Operations Research,* 24(11):1097–1100.

Morton, T. E. and Pentico, D. W. (1993). *Heuristic Scheduling Systems With Applications to Production Systems and Project Management.* Wiley Series in Engineering & Technology Management. John Wiley & Sons, Hoboken.

Nawaz, M., Enscore, Jr, E. E., and Ham, I. (1983). A Heuristic Algorithm for the m-Machine, n-Job Flow-shop Sequencing Problem. *OMEGA, The International Journal of Management Science,* 11(1):91–95.

Nowicki, E. and Smutnicki, C. (1996). A fast tabu search algorithm for the permutation flow-shop problem. *European Journal of Operational Research,* 91(1):160–175.

Nowicki, E. and Smutnicki, C. (1998). The flow shop with parallel machines: A tabu search approach. *European Journal of Operational Research,* 106(2–3):226–253.

Ogbu, F. A. and Smith, D. K. (1990). The Application of the Simulated Annealing Algorithm to the Solution of the $n/m/C_{\max}$ Flowshop Problem. *Computers and Operations Research,* 17(3):243–253.

Osman, I. H. and Potts, C. N. (1989). Simulated Annealing for Permutation Flow-shop Scheduling. *OMEGA, The International Journal of Management Science,* 17(6):551–557.

Park, Y. B., Pegden, C., and Enscore, E. (1984). A survey and evaluation of static flowshop scheduling heuristics. *International Journal of Production Research,* 22(1):127–141.

Pinedo, M. L. (2012). *Scheduling: Theory, Algorithms, and Systems.* Springer, New York, fourth edition.

Price, K., Storn, R. M., and Lampinen, J. A. (2005). *Differential Evolution: A Practical Approach to Global Optimization.* Springer, New York.

Rajendran, C. and Ziegler, H. (2004). Ant-colony algorithms for permutation flowshopn scheduling to minimize makespan/total flowtime of jobs. *European Journal of Operational Research,* 155(2):426–438.

Reeves, C. and Yamada, T. (1998). Genetic algorithms, path relinking, and the flowshop sequencing problem. *Evolutionary Computation,* 6(1):45–60.

Reeves, C. R. (1995). A genetic algorithm for flowshop sequencing. *Computers & Operations Research,* 22(1):5–13.

Resende, M. G. C., Pinho de Sousa, J., and Viana, A., editors (2004). *Metaheuristics: computer decision-making.* Kluwer Academic Publishers, Dordrecht.

Ribas, I., Companys, R., and Tort-Martorell, X. (2010). Comparing three-step heuristics for the permutation flow shop problem. *Computers & Operations Research,* 37(12):2062–2070.

Ruiz, R. and Maroto, C. (2005). A comprehensive review and evaluation of permutation flowshop heuristics. *European Journal of Operational Research,* 165(2):479–494.

Ruiz, R., Maroto, C., and Alcaraz, J. (2006). Two new robust genetic algorithms for the flowshop scheduling problem. *OMEGA, The International Journal of Management Science,* 34(5):461–476.

Ruiz, R. and Stützle, T. (2007). A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research,* 177(3):2033–2049.

Syswerda, G. (1996). Scheduling Optimization Using Genetic Algorithms. In Davis, L., editor, *Handbook of Genetic Algorithms,* pages 332–349, London. International Thomson Computer Press.

Taillard, E. (1990). Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research,* 47(1):67–74.

Turner, S. and Booth, D. (1987). Comparison of Heuristics for Flow Shop Sequencing. *OMEGA, The International Journal of Management Science,* 15(1):75–78.

Černý, V. (1985). A thermodynamical approach to the travelling salesman problem: an efficient simulation algorithm. *Journal of Optimization Theory and Applications,* 45(1):41–51.

Wang, L. and Zheng, D. Z. (2003). An effective hybrid heuristic for flow shop scheduling. *The International Journal of Advanced Manufacturing Technology,* 21(1):38–44.

Widmer, M. and Hertz, A. (1989). A new heuristic method for the flow shop sequencing problem. *European Journal of Operational Research,* 41(2):186–193.

Woo, H.-S. and Yim, D.-S. (1998). A heuristic algorithm for mean flowtime objective in flowshop scheduling. *Computers & Operations Research,* 25(3):175–182.

Chapter 10
Multi-Objective Scheduling

10.1 Introduction

Starting with Chap. 3, where scheduling models were presented, followed by Chaps. 4 and 5 where constraints and scheduling objectives were outlined, a common assumption so far in this book has been that every scheduling problem and its corresponding model has one single objective or criterion to optimise. However, internal goals, e.g. minimising tied-up capital and the related consequences with respect to utilisation and work in process, are in conflict with other internal goals or with externally oriented goals, e.g. maximising service level with its relation to due date-oriented objectives. Therefore, it is not surprising that manufacturing scheduling models have been the subject of many publications and considerations dealing with multi-criteria aspects.

The consideration of multiple criteria greatly increases the types of models to be considered, as well as the approaches and methods to solve these models. Additionally, the assessment of the quality of the solutions obtained by these methods is far from straightforward in many cases.

In this chapter we

- discuss the multi-objective nature of many manufacturing scheduling decision problems (Sect. 10.2),
- give the main definitions and notation required for dealing with multiple objectives (Sect. 10.3),
- present the most employed multi-criteria models (Sect. 10.4),
- introduce the different approaches for addressing multi-criteria scheduling (Sect. 10.5),
- address the issue of assessing the quality of algorithms in Pareto optimisation (Sect. 10.6), and
- present interfering jobs scheduling models as a general case of multi-criteria scheduling models (Sect. 10.7).

J. M. Framinan et al., *Manufacturing Scheduling Systems*,
DOI: 10.1007/978-1-4471-6272-8_10, © Springer-Verlag London 2014

10.2 Multi-Objective Nature of Many Scheduling Problems

Strictly speaking, one could say that a multi-objective problem is no more than a poorly defined single objective problem. In other words, if one needs to optimise a given scheduling problem with several simultaneous objectives is because it was not possible to verbalise and to rationalise a single-objective function—no matter how complex this function might be. However, agreeing upon a single objective is by no means an easy task in real production scheduling settings. As it has been mentioned, real-life scheduling problems are never single objective. This might seem as a gratuitous assessment. However, a close examination of how scheduling works in practice should convince the reader that this is, in fact, the truth. Below we state several issues concerning single- versus multi-objective optimisation in real production scheduling settings:

- Well-defined objectives are hard to find in practice. Shop managers have a fair amount of intuition of what makes a good or a bad schedule, but not a precise and specific objective function. Moreover, even if a well-defined objective exists, its actual value might be of little relevance in practice. Even simple objectives like minimising the number of tardy jobs can quickly turn into a debatable measure. Some jobs are not as important as others. Therefore, being tardy for some jobs is not as critical. Trying to add weights in order to cope with this issue is only a temporary fix since weights are often not enough. Things like 'this job cannot be tardy. I have added a weight of nine and the scheduling method still results in this job being tardy' easily pop out. In other occasions, a simple phone call to a client in order to advance that the delivery date for a job or order is going to change suffices to make up for a tardy job.
- Preferences or objectives vary over time. They do depend on past and current shop performance, on the mix of pending production and on market situation. Therefore, some days, the focus might be at reducing lead times, while other days machine utilisation might be of paramount importance.
- Decisions on schedules are often taken in a committee rather than by a single individual. Agreeing on a single objective in these cases is seldom feasible. Moreover, different managers at the company will have various and often conflicting goals. Top management often desires to increase revenue. Sales force desires to meet client's due dates while plant managers care about machine utilisation, to name just a few examples. As a result, even deciding on an objective to optimise can be challenging.
- The concept of 'optimal solution' is often not well understood. Plant managers have a fair understanding of what constitutes a bad or a good schedule. However, discerning between a good and an optimal schedule is not within plant personnel capabilities and is often a moot issue. Furthermore, an optimal solution for one objective might degrade another objective more than necessary.
- Problem constraints might be relaxed in view of the performance of the schedule. For example, working shifts might be reduced by sending workers home or augmented via overtime hours. Similarly, production might be subcontracted in

order to cope with additional orders. Given this potential relaxation of problem constraints, the mere concept of optimal solution is debatable at best.
- Human schedulers are more oriented towards goals or minimum/maximum achievement levels for a set of preferences rather than towards complex mathematical formulations and precise objectives. Verbalising objectives as 'reaching an 80 % service level' is much more meaningful than saying that the total tardiness is 1,700,000 time units.
- Real settings are marred with portions of data that are unknown, imprecise, constantly changing and even wrong. Under these circumstances, focusing on improving a single criterion by a few percentage points makes little, if any, sense.

Although it turns out that, in some cases, it is desirable to consider more than one criteria when scheduling, multi-criteria scheduling (and multi-criteria decision-making, in general) is substantially more difficult than dealing with a single criterion. In the latter case, for a formal deterministic model involving the minimisation of objective f, establishing if schedule S is better than S' reduces to checking if $f(S) < f(S')$. However, this is not possible, in general, for more than one objective. If we consider the minimisation objectives f_1 and f_2, and it turns out that $f_1(S) < f_1(S')$ but $f_2(S) > f_2(S')$, then it is clear that we cannot say that S is better than S' or vice versa. Therefore, in order to advance into multi-criteria scheduling, we first need some specific definitions. These are provided in the next section.

10.3 Definitions and Notation

Let us assume that we are dealing with a scheduling model with a number H of minimisation[1] objectives, for which two solutions x_1 and x_2 exist. Let us derive the following definitions:

Strong domination: A solution x_1 strongly dominates solution x_2 ($x_1 \prec\prec x_2$) if:

$$f_h(x_1) < f_h(x_2) \quad \forall h$$

Therefore, x_1 is said to strongly dominate x_2 if x_1 is strictly better than x_2 for all objectives. This is often also defined as strict domination.

(Regular) domination: Solution x_1 (regularly) dominates solution x_2 ($x_1 \prec x_2$) if the following two conditions are simultaneously satisfied:

$$(1)\, f_h(x_1) \leq f_h(x_2) \quad \forall h$$
$$(2)\, f_h(x_1) < f_h(x_2) \text{ for at least one } h = 1, 2, \ldots, H$$

[1] This does not pose a loss of generality, since all definitions can be easily adapted. Furthermore, most scheduling objectives, as shown in Chap. 5, are of the minimisation type.

This implies that x_1 is not worse than x_2 for all objectives except for one in which it is strictly better.

Weak domination: Solution x_1 weakly dominates solution x_2 ($x_1 \preceq x_2$) if:

$$f_h(x_1) \leq f_h(x_2) \quad \forall h$$

This last definition implies that x_1 is not worse than x_2 for all objectives.

Incomparability: Solutions x_1 and x_2 are said to be incomparable, and is denoted by $x_1 \| x_2$ or by $x_2 \| x_1$ if the following two conditions are simultaneously satisfied:

$$(1)\ x_1 \text{ does not weakly dominates } x_2 (x_1 \npreceq x_2)$$
$$(2)\ x_2 \text{ does not weakly dominates } x_1 (x_2 \npreceq x_1)$$

In other words, if two solutions do not weakly dominate each other, they are said to be incomparable.

Given a set of solutions A, there might be relations between the solutions contained. More specifically, one is usually interested in the dominant solutions from the set:

Weak Pareto optimum: a solution $x \in A$ is a weak Pareto optimum if and only if there is no $x' \in A$ for which $x' \prec\prec x$. In other words, $x \in A$ is a weak Pareto optimum if no other solution in set A strongly dominates x. 'Weakly efficient solution' is an alternative name for a weak Pareto optimum.

Strict Pareto optimum: a solution $x \in A$ is a strict Pareto optimum if and only if there is no $x' \in A$ for which $x' \prec x$. In other words, $x \in A$ is a strict Pareto optimum if no other solution in set A dominates x. 'Efficient solution' is an alternative name for a strict Pareto optimum.

Pareto set: a set $A' \subseteq A$ is denoted a Pareto set if and only if it contains only and all strict Pareto optima. Since it contains only efficient solutions, this set is denoted as *efficient set*. Alternative names are Pareto front or Pareto frontier.

All previous definitions can be more clearly understood by a graphical representation. Let us consider a scheduling problem with two minimisation objectives. We depict the values of the objective functions along two axis: X-axis represents the makespan ($C_{\max}$) values and the Y-axis the total tardiness ($\sum T_j$) values. Note that every possible solution x_i to this biobjective scheduling problem can be represented by a point in this graphic. We have represented some possible solutions in Fig. 10.1.

By looking at Fig. 10.1, we can also give some further definitions:

Utopian (or ideal) solution: Is a given solution with the best possible objective function value for all objectives. Often, this ideal solution does not exist as the multiple objectives are often conflictive. It has been represented in Fig. 10.1 as solution 'U'.

Nadir solution: It is the contrary to the utopian solution. It is a solution with the worst possible objective function value for all objectives. Luckily, such solution is

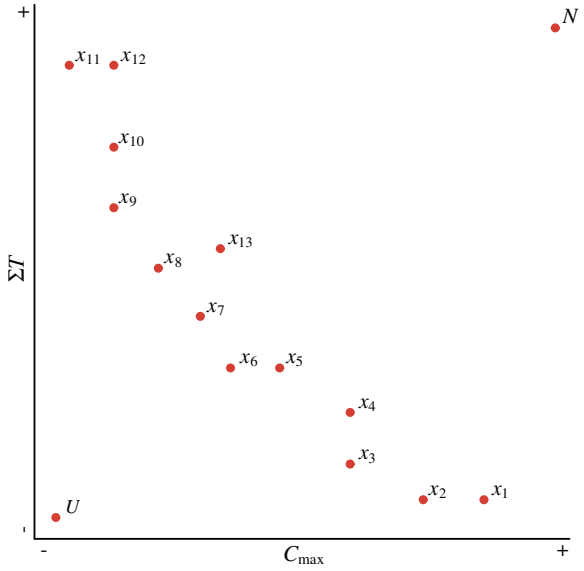

Fig. 10.1 Representation of the function objectives' values of several possible solutions

not likely to exist either. In Fig. 10.1 the nadir solution has been represented by 'N'. Solutions U and N are many times used for bounding (topologically speaking) the sets of solutions for multi-objective problems.

Following with Fig. 10.1 we have the following relations:

- Picture, for example, solutions x_1 and x_2. Both solutions have the same total tardiness value. However, solution x_2 has a lower makespan value than x_1. Therefore, we say that solution x_2 regularly dominates solution x_1 or $x_2 \prec x_1$.
- Following the previous example we also have that $x_3 \prec x_4, x_6 \prec x_5, x_9 \prec x_{10} \prec x_{12}$ and $x_{11} \prec x_{12}$.
- It can also be seen that both x_7 and x_8 have both a lower makespan and a lower total tardiness than x_{13}. Therefore, it can be stated that $x_7 \prec\prec x_{13}$ and $x_8 \prec\prec x_{13}$.
- There are some incomparable solutions in the figure. Let us pick, for example, x_1 and x_3. x_1 has a lower total tardiness value than x_3 but also it has a worse makespan value. Therefore, we cannot conclude, in a multi-objective sense, whether x_1 is better than x_3 or viceversa. We then state that x_1 and x_3 are incomparable or $x_1 \| x_3$.
- The Pareto front for the example is formed by solutions x_2, x_3, x_6, x_7, x_8, x_9, and x_{11}. This is further depicted in Fig. 10.2. Note the stepwise 'frontier'. This is indeed the case as straight lines in between Pareto solutions would imply that other solutions in between have been found.

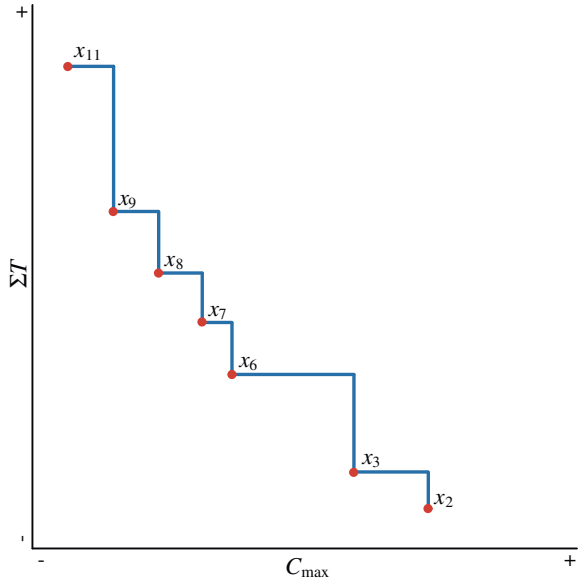

Fig. 10.2 Representation of the Pareto frontier

10.4 Multi-Objective Models

When dealing with multi-objective scheduling models, different classes of multi-objective models can be formulated depending on how the different objectives are considered in the decision problem. For instance, the decision-maker may accept some trade-off between the different objectives considered, or may be interested in that the schedule has at least a (minimum) performance with respect to one or several objectives considered.

There are many classes of models, being the most employed in manufacturing scheduling the following classes:

- Weighted combination of objectives. In this class of models, an explicit importance (weight) of each objective can be determined. As we will see, this transforms the multi-objective model into a single-objective model.
- Lexicographical optimisation. In this class of models, a ranking or importance order of the objectives exists, so they should be optimised one after another.
- ε-constraint optimisation. Only one objective is optimised at a time, while all others are kept as constraints with a known bound.
- Goal programming. In this class of models, a given goal or target value is set for each objective considered. This class of models can be seen as a multi-objective version of a feasibility model.
- Pareto optimisation. In this class of models, the goal is finding the Pareto set with respect to the objectives considered.

In Chaps. 3–5, the well-known $\alpha|\beta|\gamma$ notation was introduced. As already indicated, the third field γ covers the criterion for the manufacturing scheduling model. In order to be able to represent the different classes of multi-objective scheduling models, this notation is extended and presented in the corresponding class.

10.4.1 Weighted Combination of Objectives

The simplest approach to multi-objective scheduling problems is to combine all objective functions into a single-objective, usually by adding weights to each objective. One example is the makespan and total tardiness minimisation as follows: $\alpha C_{\max} + (1 - \alpha) \sum T_j$, where $0 \leq \alpha \leq 1$. Note that this is a convex linear combination, a reason for which this class of models is also known as Linear Convex Combination (LCC).

For three objectives F_1, F_2 and F_3, one can write something like $\mathrm{a}F_1 + \mathrm{b}F_2 + \mathrm{c}F_3$ where $0 \leq \mathrm{a}, \mathrm{b}, \mathrm{c} \leq 1$ and $\mathrm{a} + \mathrm{b} + \mathrm{c} = 1$. The advantage of such an approach is that at the end one has effectively one single-objective (a weighted sum) and therefore, existing single-objective approaches can be used with little adaptation. Indeed, some authors do not even consider this method a 'true' multi-objective approach.

Multi-criteria scheduling models of the weighted combination of objectives class are denoted as follows:

$\gamma = F_l(F_1, F_2, \ldots, F_H)$: weighted combination of objectives $F_1, F_2, \ldots, F_H$.

Therefore, $1|d_j|F_l(\sum T, \sum E)$ indicates a one machine model where each job has a due date and the objective is to minimise a LCC of the total tardiness and the total earliness.

10.4.2 Lexicographical Optimisation

Lexicographical optimisation basically entails a definition of an order among the considered objectives. Once this order is determined, the scheduling problem is optimised as regard the first objective, with no trade-offs between all other objectives. Once a solution is obtained, the second objective is optimised subject to no deterioration in the first, i.e. the second objective is optimised and a constraint is imposed so that the objective function value for the first objective is not worsened. Lexicographical optimisation is 'easy' in the sense that one has as many optimisation runs as objectives to consider. In the context of mathematical programming the first mathematical model to optimise is the following:

$$\min F_1$$

If we denote by f_1 the value obtained by the first minimisation objective, the second mathematical model goes as follows:

$$\begin{aligned} &\min F_2 \\ &s.t. \\ &F_1 \leq f_1 \end{aligned}$$

The third run would be:

$$\begin{aligned} &\min F_3 \\ &s.t. \\ &F_1 \leq f_1 \\ &F_2 \leq f_2 \end{aligned}$$

The procedure continues until the last objective is optimised. The final solution is the lexicographic optimal solution for all considered objectives in order.

Multi-criteria scheduling models of the lexicographical class are denoted as follows:

$\gamma = Lex(F_1, F_2, \ldots, F_H)$: lexicographical optimisation of objectives $F_1, F_2, \ldots, F_H$. The order indicates that F_1 is optimised first, then F_2 is optimised subject to no deterioration in F_1 and so on until F_H.

Note that the order of the objectives in the γ field in these models is very important, as different orders define different models.

According to this notation, $Fm|prmu, d_j|Lex(\max T_j, C_{max})$ indicates a model of a permutation flowshop with due dates where the maximum tardiness is minimised first and then, the makespan is minimised as long as there is no deterioration in the maximum tardiness value found first. Obviously, $Fm|prmu, d_j|Lex(C_{max}, \max T_j)$ constitutes a different model.

10.4.3 ε-Constraint Optimisation

The ε-constraint is another interesting approach in which only one objective is optimised at a time, while all others are kept as constraints with a known bound on their minimum or maximum possible values. In other words, it optimises one objective while maintaining maximum or minimum acceptable levels for all others. These levels are known as the ε-values or ε-bounds for each objective. These values are hard to set appropriately and surely depend on the combination of objectives and on the scheduling problem at hand. In general, large (or small) values for the ε-bounds allow for larger feasible regions when optimising the chosen objective and therefore, better objective values. However, this usually sacrifices all other ε-bounded criteria. Equally important is the choice of which objective to optimise first. Some of these disadvantages are clearly onset by the fact that in this method a single objective is optimised at each run, needing only an existing effective single-objective algorithm.

An example with three minimisation objectives is given next:

$$\begin{aligned} &\min F_1 \\ &s.t. \\ &F_2 \leq \varepsilon_2 \\ &F_3 \leq \varepsilon_3 \end{aligned}$$

Multi-criteria scheduling models of the ε-constraint class are denoted as follows: $\gamma = \varepsilon(F_1/F_2, F_3, \ldots, F_H)$: F_1 is optimised subject to some lower or upper bounds in the remaining objectives.

Again, the order of the objectives is important: While $1|d_j|\varepsilon(\sum C_j/\max T_j)$ defines a one machine model where the objective is to minimise the total completion time subject to an upper bound on the maximum tardiness, $1|d_j|\varepsilon(\max T_j/\sum C_j)$ seeks to minimise the maximum tardiness subject to an upper bound on the total completion time. Note, however, that the importance of the order only affects to the objective that is minimised, and not to the rest, i.e. $1|d_j|\varepsilon(\sum C_j/\max T_j, \max E_j)$ is the same model as $1|d_j|\varepsilon(\sum C_j/\max E_j, \max T_j)$.

10.4.4 Goal Programming

Goal programming (sometimes referred to as goal-attainment approach) is an interesting methodology, similar, but intrinsically different to the previous lexicographic or ε-constraint approaches. It can be viewed as a generalisation where each objective is associated with a given goal or target value to be achieved. A new objective function—referred to as achievement function—is constructed where the deviations from these set of target values are minimised. Note that it is accepted that the decision-maker is going to assume that some sacrifices will be necessary as achieving all goals will not be possible. From the perspective of mathematical programming, an example with one minimisation objective (F_1), another maximisation objective (F_2) and an equality objective (F_3) is given:

$$\begin{aligned} &\min E_1 + D_2 + E_3 + D_3 \\ &s.t. \\ &F_1 = G_1 + E_1 \\ &F_2 = G_2 - D_2 \\ &F_3 = G_3 + E_3 - D_3 \end{aligned}$$

Note that G_1, G_2 and G_3 are the target values of objectives F_1, F_2 and F_3, respectively. Similarly, the variables 'D' and 'E' modelise the defect and excess values below or above the desired target values, respectively. Therefore, by minimising the sum of $E_1 + D_2 + E_3 + D_3$ we ensure that all three objectives are as close as possible to their desired goals.

Goal programming shares the same dimensionality problems with the weighted combination of objectives. In the previous example, the objective min $E_1+D_2+E_3+D_3$ assumes that all deviations are measured in the same units, or alternatively, the orders of magnitude in which each objective is measured are comparable. If this is not the case, the same normalisation problems appear. There are many extensions of goal programming, like when weights are added in order to penalise some objectives in favour of others.

Multi-criteria scheduling models of the goal programming class are denoted as follows:

$\gamma = GP(F_1, F_2, \ldots, F_H)$: Goal Programming optimisation of objectives F_1 to F_H.

Here, the order of the objectives is not relevant for defining the model. According to this notation, $F2|prmu, d_j|GP(C_{\max}, \sum U_j)$ indicates a two-machine permutation flow shop model with due dates and the objectives of finding schedules for which their makespan and number of tardy jobs are below given bounds.

10.4.4.1 Pareto Optimisation

Pareto optimisation models provide the decision-maker with the Pareto front. Therefore, not only one solution, but many, are given to the decision-maker.

We have already stated the many drawbacks of the Pareto optimisation approach. First and foremost, obtaining a good Pareto set approximation is extremely complicated. Note that many scheduling models with a single-objective are already of the NP-hard class. Therefore, obtaining many Pareto solutions is usually hard for most scheduling settings. Second, the cardinality of the Pareto front might be extremely large, making the choice of the solution to be implemented in practice hard.

Multi-criteria scheduling models of the Pareto optimisation class are denoted as follows:

$\gamma = \#(F_1, F_2, F_3, \ldots, F_H)$: Pareto optimisation of objectives F_1 to F_H.

Note that in these models, the order of the objectives is not relevant. According to the notation, $Fm||\#(C_{max}, \sum C_j)$ denotes a model of a m-machine flowshop where the goal is finding the Pareto set for the makespan and the total completion time.

10.5 Multi-Objective Methods

In this section, we discuss how multi-criteria scheduling decision problems can be addressed. We confine our discussion to the formal sphere (see Sect. 1.5), as issues dealing with transferring the real-world problem to a formal model are roughly the same as in problems with a single criterion. We first discuss several approaches to address the problems, and then give some insights on the algorithms that can be employed.

10.5.1 Approaches to Multi-Objective Models

Approaches to multi-criteria problems can be broadly classified into three categories, which mainly depend on when and how the user intervenes:

- 'A priori' approaches. There is user intervention before the optimisation run. This information is in the form of user preferences.
- 'Interactive' optimisation methods. The user intervention occurs during the optimisation run.
- 'A posteriori' methods. The user is presented with a complete as possible Pareto front. The user selects the desired trade-off solution to be implemented in practice from the front.

In the 'a priori' category, the user must provide some information in order to guide the search. This information often entails objective preferences or weights. The 'a priori' optimisation methods often need a single optimisation run and usually one single solution is obtained as a result. This single solution already considers the information and preferences given by the user. A priori multi-objective methods are very interesting as much of the work and results obtained in the single-objective scheduling literature can be reused. The biggest drawback is that the 'a priori' information must be precise and must reflect the preferences of the user accurately. Many times it is not possible to provide this information without any knowledge of the resulting sequences and solutions. A possible workaround to this problem is to enter in a loop in which the user provides information, a solution is obtained and later analysed. Should this solution fail to address the user's preferences, the 'a priori' information can be updated accordingly and a new solution can be obtained.

Conversely, in the 'a posteriori' approach, the user is presented with the Pareto front (or at least an approximation of such Pareto front). The user can then examine as many solutions as desired with the aid of graphical tools so as to select the solution to be implemented in practice. This is an ideal approach as no preferences or any a priori information must be provided. However, this approach is not exempt of problems. First, obtaining an approximation of the Pareto front might be computationally costly and/or challenging. Let us not forget that any a posteriori multi-objective optimiser has to obtain not only one but also many dominant solutions that sufficiently cover the objective space. Due to these problems, most a posteriori optimisation methods are often limited to two simultaneous objectives. Second, there is a lingering problem of over information. Given a load of dominant solutions, the user can easily become overwhelmed by the quantity of information. However, there are some techniques that can be used to cluster solutions in the objective space so to present the user a given number of selected solutions.

Lastly, the interactive approaches seek to amalgamate the advantages of the two previous categories. With some limited information, an optimisation run is carried out and a solution is presented to the user. After examining the solution, the user provides further information, like where to go, what to avoid or what to enforce. A second optimisation is run and another solution is presented to the user. The process

goes on as much as the user desires or until he or she finds an acceptable solution. With proper training, very good solutions (albeit usually not probably Pareto optimal) can be obtained. This methodology allows to include advanced human knowledge in the process. Surprisingly, not many interactive approaches have been proposed in the multi-objective scheduling literature.

The different classes of models discussed in Sect. 10.4 can be analysed under the light of the approaches in which these models may be embedded:

- In principle, LCC models fall within the *a priori* approach. Namely, the weights of the different objectives must be given to formulate the model. However, these models can be embedded into an interactive approach where, at each iteration, the weights are changed according to the opinion of the decision-maker in order to direct the search towards new solutions.
- Lexicographical optimisation requires very little data, but it should be still considered as an 'a priori' technique since the user must provide, in the very least, the order of the objectives to optimise.
- ε-constraint models are often regarded as adequate for *a priori* approaches. However, many authors also pinpoint the need to run the method several times when adjusting the ε-bounds, which effectively turns the technique into an interactive approach. Under specific scenarios and following certain—actually, strict—conditions (T'Kindt and Billaut 2006) the ε-constraint method can generate the Pareto set.
- Goal Programming is clearly an 'a priori' method since aspiration levels or goals for each objective have to be set.
- Pareto optimisation is the typical example of an 'a posteriori' approach. Once the decision-maker is confronted with a set of dominant solutions, he or she knows that each given solution is a non-dominated solution and that improving one objective always entails a deterioration of the other objectives and in such a case, the closest non-dominated solution is available in the given Pareto front approximation. The 'a posteriori' decision is then reduced to choosing one of the non-dominated solutions to be implemented in practice.

Recall from Sect. 7.3.3 that it is possible to establish a hierarchy of the computational complexity of different scheduling problems and thus construct reductions trees such as the ones in Figs. 7.1, 7.2 and 7.3. Clearly, the two first reduction trees, which corresponding to layout and constraints, are valid in the multi-criteria context. With respect to the third reduction tree, it is valid for the different objectives within the same model. In addition, another reduction tree with respect to the models can be devised.

On one hand, it is clear that multi-criteria scheduling models are computationally more complex than their single-criterion counterpart, therefore any model $\alpha|\beta|C$ in a single-criterion model can be reduced to model $\alpha|\beta|\varepsilon(C/\ldots)$ by considering as upper bounds of the rest of the objectives a very large number. Also quite obviously, the solution obtained by any $\alpha|\beta|Lex(C,\ldots)$ model serves to obtain the solution for the single-criterion model.

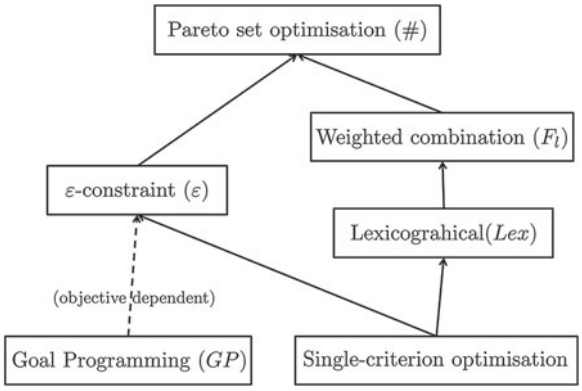

Fig. 10.3 Complexity hierarchy of multicriteria models

On the other hand, it is possible to establish a hierarchy among different classes of multi-criteria problems. More specifically, the model $\alpha|\beta|Fl(F_1, \ldots, F_H)$ reduces to $\alpha|\beta|\#(F_1, \ldots, F_H)$, as the Pareto set of the latter contains the solution given by the former, which is indeed one particular trade-off among the objectives. A similar reasoning can be done to see that $\alpha|\beta|\varepsilon(F_1, F_2, \ldots, F_H)$ reduces to $\alpha|\beta|\#(F_1, \ldots, F_H)$.

Other reductions are far from being straightforward and, in some cases, depend on the objectives considered. In Fig. 10.3 we give the reduction tree for multi-criteria models.

10.5.2 Algorithms for Multi-Criteria Scheduling Models

Given the host of different multi-objective techniques, the existing methods and algorithms for solving multi-objective scheduling problems are incredibly numerous and, in most cases, extremely model- and method-specific. Discussing them or even classifying them is beyond the scope of the book, and we refer the reader to some specific sources in Sect. 10.8. In what follows, we simply refer to one specific type of models (Pareto optimisation models) and algorithms (metaheuristics). The reason for picking Pareto optimisation models is, as we have seen, the most general approach, in the sense that some other approaches reduce to these models. Analogously, metaheuristic are chosen as they are the less model-specific type of algorithms and therefore, their study will be more valuable for a reader approaching this field for the first time.

10.5.2.1 Approximate Multi-Criteria Metaheuristics

Among the different types and variants of metaheuristics discussed in Sect. 9.4, Genetic Algorithms seems to be particularly well-suited for Pareto optimisation

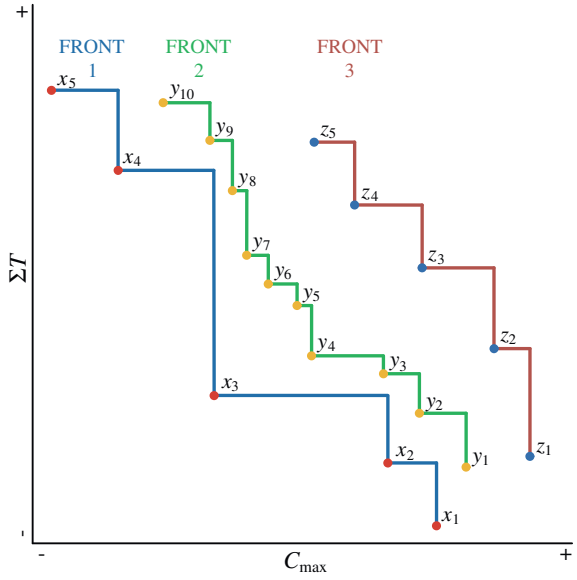

Fig. 10.4 An example of a set of solutions and the non-dominated sorting procedure

models since they are population-based approaches and keep a set of solutions. The efficient solutions within this set can be selected so this subset yields an *approximation* of the Pareto front. As a result, GAs are a natural choice for multi-objective problems.

Let us illustrate the main features of these algorithms by describing the well-known Non-dominated Sorting Genetic Algorithm (NSGA) by Srinivas and Deb (1994) and its subsequent version NSGA-II by Deb (2002). These two versions of GA have attracted a lot of interest from researchers and it is considered a benchmark algorithm for Pareto optimisation models.

NSGA basically differs from a simple GA only in the selection operator. As the name of the algorithm implies, there is a non-dominated sorting procedure (*NDS*) that is applied to the current population at each generation. Randomly selecting any individual from the population to do crossover and mutation is generally a bad idea in genetic algorithms. In multi-objective optimisation, a selection bias should be given to non-dominated solutions. *NDS* iteratively divides the entire population into different sets or fronts. The first front is formed by the non-dominated individuals in the population. Then, the individuals in the first front are removed from the population and, from the remaining individuals, the non-dominated ones are extracted again to constitute the second front. The procedure continues until all individuals have been assigned a front. An example of this procedure is graphically shown in Fig. 10.4.

As we can see from Fig. 10.4, the population is formed by 20 individuals. Individuals $x_1, x_2, \ldots, x_5$ form the first front (denoted in the figure as FRONT 1) as they are non-dominated by all others. If we remove these 5 individuals, the remaining

population of 15 individuals is dominated by solutions $y_1, y_2, \ldots, y_{10}$, which would constitute the second front (FRONT 2). Repeating the procedure would yield the third front (FRONT 3) with solutions $z_1, z_2, \ldots, z_5$.

Although FRONT 2 and FRONT 3 are not useful regarding to the final result that should be given to the decision-maker, they serve to maintain the diversity of solutions in the different iterations of the algorithm, which helps to provide the decision-maker with a diverse approximation of the Pareto front.

In Fig. 10.4, we can see how there are large gaps in the objective space between solutions x_2 and x_3 and between solutions x_3 and x_4 from FRONT 1. Actually, FRONT 1 is rather small with only five non-dominated solutions. Comparatively, FRONT 2 has twice as many solutions and, although their solutions are dominated by those in FRONT 2, they are more evenly spread out.

As mentioned before, NSGA keeps all fronts in the population since, in future iterations, it might be possible to obtain non-dominated solutions from, let us say, solutions y_5, y_6, y_7 or y_8 in FRONT 2 so to fill in the gap between solutions x_3 and x_4 in FRONT 1, to name a simple example.

The *NDS* first assigns the same large dummy fitness value to all solutions in FRONT 1. Then, a lower dummy fitness value is assigned to all solutions in FRONT 2, and so on. In order for the GA to distinguish between solutions in the same front as far as the fitness value is concerned, a 'sharing' operator is also performed. Basically, the fitness value of each individual is modified by a factor that is calculated according to the number of individuals crowding a portion of the objective space. A sharing parameter σ_{share} is used in this case. Therefore, solutions that have many other 'surrounding' individuals are penalised whereas isolated solutions in the objective space are left with their fitness values intact. Following the example given in Fig. 10.4, solution x_3 is far away from all other solutions in the first Pareto front and will be left untouched. Conversely, solution y_6 is occupying a crowded region in the second Pareto front and its fitness value would be reduced. Clearly, this mechanisms seeks to reinforce the diversity of solutions.

All other components on the NSGA method of Srinivas and Deb (1994) are basically identical to those found on GA for regular single objective problems. NSGA uses a stochastic remainder proportionate selection using the dummy fitness value returned by the *NDS* procedure and the sharing mechanism. Since NSGA was originally proposed for continuous optimisation, a binary representation and thus, binary mutation and crossover operators are used. A pseudo-algorithm listing of the Non-dominated Sorting procedure is given in Algorithm 1.

NSGA, although profusely used in both theory and in practice, suffered from some shortcomings:

- As it happens in genetic algorithms for single-objective optimisation, degradation of solutions often occurs in NSGA, especially in the later stages of the algorithm. NSGA enforces selection of non-dominant solutions in less crowded areas of the first Pareto front. These selected solutions cross and mutate and result in new solutions (offspring) that are not guaranteed to be better. Therefore, often, important genetic information is lost. In single- objective genetic algorithms, this is solved by

Algorithm 1: Non-dominated Sorting Genetic Algorithm of Srinivas and Deb (1994).

```
Input: instance data
Output: Population Pop of solutions
begin
    Initialise a population Pop of P_size individuals;
    Apply Non-Dominated Sorting;
    Apply Sharing at each obtained front;
    while Stopping criterion is not satisfied do
        Select individuals from Pop according to dummy fitness values;
        Cross individuals;
        Mutate individuals;
        Evaluate individuals;
        Include individuals in the new population;
        if new population contains Pop individuals then
            Copy new population into old population;
            Apply Non-Dominated Sorting;
            Apply Sharing at each obtained front;
    return Pop, first Pareto front
end
```

using elitism or other ways of preserving the good individuals from one generation to another.

- *NDS* is actually a computationally demanding procedure. Its computational complexity is $O(MPop^3)$, where M is the number of objectives and *Pop* is the size of the population. Recall that this procedure has to be applied at each generation of the NSGA method.
- Setting the sharing parameter σ_{share} has proven to be far from trivial and problem dependent.

All these three problems motivated the design of NSGA-II, an improved version of NSGA. NSGA-II deals with the three aforementioned caveats of the NSGA algorithm. NSGA-II includes elitism since the old and new populations are combined at the end of each generation to form a new augmented population of size $2 \cdot Pop$. A more effective Fast Non-Dominated Sorting procedure (*FNDS*) is applied. This operator is used in order to select the best fronts for the final population that is again of size *Pop*. Since all previous individuals from the old generation are considered, elitism is preserved. *FNDS* has a lower computational complexity of $O(MPop^2)$. A rank value is assigned to each individual of the population according to the front they are in. Two additional fast operators, the Crowding Distance calculation and assignment and the Crowded-Distance Sorting Operator are employed to do without the σ_{share} and to achieve fitness sharing. NSGA-II performs largely better than NSGA and than most well known multi-objective approaches. For many continuous optimisation problems, it actually achieves state-of-the-art results. Given *FNDS*, the implementation of NSGA is not difficult. All these features are key for the success of the NSGA-II in the literature.

Next, we show a metaheuristic that is an adaptation of a single-objective metaheuristic that, in contrast to GA, it employs a single solution. As we will see, the adaptation is less straightforward and special care has to be taken in the design of the metaheuristic in order to achieve a correct diversity of solutions so the algorithm does not get stuck.

The presented method is referred to as Restarted Iterated Pareto Greedy or RIPG Minella et al. (2011). In order to understand RIPG, some basic notions about the Iterated Greedy (IG) method of Ruiz and Stützle (2007) should be given. IG was originally proposed for the m-machine permutation flow shop problem with makespan criterion or $F|prmu|C_{\max}$. It basically consists of:

- destruction: d elements of a solution are iteratively removed.
- reconstruction: the d removed elements are reincorporated to the solution by means of a greedy phase.
- improvement: the solution obtained after the reconstruction is improved by means of a local search procedure.

A simulated annealing-like acceptance criterion is applied so as to temporarily accept worse solutions as incumbent ones. IG is a very simple local search-based method and it has been shown to produce state-of-the-art results for the permutation flow shop problem and makespan criterion.

The RIPG algorithm is a Pareto optimisation extension of the IG, including the following differences:

- Use of a population of solutions. While the standard IG works with only one incumbent solution and a best solution found so far, RIPG handles a population (of variable size) of non-dominated solutions as a working set. In order to provide an initial population, constructive heuristics for each one of the objectives considered can be used. Note that there is no guarantee to be non-dominated, and a greedy phase of IG is applied to these solutions in order to obtain an initial set of non-dominated solutions.
- Selection operator. Since the main logic of the IG, namely, the destruction/reconstruction, is still carried out over a single-selected solution from the working set, a selection operator is needed. RIPG employs a modified version of the Crowding Distance operator of NSGA-II where there is a selection counter keeping track of the number of times that a solution has been already selected in order to add diversity.
- Reconstruction. In RIPG, instead of fully reconstructing the selected solution after the destruction, a whole set of non-dominated solutions is built by inserting each removed job into a population of partial solutions. After the greedy phase, the working set is updated with the recently created set of non-dominated solutions from the reconstruction procedure. After each upgrade of the working set, dominated solutions are removed.
- Local search. The local search phase consists in randomly choosing one element from the selected solution, removing and reinserting it into a number of adjacent

positions (which can be seen as a curtailed local search). Furthermore, this procedure is repeated as many times as the number of times the solution has been previously selected (selection counter). This way, a deeper local search is carried out to stronger local optimal solutions.
- Restarting. In order to avoid that the method tends to get stuck, a restart operator is proposed. This operator consists in archiving the current working set, and then creating a new one with randomly generated solutions. Note that random generation of the new working set is both fast and simple. The restart mechanism is triggered after a number of iterations without changes in the cardinality of the working set.

With all previous explanations, we can see a sketch outlining of the RIPG method in Algorithm 2.

Algorithm 2: Restarted Iterated Pareto Greedy (RIPG) Algorithm of Minella et al. (2011).

```
Input: instance data
Output: Archive of non-dominated solutions
begin
    Obtain 2 · M good initial solutions applying two effective heuristics to each objective;
    Apply the Greedy Phase to each 2 · M initial solution. Generate 2 · M non-dominated sets
    of solutions;
    Include all 2 · M non-dominated sets of solutions in the working set;
    Remove dominated solutions from the working set;
    Update Archive of non-dominated solutions with the working set;
    while Stopping criterion is not satisfied do
        Select solution from the working set (Modified Crowding Selection Operator);
        Greedy Phase (destruction/reconstruction). Generate set of non-dominated solutions;
        Include set of non-dominated solutions from the Greedy Phase in the working set;
        Remove dominated solutions from the working set;
        Select solution from the working set (Modified Crowding Selection Operator);
        Local Search Phase. Generate set of non-dominated solutions;
        Include set of non-dominated solutions from the Local Search Phase in the working
        set;
        Remove dominated solutions from the working set;
        if Cardinality of the working set unchanged for 2 · n iterations then
            Update Archive of non-dominated solutions with the working set;
            Randomly generate a new working set;
    return Archive of non-dominated solutions
end
```

10.6 Comparison of Pareto Optimisation Algorithms

We already outlined in Sect. 10.2 that comparing two sets of solutions, coming from different Pareto optimisation algorithms is not an easy task, at least as compared to the evaluation of the performance of algorithms for the single-criterion case. As we have

seen, the result of a Pareto optimisation (approximate) algorithm is (an approximation of) the Pareto front. This approximation might potentially have a large number of solutions. A significant added complexity is that, as we have seen, many existing Pareto optimisation algorithms include stochastic elements, meaning that each run of these algorithms will produce, in general, different Pareto front approximations. To these problems, those also present in the single criterion case persists here.

Although there are more aspects of interest when comparing two Pareto optimisation algorithms (computation effort, among them), here we restrict to two aspects:

- Quality of the solutions offered to the decision-maker. Clearly, the decision-maker is interested in solutions yielding, in general, good values with respect to the objective functions considered.
- Quantity and diversity of these solutions. Since Pareto optimisation falls within the *a posteriori* approach, the decision-maker may consider a trade-off among the different objectives, although he/she does not know in advance such specific trade-off. Therefore, offering him/her a higher number of options (i.e. trade-offs) that are different (i.e. represent different specific balances among objectives) may be of great interest.

Regarding the quality of the solutions offered to the decision-maker, the only case in which the comparison of the outcomes of two Pareto optimisation algorithms is more or less straightforward is when the approximation of the Pareto front of the first algorithm (set A) completely dominates that of the second (set B), i.e. all and each solution in B is dominated by at least one solution in A. We can see an example in the previous Fig. 10.4. In that figure, we can assume that the FRONT 1 comes from a given algorithm A and that FRONT 2 comes from a second algorithm B. It is clear—again from the viewpoint of the solutions to be offered to the decision-maker—that algorithm A is better than algorithm B, as all solutions in FRONT 2 are dominated by at least one solution in FRONT 1.

However, as we have already noted, the quantity and diversity of the solutions offered to the decision-maker is also important. From this perspective, FRONT 2 doubles the number of solutions than FRONT 1. Furthermore, there are large gaps in the objective space between solutions x_2 and x_3 and between solutions x_3 and x_4 in FRONT 1.

Both aspects (quality and quantity/diversity) have to be balanced. For instance, while it is clear that solution y_4 from FRONT 2 is dominated by solution x_4 from FRONT 1, this dominance might be just by a few percentage points of improvement at each objective and therefore of little practical relevance for the decision-maker. He/she might be very well interested in a set with solutions of lower quality as long as it has more solutions and more evenly spread out so to have a richer basis for decision-making.

As we can see, assessing performance in multi-objective optimisation is a very complex issue in its own right. There are actually two interrelated problems involved when assessing the performance of multi-objective scheduling methods:

1. Measuring the quality of the Pareto front approximations given by a certain algorithm 'A'.
2. Comparing the out-performance of one algorithm 'A' over another algorithm 'B'.

Both issues can be dealt with the same techniques, which fall within one of the following three approaches:

- Performance measures. The set produced by an algorithm is summarised by a number indicating its quality/quantity/diversity.
- Dominance ranking. It is possible to rank a given algorithm over another based on the number of times that the resulting sets dominate each other.
- Empirical Attainment Functions (EAF). Attainment functions (AF) provide, in the objective space, the relative frequency that each region or part of the objectives space is attained by the set of solutions given by an algorithm.

Each of these three approaches present a number of advantages and disadvantages. These are discussed in the following sections.

10.6.1 Performance Measures

In order to compare two approximations of Pareto fronts, one option is to first represent each set by a single figure (*performance measure*) representing its performance, and then compare these figures. In order to provide some examples of these performance measures, we present some (rather simple and, in some cases, potentially misleading) performance measures employed in scheduling. We can make a coarse distinction between *generic* performance measures and *pairwise* performance measures. The first group of measures use a reference set S^* of solutions constituted either by the Pareto set (if these can be obtained) or a set of non-dominated solutions. The performance of a given algorithm is then measured in terms of the quality of the solutions obtained by this algorithm with respect to the solutions in S^*. Several indicators for the quality of these solutions have been proposed, being perhaps the simplest the following:

- a the number of solutions found by the algorithm,
- b^* the number of solutions in S^* found by each algorithm,
- b^*/a the percentage of non-dominated solutions (the quotient of the two first indicators).

Note that the last two measures can be also employed for comparing the performance of one algorithm relative to another (pairwise measures). In addition, other measures can be recorded, such as b the number of non-dominated solutions of each algorithm after the two algorithms are compared with each other, and b/a the percentage of non-dominated solutions (the quotient between b and a).

Note that there is an intrinsic loss of information when using performance measures as it is obviously not the same comparing two Pareto fronts with (possibly) a large number of solutions each one than just comparing two performance

measures. In addition, not all metrics—including some of those usually employed in multi-objective scheduling—are so-called *Pareto-compliant*, i.e. in some cases, a non-Pareto-compliant performance measure can assign a better value to a set B than to set A even if solutions in A dominate at least one solution in B. Again, this result highlights the fact that comparing Pareto fronts it not an easy task at all.

Some Pareto-compliant performance measures are the hypervolume (I_H) and the binary/unary Epsilon (I_ε^1) indicators. For the case of two objectives, the hypervolume indicator I_H measures the area covered by the approximated Pareto front given by an algorithm. A reference point (usually a solution with bad objective functions' values) is used for the two objectives in order to bound this area. A greater value of I_H indicates both a better convergence to as well as a good coverage of the Pareto front. Unfortunately, calculating the hypervolume can be costly. In order to have similar measures over a large benchmark, normalised and scaled variants of I_H are often employed.

The binary epsilon indicator I_ε is calculated as follows: Given two Pareto optimisation algorithms A and B, each one producing a set S_A and S_B, respectively, the indicator is defined as:

$$I_\varepsilon(A, B) = \max_{x_B} \min_{x_A} \max_{1 \le h \le H} \frac{f_j(x_A)}{f_j(x_B)}$$

where x_A and x_B are each of the solutions in S_A and S_B, respectively.

As we can see, I_ε needs to be calculated for all possible pairs of compared algorithms, something that might be impractical. An alternative unary I_ε^1 version can be proposed, where the Pareto approximation set S_B is substituted by the best-known non-dominated set or by the Pareto front if this is known. Furthermore, the interpretation of this unary version is easier since it tells us how much worse (ε) an approximation set is with respect to the best case. Therefore 'ε' gives us a direct performance measure.

An additional problem of these indicators is that in manufacturing scheduling, and particularly for due date-based criteria, the objective function might be zero. Therefore, operations of translation and/or normalisation have to be carried out.

The comparison of algorithms by means of performance measures is severely limited since a large amount of information is lost when calculating the performance measure. Picture for example the hypervolume indicator I_H. Clearly, a larger hypervolume indicates that an algorithm 'covers' a larger region of the objective space. Another algorithm might have a lower hypervolume but possibly a much more concentrated and better performance for one of the two objectives. Hence, although Pareto-compliant performance measures indicate whether an approximated Pareto front dominates or is dominated by another one, there is no way to infer in which part of the objective space each algorithm performs better. The EAF is a possible answer for this issue.

10.6.2 Dominance Ranking

Dominance ranking is not really a performance measure as there is no loss of information. It can be best observed when comparing one algorithm against another. By doing so, the number of times the solutions given by the first algorithm strongly, regularly or weakly dominate those given by the second gives a direct picture of the performance assessment among the two. However, this approach is limited because it has to be applied for each run on each instance and involves only two algorithms. Comparing many stochastic algorithms on large-sized test instances becomes quickly impractical.

10.6.3 Empirical Attainment Functions

EAF basically depict the probability for an algorithm to dominate a given point of the objective space in a single run. As many multi-objective algorithms are stochastic, different approximations to the Pareto front might be obtained in different runs even for the same instance. EAFs use colour gradients to show the relative number of times that each region of the objective space is dominated. Different from the previous performance measures, EAFs do not condense information and the behaviour over the whole Pareto front can be observed.

More formally, AFs are defined as follows: Let SS be a set of non-dominated solutions for a scheduling problem. $AF(x)$ the Attaintment Function of a solution x is defined as $AF(x) = P(\exists s \in SS\colon s \preceq x)$ which describes the probability for a given Pareto optimisation algorithm of producing, in a single run, at least one solution that weakly dominates x.

In the case of stochastic algorithms, it is not possible to express this function in a closed form but one can approximate it empirically using the outcomes of several runs. This approximation is referred to as EAF. EAFs graphically show, for a certain instance, which area is more likely to be dominated by a given algorithm. A plot is given in Fig. 10.5 taken from Minella et al. (2011) for the application of the algorithm MOSAII$_M$ of Varadharajan and Rajendran (2005) for the $Fm|prmu|\#(C_{\max}, T_{\max})$ model.

As we can see, the MOSAII$_M$ algorithm manages at obtaining a good set of solutions which is evenly spread out. The solid blue areas indicate that those parts of the objective space are dominated by solutions obtained by MOSAII$_M$ 100 % of the times. White solid areas indicate the opposite. We also see how the transition from solid blue to solid white is quick, which means that MOSAII$_M$ is robust and that obtains fairly similar Pareto fronts at each run.

EAFs show the result, over many different runs, of an algorithm over one instance. In order to compare two algorithms, Differential EAF or Diff-EAF can be employed. In short, Diff-EAF basically subtract two EAFs generated by two different algorithms for the same instance and puts the results in the same chart. By analysing Diff-EAF images, one can see in which part of the solution space an algorithm is better than the other.

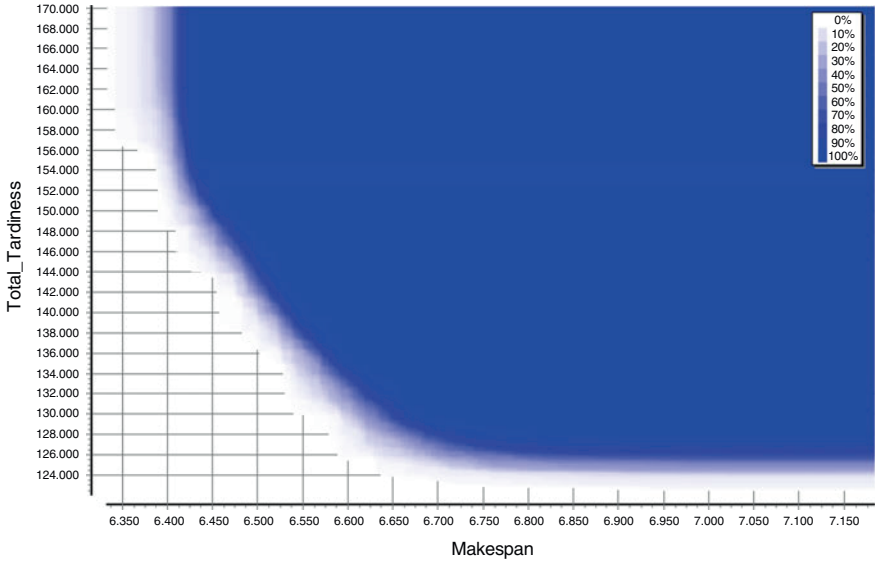

Fig. 10.5 Empirical attainment function after 100 runs of the algorithm MOSAII$_M$ of Varadharajan and Rajendran (2005) (modified by Minella et al. 2008) over instance Ta081 of Taillard (1993) with 100 jobs and 20 machines. Objectives of makespan and total tardiness

For instance, Fig. 10.6 shows a second EAF for the same instance as in Fig. 10.5 but in this case for the RIPG algorithm described in Sect. 10.5.2.1.

The Pareto front given by RIPG is closer to the axes and it contains solutions closer to the ideal point. Although both scales in Figs. 10.5 and 10.6 are identical, it is not easy to compare both plots. In these cases is where the Diff-EAFs are handy. We can subtract both figures and we plot a Diff-EAF with two gradients of colour (blue and red) to positive and negative values of Diff-EAF, respectively. This way, we are able to display the two EAFs as a plot which permits a quick and graphical identification of which algorithm is dominating each zone of the objective space. The intensity of the colour assigned for each algorithm shows the probability of dominating a point in the space of that algorithm over the other algorithm. Points of low colour intensity show lower differences between the algorithms. Notice that white or no colour indicates that algorithms cannot generate solutions dominating this region, or that both algorithms have the same probability of generating dominating points and hence the difference is zero. Figure 10.7 shows the corresponding Diff-EAF. The 'moustache' type of plot indicates that, for this instance, RIPG dominates MOSAII$_M$ over the whole objective space. Recall however, that EAFs need to be examined instance by instance.

Although several algorithms and tools are available so as to construct EAF plots, EAF are hard to code, and hard to manage. Another main drawback of both EAFs and Diff-EAFs is that one plot has to be generated for each instance and for each pair of algorithms. Furthermore, to the best of our knowledge, and at least at the time of the

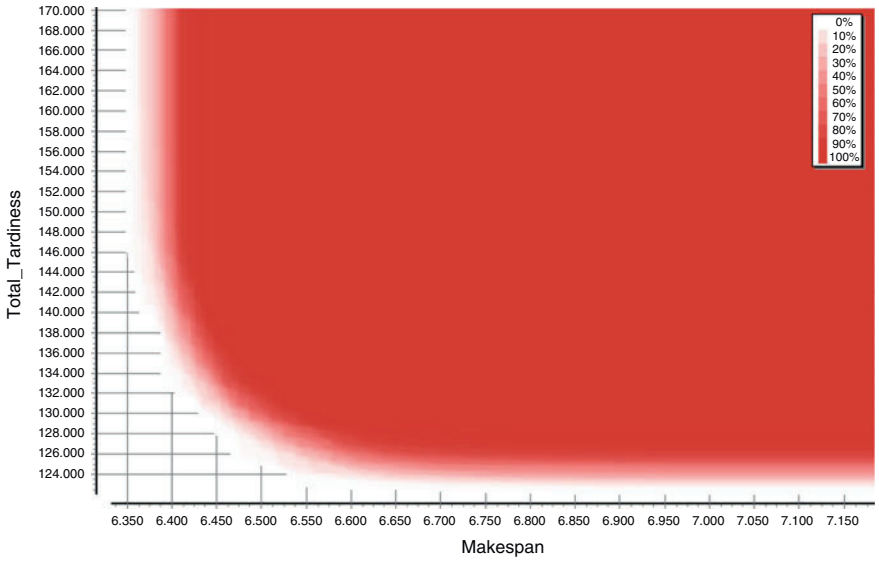

Fig. 10.6 Empirical Attainment Function (EAF) after 100 runs of the algorithm RIPG of Minella et al. (2011) over instance Ta081 of Taillard (1993) with 100 jobs and 20 machines. Objectives of makespan and total tardiness.

writing of this book, EAFs and Diff-EAFs have not been applied to multi-objective problems with more than two objectives.

10.7 Scheduling Interfering Jobs

So far, it has been assumed that the different objectives affect equally to all jobs to be scheduled. However, there are many situations in which several subsets of jobs can be identified among them, each set with different objectives. Particularly, within the rescheduling context, upon the arrival of new jobs to the shop floor, at least two sets of jobs naturally appear:

- Jobs already scheduled in an earlier scheduling decisions, but that have not been yet actually processed in the shop, and
- Arriving jobs which need to be scheduled.

One possibility is to schedule both sets anew, but in many situations it makes sense to employ different objectives for each set: Some performance measure is minimised for arriving jobs in order to obtain a short completion time for this set of jobs (such as for instance minimising their makespan or flowtime), while the objective for the existing jobs may be in line with either minimising the disruption from their initial schedule, or at least that this set of jobs does not exceed the already committed due dates.

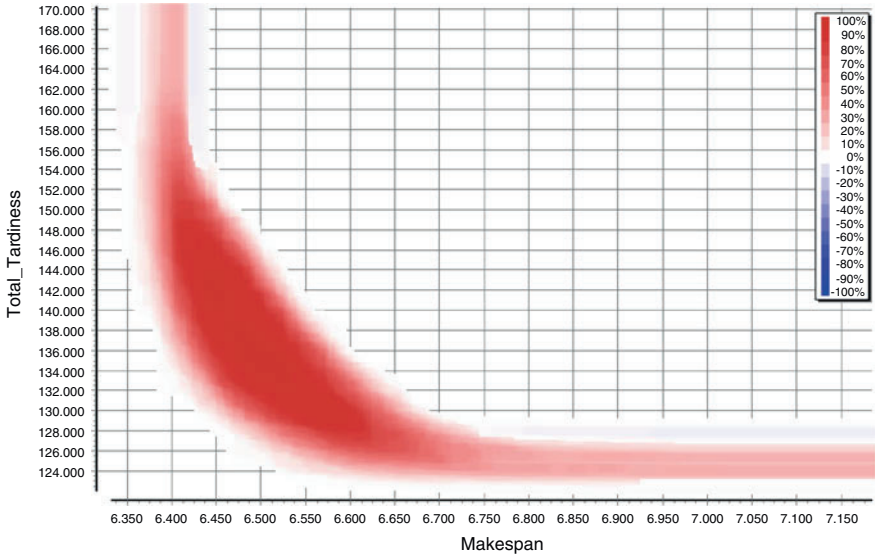

Fig. 10.7 Differential empirical attainment function (*Diff-EAF*) of the two EAFs given in Figs. 10.5 and 10.6 between algorithms MOSAII$_M$ and RIPG. Zones in blue indicate MOSAII$_M$ dominance whereas zones in red show RIPG dominance. Instance Ta081 of Taillard (1993) with 100 jobs and 20 machines. Objectives of makespan and total tardiness

So far, there is no unified name for these models (some of the most popular options are *interfering jobs*, *heterogenous-criteria scheduling* or *multi-agent scheduling*, which are, in fact, a general case of multicriteria scheduling models (note that all the previous multicriteria models are particular cases of the corresponding interfering jobs models.)

10.8 Conclusions and Further Readings

In this chapter we have given a brief introduction to multi-objective optimisation, techniques and quality assessment with special emphasis on scheduling problems. The initial motivation given in this text brings a clear conclusion: reality is multi-objective no matter how hard we try otherwise. We have presented some examples of multi-objective approaches classified mainly by how and when the decision-maker intervenes. The 'easiest' approaches are those in which the decision-maker provides information before the optimisation run. These are referred to as 'a priori' techniques. More difficult are the 'a posteriori' techniques that are basically represented by the Pareto optimisation approaches. These offer the decision-maker with a complete set of non-dominated solutions which represent the best trade-off solutions for the considered objectives.

Compared to single-objective scheduling, multi-objective techniques are less advanced and ready for action. This comes as no surprise, given the still limited penetration and usage of the many advanced existing single-objective optimisation methods. The difficulty and computational demands of multi-objective methods are also to blame here. Nevertheless, given the urgent need for these methods in practice, it is unquestionable that more research efforts must be done in order to bring advanced multi-objective algorithms to the production shops.

A very hot topic of research that is still going, at least at the time of the writing of this book, is the quality assessment of multi-objective algorithms. We have indicated how hard is to compare the outcomes of two different algorithms and the many methods that are being developed and that are currently being validated. Pareto-compliant performance indicators and EAFs seem to be the way to go.

A lot has been written about multi-objective (or multicriteria) optimisation. Some excellent books are those of Ehrgott (2005) or Collette and Siarry (2003), to name just a few. Additionally, there has been some book publications about multi-objective scheduling. The most notable contribution is the comprehensive book by T'Kindt and Billaut (2006) but there are also other less known works, as it is the book of Bagchi (1999).

Regarding the notation employed in the book, note that this is a minimum notation set as much more comprehensive listings and complete notations can be found in the books of Ehrgott (2005) and Collette and Siarry (2003) (to name just a few) and in the paper of Zitzler et al. (2003), later extended in Paquete (2005) and in Knowles et al. (2006). Furthermore, specific notation related to multi-objective scheduling is to be consulted in Vignier et al. (1999), T'Kindt and Billaut (2001), or in the excellent book of T'Kindt and Billaut (2006).

Regarding the types of models of multi-criteria scheduling, the list of references using a weighted combination of objectives in the multi-objective scheduling literature is endless. Some examples are Nagar et al. (1995); Sivrikaya-Şerifoğlu and Ulusoy (1998). Some of the earliest known papers about lexicographical approaches for scheduling problems are due to Emmons (1975). Other related references are Rajendran (1992); Gupta and Werner (1999), or Sarin and Hariharan (2000). The ε-constraint methodology has been employed by, for instance Daniels and Chambers (1990) or McCormick and Pinedo (1995). Some of the best well-known works about goal programming applied to scheduling problems are the papers of Selen and Hott (1986) and Wilson (1989). Regarding Pareto approaches, Daniels and Chambers (1990) was one of the first papers on the topic. More recent contributions are Varadharajan and Rajendran (2005), Geiger (2007) or Minella et al. (2011).

By far and long, most multi-objective metaheuristics refer to so-called evolutionary computation approach. In this regard, see the book by Bagchi (1999), or references like Deb (2001), Abraham et al. (2005), Coello et al. (2007), Coello (2004) or more recently, Knowles et al. (2008).

The issue of comparing multi-objective algorithms in a sound way is still a hot topic of research. Some important results are those of Zitzler et al. (2003), Paquete (2005), Knowles et al. (2006) or more recently, Zitzler et al. (2008). More specifically, a comprehensive study of the existing performance measures for Pareto optimisation

problems can be found in Zitzler et al. (2008) following the previous studies of Zitzler et al. (2003) and Knowles et al. (2006). The hypervolume and the epsilon indicators were introduced in Zitzler and Thiele (1999) and in Zitzler et al. (2003). A fast algorithm for the calculation of the hypervolume indicator—otherwise a very time-consuming indicator— is provided in Deb (2001). The I_ε^1 version of this indicator is provided by Knowles et al. (2006). EAFs were first proposed by Grunert da Fonseca et al. (2001) and were later analysed in more detail by Zitzler et al. (2008). Differential EAF are described in López-Ibáñez et al. (2006) and López-Ibáñez et al. (2010). Finally, a classification of interfering jobs is Perez-Gonzalez and Framinan (2013).

References

Abraham, A., Jain, L., and Goldberg, R., editors (2005). Evolutionary Multiobjective Optimization: Theoretical Advances and Applications. Springer-Verlag, London.

Bagchi, T. P. (1999). Multiobjective Scheduling by Genetic Algorithms. Kluwer Academic Publishers, Dordrecht.

Coello, C. A. (2004). Applications of Multi-Objective Evolutionary Algorithms. World Scientific Publishing Company, Singapore.

Coello, C. A., Lamont, G. B., and van Veldhuizen, D. A. (2007). Evolutionary Algorithms for Solving Multi-Objective Problems. Springer, New York, second edition.

Collette, Y. and Siarry, P. (2003). Multiobjective Optimization: Principles and Case Studies. Springer-Verlag, Berlin-Heidelberg.

Daniels, R. L. and Chambers, R. J. (1990). Multiobjective flow-shop scheduling. Naval research logistics, 37(6):981–995.

Deb, K. (2001). Multi-Objective Optimization Using Evolutionary Algorithms. John Wiley & Sons, West Sussex, England.

Deb, K. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation, 6(2):182–197.

Ehrgott, M. (2005). Multicriteria Optimization. Springer, Berlin-Heidelberg, second edition.

Emmons, H. (1975). A note on a scheduling problem with dual criteria. Naval Research Logistics Quarterly, 22(3):615–616.

Geiger, M. J. (2007). On operators and search space topology in multi-objective flow shop scheduling. European Journal of Operational Research, 181(1):195–206.

Grunert da Fonseca, V., Fonseca, C. M., and Hall, A. O. (2001). Inferential performance assessment of stochastic optimisers and the attainment function. In Zitzler, E., Deb, K., Thiele, L., Coello Coello, C. A., and Corne, D., editors, Proceedings of the First International Conference on Evolutionary Multi-Criterion Optimization, volume 1993 of Lecture Notes in Computer Science, pages 213–225, Berlin. Springer.

Gupta, J. N. D. and Werner, F. (1999). On the solution of 2-machine flow and open shop scheduling problems with secondary criteria. In 15th ISPE/IEE International Conference on CAD/CAM, Robotic, and Factories of the Future, Aguas de Lindoia, Sao Paulo, Brasil. Springer-Verlag.

Knowles, J., Corne, D., and Deb, K. (2008). Multiobjective Problem Solving From Nature: From Concepts to Applications. Springer-Verlag, Berlin-Heidelberg.

Knowles, J., Thiele, L., and Zitzler, E. (2006). A tutorial on the performance assessment of stochastic multiobjective optimizers. Technical Report 214, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, Switzerland. revised version.

López-Ibáñez, M., Paquete, L. F., and Stützle, T. (2006). Hybrid population-based algorithms for the bi-objective quadratic assignment problem. Journal of Mathematical Modelling and Algorithms, 5(1):111–137.

López-Ibáñez, M., Stützle, T., and Paquete, L. F. (2010). Graphical tools for the analysis of bi-objective optimization algorithms. In Proceedings of the 12th annual conference on Genetic and evolutionary computation (GECCO 2010), pages 1959–1962, New York, NY, USA. ACM.

Mc Cormick, S. T. and Pinedo, M. (1995). Scheduling *n* independant jobs on *m* uniform machines with both flowtime and makespan objectives: a parametric analysis. ORSA Journal on Computing, 7(1):63–77.

Minella, G., Ruiz, R., and Ciavotta, M. (2008). A review and evaluation of multi-objective algorithms for the flowshop scheduling problem. INFORMS Journal on Computing, 20(3):451–471.

Minella, G., Ruiz, R., Ciavotta, M. (2011). Restarted iterated pareto greedy algorithm for multi-objective flowshop scheduling problems. Computers & Operations Research, 38(11):1521–1533

Nagar, A., Heragu, S. S., and Haddock, J. (1995). A branch-and-bound approach for a two-machine flowshop scheduling problem. Journal of the Operational Research Society, 46(6):721–734.

Paquete, L. F. (2005). Stochastic Local Search Algorithms for Multiobjective Combinatorial Optimization: Method and Analysis. PhD thesis, Computer Science Department. Darmstadt University of Technology. Darmstadt, Germany.

Perez-Gonzalez, P. Framinan, J. M. (2014). A Common Framework and Taxonomy for Multicriteria Scheduling Problems with interfering and competing jobs: Multi-agent scheduling problems, European Journal of Operational Research, 235(1):1–16.

Rajendran, C. (1992). Two-stage flowshopnn scheduling problem with bicriteria. Journal of the Operational Research Society, 43(9):871–884.

Ruiz, R. and Stützle, T. (2007). A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. European Journal of Operational Research, 177(3):2033–2049.

Sarin, S. C. and Hariharan, R. (2000). A two machine bicriteria scheduling problem. International Journal of Production Economics, 65(2):125–139.

Selen, W. J. and Hott, D. D. (1986). A mixed-integer goal-programming formulation of the standard flowshop scheduling problem. Journal of the Operational Research Society, 37(12):1121–1128.

Sivrikaya-Şerifoğlu, F. and Ulusoy, G. (1998). A bicriteria two-machine permutation flowshopn problem. European Journal of Operational Research, 107(2):414–430.

Srinivas, N. and Deb, K. (1994). Multiobjective optimization using nondominated sorting in genetic algorithms. Evolutionary Computation, 2(3):221–248.

Taillard, E. (1993). Benchmarks for basic scheduling problems. European Journal of Operational Research, 64(2):278–285.

T'Kindt, V. and Billaut, J.-C. (2001). Multicriteria schduling problems: A survey. RAIRO Recherche operationnelle - Operations Research, 35(2):143–163.

T'Kindt, V. and Billaut, J.-C. (2006). Multicriteria Scheduling: Theory, Models and Algorithms. Springer, New York, second edition.

Varadharajan, T. and Rajendran, C. (2005). A multi-objective simulated-annealing algorithm for scheduling in flowshopns to minimize the makespan and total flowtime of jobs. European Journal of Operational Research, 167(3):772–795.

Vignier, A., Billaut, J.-C., and Proust, C. (1999). Les problèmes d'ordonnancement de type flowshop hybride: État de l'art. RAIRO Recherche opérationnelle, 33(2):117–183. In French.

Wilson, J. M. (1989). Alternative formulations of a flowshopn scheduling problem. Journal of the Operational Research Society, 40(4):395–399.

Zitzler, E., Knowles, J., and Thiele, L. (2008). Quality assessment of pareto set approximations. In Multiobjective Optimization: Interactive and Evolutionary Approaches, pages 373–404, Berlin, Heidelberg. Springer-Verlag.

Zitzler, E. and Thiele, L. (1999). Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach. IEEE Transactions on Evolutionary Computation, 3(4): 257–271.

Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M., and da Fonseca, V. G. (2003). Performance assessment of multiobjective optimizers: an analysis and review. IEEE, Transactions on Evolutionary Computation, 7(2):117–132.